February 1979

IEN: 81

TRANSMISSION CONTROL PROTOCOL

TCP
Version 4

February 1979

prepared for

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

by

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, California 90291

TABLE OF CONTENTS

PREFACE

This document describes the Transmission Control Protocol (TCP). There have been five previous editions of the TCP specification, and the present text draws heavily from them. There have been many contributors to this document both in terms of concepts and in terms of text.

Jon Postel

Editor

< INC-PROJECT, TCP-JAN-79.NLS.35, >, 17-Feb-79 00:23 JBP ;;;;

February 1979
IEN:81
Replaces: IENs 55, 44, 40, 27, 21, 5

Transmission Control Protocol

Version 4

1. INTRODUCTION

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and especially in interconnected systems of such networks.

This document describes the functions to be performed by the Transmission Control Protocol, the program that implements it, and its interface to programs or users that require its services.

1.1. Motivation

Computer communication systems are playing an increasingly important role in military, government, and civilian environments. This document primarily focuses its attention on military computer communication requirements, especially robustness in the presence of communication unreliability, but many of these problems are found in the civilian and government sector as well.

As strategic and tactical computer communication networks are developed and deployed, it is essential to provide means of interconnecting them and to provide standard interprocess communication protocols which can support a broad range of applications. In anticipation of the need for such standards, the Deputy Undersecretary of Defense for Research and Engineering has declared the Transmission Control Protocol (TCP) described herein to be a basis for DoD-wide inter-process communication protocol standardization.

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below the TCP layer. At most, the TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, the TCP should be able to operate above a wide spectrum of communication systems ranging from hard wired connections to packet-switched or circuit-switched networks.

The TCP fits into a layered protocol architecture just above a basic Internet Datagram Protocol [1] which provides a way for the TCP to send and receive variable-length segments of information enclosed in internet datagram "envelopes". The internet datagram provides a means for addressing source and destination TCPs in different networks, and that layer of protocol also deals with any fragmentation or reassembly of the TCP segments which might be required to achieve transport and delivery through multiple networks and interconnecting gateways.

Protocol Layering

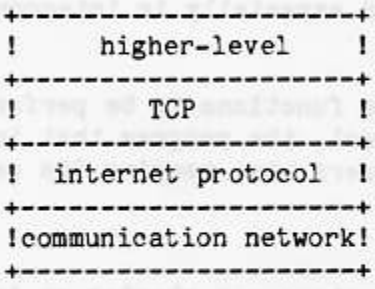


Figure 1

Much of this document is written in the context of TCP implementations which are co-resident with higher level protocols in the host computer. As a practical matter, many computer systems will be connected to networks via front-end computers which house the TCP and internet protocol layers, as well as network specific software. The TCP specification describes an interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented.

1.2. Scope

The TCP is intended to provide a reliable process-to-process interprocess communication service in a multinetwork environment. The TCP is intended to be a host-to-host protocol in common use in multiple networks.

1.3. About this Document

This document represents a specification of the behavior required of any TCP implementation, both in its interactions with higher level protocols and in its interactions with other TCPs. The rest of this section offers a very brief view of the protocol interfaces and operation. Section 2 summarizes the philosophical basis for the TCP design. Section 3 offers both a detailed description of the actions required of TCP when various events occur (arrival of new segments,

user calls, errors, etc.) and the details of the formats of TCP segments.

1.4. Interfaces

The TCP interfaces on one side to user or application processes and on the other side to a lower level protocol such as Internetwork Datagram Protocol.

The interface between an application process and the TCP is illustrated in reasonable detail. This interface consists of a set of calls much like the calls an operating system provides to application process for manipulating files. For example, there are calls to open and close connections and to send and receive letters on established connections. It is also expected that the TCP can asynchronously communicate with application programs. Although considerable freedom is permitted to TCP implementors to design interfaces which are appropriate to a particular operating system environment, this TCP specification requires a certain minimum functionality to be achieved at the TCP/user interface for any valid implementation.

The interface between TCP and lower level protocol is essentially unspecified except that it is assume there is a mechanism whereby the two can asynchronously pass information to each other. Typically, one expects the lower level protocol to specify this interface. TCP is designed to work in a very general environment of interconnected networks. Therefore, the lower level protocol which is assumed throughout this document is the Internet Datagram Protocol.

1.5. Operation

As noted above, the primary purpose of the TCP is to provide reliable logical circuit or connection service between pairs of processes. To provide this service on top of a less reliable internet communication system requires facilities in the following areas.

Basic Data Transfer
Reliability
Flow Control
Multiplexing
Connections

The basic operation of the TCP is each of these areas is described in the following paragraphs.

Basic Data Transfer:

The TCP is able to transfer a continuous stream of octets in each direction between its users by packaging some number of octets into segments for transmission through the internet system. In this stream mode, the TCP's decide when to block and forward data at their own convenience.

For users who desire a record-oriented service, the TCP also permits the user to submit records, called letters, for transmission. When the sending user indicates a record boundary (end-of-letter), this causes the TCP's to promptly forward and deliver data up to that point to the receiver. However, not all letter boundaries are given to the receiver (several letters may be delivered as a unit).

Reliability:

The TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

As long as the TCPs continue to function properly and the internet system does not become completely partitioned, no transmission errors will affect the users. All errors in the internet communication system are recovered by the TCP.

Flow Control:

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. For stream mode, the window indicates an allowed number of octets that the sender may transmit before receiving further permission. It is also possible for the TCP to operate in a mode where buffer sizes and letter boundaries are incorporated in flow control.

Multiplexing:

To allow for many process-to-process connections within a single Host, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection. That is, different connections may have a common socket on one side, but the sockets on the other sides must be different.

The binding of ports to processes is handled independently by each Host. However, it proves useful to attach frequently used processes (e.g., a "logger" or timesharing service) to fixed sockets which are made known to the public. These services can then be accessed through the known addresses. Establishing and learning the port addresses of other processes may involve more dynamic mechanisms.

Connections:

The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection. Each connection is uniquely specified by a pair of sockets identifying its two sides.

When two processes wish to communicate, their TCP's must first establish a connection (initialize the status information on each side). When their communication is complete, the connection is terminated or closed o free the resources for other uses.

Since connections must be established over the unreliable internet communication system, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

2. PHILOSOPHY

2.1. Elements of the Internetwork System

The internetwork environment consists of hosts connected to networks which are in turn interconnected via gateways. It is assumed here that the networks may be either local networks (e.g. the ETHERNET) or large networks (e.g. the ARPANET), but in any case are based on packet switching technology. The active agents that produce and consume messages are processes. Various levels of protocols in the networks, the gateways, and the hosts, support an interprocess communication system that provides two way data flow on logical connections between process ports.

We specifically assume that data is transmitted from host to host through means of a set of networks. When we say network we have in mind a packet switched network (PSN). This assumption is probably unnecessary, since a circuit switched network or a hybrid combination of the two could also be used; but for concreteness, we explicitly assume that the hosts are connected to one or more packet switches (PS) of a PSN.

The term packet is used generically here to mean the data of one transaction between a host and a packet switch. The format of data blocks exchanged between the packet switches in a network will generally not be of concern to us.

Hosts are computers attached to a network, and from the communication network's point of view, are the sources and destinations of packets. Processes are viewed as the active elements in host computers (in accordance with the fairly common definition of a process as a program in execution). Even terminals and files or other I/O devices are viewed as communicating with each other through the use of processes. Thus, all communication is viewed as inter-process communication.

Since a process may need to distinguish among several communication streams between itself and another process (or processes), we imagine that each process may have a number of ports through which it communicates with the ports of other processes.

2.2. Model of Operation

Processes transmit data by calling on the TCP and passing buffers of data as arguments. The TCP packages the data from these buffers into segments, and calls on the internet module to transmit each segment to the destination TCP. The receiving TCP places the data from a segment into the receiving users buffer and notifies the receiving user. The TCPs include control information in the segments which they use to ensure reliable ordered data transmission.

The model of internet communication is that there is a basic gateway (or internet protocol module) associated with each TCP which provides an interface to the local network. This basic gateway packages TCP segments inside internet datagrams and routes these datagrams to a destination or intermediate gateway. To transmit the datagram through the local network it is embedded in a local network packet.

The packet switches may perform further packaging, fragmentation, or other operations to achieve the delivery of the local packet to the destination gateway.

At a gateway between networks, the internet datagram is "unwrapped" from its local packet and examined to determine through which network the internet datagram should travel next. The internet datagram is then "wrapped" in a local packet suitable to the next network and routed to the next gateway.

A gateway is permitted to break up an internet datagram into smaller internet datagram fragments if this is necessary for transmission through the next network. To do this, the gateway produces a set of internet datagrams, each carrying a fragment. Fragments may be broken into smaller ones at intermediate gateways. The internet datagram fragment format is designed so that the destination gateway can reassemble fragments into internet datagrams.

A destination gateway unwraps the segment from the datagram (after reassembling the datagram, if necessary) and passes it to the destination TCP.

2.3. The Host Environment

The TCP is assumed to be a module in a time sharing operating system. The users access the TCP much like they would access the file system. The TCP may call on other operating system functions, for example, to manage data structures. The actual interface to the network is assumed to be controlled by a device driver module. The TCP does not call on the network device driver directly, but rather calls on the internet datagram protocol module which may in turn call on the device driver.

Though it is assumed here that processes are supported by the host operating system, the mechanisms of TCP do not preclude implementation of the TCP in a front-end processor. However, in such an implementation, a host-to-front-end protocol must provide the functionality to support the type of TCP-user interface described above.

