SPECIFICATION OF INTERNETWORK TRANSMISSION CONTROL PROGRAM

TCP

Version 3

Vinton G. Cerf
Advanced Research Projects Agency

Jonathan B. Postel
Information Sciences Institute

January 1978

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

(213) 822-1511

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

TABLE OF CONTENTS

PREFACE

This document describes the functions to be performed by the Internetwork Transmission Control Program (TCP) and its interface to programs or users that require its services. There have been three previous TCP specifications: The first [CDS74] defined version 1 of TCP. A second [PGR76a] was written for the Defense Communications Agency in connection with its AUTODIN II project. The third [Cerf77] defined version 2, for use in the ARPA internetwork research projects.

The AUTODIN II version differed from the original version in the following ways:

> Specification of a resynchronization mechanism was included, and fields for security and priority, which were known requirements of AUTODIN II, were added.

The Internet version 2 differed from the original version in the following ways:

> A different resynchronization procedure was introduced; an “option” field was defined for the TCP header to accommodate not only security and priority but other special features concerned with, for example, packet speech services, diagnostic timestamping, and so on.
>
> This version eliminated all error messages but for RESET and thus simplified the header format. There are still many local errors which can be reported to the user, but none of these need cross the network(s) between TCP’s.
>
> Connection closing was slightly more elaborate in Version 2 than in version 1 because the FIN signals had to be acknowledged. Furthermore, the INT and FIN facilities no longer caused flushing of the data stream. (A separate “flush” facility was tested, but eliminated, in the end.) Dealing with flow-control windows that have gone to zero was a new feature of version 2, and, finally, the reassembly of fragments into segments was more carefully specified.

In version 3 specified in this document, TCP has further evolved. The primary changes from version 2 are:

> The resynchronization mechanism has been eliminated in favor of a quiet period on initialization of the TCP.

Buffer management and letters are more tightly coupled by the coupling of the end of letter flag to a receive buffer size.

The interrupt signal has been eliminated in favor of an urgent pointer.

A further separation of the internet and TCP specific information in the packet format has been achieved, with provision for variable length addresses in the internet header.

The evolution from TCP version 2 to version 3 was influenced by many people, but special mention should be made of the work at MIT's Laboratory for Computer Science on the Data Stream Protocol (DSP) by Dave Clark and Dave Reed. Many of the specific changes introduced in version 3 were first described by Ray Tomlinson of BBN [Tomlinson77].

Although the list of participants in the TCP work is very long (see [CEHKKS77]—the final TCP project report), special acknowledgements are due to R. Kahn, R. Tomlinson, Y. Dalal, R. Karp and C. Sunshine for their active participation in the design of TCP.

This edition of the specification benefited from the comments of the following reviewers: Michael Padlipsky, Carl Sunshine, John Day, Gary Grossman, and Ray Tomlinson.

Transmission Control Protocol

Version 3

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 section introduces some of the terminology used in the remainder of the document and some of the assumptions made in the design of the protocol.

Several basic assumptions are made about process to process communication and these are listed here without further justification. The interested reader is referred to [CK74, Tomlinson74, Belsnes74, Dalal74, Dalal75, Sunshine76, CEHKKS77] for further discussion. HOSTs are computers attached to a network, and from the communication network's point of view are the sources and destinations of messages. PROCESSES are viewed as the active elements of all host computers in a network (in accordance with the fairly common definition of a process as a program in execution). Even terminals and files or other I/O media are viewed as communicating through the use of processes. Thus, all network 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.

Since port names are selected independently by each operating system, TCP, or user, they may not be unique. To provide for unique names at each TCP, we concatenate an internet ADDRESS specific to the TCP level with a port name to create a SOCKET name which will be unique throughout all networks connected together.

For example:
Network = ARPANET (number 12),
Host = ISI-TENEXA (imp 22, host 1),
Port = FTP-Server (port 3);
or

00001010-00010110-0000000000000000-1
00000000000000000000000000000011.

A pair of sockets form a CONNECTION which can be used to carry data in either direction (i.e. "full duplex"). The connection is uniquely identified by the <local socket, foreign socket> address pair, and the same local socket name can participate in multiple connections to different foreign sockets (see section 2.2).

Processes exchange finite length LETTERS as a way of communicating; thus, letter boundaries might be significant in some process-to-process communications. However, 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 that a segment never contains parts of more than one letter.

Furthermore, there is no restriction on the length of a letter. A connection might be formed to send a single long letter (a stream of bytes, in effect). In fact, processes can communicate via TCP without ever marking the end of a letter, but we think this is atypical of most anticipated use.

There is, however, a coupling between letters as transmitted and the use of buffers of data that cross the TCP/user interface. Each time an end of letter (EOL) signal 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.

We specifically assume that segments are transmitted from Host to Host through means of a PACKET SWITCHING NETWORK (PSN) [RW70, Pouzin73]. 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 [HKOCW70, Pouzin74, SW71].

Processes make use of the TCP by handing it letters (or buffers filled with parts of a letter). The TCP breaks these into segments, if necessary, and then embeds each segment in an INTERNETWORK PACKET. Each internetwork packet is in turn embedded in a LOCAL PACKET suitable for transmission from the host to one of its serving PSs. The packet switches may perform further formatting, fragmentation, or other operations to achieve the delivery of the local packet to the destination Host.

The term LOCAL PACKET is used generically here to mean the formatted bit string exchanged between a host and a packet switch. The format of bit strings exchanged between the packet switches in a PSN will generally not be of concern to us. If an internetwork packet is destined for a TCP in a foreign PSN, the packet is routed to a gateway which connects the originating PSN with an intermediate PSN or with the destination PSN. Routing of internetwork packets to the gateway may be the responsibility of the source TCP or the local PSN, depending upon the PSN services available.

One model of TCP operation is to imagine that there is a basic gateway associated with each TCP which provides an interface to the local network. This basic gateway performs routing and packet reformatting or embedding, and may also implement congestion and error control between the TCP and gateways at or intermediate to the destination TCP.

At a gateway between networks, the internetwork packet is “unwrapped” from its local packet format and examined to determine through which network the internetwork packet should travel next. The internetwork packet is then “wrapped” in a local packet format suitable to the next network and passed on to a new packet switch.

A gateway is permitted to break up a segment carried by an internetwork packet into smaller FRAGMENTS if this is necessary for transmission through the next network. To do this, the gateway produces a set of internetwork packets, each carrying a fragment. Fragments may be broken into smaller ones at intermediate gateways. The packet format is designed so that the destination TCP can reassemble fragments into segments and verify the end-to-end checksum associated with the segment. Segments, of course, can be reassembled into letters.

> Note that the design of fragmentation procedures is still an active area and this function may in the future be removed from TCP’s concerns and become entirely a gateway-to-gateway issue.

The TCP is responsible for regulating the flow of internetwork packets to and from the processes it serves, as a way of preventing its host from becoming saturated or overloaded with traffic. The TCP is also responsible for retransmitting unacknowledged packets, and for detecting duplicates. A consequence of this error detection/retransmission scheme is that the order of letters received on a given connection can also be maintained [CK74, Sunshine75]. To perform these functions, the TCP opens and closes connections between ports as described in section 4.2.

2. THE TCP INTERFACE TO THE USER

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 various TCP implementations may have different user interfaces. These will all be TCP's, as long as control messages are properly interpreted or emitted, as required. In spite of this caveat, it appears useful to have at least one concrete view of a user interface to aid in thinking about TCP-derived services.

2.1 The TCP as a Post Office

The TCP acts in many ways like a postal service since it provides a way for processes to exchange letters with each other. It sometimes happens that a process may offer some service, but not know in advance what its correspondents' addresses are. The analogy can be drawn with a mail order house which opens a post office box which can accept mail from any source. Unlike the post box, however, once a letter from a particular correspondent arrives, the resulting connection becomes specific to the correspondents until the correspondents declare otherwise—thus making the TCP more like a telephone service. Without this particularization, the TCP could not perform its flow control, sequencing, duplicate detection, end-to-end acknowledgement, and error control services.

2.2 Sockets and Addressing

We have borrowed the term SOCKET from the ARPANET terminology [CCC70, DCA76]. In general, a socket is the concatenation of an internetwork ADDRESS and a PORT identifier. A CONNECTION is fully specified by the pair of SOCKETS at each end since the same local socket name may participate in many connections to different foreign sockets.

Once the connection is specified in the OPEN command (see section 2.4.2), the TCP supplies a (short) local connection name by which the user refers to the connection in subsequent commands. As will be seen, this facilitates using connections with initially unspecified foreign sockets.

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" sockets, and that processes can only initiate connections on the sockets 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.)

Once initiated, a connection may be passed to another process that does not own the local socket (e.g. from “logger” to service process). Strictly speaking this is a reconnection issue which might be more elegantly handled by a general reconnection protocol as discussed in section 3.3. To simplify passing a connection within a single TCP, however, such “invisible” switches may be allowed, as in TENEX systems.

Of course, each connection is associated with exactly one process, and any attempt to reference that connection by another process should be treated as an error by the TCP. This prevents another process from stealing data from or inserting data into another process’ data stream, and also prevents masquerading, spoofing, or other forms of malicious mischief (given a correct implementation of TCP in a protective operating system environment).

A connection is “initiated” by the rendezvous of an arriving internetwork packet and a waiting Transmission Control Block (TCB) created by a user OPEN, SEND, URGENT, or RECEIVE command (see section 2.4). The matching of local and foreign socket identifiers determines when a successful connection has been initiated. The connection becomes “established” when sequence numbers have been synchronized in both directions as described in section 4.2.2.

