4.2 TRANSMISSION CONTROL PROTOCOL -- TCP

4.2.1 INTRODUCTION

The Transmission Control Protocol TCP [TCP:1] is the primary virtual-circuit transport protocol for the Internet suite. TCP provides reliable, in-sequence delivery of a full-duplex stream of octets (8-bit bytes). TCP is used by those applications needing reliable, connection-oriented transport service, e.g., mail (SMTP), file transfer (FTP), and virtual terminal service (Telnet); requirements for these application-layer protocols are described in [INTRO:1].

4.2.2 PROTOCOL WALK-THROUGH

4.2.2.1 Well-Known Ports: RFC-793 Section 2.7

DISCUSSION:

TCP reserves port numbers in the range 0-255 for "well-known" ports, used to access services that are standardized across the Internet. The remainder of the port space can be freely allocated to application processes. Current well-known port definitions are listed in the RFC entitled "Assigned Numbers" [INTRO:6]. A prerequisite for defining a new well-known port is an RFC documenting the proposed service in enough detail to allow new implementations.

Some systems extend this notion by adding a third subdivision of the TCP port space: reserved ports, which are generally used for operating-system-specific services. For example, reserved ports might fall between 256 and some system-dependent upper limit. Some systems further choose to protect well-known and reserved ports by permitting only privileged users to open TCP connections with those port values. This is perfectly reasonable as long as the host does not assume that all hosts protect their low-numbered ports in this manner.

4.2.2.2 Use of Push: RFC-793 Section 2.8

When an application issues a series of SEND calls without setting the PUSH flag, the TCP MAY aggregate the data internally without sending it. Similarly, when a series of segments is received without the PSH bit, a TCP MAY queue the data internally without passing it to the receiving application.

The PSH bit is not a record marker and is independent of segment boundaries. The transmitter SHOULD collapse successive PSH bits when it packetizes data, to send the largest possible segment.

A TCP MAY implement PUSH flags on SEND calls. If PUSH flags are not implemented, then the sending TCP: (1) must not buffer data indefinitely, and (2) MUST set the PSH bit in the last buffered segment (i.e., when there is no more queued data to be sent).

The discussion in RFC-793 on pages 48, 50, and 74 erroneously implies that a received PSH flag must be passed to the application layer. Passing a received PSH flag to the application layer is now OPTIONAL.

An application program is logically required to set the PUSH flag in a SEND call whenever it needs to force delivery of the data to avoid a communication deadlock. However, a TCP SHOULD send a maximum-sized segment whenever possible, to improve performance (see Section 4.2.3.4).

DISCUSSION:

When the PUSH flag is not implemented on SEND calls, i.e., when the application/TCP interface uses a pure streaming model, responsibility for aggregating any tiny data fragments to form reasonable sized segments is partially borne by the application layer.

Generally, an interactive application protocol must set the PUSH flag at least in the last SEND call in each command or response sequence. A bulk transfer protocol like FTP should set the PUSH flag on the last segment of a file or when necessary to prevent buffer deadlock.

At the receiver, the PSH bit forces buffered data to be delivered to the application (even if less than a full buffer has been received). Conversely, the lack of a PSH bit can be used to avoid unnecessary wakeup calls to the application process; this can be an important performance optimization for large timesharing hosts. Passing the PSH bit to the receiving application allows an analogous optimization within the application.

4.2.2.3 Window Size: RFC-793 Section 3.1

The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will not work. It is RECOMMENDED that implementations reserve 32-bit fields for the send and receive window sizes in the connection record and do all window computations with 32 bits.

DISCUSSION:

It is known that the window field in the TCP header is too small for high-speed, long-delay paths. Experimental TCP options have been defined to extend the window size; see for example [TCP:11]. In anticipation of the adoption of such an extension, TCP implementors should treat windows as 32 bits.

4.2.2.4 Urgent Pointer: RFC-793 Section 3.1

The second sentence is in error: the urgent pointer points to the sequence number of the LAST octet (not LAST+1) in a sequence of urgent data. The description on page 56 (last sentence) is correct.

A TCP MUST support a sequence of urgent data of any length.

A TCP MUST inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenever the Urgent pointer advances in the data stream. There MUST be a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read.

DISCUSSION:

Although the Urgent mechanism may be used for any application, it is normally used to send "interrupt"-type commands to a Telnet program (see "Using Telnet Synch Sequence" section in [INTRO:1]).

The asynchronous or "out-of-band" notification will allow the application to go into "urgent mode", reading data from the TCP connection. This allows control commands to be sent to an application whose normal input buffers are full of unprocessed data.

IMPLEMENTATION:

The generic ERROR-REPORT() upcall described in Section 4.2.4.1 is a possible mechanism for informing the application of the arrival of urgent data.

4.2.2.5 TCP Options: RFC-793 Section 3.1

A TCP MUST be able to receive a TCP option in any segment. A TCP MUST ignore without error any TCP option it does not implement, assuming that the option has a length field (all TCP options defined in the future will have length fields). TCP MUST be prepared to handle an illegal option length (e.g., zero) without crashing; a suggested procedure is to reset the connection and log the reason.

4.2.2.6 Maximum Segment Size Option: RFC-793 Section 3.1

TCP MUST implement both sending and receiving the Maximum Segment Size option [TCP:4].

TCP SHOULD send an MSS (Maximum Segment Size) option in every SYN segment when its receive MSS differs from the default 536, and MAY send it always.

If an MSS option is not received at connection setup, TCP MUST assume a default send MSS of 536 (576-40) [TCP:4].

The maximum size of a segment that TCP really sends, the "effective send MSS," MUST be the smaller of the send MSS (which reflects the available reassembly buffer size at the remote host) and the largest size permitted by the IP layer:

```
Eff.snd.MSS =

    min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize
```

where:

*   SendMSS is the MSS value received from the remote host, or the default 536 if no MSS option is received.