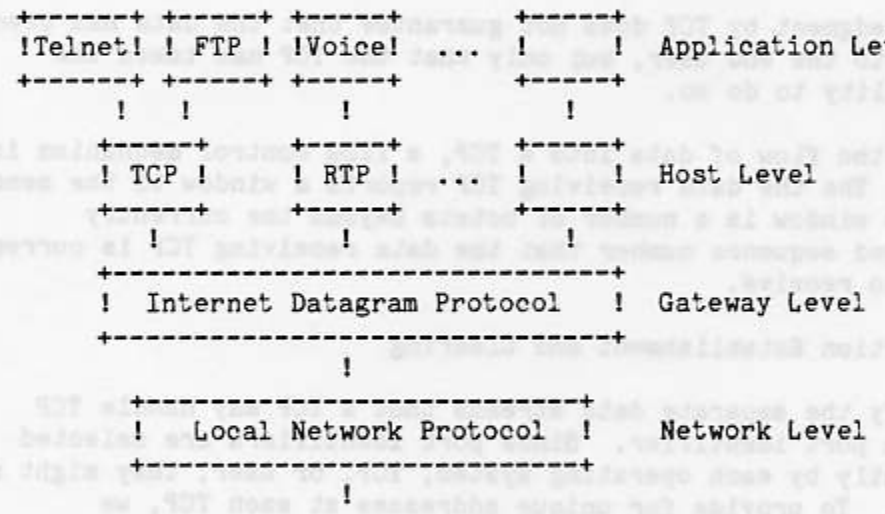
2.4. Interfaces

The TCP/user interface provides for calls made by the user on the TCP to OPEN or CLOSE a connection, to SEND or RECEIVE data, or to obtain STATUS about a connection. These call are like other calls from user programs on the operating system, for example, the calls to open, read from, and close a file.

The TCP/internet interface provides calls to send and receive datagrams addressed to TCP modules in hosts anywhere in the internet system.

2.5. Relation to Other Protocols

The following diagram illustrates the place of the TCP in the protocol hierarchy:



Protocol Relationships

Figure 2.

It is expected that the TCP will be able to support higher level protocols efficiently. It should be easy to interface higher level protocols like the ARPANET TELNET and FTP [2] to the TCP.

2.6. Reliable Communication

A stream of data sent on a TCP connection is delivered reliably and in order at the destination.

Transmission is made reliable via the use of sequence numbers and acknowledgments. Conceptually, each octet of data is assigned a sequence number. The sequence number of the first octet of data in a segment is the sequence number transmitted with that segment and is called the segment sequence number. Segments also carry an acknowledgment number which is the sequence number of the most recent data octet of transmissions in the reverse direction which has been accepted. When the TCP transmits a segment it puts a copy on a retransmission queue and starts a timer, when the acknowledgment for that data is received the segment is deleted from the queue. If the acknowledgment is not received before the timer runs out the segment is retransmitted.

An acknowledgment by TCP does not guarantee that the data has been delivered to the end user, but only that the TCP has taken the responsibility to do so.

To govern the flow of data into a TCP, a flow control mechanism is employed. The the data receiving TCP reports a window to the sending TCP. This window is a number of octets beyond the currently acknowledged sequence number that the data receiving TCP is currently prepared to receive.

2.7. Connection Establishment and Clearing

To identify the separate data streams that a TCP may handle TCP provides a port identifier. Since port identifiers are selected independently by each operating system, TCP, or user, they might not be unique. To provide for unique addresses at each TCP, we concatenate an internet address identifying the TCP with a port identifier to create a socket which will be unique throughout all networks connected together.

A connection is fully specified by the pair of sockets at the ends, since the same local socket may participate in many connections to different foreign sockets. A connection can be used to carry data in both directions, that is, it is "full duplex".

TCP's are free to associate ports with processes however they choose. However, several basic concepts seem necessary in any implementation. There must be well-known sockets which the TCP associates only with the "appropriate" processes by some means. We envision that processes may "own" ports, and that processes can only initiate connections on

the ports they own. (Means for implementing ownership is a local issue, but we envision a Request Port user command, or a method of uniquely allocating a group of ports to a given process, e.g., by associating the high order bits of a port name with a given process.)

A connection is specified in the OPEN call by the local port and foreign socket arguments. In return the TCP supplies a (short) local connection name by which the user refers to the connection in subsequent calls. There are several things that must be remembered about a connection. To store this information we imagine that there is a data structure called a Transmission Control Block (TCB). One implementation strategy would have the local connection name be a pointer to the TCB for this connection. The OPEN call also specifies whether the connection establishment is to actively pursued, or to be passively waited for.

A foreign socket of all zeros is called unspecified. The purpose behind unspecified sockets is to provide a sort of "general delivery" facility (useful for processes offering services). This is allowed only for passive OPENs.

A service process that wished to provide services for unknown other processes could issue a passive OPEN request with an unspecified foreign socket. Then a connection cold be made with any process that requested a connection to this local socket. It would help if this local socket were known to be associated with this service.

Well-known sockets are a convenient mechanism for a priori associating a socket address with a standard service. For instance, the "Telnet-Server" process might be permanently assigned to a particular socket, and other sockets might be reserved for File Transfer, Remote Job Entry, Text Generator, Echoer, and Sink processes (the last three being for test purposes). A socket address might be reserved for access to a "Look-Up" service which would return the specific socket at which a newly created service would be provided. The concept of a well-known socket is part of the TCP specification, but the assignment of sockets to services is outside this specification.

Processes can issue passive OPENs and wait for matching calls from other processes and be informed by the TCP when connections have been established. Two processes which issue calls to each other at the same time are correctly connected. This flexibility is critical for the support of distributed computing in which components act asynchronously with each other.

There are two cases for matching the sockets in the local request and an incoming segment. In the first case, the local request has fully specified the foreign socket. In this case, the match must be exact.

In the second case, the local request has left the foreign socket unspecified. In this case, any foreign socket is acceptable as long as the local sockets match.

If there are several pending passive OPENs (recorded in TCBs) with the same local socket, an incoming segment should be matched to an request with the specific foreign socket in the segment, if such an request exists, before selecting an request with an unspecified foreign socket.

The procedures to establish and clear connections utilize synchronize (SYN) and finis (FIN) control flags and involve an exchange of three messages. This exchange has been termed a three-way hand shake [3].

A connection is initiated by the rendezvous of an arriving segment containing a SYN and a waiting TCB entry created by a user OPEN command. The matching of local and foreign sockets determines when a connection has been initiated. The connection becomes "established" when sequence numbers have been synchronized in both directions.

The clearing of a connection also involves the exchange of segments, in this case carrying the FIN control flag.

2.8. Data Communication

The data that flows on a connection may be thought of as a stream of octets, or as a sequence of records. In TCP the records are called letters and are of variable length. The sending user indicates in each SEND call if the data in that call completes a letter by the setting of the end-of-letter parameter.

The length of a letter may be such that it must be broken into segments before it can be transmitted to its destination. We assume that the segments will normally be reassembled into a letter before being passed to the receiving process. A segment may contain all or a part of a letter, but a segment never contains parts of more than one letter. The end of a letter is marked by the appearance of an EOL control flag in a segment. When a TCP has a complete letter, it must not wait for more data from the sending process before passing the letter to the receiving process.

There is a coupling between letters as sent and the use of buffers of data that cross the TCP/user interface. Each time an end of letter (EOL) flag is associated with data placed into the receiving user's buffer, the buffer is returned to the user for processing even if the buffer is not filled. If a letter is longer than the user's buffer, the letter is passed to the user in buffer size units.

If the sender formed a series of letters that were exactly the size of the receivers buffers, it might occur that the receiver would get fewer end of letter notifications than the sender issued. This is because the sending TCP might be able to pack two letters in one segment for transmission, and the single EOL flag transmitted would refer to the second letter.

The TCP is responsible for regulating the flow of segments to and from the on the connections, as a way of preventing itself from becoming saturated or overloaded with traffic. This is done using a window flow control mechanism. The data receiving TCP reports to the data sending TCP a window which is the range of sequence numbers of data octets that data receiving TCP is currently prepared to accept.

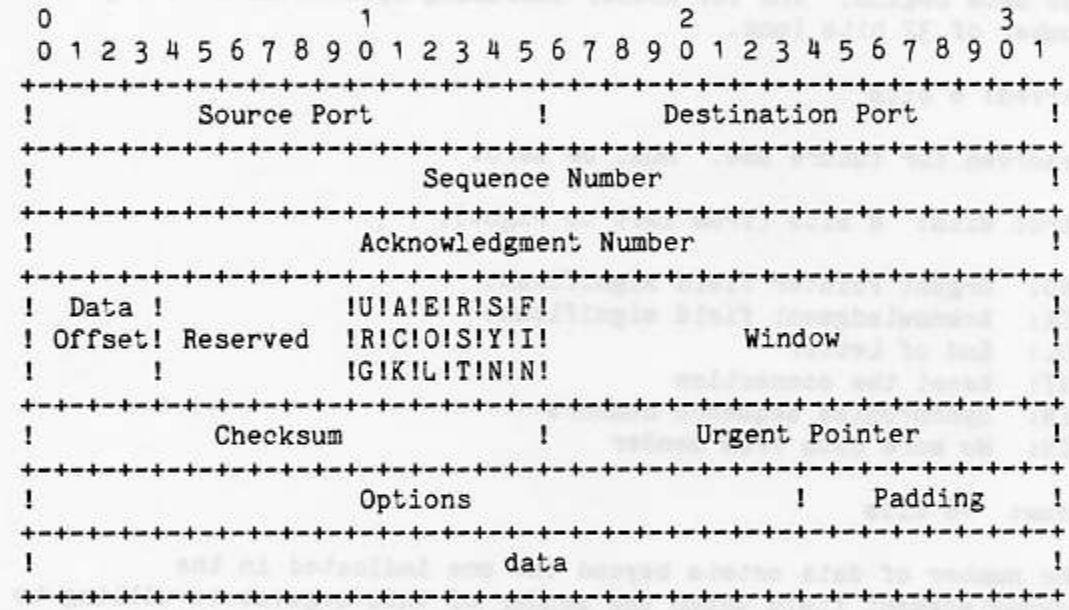
TCP also provides a means to communicate to the receiver of data that at some point further along in the data stream than the receiver is currently reading there is urgent data. TCP does not attempt to define what the user specifically do upon being notified of pending urgent data, but the general notion is that the receiving process should take action to read through the end urgent data quickly.