It is possible to specify a socket only partially by setting the PORT identifier to zero or setting both the TCP and PORT identifiers to zero. A socket of all zero is called UNSPECIFIED. The purpose behind unspecified sockets is to provide a sort of “general delivery” facility (useful for processes offering services on “well known” sockets).

There are bounds on the degree of unspecificity of socket identifiers. TCB’s must have fully specified local sockets, although the foreign socket may be fully or partly unspecified. Arriving packets must have fully specified sockets.

We employ the following notation:

x.y.z = fully specified socket with x=net, y=TCP, z=port
x.y.u = as above, but unspecified port
x.u.u = as above, but unspecified TCP and port
u.u.u = completely unspecified
with respect to implementation, u = 0 [zero]

We illustrate the principles of matching by giving all cases of incoming packets which match with existing TCB's. Generally, both the local socket field of the TCB and the destination socket field of the arriving packet must match, and the foreign field of the TCB and the source socket field of the arriving packet must match.

	TCB-local	TCB-foreign	Packet-source	Packet-destination
(a)	a.b.c	e.f.g	e.f.g	a.b.c
(b)	a.b.c	e.f.u	e.f.g	a.b.c
(c)	a.b.c	e.u.u	e.f.g	a.b.c
(d)	a.b.c	u.u.u	e.f.g	a.b.c

There are no other legal combinations of socket identifiers which match. Case (d) is typical of the ARPANET well known socket idea in which the well known socket (a.b.c) LISTENS for a connection from any (u.u.u) socket. Cases (b) and (c) can be used to restrict matching to a particular TCP or net. More elaborate masking facilities could be implemented without adverse effects, so this matching facility could be considered the minimum acceptable for TCP operation.

2.3 What is a Letter?

A letter is a sequence of one or more successive octets (8-bit bytes) on a TCP connection. The beginning of a letter is marked by a BOL control flag in a packet. The end of a letter is marked by the appearance of an EOL control flag in a packet. A letter is the minimum unit of information which must be passed from a receiving TCP to a receiving process. A TCP may pass less information to the receiving program, or it may pass more, but when a TCP has a complete letter it must not wait for more data from the remote process before passing the letter to the receiving process if the receiving process is ready to accept it.

Generally, the locations of letter boundaries are not passed to the receiving program. The exception is for non-reliable transmission (see section 4.2.8). In this case, when a section of data is missing, the data which follows must either begin on a letter boundary or contain an indication that the data does not begin on a letter boundary.

The sequence number of the first octet of data in any letters on a given connection is always equal to zero, modulo the receive buffer size. That is, whenever an EOL is transmitted, the sender advances his send sequence number by an amount (in the range 0 to buffersize–l) sufficient to consume all the unused space in the receiver's buffer. The amount of space consumed in this fashion is accounted for in the flow control mechanism in the same way as space is consumed by real data (see section 4.5). The size of the receive buffer is communicated between the TCPs in the connection establishing exchange.

The EOL interpretation permits the receiving TCP to discard letter boundary information. Higher level protocols are required to provide their own mechanism for parsing the data stream and cannot depend on the EOL mechanism. EOL also has the property that it consumes all the unused space in a buffer (as specified in the buffer size option).

2.4 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 will 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, RECEIVE, or URGENT 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 signalling 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 in section 2.5.

2.4.1 Open

Format: OPEN (local port, foreign socket [, buffer size] [, timeout])

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 source network and TCP identifiers will either be supplied by the TCP or by the processes that serve it [e.g. the program which interfaces the TCP to its packet switch or the packet switch itself]. 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 no foreign socket is specified (i.e. the foreign socket parameter is 0), then this constitutes a LISTENING local socket which can accept communication from any foreign socket. Provision is also made for partial specification of foreign sockets as described in section 2.2.

If the specified connection is already OPEN, an error is returned, otherwise 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 4.3.2.

No network traffic need be generated by the OPEN command. The first SEND or URGENT by the local user or the foreign user will typically cause the TCP to synchronize (i.e. establish) the connection, although synchronization could be immediately initiated on non-listening opens.

The buffer size, if present, indicates that the caller will always receive data from the connection in that size of buffers.

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, and is the responsibility of the TCP and not the user. 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.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

2.4.2 Send

Format: SEND(local connection name, buffer address, byte count, EOL 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 internetwork packet created from the buffer (see section 4.3.2—TCP packet format). If the EOL flag is not set, subsequent SENDs will appear to be part of the same letter.

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 packet 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.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

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 packet 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 packets 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 in section 2.5, showing the information which should be returned to the calling process.

2.4.3 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 (see section 2.4.2). 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 (see section 2.5.2).

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.

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.

Responses from the TCP which may occur as a result of this command are detailed in sections 2.5 and 4.2.9.

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.

2.4.4 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.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

2.4.5 Urgent

Format: URGENT(local connection name)

Special control information is sent to the destination indicating that urgent processing is appropriate. This facility can be used to simulate “break” signals from terminals or error or completion codes from I/O devices, for example. 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 data preceding the urgent pointer has not been consumed by the receiving process.

If the connection is not open or the calling process is not authorized to use this connection, an error is returned.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

2.4.6 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 (see section 4.3.3) 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 acknowledgement, number of buffers pending receipt (including partial ones), receive buffer size, urgent state, and default transmission timeout.

Depending on the state of the connection, on or 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.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

2.4.7 Abort

Format: ABORT (local connection name)

This commend causes all pending SENDs, URGENTs, 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, RECEIVE, or URGENT, or may simply receive an ABORT-acknowledgment. The mechanism of resetting a connection is discussed in sections 4.2.3 and 4.2.9.

Responses from the TCP which may occur as a result of this call are detailed in sections 2.5 and 4.2.9.

2.5 TCP-to-User Messages

2.5.1 Type Codes

All messages include a type code which identifies the type of user call to which the message applies. Types are:

0 — General message, spontaneously sent to user
1 — Applies to OPEN
2 — Applies to CLOSE
3 — Applies to URGENT
4 — Applies to ABORT
10 — Applies to SEND
20 — Applies to RECEIVE
30 — Applies to STATUS

2.5.2 Message Formats

All messages include the following three fields:

Type code
Local connection name
Event code

For message types 0–4 (General, Open, Close, Urgent, Abort) only these three fields are necessary.

For message type 10 (Send) one additional field is necessary:

Buffer address

For message type 20 (Receive) three additional fields are necessary:

Buffer address
Byte count (counts bytes received)
End-of-Letter flag

For message type 30 (Status) additional data might include:

Local socket, foreign socket
Send window (measures buffer space at foreign TCP)
Receive window (measures buffer space at local TCP)
Connection state (see section 4.2.9)
Number of buffers awaiting acknowledgement
Number of buffers awaiting receipt
Receive buffer size
Urgent State (urgent or not urgent)
User timeout

Once more, it is important to note that these formats are notional. Implementations which deal with buffering in different ways may or may not need to include buffer addresses in some responses, for example.

2.5.3 Event Codes

The event code specifies the particular event that the TCP wishes to communicate to the user. Generally speaking, non-zero event codes indicate important state changes or errors.

In addition to the event code, two flags may be useful to classify the event into major categories and facilitate event processing by the user:

E flag: set if event is an error

P flag: set if permanent error (otherwise, retry may succeed).

Events are encoded in 8 bits, the two high order bits being reserved for E and P flags, respectively.

Events specified so far are listed below with their codes and flag settings.

flags	code	meaning
	0	general success
E,P	1	connection illegal for this process
	2	unspecified foreign socket has become bound
E,P	3	connection not OPEN
	4	insufficient resources
E	5	foreign socket not specified
E,P	6	connection already OPEN
E	7	buffer size not acceptable
	8	unused
E,P	9	user timeout, connection aborted
	10	unused
	11	user urgent indication received
P	12	connection closing
E	13	general error
P	14	connection reset

Possible responses to each of the user commands are listed below. Section 4.2.9 offers substantially more detail.

Type 0	[general]:			2,						9,	11,	12,		14
Type 1	[open]:	0,	1,			4,		6,					13	
Type 2	[close]:	0,	1,		3,					9,			13,	14
Type 3	[urgent]:	0,	1,		3,	4,	5,			9,		12,	13,	14
Type 4	[Abort]:	0,	1,		3,								13	
Type 10	[send]:	0,	1,		3,	4,	5,			9,		12,	13,	14
Type 20	[receive]:	0,	1,		3,	4,			7,	9,		12,	13,	14
Type 30	[status]:	0,	1,		3,								13	

3. HIGHER LEVEL PROTOCOLS

3.1 Introduction

It is expected that the TCP will be able to support higher level protocols efficiently. It should be easy to interface existing ARPANET protocols like TELNET [DCA76] and FTP [DCA76] to the TCP. Support of Network Voice Protocol, and broadcast protocols, for example, has been left to version 4 TCP.

3.2 Well Known Sockets

Well known sockets are a convenient mechanism for a priori associating a socket name 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, reflector, and sink (the three being for test purposes). A socket name 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.

For compatibility with ARPANET socket naming conventions, we refer to the list of assigned sockets in RFC 739 [Postel77].

TCP implementors should note, however, that the gender and directionality of NCP sockets do not apply to TCP sockets, so that even numbered as well as odd ones can serve as well known sockets.

3.3 Reconnection Protocol

Port identifiers fall into two categories: permanent and transient. For example, a Telnet-server process is generally assigned a port identifier that is fixed and well known. Transient processes will in general have port identifiers which are dynamically assigned.

In a distributed processing environment, two processes that don’t have well known port identifiers may often wish to communicate. This can be achieved with the help of a well known process using a reconnection protocol. Such a protocol is briefly outlined using the communication facilities provided by the TCP. It essentially provides a mechanism by which port identifiers are exchanged in order to establish a connection between a pair of sockets.