*   MMS_S is the maximum size for a transport-layer message that TCP may send.

*   TCPhdrsize is the size of the TCP header; this is normally 20, but may be larger if TCP options are to be sent.

*   IPoptionsize is the size of any IP options that TCP will pass to the IP layer with the current message.

The MSS value to be sent in an MSS option must be less than or equal to:

MMS_R - 20

where MMS_R is the maximum size for a transport-layer message that can be received (and reassembled). TCP obtains MMS_R and MMS_S from the IP layer; see the generic call GET_MAXSIZES in Section 3.4.

DISCUSSION:

The choice of TCP segment size has a strong effect on performance. Larger segments increase throughput by amortizing header size and per-datagram processing overhead over more data bytes; however, if the packet is so large that it causes IP fragmentation, efficiency drops sharply if any fragments are lost [IP:9].

Some TCP implementations send an MSS option only if the destination host is on a non-connected network. However, in general the TCP layer may not have the appropriate information to make this decision, so it is preferable to leave to the IP layer the task of determining a suitable MTU for the Internet path. We therefore recommend that TCP always send the option (if not 536) and that the IP layer determine MMS_R as specified in 3.3.3 and 3.4. A proposed IP-layer mechanism to measure the MTU would then modify the IP layer without changing TCP.

4.2.2.7 TCP Checksum: RFC-793 Section 3.1

Unlike the UDP checksum (see Section 4.1.3.4), the TCP checksum is never optional. The sender MUST generate it and the receiver MUST check it.

4.2.2.8 TCP Connection State Diagram: RFC-793 Section 3.2, page 23

There are several problems with this diagram:

(a) The arrow from SYN-SENT to SYN-RCVD should be labeled with "snd SYN,ACK", to agree with the text on page 68 and with Figure 8.

(b) There could be an arrow from SYN-RCVD state to LISTEN state, conditioned on receiving a RST after a passive open (see text page 70).

(c) It is possible to go directly from FIN-WAIT-1 to the TIME-WAIT state (see page 75 of the spec).

4.2.2.9 Initial Sequence Number Selection: RFC-793 Section 3.3, page 27

A TCP MUST use the specified clock-driven selection of initial sequence numbers.

4.2.2.10 Simultaneous Open Attempts: RFC-793 Section 3.4, page 32

There is an error in Figure 8: the packet on line 7 should be identical to the packet on line 5.

A TCP MUST support simultaneous open attempts.

DISCUSSION:

It sometimes surprises implementors that if two applications attempt to simultaneously connect to each other, only one connection is generated instead of two. This was an intentional design decision; don't try to "fix" it.

4.2.2.11 Recovery from Old Duplicate SYN: RFC-793 Section 3.4, page 33

Note that a TCP implementation MUST keep track of whether a connection has reached SYN_RCVD state as the result of a passive OPEN or an active OPEN.

4.2.2.12 RST Segment: RFC-793 Section 3.4

A TCP SHOULD allow a received RST segment to include data.

DISCUSSION

It has been suggested that a RST segment could contain ASCII text that encoded and explained the cause of the RST. No standard has yet been established for such data.

4.2.2.13 Closing a Connection: RFC-793 Section 3.5

A TCP connection may terminate in two ways: (1) the normal TCP close sequence using a FIN handshake, and (2) an "abort" in which one or more RST segments are sent and the connection state is immediately discarded. If a TCP connection is closed by the remote site, the local application MUST be informed whether it closed normally or was aborted.

The normal TCP close sequence delivers buffered data reliably in both directions. Since the two directions of a TCP connection are closed independently, it is possible for a connection to be "half closed," i.e., closed in only one direction, and a host is permitted to continue sending data in the open direction on a half-closed connection.

A host MAY implement a "half-duplex" TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection. If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost.

When a connection is closed actively, it MUST linger in TIME-WAIT state for a time 2xMSL (Maximum Segment Lifetime). However, it MAY accept a new SYN from the remote TCP to reopen the connection directly from TIME-WAIT state, if it:

(1) assigns its initial sequence number for the new connection to be larger than the largest sequence number it used on the previous connection incarnation, and

(2) returns to TIME-WAIT state if the SYN turns out to be an old duplicate.

DISCUSSION:

TCP's full-duplex data-preserving close is a feature that is not included in the analogous ISO transport protocol TP4.

Some systems have not implemented half-closed connections, presumably because they do not fit into the I/O model of their particular operating system. On these systems, once an application has called CLOSE, it can no longer read input data from the connection; this is referred to as a "half-duplex" TCP close sequence.

The graceful close algorithm of TCP requires that the connection state remain defined on (at least) one end of the connection, for a timeout period of 2xMSL, i.e., 4 minutes. During this period, the (remote socket, local socket) pair that defines the connection is busy and cannot be reused. To shorten the time that a given port pair is tied up, some TCPs allow a new SYN to be accepted in TIME-WAIT state.

4.2.2.14 Data Communication: RFC-793 Section 3.7, page 40

Since RFC-793 was written, there has been extensive work on TCP algorithms to achieve efficient data communication. Later sections of the present document describe required and recommended TCP algorithms to determine when to send data (Section 4.2.3.4), when to send an acknowledgment (Section 4.2.3.2), and when to update the window (Section 4.2.3.3).

DISCUSSION:

One important performance issue is "Silly Window Syndrome" or "SWS" [TCP:5], a stable pattern of small incremental window movements resulting in extremely poor TCP performance. Algorithms to avoid SWS are described below for both the sending side (Section 4.2.3.4) and the receiving side (Section 4.2.3.3).

In brief, SWS is caused by the receiver advancing the right window edge whenever it has any new buffer space available to receive data and by the sender using any incremental window, no matter how small, to send more data [TCP:5]. The result can be a stable pattern of sending tiny data segments, even though both sender and receiver have a large total buffer space for the connection. SWS can only occur during the transmission of a large amount of data; if the connection goes quiescent, the problem will disappear. It is caused by typical straightforward implementation of window management, but the sender and receiver algorithms given below will avoid it.