3. FUNCTIONAL SPECIFICATION

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Datagram Protocol header carries several information fields, including the source and destination host addresses [1]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 3.

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

The sequence number of the first data octet in this segment.

Acknowledgment Number: 32 bits

If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header including options is an integral number of 32 bits long.

Reserved: 6 bits

Reserved for future use. Must be zero.

Control Bits: 8 bits (from left to right):

URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
EOL: End of Letter
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

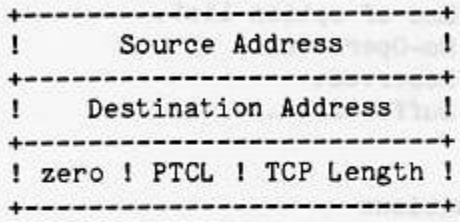
The number of data octets beyond the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a 96 bit pseudo header prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the

TCP protection against misrouted segments. This information is carried in the Internet Datagram Protocol and is transfered across the TCP/Network interface.



The TCP Length is the TCP header plus the data length in octets (this is not a transmitted quantity).

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field should only be interpreted in segments with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. All options have the same basic format:

Option kind: 8 bits

Option length: 8 bits

Length in octets (including the two octets of length and kind information)

There are two special cases for options.

The first is the End-of-Options option. Only one octet is associated with this option, the kind octet itself.

The second is the No-Operation option and is also one octet long.

Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option should be header padding (i.e., zero).

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
0	-	End of option list.
1	-	No-Operation.
100	-	Reserved.
105	4	Buffer Size.

Specific Option Definitions

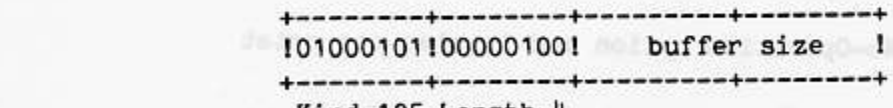
End of Option List

This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header.

No-Operation

This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

Buffer Size



Buffer Size Option Data: 16 bits

If this option is present, then it communicates the receive buffer size at the TCP which sends this segment. This field should only be sent in segments with the SYN control bit set. If this option is not used, the default buffer size of one octet is assumed.

Padding: variable

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

3.2. Terminology

Before we can discuss very much about the operation of the TCP we need to introduce some detailed terminology. The maintenance of a TCP connection requires the remembering of several facts. We conceive of these facts being stored in a connection record called a Transmission Control Block or TCB. Among the facts stored in the TCB are the local and remote socket numbers, pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition several variables relating to the send and receive sequence numbers are stored in the TCB.

Send Sequence Variables

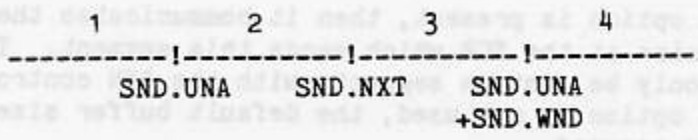
```
SND.UNA - send unacknowledged
SND.NXT - send sequence
SND.WND - send window
SND.BS  - send buffer size
SND.UP  - send urgent pointer
SND.WL  - send sequence number used for last window update
ISS     - initial send sequence number
```

Receive Sequence Variables

```
RCV.NXT - receive sequence
RCV.WND - receive window
RCV.BS  - receive buffer size
RCV.UP  - receive urgent pointer
IRS     - initial receive sequence number
```

The following diagrams may help to relate some of these variables to the sequence space.

Send Sequence Space

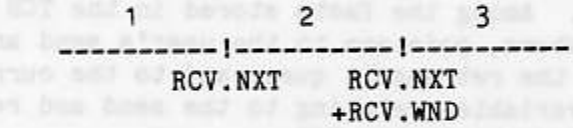


1 - old sequence numbers which have been acknowledged
2 - sequence numbers of unacknowledged data
3 - sequence numbers allowed for new data transmission
4 - future sequence numbers which are not yet allowed

Send Sequence Space

Figure 4.

Receive Sequence Space



1 - old sequence numbers which have been acknowledged
2 - sequence numbers allowed for new reception
3 - future sequence numbers which are not yet allowed

Receive Sequence Space

Figure 5.

There are also some variables used frequently in the discussion that take their values from the fields of the current segment.

Current Segment Variables

SEG.SEQ - segment sequence number
SEG.ACK - segment acknowledgment number
SEG.LEN - segment length
SEG.WND - segment window
SEG.UP - segment urgent pointer

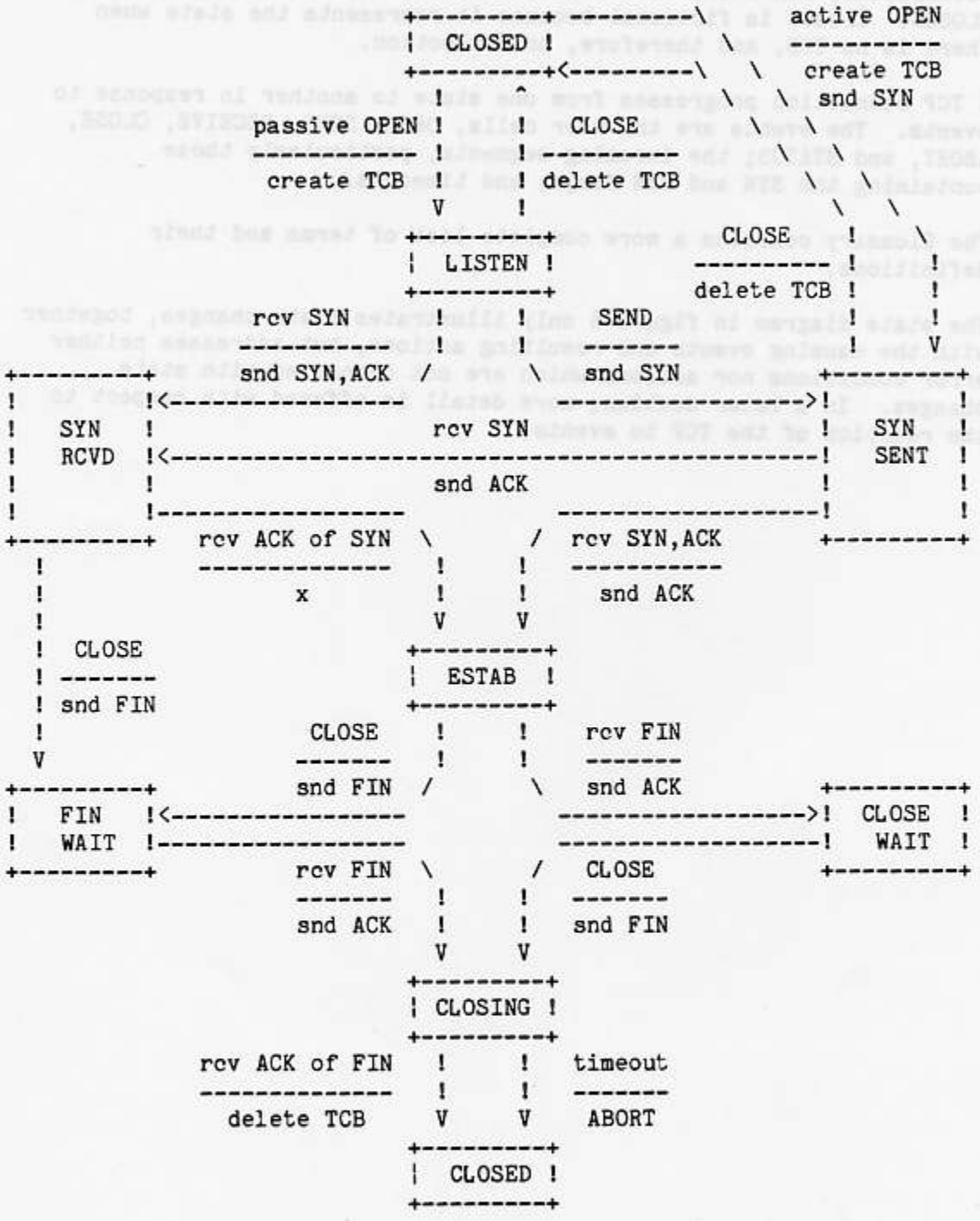
A connection progresses through a series of states during its lifetime. The states are: LISTEN, SYN-SENT, SYN-RECEIVED,

ESTABLISHED, FIN-WAIT, CLOSE-WAIT, CLOSING, and the fictional state CLOSED. Closed is fictional because it represents the state when there is no TCB, and therefore, no connection.

A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN and FIN flags; and timeouts.

The Glossary contains a more complete list of terms and their definitions.

The state diagram in figure 6 only illustrates state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions which are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP to events.



TCP Connection State Diagram
Figure 6.

3.3. Sequence Numbers

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. This mechanism allows for straight-forward duplicate detection in the presence of retransmission.

It is essential to remember that the actual sequence number space is finite, though very large. This space ranges from 0 to 2**32 - 1. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2**32. This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from 2**32 - 1 to 0 again. There are some subtleties to computer modulo arithmetic so great care should be taken in programming these tests. The typical kinds of sequence number comparisons which the TCP must perform include:

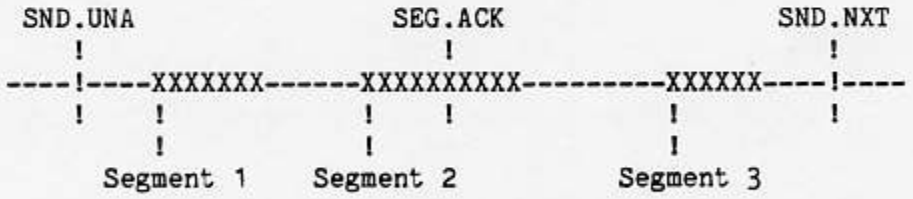
(a) Determining that an acknowledgment refers to some sequence number sent but not yet acknowledged.