Such a protocol can be used to achieve the dynamic establishment of new connections in order to have multiple processes solving a problem co-operatively, or to provide a user process access to a server-application process via a server-exec process, when the server-exec’s end of the connection can not be invisibly passed to the server-application process.

A paper on this subject by R. Schantz [Schantz74] discusses some of the issues associated with reconnection, and some of the ideas contained therein went into the design of the protocol outlined below.

In the ARPANET, a protocol (called the Initial Connection Protocol [Postel72]) was implemented which would allow a process to connect to a well known socket, thus making an implicit request for service, and then be switched to another socket so that the well known socket could be freed for use by others. Since sockets in our TCP are permitted to participate in more than one connection name, this facility may not be explicitly needed (i.e. connections <A,B> and <A,C> are distinguishable).

However, the well known socket may be in one network and the actual service socket(s) may be in another network (or at least in another TCP). Thus, the invisible switching of a connection from one port to another within a TCP may not be sufficient as an “Initial Connection Protocol”. Let Nx be a network identifier, and Tx be a TCP identifier. We imagine that a process wishes to use socket N1.T1.Q to access well known socket N2.T2.P. However, the process associated with socket N2.T2.P will actually start up a new process somewhere which will use N3.T3.S as its server socket. The N(i) and T(i) may be distinct or the same. The user will send to N2.T2.P the relevant user information such as user name, password, and account. This intermediate server will start up the actual server process and send to Nl.Tl.Q the actual service socket identifier: N3.T3.S. The connection (N1.T1.Q,N2.T2.P) can then be closed, and the user can do a RECEIVE on (Nl.Tl.Q,N3.T3.S). The serving process can SEND on (N3.T3.S,Nl.Tl.Q). There are many variations on this scheme, some involving the user process doing a RECEIVE on a different socket (e.g. (Nl.Tl.X,U.U.U)) with the server doing SEND on (N3.T3.S,Nl.Tl.X).

Without showing all the detail of synchronization of sequence numbers and the like, we can illustrate the exchange as shown below.

```
USER                                      SERVER
                                      1. RECEIVE(N2.T2.P,U.U.U)
1. SEND(N1.T1.Q,N2.T2.P)       ==>
                               <==    2. SEND(N2.T2.P,N1.T1.Q) with "N3.T3.S" as data
2. RECEIVE(N1.T1.Q,N2.T2.P)
3. CLOSE(N1.T1.Q,N2.T2.P)      ==>
                               <==    3. CLOSE(N2.T2.P,N1.T1.Q)
4. RECEIVE(N1.T1.Q,N3.T3.S)
                               <==    4. SEND(N3.T3.S,N1.T1.Q)
```

Reconnection Protocol Example

Figure 3.3-1

At this point, a connection is open between N1.T1.Q and N3.T3.S. A variation might be to have the user do an extra RECEIVE on (Nl.T1.X,U.U.U) and have the data “N1.T1.X” be sent in the first user SEND. Then, the server can start up the real serving process and do a SEND on (N3.T3.S,N1.T1.X) without having to send the “N3.T3.S” data to the user. Or perhaps both server and receiver exchange this data, to assure security of the ultimate connection (i.e. some wild process might try to connect to N1.T1.X if it is merely RECEIVING on foreign socket U.U.U).

We do not propose any specific reconnection protocol here, but leave this to further deliberation, since it is really a user level protocol issue.

Further work on reconnection is in progress and version 4 of TCP may include provisions for reconnection via TCP control exchanges.

4. TCP DESIGN

4.1 Introduction

The TCP is designed to offer highly reliable, sequenced, and flow-controlled interprocess communication across network boundaries. A fundamental notion in the design is that every octet (8 bit byte) of data in an internetwork packet has a sequence number. Since every octet is sequenced, each of them can be acknowledged individually or collectively. In particular, 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 straightforward duplicate detection in the presence of retransmission.

> This also permits gateways to fragment packets as needed to get them across networks with short packet sizes. There is current discussion of how and where fragmentation should be done, and it may be that in version 4 TCP fragmentation is removed with the view that it is an internet function not specific to TCP.

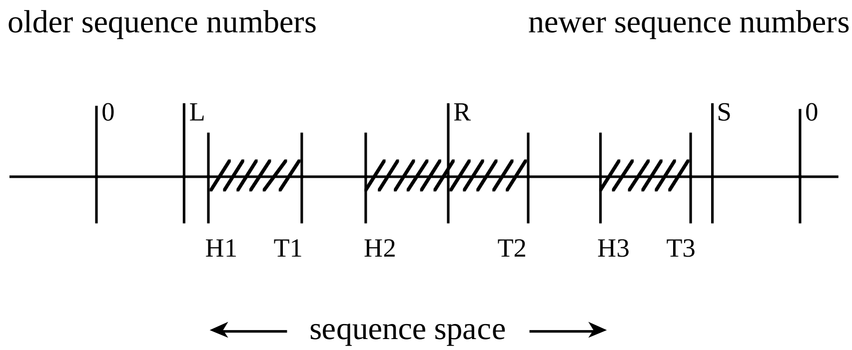
It is essential to remember that the actual sequence number space is finite, though very large. In the current design, this space ranges from 0 to 2**32–1. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2**32. This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from 2**32–1 to 0 again. The typical kinds of sequence number comparisons which the TCP must perform include:

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

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

(c) determining that an incoming packet contains sequence numbers which are expected (i.e. that the packet “overlaps” the receive window).

The TCP typically maintains status information about each connection, as is illustrated in figure 4.1-1, below.



TCP State Information for Sending Sequence Space

Figure 4.1-1

L = oldest, unacknowledged sequence number
S = next sequence number to be sent
A = acknowledgement (next sequence number expected by the acknowledging TCP)
H(i) = first sequence number of the i-th packet
T(i) = last sequence number of the i-th packet

An acceptable acknowledgement, A, is one for which the inequality below holds:

$$0 < (A - L) <= (S - L) \quad (4.1\text{-}1)$$

We will often write equation (4.1-1) in the form below:

$$L < A <= S \quad (4.1\text{-}1')$$

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

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

$$0 < (T(i) - L) < (A - L) \quad (4.1\text{-}2)$$

In this instance, H(i) and T(i) are related by the equation:

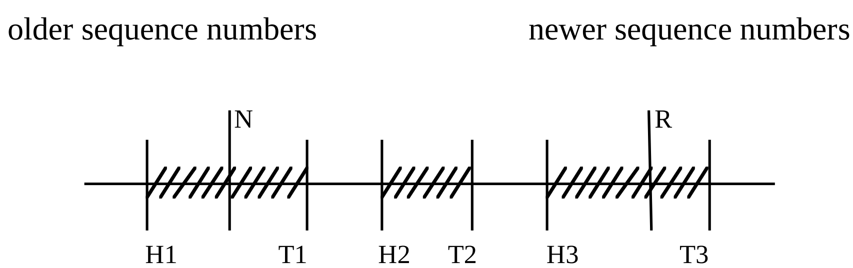
$$T(i) = H(i) + n(i) - 1 \quad (4.1\text{-}3)$$

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

Finally, a packet is judged to occupy a portion of valid receive sequence space if

$$0 <= (T - N) < (R - N) \tag{4.1-4}$$

Where T is the last sequence number occupied by the packet, N is the next sequence number expected on an incoming packet, and R is the right edge of the receive window, as shown in figure 4.1-2.



Receive Sequence State Information
Figure 4.1-2

N = next sequence number expected on incoming packets
R = last sequence number expected on incoming packets, plus one
H(i) = first sequence number occupied by the i-th incoming packet
T(i) = last sequence number occupied by the i-th incoming packet

R and N in figure 4.1-2 are related by the equation:

$$R = N + W \tag{4.1-5}$$

Where W = the receive window size.

Note that the acceptance test for a packet, since it requires the end of a packet to lie in the window, is somewhat more restrictive than is absolutely necessary. If at least the first sequence number of the packet lies in the receive window, or if some part of the packet lies in the receive window, then the packet might be judged acceptable. Thus, in figure 4.1-2, at least packets 1

(H(l)–T(l)) and 2 (H(2)–T(2)) are acceptable by the strict rule and packet 3 (H(3)–T(3)) may or may not be, depending on the strictness of interpretation of the rule.

Note that when R = N, the receive window is zero and no packets should be acceptable except ACK packets. Thus, it should be possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs on a non-zero send window.

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 packet data space (see section 4.3.2 for typical internet TCP packet format). Consequently, we must adopt rules for implicitly assigning sequence numbers to control. In version 3 these sequenced controls have been reduced to only the SYN and FIN controls which 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 packet in which it occurs, while the FIN is considered to occur after the last actual data octet in a packet in which it occurs. The packet length includes both data and sequence-space-occupying controls.

The main jobs of the TCP are:

a. Connection management (establishing and closing full-duplex connections)

b. “Packetizing” of user letters into segments for internet transmission

c. Reassembly of fragments into segments and segments into letters. (Note that the reassembly of fragments into segments may become an internet protocol function and of no concern to TCP in version 4.)

d. Flow control, sequencing, duplicate detection, and retransmission for each connection.

e. Reacting to user requests for service

In the sections which follow, we elaborate on the way in which the TCP is designed to carry out each of these tasks.

4.2 Connection Management

4.2.1 Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. Now 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 packets from previous incarnations of the connection?”. This problem becomes harmfully 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.