Another important TCP performance issue is that some applications, especially remote login to character-at-a-time hosts, tend to send streams of one-octet data segments. To avoid deadlocks, every TCP SEND call from such applications must be "pushed", either explicitly by the application or else implicitly by TCP. The result may be a stream of TCP segments that contain one data octet each, which makes very inefficient use of the Internet and contributes to Internet congestion. The Nagle Algorithm described in Section 4.2.3.4 provides a simple and effective solution to this problem. It does have the effect of clumping characters over Telnet connections; this may initially surprise users accustomed to single-character echo, but user acceptance has not been a problem.

Note that the Nagle algorithm and the send SWS avoidance algorithm play complementary roles in improving performance. The Nagle algorithm discourages sending tiny segments when the data to be sent increases in small increments, while the SWS avoidance algorithm discourages small segments resulting from the right window edge advancing in small increments.

A careless implementation can send two or more acknowledgment segments per data segment received. For example, suppose the receiver acknowledges every data segment immediately. When the application program subsequently consumes the data and increases the available receive buffer space again, the receiver may send a second acknowledgment segment to update the window at the sender. The extreme case occurs with single-character segments on TCP connections using the Telnet protocol for remote login service. Some implementations have been observed in which each incoming 1-character segment generates three return segments: (1) the acknowledgment, (2) a one byte increase in the window, and (3) the echoed character, respectively.

4.2.2.15 Retransmission Timeout: RFC-793 Section 3.7, page 41

The algorithm suggested in RFC-793 for calculating the retransmission timeout is now known to be inadequate; see Section 4.2.3.1 below.

Recent work by Jacobson [TCP:7] on Internet congestion and TCP retransmission stability has produced a transmission algorithm combining "slow start" with "congestion avoidance". A TCP MUST implement this algorithm.

If a retransmitted packet is identical to the original packet (which implies not only that the data boundaries have not changed, but also that the window and acknowledgment fields of the header have not changed), then the same IP Identification field MAY be used (see Section 3.2.1.5).

IMPLEMENTATION:

Some TCP implementors have chosen to "packetize" the data stream, i.e., to pick segment boundaries when segments are originally sent and to queue these segments in a "retransmission queue" until they are acknowledged. Another design (which may be simpler) is to defer packetizing until each time data is transmitted or retransmitted, so there will be no segment retransmission queue.

In an implementation with a segment retransmission queue, TCP performance may be enhanced by repacketizing the segments awaiting acknowledgment when the first retransmission timeout occurs. That is, the outstanding segments that fitted would be combined into one maximum-sized segment, with a new IP Identification value. The TCP would then retain this combined segment in the retransmit queue until it was acknowledged. However, if the first two segments in the retransmission queue totalled more than one maximum-sized segment, the TCP would retransmit only the first segment using the original IP Identification field.

4.2.2.16 Managing the Window: RFC-793 Section 3.7, page 41

A TCP receiver SHOULD NOT shrink the window, i.e., move the right window edge to the left. However, a sending TCP MUST be robust against window shrinking, which may cause the "useable window" (see Section 4.2.3.4) to become negative.

If this happens, the sender SHOULD NOT send new data, but SHOULD retransmit normally the old unacknowledged data between SND.UNA and SND.UNA+SND.WND. The sender MAY also retransmit old data beyond SND.UNA+SND.WND, but SHOULD NOT time out the connection if data beyond the right window edge is not acknowledged. If the window shrinks to zero, the TCP MUST probe it in the standard way (see next Section).

DISCUSSION:

Many TCP implementations become confused if the window shrinks from the right after data has been sent into a larger window. Note that TCP has a heuristic to select the latest window update despite possible datagram reordering; as a result, it may ignore a window update with a smaller window than previously offered if neither the sequence number nor the acknowledgment number is increased.

4.2.2.17 Probing Zero Windows: RFC-793 Section 3.7, page 42

Probing of zero (offered) windows MUST be supported.

A TCP MAY keep its offered receive window closed indefinitely. As long as the receiving TCP continues to send acknowledgments in response to the probe segments, the sending TCP MUST allow the connection to stay open.

DISCUSSION:

It is extremely important to remember that ACK (acknowledgment) segments that contain no data are not reliably transmitted by TCP. If zero window probing is not supported, a connection may hang forever when an ACK segment that re-opens the window is lost.

The delay in opening a zero window generally occurs when the receiving application stops taking data from its TCP. For example, consider a printer daemon application, stopped because the printer ran out of paper.

The transmitting host SHOULD send the first zero-window probe when a zero window has existed for the retransmission timeout period (see Section 4.2.2.15), and SHOULD increase exponentially the interval between successive probes.

DISCUSSION:

This procedure minimizes delay if the zero-window condition is due to a lost ACK segment containing a window-opening update. Exponential backoff is recommended, possibly with some maximum interval not specified here. This procedure is similar to that of the retransmission algorithm, and it may be possible to combine the two procedures in the implementation.

4.2.2.18 Passive OPEN Calls: RFC-793 Section 3.8

Every passive OPEN call either creates a new connection record in LISTEN state, or it returns an error; it MUST NOT affect any previously created connection record.

A TCP that supports multiple concurrent users MUST provide an OPEN call that will functionally allow an application to LISTEN on a port while a connection block with the same local port is in SYN-SENT or SYN-RECEIVED state.

DISCUSSION:

Some applications (e.g., SMTP servers) may need to handle multiple connection attempts at about the same time. The probability of a connection attempt failing is reduced by giving the application some means of listening for a new connection at the same time that an earlier connection attempt is going through the three-way handshake.

IMPLEMENTATION:

Acceptable implementations of concurrent opens may permit multiple passive OPEN calls, or they may allow "cloning" of LISTEN-state connections from a single passive OPEN call.