(b) Determining that all sequence numbers occupied by a segment have been acknowledged (e.g., to remove the segment from a retransmission queue).

(c) Determining that an incoming segment contains sequence numbers which are expected (i.e., that the segment "overlaps" the receive window).

On send connections the following comparisons are needed:

older sequence numbers newer sequence numbers



Sending Sequence Space Information

Figure 7.

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

SEG.ACK = acknowledgment (next sequence number expected by the acknowledging TCP)

SEG.SEQ(i) = first sequence number of the i-th segment

SEG.SEQ+SEG.LEN-1(i) = last sequence number of the i-th segment

An acceptable acknowledgment, SEG.ACK, is one for which the inequality below holds:

0 < (SEG.ACK - SND.UNA) =< (SND.NXT - SND.UNA)

or:

SND.UNA < SEG.ACK =< SND.NXT

Note that all arithmetic is modulo 2**32 and that comparisons are unsigned. "=<" means "less than or equal".

Similarly, the determination that a particular segment has been fully acknowledged can be made if the inequality below holds:

0 < (SEG.SEQ+SEG.LEN-1(i) - SND.UNA) < (SEG.ACK - SND.UNA)

SEG.LEN(i) is the number of octets occupied by the data in the segment. It is important to note that SEG.LEN(i) must be non-zero; segments which do not occupy any sequence space (e.g., empty acknowledgment segments) are never placed on the retransmission queue, so would not go through this particular test.

On receive connections the following comparisons are needed:

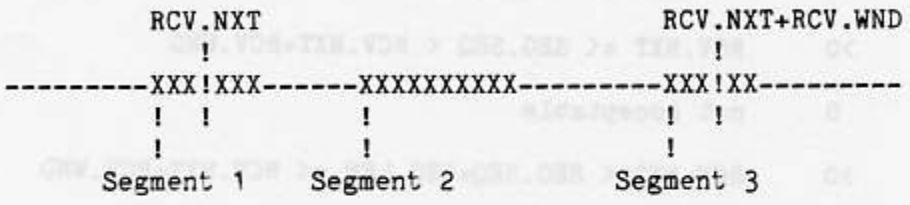


Figure 8.

RCV.NXT = next sequence number expected on incoming segments

RCV.NXT+RCV.WND = last sequence number expected on incoming segments, plus one

SEG.SEQ(i) = first sequence number occupied by the i-th incoming segment

SEG.SEQ+SEG.LEN-1(i) = last sequence number occupied by the i-th incoming segment

A segment is judged to occupy a portion of valid receive sequence space if

0 =< (SEG.SEQ+SEG.LEN-1 - RCV.NXT) < (RCV.NXT+RCV.WND - RCV.NXT)

SEG.SEQ+SEG.LEN-1 is the last sequence number occupied by the segment, RCV.NXT is the next sequence number expected on an incoming segment, and RCV.NXT+RCV.WND is the right edge of the receive window.

Actually, it is a little more complicated than this. Due to zero windows and zero length segments, we have four cases for the acceptability of an incoming segment:

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT < SEG.SEQ+SEG.LEN =< RCV.NXT+RCV.WND

Note that the acceptance test for a segment, since it requires the end of a segment to lie in the window, is somewhat more restrictive than is absolutely necessary. If at least the first sequence number of the segment lies in the receive window, or if some part of the segment lies in the receive window, then the segment might be judged acceptable. Thus, in figure 8, at least segments 1 and 2 are acceptable by the strict rule and segment 3 may or may not be, depending on the strictness of interpretation of the rule.

Note that when RCV.NXT = RCV.NXT+RCV.WND, the receive window is zero and no segments should be acceptable except ACK segments. Thus, it should be possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs.

We have taken advantage of the numbering scheme to protect certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can be retransmitted and acknowledged without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length includes both data and sequence space occupying controls.

Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. A connection is defined by a pair of sockets. New instances of a connection will be referred to as incarnations of the connection. The problem that arises owing to this is -- "how does the TCP identify duplicate segments from previous

incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

To avoid confusion we must prevent segments from being emitted with sequence numbers which duplicate those which are still in the network. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds. Thus, the ISN cycles approximately every 4.55 hours. Since we assume that segments will stay in the network no more than tens of seconds or minutes, at worst, we can reasonably assume that ISN's will be unique.

For each connection there is a send sequence number and a receive sequence number. The initial send sequence number (ISS) is chosen by the data sending TCP, and the initial receive sequence number (IRS) is learned during the connection establishing procedure.

For a connection to be established or initialized, the two TCP's must synchronize on each other's initial sequence numbers. This is done in an exchange of connection establishing messages carrying a control bit called "SYN" (for synchronize) and the initial sequence numbers. As a shorthand, messages carrying the SYN bit are also called "SYNs". Hence, the solution requires a suitable mechanism for picking an initial sequence number and a slightly involved handshake to exchange the ISN's. A "three way handshake" is necessary because sequence numbers are not tied to a global clock in the network, and TCP's may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN.

The "three way handshake" and the advantages of a "clock-driven" scheme are discussed in [3].

Knowing When to Keep Quiet

To be sure that a TCP does not create a segment that carries a sequence number which may be duplicated by an old segment remaining in the network, the TCP must keep quiet for a maximum segment lifetime (MSL) before assigning any sequence numbers upon starting up or recovering from a crash in which memory of sequence numbers in use was lost. For this specification the MSL is taken to be 2 minutes. This value may be changed if experience indicates it is desirable to do so.

Note that if a TCP is reinitialized in some sense, yet retains its memory of sequence numbers in use, then it need not wait at all; it must only be sure to use sequence numbers larger than those recently used.

It should be noted that this strategy does not protect against spoofing or other replay type duplicate message problems.

3.4. Establishing a connection

The "three-way handshake" is essentially a unidirectional attempt to establish a connection; i.e., there is an initiator and a responder. The TCP can also establish a connection when a simultaneous initiation occurs. A simultaneous attempt occurs when one TCP receives a "SYN" segment which carries no acknowledgment after it has sent a "SYN". Of course, the arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases. Several examples of connection initiation are offered below. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the user until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state).

The simplest three-way handshake is shown in figure 9 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

```
    TCP A                                                    TCP B

1.  CLOSED                                                   LISTEN

2.  SYN-SENT    --> <SEQ=100><CTL=SYN>                   --> SYN-RECEIVED

3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>          --> ESTABLISHED

5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Basic 3-Way Handshake for Connection Synchronization

Figure 9.

In line 2 of figure 9, TCP A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP B sends a SYN and acknowledges the SYN it received from TCP A. Note that the acknowledgment field indicates TCP B is now expecting to hear sequence 101, acknowledging the SYN which occupied sequence 100.

At line 4, TCP A responds with an empty segment containing an ACK for TCP B's SYN; and in line 5, TCP A sends some data. Note that the sequence number of the segment in line 5 is the same as in line 4 because the ACK does not occupy sequence number space (if it did, we would wind up ACKing ACK's!).

Simultaneous initiation is only slightly more complex, as is shown in figure 10. Each TCP cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, has been devised. A TCP which receives a reset message first verifies that the ACK field of the reset acknowledges something the TCP sent (otherwise, the message is ignored). If the receiving TCP is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT, CLOSE-WAIT, CLOSING), it aborts the connection and informs its user. We discuss this latter case under "half-open" connections below.

```
    TCP A                                                  TCP B

1.  CLOSED                                                 CLOSED

2.  SYN-SENT     --> <SEQ=100><CTL=SYN>                    ...

3.  SYN-RECEIVED <-- <SEQ=300><CTL=SYN>                    <-- SYN-SENT

4.               ... <SEQ=100><CTL=SYN>                    --> SYN-RECEIVED

5.  SYN-RECEIVED --> <SEQ=101><ACK=301><CTL=ACK>           ...

6.  ESTABLISHED  <-- <SEQ=301><ACK=101><CTL=ACK>           <-- SYN-RECEIVED

7.               ... <SEQ=101><ACK=301><CTL=ACK>           --> ESTABLISHED
```

Simultaneous Connection Synchronization

Figure 10.

```
    TCP A                                                          TCP B

1.  CLOSED                                                         LISTEN

2.  SYN-SENT     --> <SEQ=100><CTL=SYN>                            ...

3.  (duplicate)  ... <SEQ=1000><CTL=SYN>                           --> SYN-RECEIVED

4.  SYN-SENT     <-- <SEQ=300><ACK=1001><CTL=SYN,ACK>              <-- SYN-RECEIVED

5.  SYN-SENT     --> <SEQ=1001><RST><ACK=301>                      --> LISTEN
                                                                       (ACK is ok)

6.               ... <SEQ=100><SYN>                                --> SYN-RECEIVED

7.  SYN-SENT     <-- <SEQ=400><ACK=101><CTL=SYN,ACK>               <-- SYN-RECEIVED

8.  ESTABLISHED  --> <SEQ=101><ACK=401><CTL=ACK>                    --> ESTABLISHED
```

Recovery from Old Duplicate SYN

Figure 11.

As a simple example of recovery from old duplicates, consider

figure 11. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ and ACK fields selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.

Half-Open Connections and Other Anomalies

An established connection is said to be "half-open" if one of the TCP's has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is mildly involved.

If at site A the connection no longer exists, then an attempt by the user at site B to send any data on it will result in the site B TCP receiving a reset control message. Such a message should indicate to the site B TCP that something is wrong, and it is expected to abort the connection.

Assume that two user processes A, and B, are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, A is likely to start again from the beginning or from a recovery point. As a result A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local TCP. In an attempt to establish the connection A's TCP will send a segment containing SYN. This scenario leads to the example shown in figure 12. After TCP A crashes, the user attempts to re-open the connection. TCP B, in the meantime, thinks the connection is open.