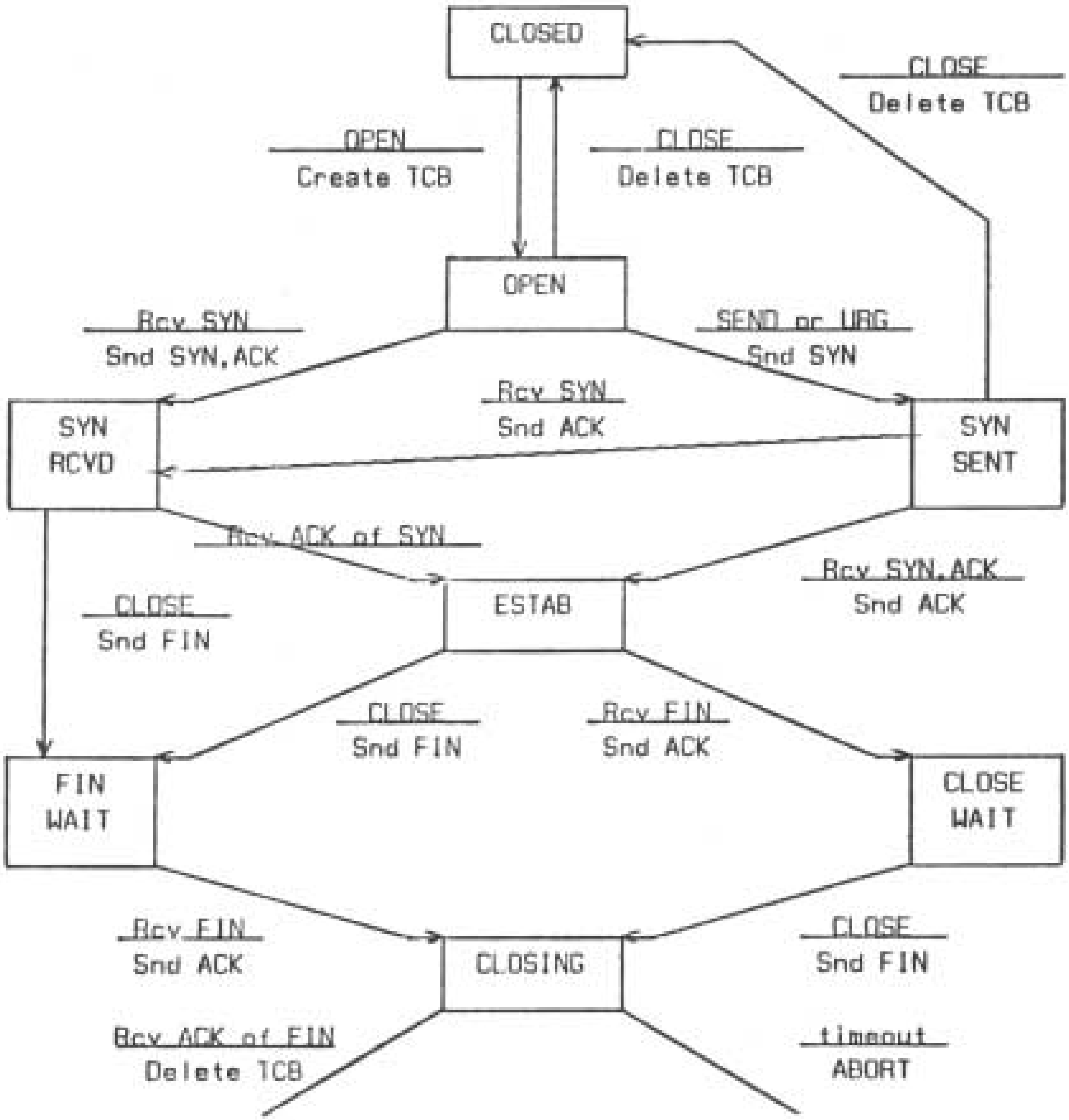
The essence of the solution [Tomlinson74] is that the initial sequence number [ISN] must be chosen so that a particular sequence number can never refer to an “old” octet. Once the connection is established the sequencing mechanism provided by the TCP filters out duplicates.

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 packet 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 [Tomlinson74]. More on the subject, and algorithms for implementing the clock-driven scheme can be found in [Dalal74, Dalal75, Cerf76b].

4.2.2 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” packet which carries no acknowledgement after having sent a “SYN” earlier. Of course, the arrival of an old duplicate “SYN” packet can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of “reset” packets can disambiguate these cases. Several examples of connection initiation are offered below, using a notation due to Tomlinson. Although these examples do not show connection synchronization using data-carrying packets, 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 (see figure 4.2-1)).



TCP Connection State Diagram

Figure 4.2-1

The simplest three-way handshake is shown in figure 4.2-2 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP packet from TCP A to TCP B, or arrival of a packet at B from

A. Left arrows (<--), indicate the reverse. Ellipsis (…) indicates a packet which is still in the network (delayed). An "XXX" indicates a packet which is lost or rejected. Comments appear in parentheses. TCP states are keyed to those in figure 4.2-1, and represent the state AFTER the departure or arrival of the packet (whose contents are shown in the center of each line). Packet 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, generally, in the interest of clarity.

```
    TCP A                                                   TCP B
1.  OPEN                                                    OPEN
2.  SYN-SENT     -->  <SEQ 100><SYN>                   -->  SYN-RECEIVED
3.  ESTABLISHED  <--  <SEQ 300><SYN><ACK 101>          <--  SYN-RECEIVED
4.  ESTABLISHED  -->  <SEQ 101><ACK 301>               -->  ESTABLISHED
5.  ESTABLISHED  -->  <SEQ 101><ACK 301><DATA>         -->  ESTABLISHED
```

Basic 3-Way Handshake for Connection Synchronization

Figure 4.2-2

In line 2 of figure 4.2-2, TCP A begins by sending a SYN packet 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 (per figure 4.1-3), the acknowledgement field indicates TCP B is now expecting to hear sequence 101, implicitly acknowledging the SYN which occupied sequence 100.

At line 4, TCP A responds with an empty packet containing an ACK for TCP B's SYN, and in line 5, TCP A sends some data. Note that the sequence number of the packet 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 4.2-3. Each TCP cycles from OPEN 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 OPEN 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” connection in section 4.2.3.

	TCP A				TCP B
1.	OPEN				OPEN
2.	SYN-SENT	-->	<SEQ 100><SYN>	…	
3.	SYN-RECEIVED	<--	<SEQ 300><SYN>	<--	SYN-SENT
4.		…	<SEQ 100><SYN>	-->	SYN-RECEIVED
5.	SYN-RECEIVED	-->	<SEQ 101><ACK 301>	…	
6.	ESTABLISHED	<--	<SEQ 301><ACK 101>	<--	SYN-RECEIVED
7.		…	<SEQ 101><ACK 301>	-->	ESTABLISHED

Simultaneous Connection Synchronization

Figure 4.2-3

	TCP A				TCP B
1.	OPEN				OPEN
2.	SYN-SENT	-->	<SEQ 100><SYN>	…	
3.	(duplicate)	…	<SEQ 1000><SYN>	-->	SYN-RECEIVED
4.	SYN-SENT	<--	<SEQ 300><SYN><ACK 1001>	<--	SYN-RECEIVED
5.	SYN- SENT	-->	<SEQ 1001><RST><ACK 301>	-->	OPEN (ACK is ok)
6.		…	<SEQ 100><SYN>	-->	SYN-RECEIVED
7.	SYN- SENT	<--	<SEQ 400><SYN><ACK 101>	<--	SYN-RECEIVED
8.	ESTABLISHED	-->	<SEQ 101><ACK 401>	-->	ESTABLISHED

Recovery from Old Duplicate SYN

Figure 4.2-4

As a simple example of recovery from old duplicates, consider figure 4.2-4. At line 3, and 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 packet believable. TCP B, on receiving the RST, returns to the OPEN 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.

4.2.3 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 packet containing SYN. This scenario leads to the example shown in figure 4.2-5. 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.  OPEN                                           ESTABLISHED
3.  SYN-SENT  -->  <SEQ 400><SYN>             -->  (??)
4.  (!!)      <--  <SEQ 300><ACK 100>         <--  ESTABLISHED
5.  SYN-SENT  -->  <SEQ 100><RST><ACK 300>    -->  (Abort!!)
```

Half-Open Connection Discovery

Figure 4.2-5

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 packet 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 4.2-6. 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.

In figure 4.2-7, 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 OPEN state.

```
    TCP A                                               TCP B
1.  (CRASH)                                      (send 300, receive 100)
2.  (??)         <--  <SEQ 300><ACK 100><DATA 10>  <--  ESTABLISHED
3.               -->  <SEQ 100><RST><ACK 310>      -->  (ABORT!!)
```

Active Side Causes Half-Open Connection Discovery

Figure 4.2-6