4.2.2.19 Time to Live: RFC-793 Section 3.9, page 52

RFC-793 specified that TCP was to request the IP layer to send TCP segments with TTL = 60. This is obsolete; the TTL value used to send TCP segments MUST be configurable. See Section 3.2.1.7 for discussion.

4.2.2.20 Event Processing: RFC-793 Section 3.9

While it is not strictly required, a TCP SHOULD be capable of queueing out-of-order TCP segments. Change the "may" in the last sentence of the first paragraph on page 70 to "should".

DISCUSSION:

Some small-host implementations have omitted segment queueing because of limited buffer space. This omission may be expected to adversely affect TCP throughput, since loss of a single segment causes all later segments to appear to be "out of sequence".

In general, the processing of received segments MUST be implemented to aggregate ACK segments whenever possible. For example, if the TCP is processing a series of queued segments, it MUST process them all before sending any ACK segments.

Here are some detailed error corrections and notes on the Event Processing section of RFC-793.

(a) CLOSE Call, CLOSE-WAIT state, p. 61: enter LAST-ACK state, not CLOSING.

(b) LISTEN state, check for SYN (pp. 65, 66): With a SYN bit, if the security/compartment or the precedence is wrong for the segment, a reset is sent. The wrong form of reset is shown in the text; it should be:

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

(c) SYN-SENT state, Check for SYN, p. 68: When the connection enters ESTABLISHED state, the following variables must be set:

```
SND.WND <- SEG.WND
SND.WL1 <- SEG.SEQ
SND.WL2 <- SEG.ACK
```

(d) Check security and precedence, p. 71: The first heading "ESTABLISHED STATE" should really be a list of all states other than SYN-RECEIVED: ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, and TIME-WAIT.

(e) Check SYN bit, p. 71: "In SYN-RECEIVED state and if the connection was initiated with a passive OPEN, then return this connection to the LISTEN state and return. Otherwise...".

(f) Check ACK field, SYN-RECEIVED state, p. 72: When the connection enters ESTABLISHED state, the variables listed in (c) must be set.

(g) Check ACK field, ESTABLISHED state, p. 72: The ACK is a duplicate if SEG.ACK =< SND.UNA (the = was omitted). Similarly, the window should be updated if: SND.UNA =< SEG.ACK =< SND.NXT.

(h) USER TIMEOUT, p. 77:

It would be better to notify the application of the timeout rather than letting TCP force the connection closed. However, see also Section 4.2.3.5.

4.2.2.21 Acknowledging Queued Segments: RFC-793 Section 3.9

A TCP MAY send an ACK segment acknowledging RCV.NXT when a valid segment arrives that is in the window but not at the left window edge.

DISCUSSION:

RFC-793 (see page 74) was ambiguous about whether or not an ACK segment should be sent when an out-of-order segment was received, i.e., when SEG.SEQ was unequal to RCV.NXT.

One reason for ACKing out-of-order segments might be to support an experimental algorithm known as "fast retransmit". With this algorithm, the sender uses the "redundant" ACK's to deduce that a segment has been lost before the retransmission timer has expired. It counts the number of times an ACK has been received with the same value of SEG.ACK and with the same right window edge. If more than a threshold number of such ACK's is received, then the segment containing the octets starting at SEG.ACK is assumed to have been lost and is retransmitted, without awaiting a timeout. The threshold is chosen to compensate for the maximum likely segment reordering in the Internet. There is not yet enough experience with the fast retransmit algorithm to determine how useful it is.

4.2.3 SPECIFIC ISSUES

4.2.3.1 Retransmission Timeout Calculation

A host TCP MUST implement Karn's algorithm and Jacobson's algorithm for computing the retransmission timeout ("RTO").

- o Jacobson's algorithm for computing the smoothed round-trip ("RTT") time incorporates a simple measure of the variance [TCP:7].

- o Karn's algorithm for selecting RTT measurements ensures that ambiguous round-trip times will not corrupt the calculation of the smoothed round-trip time [TCP:6].

This implementation also MUST include "exponential backoff" for successive RTO values for the same segment. Retransmission of SYN segments SHOULD use the same algorithm as data segments.

DISCUSSION:

There were two known problems with the RTO calculations specified in RFC-793. First, the accurate measurement of RTTs is difficult when there are retransmissions. Second, the algorithm to compute the smoothed round-trip time is inadequate [TCP:7], because it incorrectly assumed that the variance in RTT values would be small and constant. These problems were solved by Karn's and Jacobson's algorithm, respectively.

The performance increase resulting from the use of these improvements varies from noticeable to dramatic. Jacobson's algorithm for incorporating the measured RTT variance is especially important on a low-speed link, where the natural variation of packet sizes causes a large variation in RTT. One vendor found link utilization on a 9.6kb line went from 10% to 90% as a result of implementing Jacobson's variance algorithm in TCP.

The following values SHOULD be used to initialize the estimation parameters for a new connection:

(a) RTT = 0 seconds.

(b) RTO = 3 seconds. (The smoothed variance is to be initialized to the value that will result in this RTO).

The recommended upper and lower bounds on the RTO are known to be inadequate on large internets. The lower bound SHOULD be measured in fractions of a second (to accommodate high speed LANs) and the upper bound should be 2*MSL, i.e., 240 seconds.

DISCUSSION:

Experience has shown that these initialization values are reasonable, and that in any case the Karn and Jacobson algorithms make TCP behavior reasonably insensitive to the initial parameter choices.

4.2.3.2 When to Send an ACK Segment

A host that is receiving a stream of TCP data segments can increase efficiency in both the Internet and the hosts by sending fewer than one ACK (acknowledgment) segment per data segment received; this is known as a "delayed ACK" [TCP:5].

A TCP SHOULD implement a delayed ACK, but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment.