```
    TCP A                                                TCP B

1.  (CRASH)                                  (send 300,receive 100)

2.  CLOSED                                               ESTABLISHED

3.  SYN-SENT --> <SEQ=400><CTL=SYN>                  --> (??)

4.  (!!)     <-- <SEQ=300><ACK=100><CTL=ACK>      <-- ESTABLISHED

5.  SYN-SENT --> <SEQ=100><ACK=300><CTL=RST,ACK> --> (Abort!!)

6.                                                       CLOSED

7.  SYN-SENT --> <SEQ=400><SYN><CTL=SYN>             --> CLOSED

8.  (Abort!!)<-- <SEQ=xxx><ACK=401><CTL=RST,ACK> <-- CLOSED

9.  CLOSED                                               CLOSED
```

Half-Open Connection Discovery

Figure 12.

When the SYN arrives at line 3, TCP B, being in a synchronized state, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will continue to retransmit its SYN; and if the user at TCP B re-opens the connection, eventually everything will work out.

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in figure 13. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

```
      TCP A                                                TCP B

1.  (CRASH)                                    (send 300,receive 100)

2.  (??)    <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK> <-- ESTABLISHED

3.          --> <SEQ=100><ACK=310><CTL=RST,ACK>      --> (ABORT!!)
```

Active Side Causes Half-Open Connection Discovery

Figure 13.

In figure 14, we find the two TCP's A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

```
      TCP A                                                TCP B

1.  LISTEN                                                 LISTEN

2.          ... <SEQ=Z><CTL=SYN>                  -->  SYN-RECEIVED

3.  (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>        <--  SYN-RECEIVED

4.          --> <SEQ=Z+1><ACK=X+1><CTL=RST,ACK>  -->  (return to LISTEN!)