```
    TCP A                                               TCP B
1.  OPEN                                                OPEN
2.               …    <SEQ Z><SYN>                 -->  SYN-RECEIVED
3.  (??)         <--  <SEQ X><SYN><ACK Z+1>        <--  SYN-RECEIVED
4.               -->  <SEQ Z+1><RST><ACK X+1>      -->  (return to OPEN!)
5.  OPEN                                                OPEN
```

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 4.2-7

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 packet arrives which apparently is not intended for the current or a future instantiation of the connection. A reset should not be sent if it is not clear that this is the case. Thus, if any packet arrives for a nonexistent connection, a reset should be sent. If a packet ACKs something which has never been sent on the current connection, send reset.

1. If the connection is in any non-synchronized state (OPEN, SYN-SENT, SYN-RECEIVED) or if the connection does not exist, a reset (RST) should be formed and sent for any packet that does not acknowledge something the receiver sent earlier. The RST should take its SEQ field from the ACK field of the offending packet (if it has one) and its ACK field should acknowledge all data and control in the offending packet.

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

Reset Processing

All RST (reset) packets are validated by checking their ACK-fields and SEQ fields (if appropriate). If the RST acknowledges something the receiver sent (but has not yet received acknowledgment for), the RST must be valid. RST packets will have ACK fields which acknowledge any data and control in the offending packet to assure acceptability of the RST.

The receiver of a RST first validates it, then changes state. If the receiver was in a non-synchronized state (OPEN, SYN-SENT, SYN-RECEIVED) it returns to the OPEN state (possibly modifying the foreign socket specification in the process—see section 4.3.3). If the receiver was in a synchronized state (ESTABLISHED, FIN-WAIT, CLOSE-WAIT, CLOSING), it aborts the connection and advises the user (see section 2.4.3—error 14).

4.2.4 Knowing When to Keep Quiet

A basic goal of the TCP design is to prevent packets 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. The ISN thus cycles every 4.55 hours, approximately. Since we assume that packets 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.

To be sure that a TCP does not create a packet that carries a sequence number which may be duplicated by an old packet remaining in the network the TCP must keep quiet for a maximum packet lifetime (MPL) 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 MPL 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.

4.2.5 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 signalled 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:

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

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

c) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN packet can be constructed and placed on the outgoing packet 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 packets 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 (see figure 4.2-1). It should be noted that a TCP receiving a FIN will ACK but not send its own FIN until the 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 (see Event Codes, section 2.4.3). 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 (see 2.4.3).

Case 3: both users close simultaneously

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

4.2.6 End of Letter Sequence Number Adjustments

The difference between the sequence numbers of the first octets of data in any pair of letters on a given connection is always equal zero modulo the receive buffer size. That is, whenever an EOL is transmitted, the sender advances his send sequence number by an amount (in the range 0 to buffer size–1) sufficient to consume all the unused space in the receiver's buffer. The amount of space consumed in this fashion is deducted from the send window just as is the space consumed by actual data.

The idea is that an EOL signals the consumption of the rest of the space in the buffer and that the data sequence numbers reflect that. 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 packet that contains a 16 bit buffer size field in the TCP header, the presence of the field being signaled by a BSZ control bit.

If a letter starts at sequence number x and is n octets long and the buffer size is m octets, then the next letter starts at $x+im$, where i is a positive integer such that $im > n > (i-1)m$.

If a buffer size is specified and then all receive buffers provided by the user must be exactly that size, otherwise the TCP should return an error indication.

4.2.7 The Communication of Urgent Information

The urgent mechanism is used to indicate the need for special processing of the data traversing the connection. This mechanism permits a point in the data stream to be designated as the end of “urgent” information. Whenever this point is beyond the left window edge at the receiving TCP, that TCP so informs the application program, so the program can switch into a mode of operation intended to scan through the data up to the urgent pointer in an attempt to extract the urgent information. The exact nature of this scan depends on the higher level protocol being employed, but would typically involve discarding information.

As soon as an urgent pointer is in advance of the left edge the TCP should tell the user to go into “read fast” mode, when left edge catches up to urgent pointer the TCP should tell user to go into “read 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 pointer which is carried in a field of all packets transmitted while the urgent pointer exceeds the left window edge. A control bit (URG) indicates that the packet contains a 16-bit field which should be added to the packet sequence number to yield the urgent pointer. The absence of this bit indicates that the urgent pointer has not changed.

It should be mentioned that coordinating the urgent pointer with a letter boundary acts to insure timely delivery of the urgent information to the destination process.

4.2.8 The Possibility of Less than Reliable Communication

As a future development TCP may be called on to support other types of applications that require different types of service. One feature included at this time to enable such development is the beginning of letter flag, or BOL, which could be used in conjunction with EOL (end of letter) in a mode of operation where the receiver acknowledges everything to keep retransmissions at a minimum to provide a special type of service. In this mode TCP provides the user with complete letters but allows letters to be lost in between the ones actually delivered. For this mode the TCP must be able to find the beginning of a letter as well as the end. (Actually this could be done without a special BOL since the end of one letter is the beginning of the next, but a BOL allows a slight improvement in the probability of finding whole letters.)

4.2.9 TCP Connection State Transitions

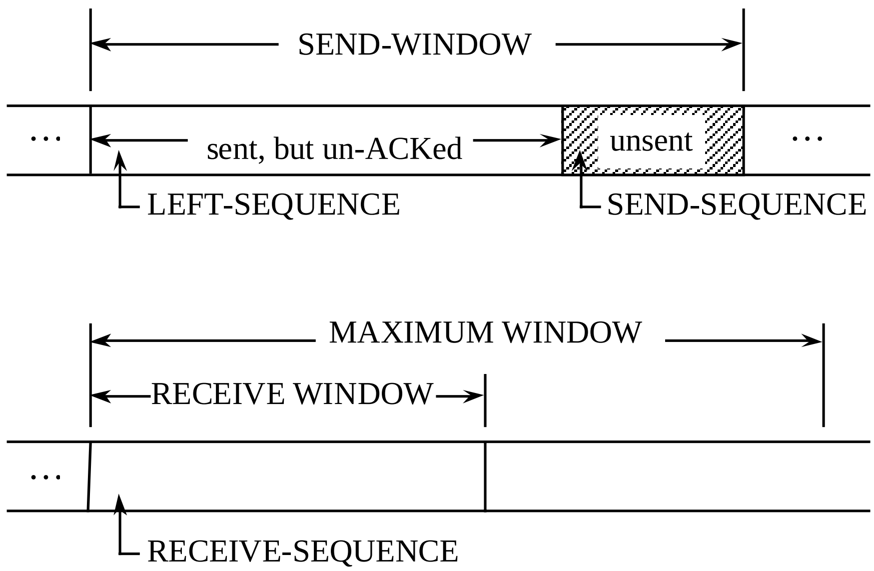
The foregoing sections on connection management were succinctly represented with a simple state diagram, shown in figure 4.2-1. The figure only illustrates state changes (and actions which occur as a result), but addresses neither error conditions nor actions which are not connected with state changes. In this section, more detail is offered with respect to the reaction of the TCP to various events (user command, packet arrivals). The characterization of TCP processing of control packets and reaction to user commands is relatively terse. Certain implementation choices can make the realization of the specified processing fairly compact, but these implementation issues are dealt with in sections 4.3–4.5. For the sake of compactness, this section deliberately avoids much explanatory material which can be found in the implementation sections. Thus, this section is intended more as a reference than as a tutorial, and really requires exposure to sections 4.3–4.5 to be fully useful.

Furthermore, it should be kept in mind that some control information occupies sequence number space along with data (see figure 4.1-3). This latter point means that there is a natural order in which to process the data and control portions of an incoming packet and that certain controls will change the connection state BEFORE later control or data (i.e., those assigned higher sequence numbers) is processed. An implementation could take advantage of this sequencing to keep track of which portions of a packet (data and control) had already been processed. Note that by assigning sequence numbers to some control bits, it is possible to use the normal acknowledgment mechanisms to acknowledge receipt of control information and to filter out duplicates.

A natural way to think about incoming packet processing 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 queued and processed in sequence number order. We are, in this view, ignoring for the moment the problem of reassembling segments that were fragmented at gateways, or which overlap other, already received, packets.

We have chosen to organize the description according to the connection state, to key the description to figure 4.2-1. In the following specifications the user events are mutually exclusive, while the incoming packet may call for some or all of the steps described to be carried out. When a packet causes a state change, but carries more data or control which should be processed, it is appropriate to continue processing in the new state, but processing of the packet’s acknowledgment field or sequence number field should not be repeated (lest a packet which looked valid before appear to be an old duplicate or have a bad acknowledgment field as an artifact of the state change).

A TCP must typically maintain certain state information about each connection in order to sequence packets. For reference, we present a list of terms below (see section 4.3 for more detail) which are used in the action summaries for each state (also see figure 4.2-8).



Sequence Number Management
Figure 4.2-8

Glossary of terms

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

BOL—A control bit (Begin of Letter) occupying no sequence space, indicating that this packet begins a logical letter with the first data octet in the packet.

BSZ—A control bit (buffer size) in the incoming SYN packet, occupying no sequence space, used to indicate the presence of the buffer size field.

BUFFER-SIZE—A control field (buffer size) in the incoming SYN packet, occupying no sequence space, used to state the receive data buffer size of the sender of this control. May only be sent in a packet that also carries a SYN.

EOL—A control bit (End of Letter) occupying no sequence space, indicating that this packet ends a logical letter with the last data octet in the packet. 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.

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 transmit “window.”

PKT-ACKNOWLEDGMENT—The sequence number in the acknowledgment field of the arriving packet.