DISCUSSION:

A delayed ACK gives the application an opportunity to update the window and perhaps to send an immediate response. In particular, in the case of character-mode remote login, a delayed ACK can reduce the number of segments sent by the server by a factor of 3 (ACK, window update, and echo character all combined in one segment).

In addition, on some large multi-user hosts, a delayed ACK can substantially reduce protocol processing overhead by reducing the total number of packets to be processed [TCP:5]. However, excessive delays on ACK's can disturb the round-trip timing and packet "clocking" algorithms [TCP:7].

4.2.3.3 When to Send a Window Update

A TCP MUST include a SWS avoidance algorithm in the receiver [TCP:5].

IMPLEMENTATION:

The receiver's SWS avoidance algorithm determines when the right window edge may be advanced; this is customarily known as "updating the window". This algorithm combines with the delayed ACK algorithm (see Section 4.2.3.2) to determine when an ACK segment containing the current window will really be sent to the receiver. We use the notation of RFC-793; see Figures 4 and 5 in that document.

The solution to receiver SWS is to avoid advancing the right window edge RCV.NXT+RCV.WND in small increments, even if data is received from the network in small segments.

Suppose the total receive buffer space is RCV.BUFF. At any given moment, RCV.USER octets of this total may be tied up with data that has been received and acknowledged but which the user process has not yet consumed. When the connection is quiescent, RCV.WND = RCV.BUFF and RCV.USER = 0.

Keeping the right window edge fixed as data arrives and is acknowledged requires that the receiver offer less than its full buffer space, i.e., the receiver must specify a RCV.WND that keeps RCV.NXT+RCV.WND constant as RCV.NXT increases. Thus, the total buffer space RCV.BUFF is generally divided into three parts:

```
          |<------- RCV.BUFF ---------------->|
               1             2            3
      ----|---------|------------------|------|----
                 RCV.NXT               ^
                                    (Fixed)

      1 - RCV.USER =  data received but not yet consumed;
      2 - RCV.WND =   space advertised to sender;
      3 - Reduction = space available but not yet
                      advertised.
```

The suggested SWS avoidance algorithm for the receiver is to keep RCV.NXT+RCV.WND fixed until the reduction satisfies:

```
     RCV.BUFF - RCV.USER - RCV.WND  >=

            min( Fr * RCV.BUFF, Eff.snd.MSS )
```

where Fr is a fraction whose recommended value is 1/2, and Eff.snd.MSS is the effective send MSS for the connection (see Section 4.2.2.6). When the inequality is satisfied, RCV.WND is set to RCV.BUFF-RCV.USER.

Note that the general effect of this algorithm is to advance RCV.WND in increments of Eff.snd.MSS (for realistic receive buffers: Eff.snd.MSS < RCV.BUFF/2). Note also that the receiver must use its own Eff.snd.MSS, assuming it is the same as the sender's.

4.2.3.4 When to Send Data

A TCP MUST include a SWS avoidance algorithm in the sender.

A TCP SHOULD implement the Nagle Algorithm [TCP:9] to coalesce short segments. However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection. In all cases, sending data is also subject to the limitation imposed by the Slow Start algorithm (Section 4.2.2.15).

DISCUSSION:
The Nagle algorithm is generally as follows:

If there is unacknowledged data (i.e., SND.NXT > SND.UNA), then the sending TCP buffers all user data (regardless of the PSH bit), until the outstanding data has been acknowledged or until the TCP can send a full-sized segment (Eff.snd.MSS bytes; see Section 4.2.2.6).

Some applications (e.g., real-time display window updates) require that the Nagle algorithm be turned off, so small data segments can be streamed out at the maximum rate.

IMPLEMENTATION:

The sender's SWS avoidance algorithm is more difficult than the receivers's, because the sender does not know (directly) the receiver's total buffer space RCV.BUFF. An approach which has been found to work well is for the sender to calculate Max(SND.WND), the maximum send window it has seen so far on the connection, and to use this value as an estimate of RCV.BUFF. Unfortunately, this can only be an estimate; the receiver may at any time reduce the size of RCV.BUFF. To avoid a resulting deadlock, it is necessary to have a timeout to force transmission of data, overriding the SWS avoidance algorithm. In practice, this timeout should seldom occur.

The "useable window" [TCP:5] is:

```
U = SND.UNA + SND.WND - SND.NXT
```

i.e., the offered window less the amount of data sent but not acknowledged. If D is the amount of data queued in the sending TCP but not yet sent, then the following set of rules is recommended.

Send data:

(1) if a maximum-sized segment can be sent, i.e, if:

```
min(D,U) >= Eff.snd.MSS;
```

(2) or if the data is pushed and all queued data can be sent now, i.e., if:

```
[SND.NXT = SND.UNA and] PUSHED and D <= U
```

(the bracketed condition is imposed by the Nagle algorithm);

(3) or if at least a fraction Fs of the maximum window can be sent, i.e., if:

```
[SND.NXT = SND.UNA and]

        min(D.U) >= Fs * Max(SND.WND);
```

(4) or if data is PUSHed and the override timeout occurs.

Here Fs is a fraction whose recommended value is 1/2. The override timeout should be in the range 0.1 - 1.0 seconds. It may be convenient to combine this timer with the timer used to probe zero windows (Section 4.2.2.17).

Finally, note that the SWS avoidance algorithm just specified is to be used instead of the sender-side algorithm contained in [TCP:5].

4.2.3.5 TCP Connection Failures

Excessive retransmission of the same segment by TCP indicates some failure of the remote host or the Internet path. This failure may be of short or long duration. The following procedure MUST be used to handle excessive retransmissions of data segments [IP:11]:

(a) There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment. R1 and R2 might be measured in time units or as a count of retransmissions.

(b) When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice (see Section 3.3.1.4) to the IP layer, to trigger dead-gateway diagnosis.

(c) When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection.