5.  LISTEN                                                 LISTEN
```

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 14.

A variety of other cases are possible, all of which are accounted for by the following rules for RST generation and processing.

Reset Generation

As a general rule, reset (RST) should be sent whenever a segment arrives which apparently is not intended for the current or a future incarnation of the connection. A reset should not be sent if it is not clear that this is the case. Thus, if any segment arrives for a nonexistent connection, a reset should be sent. If a segment ACKs

something which has never been sent on the current connection, then one of the following two cases applies.

1. If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED) or if the connection does not exist, a reset (RST) should be formed and sent for any segment that acknowledges something not yet sent. The RST should take its SEQ field from the ACK field of the offending segment (if the ACK control bit was set), and its ACK field should acknowledge all data and control in the offending segment. This is done to make the segment believable to the remote TCP. The sequence field will contain the next sequence the remote TCP expects, and the acknowledgment field will acknowledge everything the remote TCP claims to have sent.

2. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT, CLOSE-WAIT, CLOSING), any unacceptable segment should elicit only an empty acknowledgment segment containing the current send-sequence number and an acknowledgment indicating the next sequence number expected to be received.

Reset Processing

All RST (reset) segments are validated by checking their ACK-fields (and SEQ fields if in a synchronized state). If the RST acknowledges something the receiver sent (but has not yet received acknowledgment for), the RST must be valid. RST segments will have ACK fields which acknowledge any data and control in the offending segment to assure acceptability of the RST. In particular, the RST is valid if SND.UNA=<SEG.ACK=<SND.NXT.

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was in SYN-RECEIVED state and had previously been in the LISTEN state then the receiver returns to the LISTEN state, otherwise the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the user and goes to the CLOSED state.

3.5. Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The notion of closing a full-duplex connection is subject to ambiguous interpretation, of course, since it may not be obvious how to treat the receiving side of the connection. We have chosen to treat CLOSE in a simplex fashion. The user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side

has CLOSED. We assume that the TCP will unilaterally inform a user, even if no RECEIVEs are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP.

There are essentially three cases:

1) The user initiates by telling the TCP to CLOSE the connection

2) The remote TCP initiates by sending a FIN control signal

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT state. RECEIVEs are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN and delete the connection. It should be noted that a TCP receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

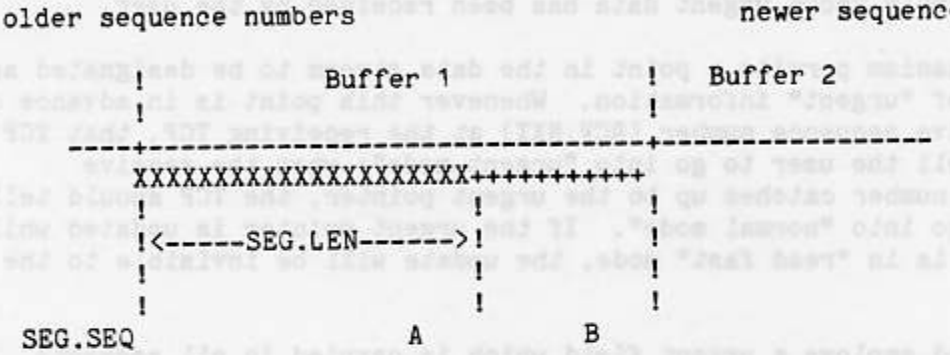
Case 2: TCP receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP can ACK it and tell the user that the connection is closing. The user should respond with a CLOSE, upon which the TCP can send a FIN to the other TCP. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after a timeout the connection is aborted and the user is told.

Case 3: both users close simultaneously

A simultaneous CLOSE by users at both ends of a connection causes FIN segments to be exchanged. When all segments preceding the FINs have been processed and acknowledged, each TCP can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

And, whenever an EOL is received, the receiver advances its receive sequence number, RCV.NXT, by an amount sufficient to consume all the unused space in the receiver's buffer. The amount of space consumed in this fashion is subtracted from the receive window just as is the space consumed by actual data.



XXX - data octets from segment
+++ - phantom data

<----- sequence space ----->

End of Letter Adjustment

Figure 15.

In the case illustrated above, if the segment does not carry an EOL flag the next value of SND.NXT or RCV.NXT will be A. If it does carry an EOL flag the next value will be B.

The exchange of buffer size and sequencing information is done in units of octets. If no buffer size is stated, then the buffer size is assumed to be 1 octet. The receiver tells the sender the size of the buffer in a SYN segment that contains the 16 bit buffer size data in an option field in the TCP header.

Each EOL advances the sequence number (SN) to the next buffer boundary

SN <- SEG.SEQ+SEG.LEN+B-1-((SEG.SEQ+SEG.LEN+B-1-(IS+1)B)

where IS is the initial sequence number, and B is the buffer size.

The CLOSE user call implies an end of letter, as does the FIN control flag in an incoming segment.

The Communication of Urgent Information

The objective of the TCP urgent mechanism is to allow the sending user to stimulate the receiving user to accept some urgent data and to permit the receiving TCP to indicate to the receiving user when all the currently known urgent data has been received by the user.

This mechanism permits a point in the data stream to be designated as the end of "urgent" information. Whenever this point is in advance of the receive sequence number (RCV.NXT) at the receiving TCP, that TCP should tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP should tell user to go into "normal mode". If the urgent pointer is updated while the user is in "read fast" mode, the update will be invisible to the user.

The method employs a urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful and should be added to the segment sequence number to yield the urgent pointer. The absence of this flag indicates that the urgent pointer has not changed.

To send an urgent indication the user must also send at least one data octet. If the sending user also indicates end of letter timely delivery of the urgent information to the destination process is enhanced.

Managing the Window

The window sent in each segment indicates the range of sequence number the sender of the window (the data receiver) is currently prepared to accept. There is an assumption that this is somehow related to to the currently available data buffer space available for this connection.

Indicating a large window encourages transmissions. If more data arrives than can be accepted this will result in excessive retransmissions, adding unnecessarily to the load on the network and the TCPs. Indicating a small window may restrict the transmission of data to the point of introducing a round trip delay between each new segment transmitted.

The mechanisms provided allow a TCP to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged.

The sending TCP must be prepared to accept and send at least one octet of new data even if the send window is zero. This is essential to

guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other.

Users must keep reading connections they close for sending until the TCP says no more data.

In a connection with a one way data flow the window information will be carried in acknowledgment segments that all have the same sequence number so there will be no way to reorder them if they arrive out of order. This is not a serious problem, but it will allow the window information to be on occasion temporarily based on old reports from the data receiver.

3.7. Interfaces

There are of course two interfaces of concern: the user/TCP interface and the TCP/network interface. We have a fairly elaborate model of the user/TCP interface, but only a sketch of the interface to the lower level protocol module.

User/TCP Interface

The functional description of user commands to the TCP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that different TCP implementations may have different user interfaces. However, all TCP's must provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all TCP implementations.

TCP User Commands

The following sections functionally characterize a USER/TCP interface. The notation used is similar to most procedure or function calls in high level languages, but this usage is not meant to rule out trap type service calls (e.g., SVC's, UUO's, EMT's).

The user commands described below specify the basic functions the TCP must perform to support interprocess communication. Individual implementations should define their own exact format, and may provide combinations or subsets of the basic functions in single calls. In particular, some implementations may wish to automatically OPEN a connection on the first SEND or RECEIVE issued by the user for a given connection.

In providing interprocess communication facilities, the TCP must not only accept commands, but must also return information to the processes it serves. The latter consists of:

(a) general information about a connection (e.g., interrupts, remote close, binding of unspecified foreign socket).

(b) replies to specific user commands indicating success or various types of failure.

Although the means for signaling user processes and the exact format of replies will vary from one implementation to another, it would promote common understanding and testing if a common set of codes were adopted. Such a set of event codes is described below.

OPEN

Format: OPEN (local port, foreign socket, active/passive [, buffer size] [, timeout]) -> local connection name

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the local network and TCP identifiers for the source address will either be supplied by the TCP or by the processes that serve it (e.g., the program which interfaces the TCP network). These considerations are the result of concern about security, to the extent that no TCP be able to masquerade as another one, and so on. Similarly, no process can masquerade as another without the collusion of the TCP.

If the active/passive flag is set to passive, then this is a call to LISTEN for an incoming connection. A passive open may have either a fully specified foreign socket to wait for a particular connection or an unspecified foreign socket to wait for any call. A fully specified passive call can be made active by the subsequent execution of a SEND.

A full-duplex transmission control block (TCB) is created and partially filled in with data from the OPEN command parameters. The TCB format is described in more detail in section 5.4.

On an active OPEN command, the TCP will begin the procedure to synchronize (i.e., establish) the connection at once.

The buffer size, if present, indicates that the caller will always receive data from the connection in that size of buffers. This buffer size is a measure of the buffer between the user and

the local TCP. The buffer size between the two TCP's may be different.

The timeout, if present, permits the caller to set up a timeout for all buffers transmitted on the connection. If a buffer is not successfully delivered to the destination within the timeout period, the TCP will abort the connection. The present global default is 30 seconds. The buffer retransmission rate may vary; most likely, it will be related to the measured time for responses from the remote TCP.

Depending on the TCP implementation, either a local connection name will be returned to the user by the TCP, or the user will specify this local connection name (in which case another parameter is needed in the call). The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

Send

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

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the connection has not been opened, the SEND is considered an error. Some implementations may allow users to SEND first; in which case, an automatic OPEN would be done. If the calling process is not authorized to use this connection, an error is returned.

If the EOL flag is set, the data is the End Of a Letter, and the EOL bit will be set in the last TCP segment created from the buffer. If the EOL flag is not set, subsequent SENDs will appear to be part of the same letter.

If the URGENT flag is set, segments resulting from this call will have the urgent pointer set to indicate that all of the data associated with this call is urgent. This facility, for example, can be used to simulate "break" signals from terminals or error or completion codes from I/O devices. The semantics of this signal to the receiving process are unspecified. The receiving TCP will signal the urgent condition to the receiving process as long as the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of urgent is to stimulate the receiver to accept some urgent data and to indicate to the receiver when all the currently known urgent data has been received.

The number of times the sending user's TCP signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

If no foreign socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a foreign segment arriving for the local socket]) then the designated buffer is sent to the implied foreign socket. In general, users who make use of OPEN with an unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

However, if a SEND is attempted before the foreign socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. In some implementations the TCP may notify the user when an unspecified socket is bound.

If a timeout is specified, then the current timeout for this connection is changed to the new one.

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. However, this simple method is both highly subject to deadlocks (for example, both sides of the connection might try to do SENDs before doing any RECEIVEs) and offers poor performance, so it is not recommended. A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

We have implicitly assumed an asynchronous user interface in which a SEND later elicits some kind of SIGNAL or pseudo-interrupt from the serving TCP. An alternative is to return a response immediately. For instance, SENDs might return immediate local acknowledgment, even if the segment sent had not been acknowledged by the distant TCP. We could optimistically assume eventual success. If we are wrong, the connection will close anyway due to the timeout. In implementations of this kind (synchronous), there will still be some asynchronous signals, but these will deal with the connection itself, and not with specific segments or letters.

NOTA BENE: In order for the process to distinguish among error or success indications for different SENDs, it might be appropriate for the buffer address to be returned along with the

coded response to the SEND request. We will offer an example event code format below, showing the information which should be returned to the calling process.

Receive

Format: RECEIVE (local connection name, buffer address, byte count)

This command allocates a receiving buffer associated with the specified connection. If no OPEN precedes this command or the calling process is not authorized to use this connection, an error is returned.

In the simplest implementation, control would not return to the calling program until either the buffer was filled, or some error occurred, but this scheme is highly subject to deadlocks. A more sophisticated implementation would permit several RECEIVEs to be outstanding at once. These would be filled as letters, segments or fragments arrive. This strategy permits increased throughput at the cost of a more elaborate scheme (possibly asynchronous) to notify the calling program that a letter has been received or a buffer filled.

If insufficient buffer space is given to reassemble a complete letter, the EOL flag will not be set in the response to the RECEIVE. The buffer will be filled with as much data as it can hold. The last buffer required to hold the letter is returned with EOL signalled.