PKT-LENGTH—The amount of sequence number space occupied by a packet, including any controls which occupy sequence space.

RECEIVE-SEQUENCE—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 packets overlapping the range RECEIVE-SEQUENCE to RECEIVE-SEQUENCE + RECEIVE-WINDOW – 1 carry acceptable data or control. Packets containing sequence numbers entirely outside of this range are considered duplicates and discarded. This topic is discussed in detail in section 4.5 on window allocation policies.

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 packet, whether it should honor the reset command or ignore it. In no case does receipt of a packet containing RST give rise to a RST in response.

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, see section 4.2.1) 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 packets from the remote (data receiving) TCP. The range of sequence numbers which may be emitted by a TCP lies between SEND-SEQUENCE and LEFT-SEQUENCE + SEND-WINDOW – 1.

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

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—An optional control field present 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.

State and Event Descriptions

Certain error responses shown below are generic. See section 2.5 for details on TCP-to-user messages. User commands referencing connections that do not exist receive "connection not open" (EP3) and references to connections not accessible to the caller receive "connection illegal for this process" (EP1). We have not repeated these generic responses in each description of action performed for each connection state. Overt attempts to SEND or signal URGENT on a connection with unspecified foreign socket results in a "foreign socket unspecified" (E5) response.

CLOSED STATE (i.e. connection does not exist)

User Commands

1. OPEN

   Create a new transmission control block TCB to hold connection state information. Fill in local socket identifier, foreign socket if present (the connection is passively "listening" if the foreign socket is unspecified), and user timeout information. Some implementations may issue SYN packets if the foreign socket is fully specified. In this case, an initial sequence number (ISN) is selected and a SYN packet formed and sent. The LEFT-SEQUENCE is set to ISN, the SEND-SEQUENCE to ISN + 1, and SYN-SENT state is entered.

If the caller does not have access to the local socket specified, return “connection illegal for this process” (EP1). If there is no room to create a new connection, return “insufficient resources” (4).

2. SEND, URGENT, CLOSE, ABORT, RECEIVE, STATUS

   Error return “Connection not open” (EP3).

   If the user should no have access to such a connection, “connection illegal for this process” (EP1) may be returned.

Incoming Packets.

All incoming packets are discarded. For an incoming packet containing an ACK, except for incoming RST packets which should be ignored, a RST is created with a sequence number (PKT-SEQUENCE) equal to the acknowledgment field (PKT-ACKNOWLEDGMENT) of the incoming packet (if it has one; otherwise PKT-SEQUENCE is set to zero or, ISN). The acknowledgment field of the RST should be set to the sum of the incoming PKT-SEQUENCE and PKT-LENGTH. The RST and ACK control bits for the outbound packet should be set (see figure 4.2-6).

OPEN STATE

User Commands

1. OPEN

   Return “already OPEN” (EP6)

2. SEND or URGENT

   Select an ISN, send a SYN packet, set LEFT-SEQUENCE to ISN and SEND-SEQUENCE to ISN + 1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN packet or queued for transmission after entering ESTABLISHED state.

URGENT can be sent as a combination SYN, URG packet (see figure 4.1-3 and section 4.3.2). If there is no room to queue the request, respond with “insufficient resources” (4).

3. RECEIVE

   Queue request if there is space, or respond with “insufficient resources” (4)

4. CLOSE

   Delete TCB, return “ok” (0). Any outstanding RECEIVES should be returned with “closing” responses (P12).

5. ABORT

   Delete TCB, return “ok” (0); any outstanding RECEIVES should be returned with “connection reset” (P14) responses.

6. STATUS

   Return state = OPEN.

Incoming Packets

1. ACK

   Any acknowledgement is bad if it arrives on a connection still in the OPEN state. A reset (RST) packet should be formed for any arriving ACK-bearing Packet, except another RST. The RST should be formatted as follows:

   <SEQ PKT-ACKNOWLEDGMENT><RST><ACK PKT-SEQUENCE + PKT-LENGTH>

   Thus the RST will acknowledge any text or control in the offending packet.

2. SYN

   RECEIVE-SEQUENCE should be set to PKT-SEQUENCE + l and any other control or text should be queued for processing later. ISN should be selected and a SYN packet sent of the form,

<SEQ ISN><SYN><ACK RECEIVE-SEQUENCE>

SEND-SEQUENCE should be set to ISN + 1 and LEFT-SEQUENCE to ISN. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control (combined with SYN) will be processed in the SYN-RECEIVED state. Processing of SYN and ACK should not be repeated.

3. Other text or control

   Any other control or text-bearing packet (not containing SYN) will have an ACK and thus will be discarded by the ACK processing. An incoming RST packet could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection.

SYN-SENT STATE

User Commands

1. OPEN

   Return “already OPEN” (EP6)

2. SEND or URGENT

   Queue for processing after the connection is ESTABLISHED or packetize, starting with the current SEND-SEQUENCE number. Typically, nothing can be sent yet, anyway, because the send window has not yet been set by the other side. If no space, return “insufficient resources” (4).

3. RECEIVE

   Queue for later processing unless there is no room, in which case return “insufficient resources” (4).

4. CLOSE

   Delete the TCB and return “closing” (P12) responses to any queued SENDs, RECEIVES, or URGENTs.

5. ABORT

   Delete the TCB and return “reset” (P14) responses to any queued SENDS, RECEIVES, or URGENTs.

6. STATUS

   Return state = SYN-SENT; SEND-SEQUENCE, RECEIVE-WINDOW

Incoming packets

1. ACK

   If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE then the ACK is acceptable. LEFT-SEQUENCE should be advanced to equal PKT-ACKNOWLEDGMENT, and any packet(s) on the retransmission queue which are thereby acknowledged should be removed.

   If the packet acknowledgment is not acceptable, a RST packet should be formed (except when the offending packet is also a RST) which carries the PKT-ACKNOWLEDGMENT as a sequence number, and acknowledges all text and control of the offending packet.

2. SYN

   RECEIVE-SEQUENCE should be set to PKT-SEQUENCE + 1 and any packet text or control queued for later processing. If the packet has an ACK, change the connection state to ESTABLISHED, otherwise enter SYN-RECEIVED. In any case, form an ACK packet:

   <SEQ SEND-SEQUENCE><ACK RECEIVE-SEQUENCE>

   and send it.

3. RST

   Delete TCB, enter CLOSED state.

4. Other text or control.

   Incoming packets with other control or text combined with SYN will be processed in SYN-RECEIVED or ESTABLISHED state. Arriving packets which do not contain SYN are either old duplicates or out-of-order arrivals. Since these must contain ACK fields, they will have been discarded by earlier ACK processing.

5. User Timeout.

   If the user timeout expires on a packet in the retransmission queue, abort the connection, notifying the user “retransmission timeout, connection aborted” (EP9), and flushing all queues, returning RECEIVES, SENDS or URGENTs with the same error (EP9). Delete the TCB.

SYN-RECEIVED STATE

User Commands

1. OPEN

   Return “already OPEN” (EP6)

2. SEND or URGENT

   Queue for later processing after entering ESTABLISHED state, or packetize and queue for output. If no space to queue, respond with “insufficient resources” (4)

3. RECEIVE

   Queue for processing after entering ESTABLISHED state. If there is no room to queue this request, respond with “insufficient resources” (4).

4. CLOSE

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

5. ABORT

Delete TCB, send a RST of the form;

<SEQ SEND-SEQUENCE><RST><ACK RECEIVE-SEQUENCE>

and return any unprocessed SENDs, URGENTs, or RECEIVEs with “reset” code (P14).

6. STATUS

   Return state = SYN-RECEIVED, LEFT-SEQUENCE, SEND-SEQUENCE, SEND-WINDOW, RECEIVE-SEQUENCE, RECEIVE-WINDOW, and other desired statistics number of (SEND, RECEIVE buffers queued), packets queued for reassembly, for retransmission, etc.

Incoming Packets

1. Check PKT-SEQUENCE

   If RECEIVE-SEQUENCE < PKT-SEQUENCE + MAX (0,PKT-LENGTH – l) < RECEIVE-SEQUENCE + RECEIVE-WINDOW then the packet sequence number is acceptable. If not, form a reset (RST) packet:

   <SEQ PKT-ACKNOWLEDGMENT> <RST>
   <ACK PKT-SEQUENCE + TEXT-LENGTH>

   If the incoming packet is RST or has no ACK, discard it, and do not send RST formed above. Note that the test above guarantees that the last sequence number used by the packet lies in the receive-window. The special “MAX” operation makes certain that empty ACK packets, whose length are 0, will be accepted. If the RECEIVE-WINDOW is zero, no packets will be acceptable, but special allowance should be made to accept valid ACKs.

   Insisting that PKT-SEQUENCE (i.e., the first sequence number occupied by the packet) lie in the RECEIVE-WINDOW could lead to deadlock in the case of alternate gateway routing and different fragmentation.

   A Scenario:

   Assume the receivers RECEIVE-SEQUENCE is 1.

The sender transmits a packet (p1) containing data octets 1 through 8.

Gateway A fragments p1 into two new packets, the first (p2) carries data octets 1 through 4, and the second (p3) carries data octets 5 through 8.

Packet p2 arrives at the receiver and is found acceptable. The receiver sets the RECEIVE-SEQUENCE to 5.

Gateway A breaks.

The sender timesout and retransmits p1 as p4.

The receiver finds p3 afflicted with errors and discards it.

Gateway B fragments p4 into three new packets, the first (p5) carries data octets 1 through 3, the second (p6) carries data octets 4 through 6, the third (p7) carries data octets 7 and 8.

When p5 arrives at the receiver it is acknowledged then discarded since it is completely below the RECEIVE-SEQUENCE.

When p6 arrives at the receiver it is acknowledged then if the special MAX function were not used it would be discarded since its PKT-SEQUENCE is below the RECEIVE-SEQUENCE.

A deadlock would develop if p6 were discarded, and if when the sender retransmitted it always sent the complete contents of the original packet p1.

2. ACK

If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE then set LEFT-SEQUENCE = PKT-ACKNOWLEDGMENT, remove any acknowledged packets from the retransmission queue, and enter ESTABLISHED state.

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

3. RST

   If the packet has passed sequence and acknowledgment tests, it is valid. Return this connection to OPEN state. The user need not be informed. All packets on the retransmission queue should be removed. All packetized buffers must be assigned new sequence numbers, so they should be requeued for re-packetizing.

4. Other text or control