(d) An application MUST be able to set the value for R2 for a particular connection. For example, an interactive application might set R2 to "infinity," giving the user control over when to disconnect.

(d) TCP SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see Section 4.2.4.1), when R1 is reached and before R2. This will allow a remote login (User Telnet) application program to inform the user, for example.

The value of R1 SHOULD correspond to at least 3 retransmissions, at the current RTO. The value of R2 SHOULD correspond to at least 100 seconds.

An attempt to open a TCP connection could fail with excessive retransmissions of the SYN segment or by receipt of a RST segment or an ICMP Port Unreachable. SYN retransmissions MUST be handled in the general way just described for data retransmissions, including notification of the application layer.

However, the values of R1 and R2 may be different for SYN and data segments. In particular, R2 for a SYN segment MUST be set large enough to provide retransmission of the segment for at least 3 minutes. The application can close the connection (i.e., give up on the open attempt) sooner, of course.

DISCUSSION:
Some Internet paths have significant setup times, and the number of such paths is likely to increase in the future.

4.2.3.6 TCP Keep-Alives

Implementors MAY include "keep-alives" in their TCP implementations, although this practice is not universally accepted. If keep-alives are included, the application MUST be able to turn them on or off for each TCP connection, and they MUST default to off.

Keep-alive packets MUST only be sent when no data or acknowledgement packets have been received for the connection within an interval. This interval MUST be configurable and MUST default to no less than two hours.

It is extremely important to remember that ACK segments that contain no data are not reliably transmitted by TCP. Consequently, if a keep-alive mechanism is implemented it MUST NOT interpret failure to respond to any specific probe as a dead connection.

An implementation SHOULD send a keep-alive segment with no data; however, it MAY be configurable to send a keep-alive segment containing one garbage octet, for compatibility with erroneous TCP implementations.

DISCUSSION:

A "keep-alive" mechanism periodically probes the other end of a connection when the connection is otherwise idle, even when there is no data to be sent. The TCP specification does not include a keep-alive mechanism because it could: (1) cause perfectly good connections to break during transient Internet failures; (2) consume unnecessary bandwidth ("if no one is using the connection, who cares if it is still good?"); and (3) cost money for an Internet path that charges for packets.

Some TCP implementations, however, have included a keep-alive mechanism. To confirm that an idle connection is still active, these implementations send a probe segment designed to elicit a response from the peer TCP. Such a segment generally contains SEG.SEQ = SND.NXT-1 and may or may not contain one garbage octet of data. Note that on a quiet connection SND.NXT = RCV.NXT, so that this SEG.SEQ will be outside the window. Therefore, the probe causes the receiver to return an acknowledgment segment, confirming that the connection is still live. If the peer has dropped the connection due to a network partition or a crash, it will respond with a RST instead of an acknowledgment segment.

Unfortunately, some misbehaved TCP implementations fail to respond to a segment with SEG.SEQ = SND.NXT-1 unless the segment contains data. Alternatively, an implementation could determine whether a peer responded correctly to keep-alive packets with no garbage data octet.

A TCP keep-alive mechanism should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure.

4.2.3.7 TCP Multihoming

If an application on a multihomed host does not specify the local IP address when actively opening a TCP connection, then the TCP MUST ask the IP layer to select a local IP address before sending the (first) SYN. See the function GET_SRCADDR() in Section 3.4.

At all other times, a previous segment has either been sent or received on this connection, and TCP MUST use the same local address is used that was used in those previous segments.

4.2.3.8 IP Options

When received options are passed up to TCP from the IP layer, TCP MUST ignore options that it does not understand.

A TCP MAY support the Time Stamp and Record Route options.

An application MUST be able to specify a source route when it actively opens a TCP connection, and this MUST take precedence over a source route received in a datagram.

When a TCP connection is OPENed passively and a packet arrives with a completed IP Source Route option (containing a return route), TCP MUST save the return route and use it for all segments sent on this connection. If a different source route arrives in a later segment, the later definition SHOULD override the earlier one.

4.2.3.9 ICMP Messages

TCP MUST act on an ICMP error message passed up from the IP layer, directing it to the connection that created the error. The necessary demultiplexing information can be found in the IP header contained within the ICMP message.

- o Source Quench

  TCP MUST react to a Source Quench by slowing transmission on the connection. The RECOMMENDED procedure is for a Source Quench to trigger a "slow start," as if a retransmission timeout had occurred.

- o Destination Unreachable -- codes 0, 1, 5

  Since these Unreachable messages indicate soft error conditions, TCP MUST NOT abort the connection, and it SHOULD make the information available to the application.

DISCUSSION:

TCP could report the soft error condition directly to the application layer with an upcall to the ERROR_REPORT routine, or it could merely note the message and report it to the application only when and if the TCP connection times out.

- Destination Unreachable -- codes 2-4

  These are hard error conditions, so TCP SHOULD abort the connection.

- Time Exceeded -- codes 0, 1

  This should be handled the same way as Destination Unreachable codes 0, 1, 5 (see above).

- Parameter Problem

  This should be handled the same way as Destination Unreachable codes 0, 1, 5 (see above).

4.2.3.10 Remote Address Validation

A TCP implementation MUST reject as an error a local OPEN call for an invalid remote IP address (e.g., a broadcast or multicast address).

An incoming SYN with an invalid source address must be ignored either by TCP or by the IP layer (see Section 3.2.1.3).

A TCP implementation MUST silently discard an incoming SYN segment that is addressed to a broadcast or multicast address.

4.2.3.11 TCP Traffic Patterns

IMPLEMENTATION:

The TCP protocol specification [TCP:1] gives the implementor much freedom in designing the algorithms that control the message flow over the connection -- packetizing, managing the window, sending acknowledgments, etc. These design decisions are difficult because a TCP must adapt to a wide range of traffic patterns. Experience has shown that a TCP implementor needs to verify the design on two extreme traffic patterns:

- o Single-character Segments

  Even if the sender is using the Nagle Algorithm, when a TCP connection carries remote login traffic across a low-delay LAN the receiver will generally get a stream of single-character segments. If remote terminal echo mode is in effect, the receiver's system will generally echo each character as it is received.