The remaining parts of a partly delivered letter will be placed in buffers as they are made available via successive RECEIVEs. If a number of RECEIVEs are outstanding, they may be filled with parts of a single long letter or with at most one letter each. The event codes associated with each RECEIVE will indicate what is contained in the buffer.

If a buffer size was given in the OPEN call, then all buffers presented in RECEIVE calls must be of exactly that size, or an error indication will be returned.

The URGENT flag will be set only if the receiving user has previously been informed via a general event, that urgent data is waiting. The receiving user should thus be in "read-fast" mode. If the URGENT flag is on, additional urgent data remains. If the URGENT flag is off, this call to RECEIVE has returned all the urgent data, and the user may now leave "read-fast" mode.

To distinguish among several outstanding RECEIVEs and to take care of the case that a letter is smaller than the buffer supplied, the event code is accompanied by both a buffer pointer and a byte count indicating the actual length of the letter received.

Alternative implementations of RECEIVE might have the TCP allocate buffer storage, or the TCP might share a ring buffer with the user. Variations of this kind will produce obvious variation in user interface to the TCP.

Close

Format: CLOSE(local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the other side may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not receive any more." It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP gives up.

The user may CLOSE the connection at any time on his own initiative, or in response to various prompts from the TCP (e.g., remote close executed, transmission timeout exceeded, destination inaccessible).

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the CLOSE command will result in error responses.

Close also implies end of letter.

Status

Format: STATUS(local connection name)

This is an implementation dependent user command and could be excluded without adverse effect. Information returned would typically come from the TCB associated with the connection.

This command returns a data block containing the following information:

local socket, foreign socket, local connection name, receive window, send window, connection state, number of buffers awaiting acknowledgment, number of buffers pending receipt (including partial ones), receive buffer size, urgent state, and default transmission timeout.

Depending on the state of the connection, or on the implementation itself, some of this information may not be available or meaningful. If the calling process is not authorized to use this connection, an error is returned. This prevents unauthorized processes from gaining information about a connection.

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be aborted, the TCB to be removed, and a special RESET message to be sent to the TCP on the other side of the connection. Depending on the implementation, users may receive abort indications for each outstanding SEND or RECEIVE, or may simply receive an ABORT-acknowledgment.

TCP-to-User Messages

It is assumed that the operating system environment provides a means for the TCP to asynchronously signal the user program. When the TCP does signal a user program certain information is passed to the user. Often in the specification the information will be an error message. In other cases there will be information relating to the completion of processing a SEND or RECEIVE or other user call.

The following information is provided:

Local Connection Name	Always
Response String	Always
Buffer Address	Send & Receive
Byte count (counts bytes received)	Receive
End-of-Letter flag	Receive
End-of-Urgent flag	Receive

TCP/Network Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. One case is that of the ARPA internetwork system where the lower level module is the Internet Datagram Protocol [1]. In most cases the following simple interface would be adequate.

The following two calls satisfy the requirements for the TCP to internet protocol module communication:

SEND (dest, BufPTR, len)

where:

dest = destination address
BufPTR = buffer pointer
len = length of buffer

Response:

OK = sent ok
Error = error in arguments or local network error

RECV (BufPTR)

Response:

OK = received ok with the additional information:
source address and length
Error = error in arguments or local network error

When the TCP sends a segment, it executes the SEND call supplying all the arguments. The internet protocol module, on receiving this call, checks the arguments and prepares and sends the message. If the arguments are good and the segment is accepted by the local network, the call returns successfully. If either the arguments are bad, or the segment is not accepted by the local network, the call returns unsuccessfully. On unsuccessful returns, a reasonable report should be made as to the cause of the problem, but the details of such reports are up to individual implementations.

When a segment arrives at the internet protocol module from the local network, either there is a pending RECV call from TCP or there is not. In the first case, the pending call is satisfied by passing the information from the segment to the TCP. In the second case, the TCP is notified of a pending segment.

The notification of a TCP may be via a pseudo interrupt or similar mechanism, as appropriate in the particular operating system environment of the implementation.

A TCP's RECV call may then either be immediately satisfied by a pending segment, or the call may be pending until a segment arrives.

We note that the Internet Datagram Protocol provides arguments for a type of service and for a time to live. TCP uses the following settings for these parameters:

type of service = Priority: none, Package: stream, Reliability: higher, Preference: speed, Speed: higher; or 00110110.

time to live = one minute, or 00111100.

3.8. Event Processing

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT
RETRANSMISSION TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera is modulo 2**32 the size of the sequence number space. Also note that "=<" means less than or equal to.

A natural way to think about processing incoming segments is to imagine that they are first tested for proper sequence number (i.e., that their contents lie in the range of the expected "receive window" in the sequence number space) and then that they are generally queued and processed in sequence number order. We are ignoring the problem of segments that overlap other, already received, segments.

OPEN Call

CLOSED STATE (i.e., TCB does not exist)

Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, foreign socket, and user timeout information. If active and the foreign socket is unspecified, return "error: foreign socket unspecified"; if active and the foreign socket is specified, issue a SYN segment. An initial send sequence number (ISS) is selected and a SYN segment of the form <SEQ=ISS><CTL=SYN> is sent. Set SND.UNA to ISS, SND.NXT to ISS+1, enter SYN-SENT state, and return.

If the caller does not have access to the local socket specified, return "error: connection illegal for this process". If there is no room to create a new connection, return "error: insufficient resources".

LISTEN STATE
SYN-SENT STATE
SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT STATE
CLOSE-WAIT STATE
CLOSING STATE

Return "error: connection already exists".

SEND Call

CLOSED STATE (i.e., TCB does not exist)

If the user should no have access to such a connection, then return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS, send a SYN segment, set SND.UNA to ISS and SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command should be sent with the first data segment sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

SYN-SENT STATE

Queue for processing after the connection is ESTABLISHED. Typically, nothing can be sent yet, anyway, because the send window has not yet been set by the other side. If no space, return "error: insufficient resources".

SYN-RECEIVED STATE

Queue for later processing after entering ESTABLISHED state. If no space to queue, respond with "error: insufficient resources".

ESTABLISHED STATE

Segmentize the buffer, send or queue it for output, with a piggy-backed acknowledgment (acknowledgment value = SND.UNA) with the data (this is not required, but there is no advantage in not doing so). If there is insufficient space to remember this buffer, simply return "error: insufficient resources".

If remote buffer size is not one octet; then, if this is the end of a letter, do end-of-letter/buffer-size adjustment processing. Let ISS be the initial send sequence number used on this connection (the SYN sequence number), OSS be the send sequence before sending this segment, NSS the send sequence after sending

this segment, RB be the remote buffer size, and L the number of octets in this segment. Then:

if EOL = 0 then NSS <- OSS + L

if EOL = 1 then NSS <- OSS+L+RB-1-((OSS+L+RB-1-(ISS+1))modRB)

Set SND.NXT <- NSS.

FIN-WAIT STATE

Return "error: connection closing" and do not service request.

CLOSE-WAIT STATE

Segmentize any text to be sent and queue for output. If there is insufficient space to remember the SEND, return "error: insufficient resources"

CLOSING STATE

Respond with "error: connection closing"

RECEIVE Call

CLOSED STATE (i.e., TCB does not exist)

If the user should no have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Queue request if there is space, or respond with "error: insufficient resources".

SYN-SENT STATE

Queue for later processing unless there is no room, in which case return "error: insufficient resources".

SYN-RECEIVED STATE

Queue for processing after entering ESTABLISHED state. If there is no room to queue this request, respond with "error: insufficient resources".

ESTABLISHED STATE

If insufficient incoming segments are queued to satisfy the request, queue the request. If there is no queue space to remember the RECEIVE, respond with "error: insufficient resources".

Reassemble queued incoming segments into receive buffer and return to user. Mark "end of letter" (EOL) if this is the case.

When the TCP takes responsibility for delivering data to the user that fact must be communicated to the sender via an acknowledgment. The formation of such an acknowledgment is described below in the discussion of processing an incoming segment.

FIN-WAIT STATE

Reassemble and return a letter, or as much as will fit, in the user buffer. Queue the request if it cannot be serviced immediately.

CLOSE-WAIT STATE

Since the remote side has already sent FIN, RECEIVEs must be satisfied by text already reassembled, but not yet delivered to the user. If no reassembled segment text is awaiting delivery, the RECEIVE should get a "error: connection closing" response. Otherwise, any remaining text can be used to satisfy the RECEIVE.

CLOSING STATE

Return "error: connection closing"

CLOSE Call

CLOSED STATE (i.e., TCB does not exist)

If the user should no have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVEs should be returned with "error: closing" responses. Delete TCB, return "ok".

SYN-SENT STATE

Delete the TCB and return "error: closing" responses to any queued SENDs, or RECEIVEs.

SYN-RECEIVED STATE

Queue for processing after entering ESTABLISHED state or segmentize and send FIN segment. If the latter, enter FIN-WAIT state.

ESTABLISHED STATE

Queue this until all preceding SENDs have been segmentized, then form a FIN segment and send it. In any case, enter FIN-WAIT state.

FIN-WAIT STATE

Strictly speaking, this is an error and should receive a "error: connection closing" response. An "ok" response would be acceptable, too, as long as a second FIN is not emitted.

CLOSE-WAIT STATE

Queue this request until all preceding SENDs have been segmentized; then send a FIN segment, enter CLOSING state.

CLOSE Call

CLOSING STATE

Respond with "error: connection closing"

ABORT Call

CLOSED STATE (i.e., TCB does not exist)

If the user should no have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVEs should be returned with "error: connection reset" responses. Delete TCB, return "ok".

SYN-SENT STATE

Delete the TCB and return "reset" responses to any queued SENDs, or RECEIVEs.

SYN-RECEIVED STATE

Send a RST of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=RST,ACK>

and return any unprocessed SENDs, or RECEIVEs with "reset" code, delete the TCB.

ESTABLISHED STATE

Send a reset segment:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=RST,ACK>

All queued SENDs and RECEIVEs should be given "reset" responses; all segments queued for transmission (except for the RST formed above) or retransmission should be flushed, delete the TCB.

FIN-WAIT STATE

A reset segment (RST) should be formed and sent:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=RST,ACK>

Outstanding SENDs, RECEIVEs, CLOSEs, and/or segments queued for retransmission, or segmentizing, should be flushed, with "connection reset" notification to the user, delete the TCB.

CLOSE-WAIT STATE

Flush any pending SENDs and RECEIVEs, returning "connection reset" responses for them. Form and send a RST segment:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=RST,ACK>

Flush all segment queues and delete the TCB.

CLOSING STATE

Respond with "ok" and delete the TCB; flush any remaining segment queues. If a CLOSE command is still pending, respond "error: connection reset".

STATUS Call

CLOSED STATE (i.e., TCB does not exist)

If the user should no have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Return "state = LISTEN", and the TCB pointer.

SYN-SENT STATE

Return "state = SYN-SENT", and the TCB pointer.

SYN-RECEIVED STATE

Return "state = SYN-RECEIVED", and the TCB pointer.

ESTABLISHED STATE

Return "state = ESTABLISHED", and the TCB pointer.

FIN-WAIT STATE

Return "state = FIN-WAIT", and the TCB pointer.

CLOSE-WAIT STATE

Return "state = CLOSE-WAIT", and the TCB pointer.

CLOSING STATE

Return "state = CLOSING", and the TCB pointer.

SEGMENT ARRIVES

If the state is CLOSED (i.e., TCB does not exist) then

all data in the incoming segment is discarded. An incoming segment containing a RST is discarded. An incoming segment not containing a RST causes a RST to be sent in response. The acknowledgment and sequence field values are selected to make the reset sequence acceptable to the TCP that sent the offending packet. If the ACK bit is off, sequence number zero is used. Then return

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

If the state is LISTEN then

first check for an ACK

Any acknowledgment is bad if it arrives on a connection still in the LISTEN state. An acceptable reset segment should be formed for any arriving ACK-bearing segment, except another RST. The RST should be formatted as follows:

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

Thus, the RST will acknowledge any text or control in the offending segment. Return.

An incoming RST should be ignored. Return.

if there was no ACK then check for a SYN

If the SYN bit is set, RCV.NXT should be set to SEG.SEQ+1 and any other control or text should be queued for processing later. ISS should be selected and a SYN segment sent of the form:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

SND.NXT should be set to ISS+1 and SND.UNA to ISS. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control or data (combined with SYN) will be processed in the SYN-RECEIVED state, but processing of SYN and ACK should not be repeated.

This segment may also include data and control bits (e.g., URG, EOL) which were queued for transmission.

if there was no SYN but there was other text or control