   If there is other control or text in the packet, it can be processed when the connection enters the ESTABLISHED state.

5. User Timeout

   If the user timeout expires on any packet in the retransmission queue, flush all queues, return outstanding SENDs, URGENTs or RECEIVEs with “user timeout, connection aborted” (EP9), and delete the TCB.

ESTABLISHED STATE

User Commands

1. OPEN

   Respond with “already OPEN” (EP6)

2. SEND or URGENT

   Packetize the buffer, send or queue it for output. If there is insufficient space to remember this buffer, simply respond with “insufficient resources” (4).

3. RECEIVE

   Reassemble queued incoming segments into receive buffer, and return to user. Mark “end of letter” (EOL) if this is the case. If buffer size is not one octet then do end-of-letter/buffer-size adjustment processing. 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 “insufficient resources” (4).

4. CLOSE

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

5. ABORT

   Delete TCB and send a reset packet:

   <SEQ SEND-SEQUENCE><RST><ACK RECEIVE-SEQUENCE>

   All queued SENDs, URGENTs, and RECEIVEs should be given “reset” responses (P14); all packets queued for transmission (except for the RST formed above) or retransmission should be flushed.

6. STATUS

   Return state = ESTABLISHED; SEND-SEQUENCE, LEFT-SEQUENCE, SEND-WINDOW, RECEIVE-SEQUENCE, RECEIVE-WINDOW, and other statistics, as desired.

Incoming Packets

1. Check PKT-SEQUENCE

   All packets are generally processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in PKT-SEQUENCE order. If a packet’s contents straddle the boundary between old and new, only the new parts should be processed.

   If RECEIVE-SEQUENCE <= PKT-SEQUENCE + MAX(PKT-LENGTH-l,0) < RECEIVE-SEQUENCE + RECEIVE-WINDOW then packet is acceptable. Otherwise if PKT-LENGTH is non-zero, an empty acknowledgment packet should be sent:

   <SEQ SEND-SEQUENCE><ACK RECEIVE-SEQUENCE>

   In any case, unacceptable packets should be discarded.

2. ACK

   If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE then set LEFT-SEQUENCE = PKT-ACKNOWLEDGMENT. Any packets 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” (0) response). If the ACK is a duplicate, it can be ignored.

3. RST

   All pending RECEIVEs, SENDs, and URGENTs receive “reset” (P14) responses. All packet queues are flushed. The TCB is deleted. User also receives an unsolicited general “reset” signal (P14).

4. SYN

   Ignore the SYN. A packet carrying a SYN could not have passed through the sequence check unless it had control or text lying beyond the SYN which was acceptable. To prevent repeat processing of controls or text, such packets could be “marked” so that all duplicate control or text is removed before they exit sequence-number check. Other marking strategies could be employed to achieve the same effect.

5. URG

   Signal user that remote side has urgent data (P11) 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.

6. Packet text

   Once in the ESTABLISHED state, it is possible to deliver packet text to user RECEIVE buffers. Some preliminary packet reassembly may be required to form valid segments from fragments created at a gateway. 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 the buffer size is not one octet then the end-of-letter/buffer-size adjustment processing must be done.

7. FIN

   An ACK packet should be sent, acknowledging the FIN. The user should be signalled “connection closing” (P12) and similar responses should be returned for any outstanding RECEIVEs which cannot be satisfied. Connection state should be changed to CLOSE-WAIT.

8. User Timeout

   If the user timeout expires on a packet in the retransmission queue, flush all queues, return “user timeout, connection aborted” (EP9) for all outstanding SENDs, URGENTs, and RECEIVEs, and delete the TCB. The user should receive an unsolicited message of the same form (EP9).

FIN-WAIT STATE

User-Commands

1. OPEN

   Return “already OPEN” (EP6)

2. SEND or URGENT

   Return “connection closing” (EP12) and do not service request.

3. RECEIVE

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

4. CLOSE

   Strictly speaking, this is an error and should receive a “connection closing” (EP12) response. An “ok” (0) response would be acceptable, too, as long as a second FIN is not emitted.

5. ABORT

A reset packet (RST should be formed and sent:

<SEQ SEND-SEQUENCE><RST><ACK RECEIVE-SEQUENCE>

Outstanding SENDs, URGENTs, RECEIVEs, CLOSEs, and/or packets queued for retransmission, or packetizing, should be flushed, with appropriate “connection reset” (P12).

6. STATUS

   Respond with state = FIN-WAIT, SEND-SEQUENCE, LEFT-SEQUENCE, SEND-WINDOW, RECEIVE-SEQUENCE, RECEIVE-WINDOW, and other statistical information, as desired.

Incoming packets

1. Check PKT-SEQUENCE

   If RECEIVE-SEQUENCE <= PKT-SEQUENCE + MAX(PKT-LENGTH-1,0) < RECEIVE-SEQUENCE + RECEIVE-WINDOW then packet sequence is acceptable. Otherwise, if PKT-LENGTH is non-zero, an ACK packet should be sent:

   <SEQ SEND-SEQUENCE><ACK RECEIVE-SEQUENCE>

   In any case, an unacceptable packet should be discarded.

2. ACK

   If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE, then LEFT-SEQUENCE should be advanced appropriately and any acknowledged packets deleted from the retransmission queue. SENDs or URGENTs which are thereby completed can also be acknowledged to the user. ACK’s outside of the SEND-WINDOW can be ignored. If the retransmission queue is empty, the user’s CLOSE can be acknowledged (“OK” (0)) and the TCB deleted.

3. RST

   All RECEIVEs, SENDs, and URGENTs still outstanding should receive “reset” (P14) responses. All packet queues should be flushed and the connection TCB deleted. User should also receive an unsolicited general “connection reset” (P14) signal.

4. SYN

   This case should not occur, since a duplicate of the SYN which started the current incarnation will have been filtered in the PKT-SEQUENCE processing. Other SYN’s could not have passed the PKT-SEQUENCE check at all (see SYN processing for ESTABLISHED state).

5. URG

   Signal the user that the remote side has urgent data (P11) 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.

6. Packet Text

   If there are outstanding RECEIVEs, they should be satisfied, if possible, with the text of this packet, 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.

7. FIN

   The FIN should be acknowledged. Return any remaining RECEIVEs with “connection closing” (P12) and advise user that connection is closing with a general signal (P12). If the retransmission queue is not empty, then enter CLOSING state, otherwise, delete the TCB.

8. User Timeout

   If the user timeout expires on a packet in the retransmission queue, flush all queues, return “user timeout, connection aborted” messages for all outstanding SENDs, RECEIVEs, CLOSES or URGENTs, send an unsolicited general message of the same form to the user, and delete the TCB.

CLOSE-WAIT STATE

User Commands

1. OPEN

   Return “already OPEN” error (EP6)

2. SEND or URGENT

   Packetize any text to be sent and queue for output. If there is insufficient space to remember the SEND or URGENT, return “insufficient resources” (4)

3. RECEIVE

   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 packet text is awaiting delivery, the RECEIVE should get a “connection closing” (P12) response. Otherwise, any remaining text can be used to satisfy the RECEIVE. In implementations which do not acknowledge packets until they have been delivered into user buffers, the FIN packet which led to the CLOSE-WAIT state will not be processed until all preceding packet text has been delivered into user buffers. Consequently, for such an implementation, all RECEIVEs in CLOSE-WAIT state will receive the “connection closing” (P12) response.

4. CLOSE

   Queue this request until all preceding SENDs or URGENTs have been packetized; then send a FIN packet, enter CLOSING state.

5. ABORT

   Flush any pending SENDs, RECEIVEs and URGENTs, returning “connection reset” (P14) responses for them. Form and send a RST packet:

   <SEQ SEND-SEQUENCE><RST><ACK RECEIVE-SEQUENCE>

   Flush all packet queues and delete the TCB.

6. STATUS

   Return state = CLOSE-WAIT, all other TCB values as for ESTABLISHED case.

Incoming Packets

1. Check PKT-SEQUENCE

   If RECEIVE-SEQUENCE <= PKT-SEQUENCE + MAX(PKT-LENGTH-l,0) < RECEIVE-SEQUENCE + RECEIVE-WINDOW then the packet sequence is acceptable. Otherwise, if PKT-LENGTH is non-zero, an ACK should be sent:

   <SEQ SEND-SEQUENCE><ACK RECEIVE-SEQUENCE>

   Unacceptable packets should be discarded. Others should be processed in sequence number order.

2. ACK

   If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE, then LEFT-SEQUENCE should be advanced appropriately and any acknowledged packets removed from the retransmission queue. Completed SENDs or URGENTs should be acknowledged to the user (“OK” (0) returns). ACK’s which are outside the receive window can be ignored.

3. RST

   All RECEIVEs, SENDs, and URGENTs still outstanding should receive “reset” (P14) responses. Packet queues should be flushed and the TCB deleted. The user should also received an unsolicited general “connection reset” signal (P14).

4. SYN

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

5. URG

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

6. Packet text

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

7. FIN

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

8. User Timeout

   If the user timeout expires on a packet in the retransmission queue, flush all queues, return “user timeout, connection aborted” (EP9) for any outstanding SENDs, RECEIVEs or URGENTs, send an unsolicited general message of the same form to the user and delete the TCB.

CLOSING STATE

User Commands

1. OPEN

   Respond with “already OPEN” (EP6)

2. SEND, URGENT

   Respond with “connection closing” (EP12)

3. RECEIVE

   Respond with “connection closing” (EP12)

4. CLOSE

   Respond with “connection closing” (EP12)

5. ABORT

Respond with “OK” (0) and delete the TCB; flush any remaining packet queues. If a CLOSE command is still pending, respond “connection reset” (P14).

6. STATUS

   Return State = CLOSING along with other TCP parameters.

Incoming packets

1. Check PKT-SEQUENCE

   If RECEIVE-SEQUENCE <= PKT-SEQUENCE + MAX(PKT-LENGTH-l,0) < RECEIVE-SEQUENCE + RECEIVE-WINDOW then packet sequence is acceptable. Otherwise, if PKT-LENGTH is non-zero, an ACK packet should be formed and sent:

   <SEQ SEND-SEQUENCE><ACK RECEIVE-SEQUENCE>

   In any case, an unacceptable packet should be discarded.

2. ACK

   If LEFT-SEQUENCE < PKT-ACKNOWLEDGMENT <= SEND-SEQUENCE, then LEFT-SEQUENCE should be advanced and any acknowledged packets deleted from the retransmission queue. SENDs or URGENTs which are thereby completed can also be acknowledged to the user. ACK’s outside of the SEND-WINDOW can be ignored.

3. RST

   Any outstanding RECEIVEs, SEND, and URGENTs should receive “reset” responses (P14). All packet queues should be flushed and the TCB deleted. Users should also receive an unsolicited general “connection reset” (P14) signal.

4. Packet text or control