- o Bulk Transfer

  When TCP is used for bulk transfer, the data stream should be made up (almost) entirely of segments of the size of the effective MSS. Although TCP uses a sequence number space with byte (octet) granularity, in bulk-transfer mode its operation should be as if TCP used a sequence space that counted only segments.

Experience has furthermore shown that a single TCP can effectively and efficiently handle these two extremes.

The most important tool for verifying a new TCP implementation is a packet trace program. There is a large volume of experience showing the importance of tracing a variety of traffic patterns with other TCP implementations and studying the results carefully.

4.2.3.12 Efficiency

IMPLEMENTATION:
Extensive experience has led to the following suggestions for efficient implementation of TCP:

(a) Don't Copy Data

In bulk data transfer, the primary CPU-intensive tasks are copying data from one place to another and checksumming the data. It is vital to minimize the number of copies of TCP data. Since the ultimate speed limitation may be fetching data across the memory bus, it may be useful to combine the copy with checksumming, doing both with a single memory fetch.

(b) Hand-Craft the Checksum Routine

A good TCP checksumming routine is typically two to five times faster than a simple and direct implementation of the definition. Great care and clever coding are often required and advisable to make the checksumming code "blazing fast". See [TCP:10].

(c) Code for the Common Case

TCP protocol processing can be complicated, but for most segments there are only a few simple decisions to be made. Per-segment processing will be greatly speeded up by coding the main line to minimize the number of decisions in the most common case.

4.2.4 TCP/APPLICATION LAYER INTERFACE

4.2.4.1 Asynchronous Reports

There MUST be a mechanism for reporting soft TCP error conditions to the application. Generically, we assume this takes the form of an application-supplied ERROR_REPORT routine that may be upcalled [INTRO:7] asynchronously from the transport layer:

```
ERROR_REPORT(local connection name, reason, subreason)
```

The precise encoding of the reason and subreason parameters is not specified here. However, the conditions that are reported asynchronously to the application MUST include:

* ICMP error message arrived (see 4.2.3.9)

* Excessive retransmissions (see 4.2.3.5)

* Urgent pointer advance (see 4.2.2.4).

However, an application program that does not want to receive such ERROR_REPORT calls SHOULD be able to effectively disable these calls.

DISCUSSION:

These error reports generally reflect soft errors that can be ignored without harm by many applications. It has been suggested that these error report calls should default to "disabled," but this is not required.

4.2.4.2 Type-of-Service

The application layer MUST be able to specify the Type-of-Service (TOS) for segments that are sent on a connection. It not required, but the application SHOULD be able to change the TOS during the connection lifetime. TCP SHOULD pass the current TOS value without change to the IP layer, when it sends segments on the connection.

The TOS will be specified independently in each direction on the connection, so that the receiver application will specify the TOS used for ACK segments.

TCP MAY pass the most recently received TOS up to the application.

DISCUSSION

Some applications (e.g., SMTP) change the nature of their communication during the lifetime of a connection, and therefore would like to change the TOS specification.

Note also that the OPEN call specified in RFC-793 includes a parameter ("options") in which the caller can specify IP options such as source route, record route, or timestamp.

4.2.4.3 Flush Call

Some TCP implementations have included a FLUSH call, which will empty the TCP send queue of any data for which the user has issued SEND calls but which is still to the right of the current send window. That is, it flushes as much queued send data as possible without losing sequence number synchronization. This is useful for implementing the "abort output" function of Telnet.

4.2.4.4 Multihoming

The user interface outlined in sections 2.7 and 3.8 of RFC-793 needs to be extended for multihoming. The OPEN call MUST have an optional parameter:

```
OPEN( ... [local IP address,] ... )
```

to allow the specification of the local IP address.

DISCUSSION:
Some TCP-based applications need to specify the local IP address to be used to open a particular connection; FTP is an example.

IMPLEMENTATION:
A passive OPEN call with a specified "local IP address" parameter will await an incoming connection request to that address. If the parameter is unspecified, a passive OPEN will await an incoming connection request to any local IP address, and then bind the local IP address of the connection to the particular address that is used.

For an active OPEN call, a specified "local IP address" parameter will be used for opening the connection. If the parameter is unspecified, the networking software will choose an appropriate local IP address (see Section 3.3.4.2) for the connection

4.2.5 TCP REQUIREMENT SUMMARY

```
                                                 |        | | | |S| |
                                                 |        | | | |H| |F
                                                 |        | | | |O|M|o
                                                 |        | |S| |U|U|o
                                                 |        | |H| |L|S|t
                                                 |        |M|O| |D|T|n
                                                 |        |U|U|M| | |o
                                                 |        |S|L|A|N|N|t
                                                 |        |T|D|Y|O|O|t
FEATURE                                          |SECTION | | | |T|T|e
-------------------------------------------------|--------|-|-|-|-|-|--
                                                 |        | | | | | |
Push flag                                        |        | | | | | |
  Aggregate or queue un-pushed data              |4.2.2.2 | | |x| | |
  Sender collapse successive PSH flags           |4.2.2.2 | |x| | | |
  SEND call can specify PUSH                     |4.2.2.2 | | |x| | |
```