Any other control or text-bearing segment (not containing SYN) will have an ACK and thus will be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you won't get here, but if you do, drop the segment, and return.

If the state is SYN-SENT then

first check for an ACK

If SND.UNA < SEG.ACK =< SND.NXT then the ACK is acceptable. SND.UNA should be advanced to equal SEG.ACK, and any segments on the retransmission queue which are thereby acknowledged should be removed.

If the segment acknowledgment is not acceptable and the RST bit is off, send an acceptable RST segment of the form:

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

and discard the segment. Return.

if the ACK is ok, check the RST bit

If the RST bit is set then signal the user "error: connection reset", enter CLOSED state, drop the segment, delete TCB, and return.

if the ACK is ok and it was not a RST, check the SYN bit

If the SYN bit is on then, RCV.NXT should be set to SEG.SEQ+1. If SND.UNA > ISS (our SYN has been ACKed), change the connection state to ESTABLISHED, otherwise enter SYN-RECEIVED. In any case, form an ACK segment:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

and send it. Data or controls which were queued for transmission may be included.

If there are other controls or text in the segment then continue processing at the fifth step below where the URG bit is checked, otherwise return.

Otherwise,

first check sequence number

SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT STATE
CLOSE-WAIT STATE
CLOSING STATE

Segments are processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in SEG.SEQ order. If a segment's contents straddle the boundary between old and new, only the new parts should be processed.

There are four cases for the acceptability test for an incoming segment:

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT < SEG.SEQ+SEG.LEN =< RCV.NXT+RCV.WND

Note that the test above guarantees that the last sequence number used by the segment lies in the receive-window. If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs.

If an incoming segment is not acceptable, an acknowledgment should be sent in reply:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

If the incoming segment is unacceptable, drop it and return.

second check the ACK field,

SYN-RECEIVED STATE

If the RST bit is off and SND.UNA < SEG.ACK =< SND.NXT then set SND.UNA <- SEG.ACK, remove any acknowledged segments from the retransmission queue, and enter ESTABLISHED state.

If the segment acknowledgment is not acceptable, form a reset segment, as for the bad sequence case above, and send it, unless the incoming segment is an RST, in which case, it should be discarded, then return.

ESTABLISHED STATE

If SND.UNA < SEG.ACK =< SND.NXT then set SND.UNA <- SEG.ACK. Any segments on the retransmission queue which are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers which have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate, it can be ignored.

If the segment passes the sequence number and acknowledgment number tests the send window should be updated. If SND.WL =< SEG.SEQ set SND.WND <- SEG.WND and set SND.WL <- SEG.SEQ.

If the remote buffer size is not one, then the end-of-letter/buffer-size adjustment to sequence numbers may have an effect on the next expected sequence number to be acknowledged. It is possible that the remote TCP will acknowledge with a SEG.ACK equal to a sequence number of an octet that was skipped over at the end of a letter. This a mild error on the remote TCPs part, but not cause for alarm.

FIN-WAIT STATE

In addition to the processing for the ESTABLISHED state, if the retransmission queue is empty, the user's CLOSE can be acknowledged ("ok") but do not delete the TCB.

CLOSE-WAIT STATE
CLOSING STATE

. Do the same processing as for the ESTABLISHED state.

third, check the RST bit,

SYN-RECEIVED STATE

If the segment has passed sequence and acknowledgment tests, it is valid. If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then return this connection to LISTEN state. The user need not be informed. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state) then the connection was refused, signal the user "connection refused". In either case, all segments on the retransmission queue should be removed.

ESTABLISHED
FIN-WAIT
CLOSE-WAIT
CLOSING STATE

Any outstanding RECEIVEs and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

fourth, check the SYN bit, Grab=10;

SYN-RECEIVED
ESTABLISHED STATE

The segment sequence number must be in the receive window; if not, ignore the segment. If the SYN is on and the segment sequence and the receive sequence are equal, then everything is ok and no action is needed; but if they are not equal, there is an error and a reset must be sent.

If a reset must be sent it is formed as follows:

<SEQ SEG.ACK> <RST> <ACK SEG.SEQ+SEG.LEN>

The connection must be aborted as if a RST had been received.

FIN-WAIT STATE
CLOSE-WAIT STATE

This case should not occur, since a duplicate of the SYN which started the current connection incarnation will have been filtered in the SEG.SEQ processing. Other SYN's will have been rejected by this test as well (see SYN processing for ESTABLISHED state).

fifth, check the URG bit,

ESTABLISHED STATE
FIN-WAIT STATE

Signal the user that the remote side has urgent data if the urgent pointer is in advance of the data consumed. If the user has already been signalled (or is still in the "urgent mode") for this continuous sequence of urgent data, do not signal the user again.

CLOSE-WAIT STATE
CLOSING

This should not occur, since a FIN has been received from the remote side. Ignore the URG.

sixth, process the segment text,

ESTABLISHED STATE

Once in the ESTABLISHED state, it is possible to deliver segment text to user RECEIVE buffers. Text from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries an EOL flag, then the user is informed, when the buffer is returned, that an EOL has been received.

If buffer size is not one octet, then do end-of-letter/buffer-size adjustment processing. Let IRS be the initial receive sequence number used on this connection (then SYN sequence number), ORS be the receive sequence before receiving this segment, NRS the receive sequence after receiving this segment, LB be the local buffer size, and L the number of octets in this segment. Then:

if EOL = 0 then NRS <- ORS + L

SEGMENT ARRIVES

if EOL = 0 then NRS <- ORS+L+LB-1-((ORS+L+LB-1-(IRS+1))modLB)

Set RCV.NXT <- NRS.

When the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data. Send an acknowledgment of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

This acknowledgment should be piggybacked on a segment being transmitted if possible without incurring undue delay.

FIN-WAIT STATE

If there are outstanding RECEIVEs, they should be satisfied, if possible, with the text of this segment; remaining text should be queued for further processing. If a RECEIVE is satisfied, the user should be notified, with "end-of-letter" (EOL) signal, if appropriate.

CLOSE-WAIT STATE

This should not occur, since a FIN has been received from the remote side. Ignore the segment text.

seventh, check the FIN bit,

Send an acknowledgment for the FIN. Signal the user "connection closing", and return any pending RECEIVEs with same message. Note that FIN implies EOL for any segment text not yet delivered to the user. If the current state is ESTABLISHED, enter the CLOSE-WAIT state. If the current state is FIN-WAIT, enter the CLOSING state.

and return.

USER TIMEOUT

For any state if the user timeout expires, flush all queues, signal the user "error: connection aborted due to user timeout" in general and for any outstanding calls, delete the TCB, and return.

RETRANSMISSION TIMEOUT

For any state if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again, reinitialize the retransmission timer, and return.

GLOSSARY

1822
: BBN Report 1822, "The Specification of the Interconnection of a Host and an IMP". The specification of interface between a host and the ARPANET.

ACK
: A control bit (acknowledge) occupying no sequence space, which indicates that the acknowledgment field of this segment specifies the next sequence number the sender of this segment is expecting to receive, hence acknowledging receipt of all previous sequence numbers.

ARPANET message
: The unit of transmission between a host and an IMP in the ARPANET. The maximum size is about 1012 octets (8096 bits).

ARPANET packet
: A unit of transmission used internally in the ARPANET between IMPs. The maximum size is about 126 octets (1008 bits).

buffer size
: An option (buffer size) used to state the receive data buffer size of the sender of this option. May only be sent in a segment that also carries a SYN.

connection
: A logical communication path identified by a pair of sockets.

datagram
: A message sent in a packet switched computer communications network.

Destination Address
: The destination address, usually the network and host identifiers.

EOL
: A control bit (End of Letter) occupying no sequence space, indicating that this segment ends a logical letter with the last data octet in the segment. If this end of letter causes a less than full buffer to be released to the user and the connection buffer size is not one octet then the end-of-letter/buffer-size adjustment to the receive sequence number must be made.

FIN
: A control bit (finis) occupying one sequence number, which indicates that the sender will send no more data or control occupying sequence space.

fragment
: A portion of a logical unit of data, in particular an internet fragment is a portion of an internet datagram.

FTP
: A file transfer protocol.

header
: Control information at the beginning of a message, segment, fragment, packet or block of data.

host
: A computer. In particular a source or destination of messages from the point of view of the communication network.

Identification
: An Internet Datagram Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.

IMP
: The Interface Message Processor, the packet switch of the ARPANET.

internet address
: A source or destination address specific to the host level.

internet datagram
: The unit of data exchanged between an internet module and the higher level protocol together with the internet header.

internet fragment
: A portion of the data of an internet datagram with an internet header.

IRS
: The Initial Receive Sequence number. The first sequence number used by the sender on a connection.

ISN
: The Initial Sequence Number. The first sequence number used on a connection. Selected on a clock based procedure.

ISS

The Initial Send Sequence number. The first sequence number used by the sender on a connection.

leader

Control information at the beginning of a message or block of data. In particular, in the ARPANET, the control information on an ARPANET message at the host-IMP interface.

left sequence

This is the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged sequence number) and is sometimes referred to as the left edge of the send window.

letter

A logical unit of data, in particular the logical unit of data transmitted between processes via TCP.

local packet

The unit of transmission within a local network.

module

An implementation, usually in software, of a protocol or other procedure.

MSL

Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system.

octet

An eight bit byte.

Options

An Option field may contain several options, and each option may be several octets in length. The options are used primarily in testing situations; for example, to carry timestamps. Both the Internetwork Protocol and TCP provide for options fields.

packet

A package of data with a header which may or may not be logically complete. More often a physical packaging than a logical packaging of data.

port
: The portion of a socket that specifies which logical input or output channel of a process is associated with the data.

process
: A program in execution. A source or destination of data from the point of view of the TCP or other host-to-host protocol.

PS
: A Packet Switch. For example, an IMP.

PSN
: A Packet Switched Network. For example, the ARPANET.

RCV.BS
: receive buffer size, the remote buffer size

RCV.NXT
: receive next sequence number

RCV.UP
: receive urgent pointer

RCV.WND
: receive window

receive next sequence number
: This is the next sequence number the local TCP is expecting to receive.

receive window
: This represents the sequence numbers the local (receiving) TCP is willing to receive. Thus, the local TCP considers that segments overlapping the range RCV.NXT to RCV.NXT + RCV.WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.

RST
: A control bit (reset), occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can determine, based on the sequence number and acknowledgment fields of the incoming segment, whether it should honor the reset command or ignore it. In no case does receipt of a segment containing RST give rise to a RST in response.

RTP
: Real Time Protocol: A host-to-host protocol for communication of time critical information.

Rubber EOL
: An end of letter (EOL) requiring a sequence number adjustment to align the beginning of the next letter on a buffer boundary.

SEG.ACK
: segment acknowledgment

SEG.LEN
: segment length

SEG.SEQ
: segment sequence

SEG.UP
: segment urgent pointer field

SEG.WND
: segment window field

segment
: A logical unit of data, in particular a TCP segment is the unit of data transfered between a pair of TCP modules.

segment acknowledgment
: The sequence number in the acknowledgment field of the arriving segment.

segment length
: The amount of sequence number space occupied by a segment, including any controls which occupy sequence space.

segment sequence
: The number in the sequence field of the arriving segment.

send sequence
: This is the next sequence number the local (sending) TCP will use on the connection. It is initially selected from an initial sequence number curve (ISN) and is incremented for each octet of data or sequenced control transmitted.

send window
: This represents the sequence numbers which the remote (receiving) TCP is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP. The range of sequence numbers which may be emitted by a TCP lies between SND.NXT and SND.UNA + SND.WND - 1.

SND.BS
: send buffer size, the local buffer size

SND.NXT
: send sequence

SND.UNA
: left sequence

SND.UP
: send urgent pointer

SND.WND
: send window

socket
: An address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Address
: The source address, usually the network and host identifiers.

SYN
: A control bit in the incoming segment, occupying one sequence number, used at the initiation of a connection, to indicate where the sequence numbering will start.

TCP
: Transmission Control Protocol: A host-to-host protocol for reliable communication in internetwork environments.

TOS
: Type of Service, an Internet Datagram Protocol field.

Type of Service
: An Internet Datagram Protocol field which indicates the type of service for this internet fragment.

URG

A control bit (urgent), occupying no sequence space, used to indicate that the receiving user should be notified to do urgent processing as long as there is data to be consumed with sequence numbers less than the value indicated in the urgent pointer.

urgent pointer

A control field meaningful only when the URG bit is on. This field communicates the value of the urgent pointer which indicates the data octet associated with the sending user's urgent call.

XNET

A cross-net debugging protocol.

REFERENCES

[1]

Postel,J. (ed.), "Internetwork Datagram Protocol Specification - Version 4," Defense Advanced Research Projects Agency, Information Processing Techniques Office, IEN 80, February 1979.

[2]

Feinler, E. and J. Postel, ARPANET Protocol Handbook, Network Information Center, Stanford Research Institute, Menlo Park, CA, January 1978.

[3]

Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols," Computer Networks, Vol. 2, No. 6. December 1978, pp. 454-473.