If cannot: sender buffer indefinitely	4.2.2.2					x	
If cannot: PSH last segment	4.2.2.2	x					
Notify receiving ALP of PSH	4.2.2.2			x			1
Send max size segment when possible	4.2.2.2		x				
Window							
Treat as unsigned number	4.2.2.3	x					
Handle as 32-bit number	4.2.2.3		x				
Shrink window from right	4.2.2.16				x		
Robust against shrinking window	4.2.2.16	x					
Receiver's window closed indefinitely	4.2.2.17			x			
Sender probe zero window	4.2.2.17	x					
First probe after RTO	4.2.2.17		x				
Exponential backoff	4.2.2.17		x				
Allow window stay zero indefinitely	4.2.2.17	x					
Sender timeout OK conn with zero wind	4.2.2.17					x	
Urgent Data							
Pointer points to last octet	4.2.2.4	x					
Arbitrary length urgent data sequence	4.2.2.4	x					
Inform ALP asynchronously of urgent data	4.2.2.4	x					1
ALP can learn if/how much urgent data Q'd	4.2.2.4	x					1
TCP Options							
Receive TCP option in any segment	4.2.2.5	x					
Ignore unsupported options	4.2.2.5	x					
Cope with illegal option length	4.2.2.5	x					
Implement sending & receiving MSS option	4.2.2.6	x					
Send MSS option unless 536	4.2.2.6		x				
Send MSS option always	4.2.2.6			x			
Send-MSS default is 536	4.2.2.6	x					
Calculate effective send seg size	4.2.2.6	x					
TCP Checksums							
Sender compute checksum	4.2.2.7	x					
Receiver check checksum	4.2.2.7	x					
Use clock-driven ISN selection	4.2.2.9	x					
Opening Connections							
Support simultaneous open attempts	4.2.2.10	x					
SYN-RCVD remembers last state	4.2.2.11	x					
Passive Open call interfere with others	4.2.2.18					x	
Function: simultan. LISTENs for same port	4.2.2.18	x					
Ask IP for src address for SYN if necc.	4.2.3.7	x					
Otherwise, use local addr of conn.	4.2.3.7	x					
OPEN to broadcast/multicast IP Address	4.2.3.14					x	
Silently discard seg to bcast/mcast addr	4.2.3.14	x					

Closing Connections							
RST can contain data	4.2.2.12		x				
Inform application of aborted conn	4.2.2.13	x					
Half-duplex close connections	4.2.2.13			x			
Send RST to indicate data lost	4.2.2.13		x				
In TIME-WAIT state for 2xMSL seconds	4.2.2.13	x					
Accept SYN from TIME-WAIT state	4.2.2.13			x			
Retransmissions							
Jacobson Slow Start algorithm	4.2.2.15	x					
Jacobson Congestion-Avoidance algorithm	4.2.2.15	x					
Retransmit with same IP ident	4.2.2.15			x			
Karn's algorithm	4.2.3.1	x					
Jacobson's RTO estimation alg.	4.2.3.1	x					
Exponential backoff	4.2.3.1	x					
SYN RTO calc same as data	4.2.3.1		x				
Recommended initial values and bounds	4.2.3.1		x				
Generating ACK's:							
Queue out-of-order segments	4.2.2.20		x				
Process all Q'd before send ACK	4.2.2.20	x					
Send ACK for out-of-order segment	4.2.2.21			x			
Delayed ACK's	4.2.3.2		x				
Delay < 0.5 seconds	4.2.3.2	x					
Every 2nd full-sized segment ACK'd	4.2.3.2	x					
Receiver SWS-Avoidance Algorithm	4.2.3.3	x					
Sending data							
Configurable TTL	4.2.2.19	x					
Sender SWS-Avoidance Algorithm	4.2.3.4	x					
Nagle algorithm	4.2.3.4		x				
Application can disable Nagle algorithm	4.2.3.4	x					
Connection Failures:							
Negative advice to IP on R1 retxs	4.2.3.5	x					
Close connection on R2 retxs	4.2.3.5	x					
ALP can set R2	4.2.3.5	x					1
Inform ALP of R1<=retxs<R2	4.2.3.5		x				1
Recommended values for R1, R2	4.2.3.5		x				
Same mechanism for SYNs	4.2.3.5	x					
R2 at least 3 minutes for SYN	4.2.3.5	x					
Send Keep-alive Packets:	4.2.3.6			x			
- Application can request	4.2.3.6	x					
- Default is "off"	4.2.3.6	x					
- Only send if idle for interval	4.2.3.6	x					
- Interval configurable	4.2.3.6	x					

- Default at least 2 hrs.	4.2.3.6	x					
- Tolerant of lost ACK's	4.2.3.6	x					
IP Options							
Ignore options TCP doesn't understand	4.2.3.8	x					
Time Stamp support	4.2.3.8			x			
Record Route support	4.2.3.8			x			
Source Route:							
ALP can specify	4.2.3.8	x					1
Overrides src rt in datagram	4.2.3.8	x					
Build return route from src rt	4.2.3.8	x					
Later src route overrides	4.2.3.8		x				
Receiving ICMP Messages from IP	4.2.3.9	x					
Dest. Unreach (0,1,5) => inform ALP	4.2.3.9		x				
Dest. Unreach (0,1,5) => abort conn	4.2.3.9					x	
Dest. Unreach (2-4) => abort conn	4.2.3.9		x				
Source Quench => slow start	4.2.3.9		x				
Time Exceeded => tell ALP, don't abort	4.2.3.9		x				
Param Problem => tell ALP, don't abort	4.2.3.9		x				
Address Validation							
Reject OPEN call to invalid IP address	4.2.3.10	x					
Reject SYN from invalid IP address	4.2.3.10	x					
Silently discard SYN to bcast/mcast addr	4.2.3.10	x					
TCP/ALP Interface Services							
Error Report mechanism	4.2.4.1	x					
ALP can disable Error Report Routine	4.2.4.1		x				
ALP can specify TOS for sending	4.2.4.2	x					
Passed unchanged to IP	4.2.4.2		x				
ALP can change TOS during connection	4.2.4.2		x				
Pass received TOS up to ALP	4.2.4.2			x			
FLUSH call	4.2.4.3			x			
Optional local IP addr parm. in OPEN	4.2.4.4	x					

FOOTNOTES:

(1) "ALP" means Application-Layer program.