   No other control or text should be sent by the remote side, so packets containing non-zero PKT-LENGTH should be ignored.

5. User Timeout

If the user timeout expires on a packet in the retransmission queue, flush all queues, return “user timeout, connection aborted” (EP9) responses for all outstanding SENDs, URGENTs, RECEIVEs, or CLOSEs, send an unsolicited message of the same form (EP9) to the user and delete the TCB.

4.3 TCP Data Structures

Our basic view of internetworking is that all internetwork packets (TCP and otherwise) have a basic internet header consisting of source/destination address, data and header length fields, and format indicator. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of internet protocols other than TCP, and for experimentation with TCP variations.

4.3.1 Internetwork Packet Format

In this section, we offer a terse descriptive summary of the contents of the internetwork header.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  TOS  |    Format     |          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  IH Length=20 | DAL=5 | SAL=3 |           Destination         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Destination continued            |    Source     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Source continued       |        Internet Options       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Internet Options continued            |   Padding     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Internet Packet Header Format

Note that each tick mark represents one bit position.

Figure 4.3-1

About Addresses:

An address is a variable length quantity (in multiples of octets). It is intended for the first octet of an address to be interpreted as a network identifier, and that the rest of the address identifies a host within that network. The address field is allowed to be even longer than that with the view that a host may multiplex between several functions, or further route messages based on the additional address bits.

If a host were to support two instances of TCP they could be assigned distinct addresses by using an additional octet of address beyond that needed to identify the host. Other examples of such processes are the XNET (cross-network debugger) server process, the gateway control process, or the packet echoer process. There is also the possibility of placing another whole layer of addressing hierarchy in this position.

The 8 bit network number, which is the first octet of the variable length address, has a value as specified in RFC 739 [Postel77]. In any case the latest information can be obtained from Jon Postel.

Version: 4 bits

There is a Version field which indicates the "shape", or format, of the Internet portion. This is version 0. Subsequent versions may employ different formats.

TOS: 4 bits

Type of service. To be defined later.

Format: 8 bits

There is a Format field which indicates the "shape", or format, of the rest (Protocol Specific portion) of the packet is (examples of format values are TCP-3, TCP-4, DSP). The format field has values as specified in RFC 739 or its successor. In any case the latest information can be obtained from Jon Postel.

Total Length: 16 bits

Total Length is the length of the packet in octets including Internet Header, Protocol Specific Header (in our case TCP Header), and Data.

IH Length: 8 bits

Internet Header Length is the length of the Internet Header in octets, and thus points to the beginning of the Protocol Specific (e.g. TCP) Header.

DAL: 4 bits

Destination Address Length in octets.

SAL: 4 bits

Source Address Length in octets.

Destination: variable

The destination address, DAL octets in length.

Source: variable

The source address, SAL octets in length.

Internet Options: variable

To be defined.

Padding: variable

Padding fields are used to ensure that the Protocol Specific (e.g. TCP) Header and the Data begin on 32 bit word boundaries.

4.3.2 TCP Packet Format

In this section, we offer a terse descriptive summary of the contents of the TCP header.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgement Number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|x x x x x x U B B E A B E R S F|                               |
|x x x x x x R S O O C O O S Y I|           Window              |
|x x x x x x G Z S S K L L T N N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Port                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Source Port                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data Offset  |   Reserved    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Urgent Pointer*        |         Buffer Size*          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        TCP Options            |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

TCP Header Format

Note that one tick mark represents one bit position.

Figure 4.3-2.

Note:

Fields marked with an asterisk are omitted if the corresponding control flag is 0.

Sequence Number: 32 bits

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

Acknowledgement Number: 32 bits

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

Control Bits: 16 bits (from left to right):

bits 0–5:		unused
bit 6:	URG:	Urgent Pointer field present
bit 7:	BSZ:	Buffer Size field present
bit 8:	BOS:	Beginning of Segment
bit 9:	EOS:	End of Segment
bit 10:	ACK:	Acknowledgment field significant
bit 11:	BOL:	Begin of Letter
bit 12:	EOL:	End of Letter
bit 13:	RST:	Reset the connection
bit 14:	SYN:	Synchronize sequence numbers
bit 15:	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 packet is willing to accept.

Destination Port: 32 bits

Source Port: 32 bits

Data Offset: 8 bits

The number of octets in the TCP Header. This indicates where the data begins.

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, except that unchecksummed option fields are replaced with zeros in the computation (see below). If a packet 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 packet.

Urgent Pointer: 16 bits

If this field is present then it communicates the current value of the urgent pointer as a positive offset from the sequence number in this packet. This field should only be sent in packets with the URG control bit set.

Buffer Size: 16 bits

If this field is present then it communicates the receive buffer size for process at the TCP which sends this packet. This field should only be sent in packets with both the BSZ and SYN control bits set.

TCP Options: variable

Options may occupy space at the end of the TCP header, and are a multiple of 8 bits in length. All options have the same basic format:

Option length: 8 bits

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

Option kind: 8 bits

C: 1 bit

If set, this option is not included in the checksum calculation. That is, in the default case (C=0) the option is included in the checksum.

P: 1 bit

If set, this option is protocol specific (i.e. option is interpreted based on format and version specified in the internet packet header fields).

Option identifier: 6 bits

There are two special cases for options.

The first is an option whose length field is zero. This marks the end of the option list. Only one octet is associated with this option, the length octet itself.

The second is an option whose length field is one. This option serves as padding and is also one octet long. This option does not terminate the option list.

Note that the list of options may be shorter than the header length field might imply. No guarantees concerning the content of the header beyond the end-of-option mark are made. The two special options are included in the checksum of the packet.

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
--	0	End of option list (checksummed), occupies one octet only
--	1	Padding (checksummed), occupies one octet only

0XX	---	Checksummed and protocol independent
000	---	Reserved

1XX	---	Checksummed, protocol dependent (TCP)
100	---	Reserved
101	4	Packet Label-sequence number for debugging purposes
102	4	Secure Open—used by TCP's communicating with BCR security system
103	2	Secure Close—used by TCP's communicating with BCR security system
-------------		
2XX	---	Not checksummed and Protocol independent
200	---	Reserved
204	variable	Internetwork timestamp field; used to accumulate timestamping information during internet transit
205	variable	Satellite timestamp field; (as above)

3XX	---	Not checksummed and protocol dependent
304	6	Internal TCP timestamp for diagnostics

Padding:

Padding fields are used to ensure that the protocol specific (e.g. TCP) header and the data begin on 32 bit word boundaries.

4.3.3 Transmission Control Block

It is highly likely that any implementation will include shared data structures among parts of the TCP and some asynchronous means of signalling users when letters have been delivered.

One typical data structure is the Transmission Control Block (TCB) which is created and maintained during the lifetime of a given connection. The TCB contains the following information (field sizes and content are notional only and may vary from one implementation to another):

Local connection name: 16 bits

Local socket: variable (fixed for any given TCP)

Local address: variable (fixed for any given TCP)

Local port: 32 bits

Foreign socket: variable

Foreign address: variable

Foreign port: 32 bits

Receive window size in octets: 16 bits

Receive left window edge (next sequence number expected): 32 bits

Send window size in octets: 16 bits

Send left window edge (earliest unacknowledged octet): 32 bits

Next packet sequence number to send: 32 bits

Last sequence number used to update send window (make sure that only the most recent window information is used): 32 bits

Send Buffer Size: 16 bits

Receive Buffer Size: 16 bits

Send Urgent Pointer: 16 bits

Receive Urgent Pointer: 16 bits

Connection state: 4 bits

See figure 4.2-1 for basic state diagram.

CLOSED (0), OPEN (1), SYN-SENT (2), SYN-RECEIVED (3), ESTABLISHED (4), CLOSE-WAIT (5), FIN-WAIT (6), CLOSING (7).

Foreign connection specification (U,U.N,U.T,U.P): 4 bits

U.N is set if the foreign network was not specified in the OPEN command. U.T is set if the foreign TCP was not specified in the OPEN command. U.P is set if the foreign Port was not specified in the OPEN command. U is set if any of U.N, U.T, or U.P are set. U.T implies U.P and U.N implies both U.T and U.P (see section 2.20). U.N, U.T, and U.P are used to remember the specificity of the foreign socket at the initial OPEN so that a RST (reset) will return the foreign socket to its proper state. U is reset (i.e. made false) when a SYN is received, but may be set again on receipt of RST, depending upon U.N, U.T. or U.P. Once in the ESTABLISHED state, U.N, U.T, and U.P can be reset, since the connection will not return to OPEN on receiving RST after it has become ESTABLISHED.

Retransmission timeout: 16 bits

Head of Send buffer queue [buffers SENT from user to TCP, but not packetized]: 16 bits

Tail of Send buffer queue: 16 bits

Pointer to last octet packetized in partially packetized buffer (refers to the buffer at the head of the queue): 16 bits

Head of Send packet queue: 16 bits

Tail of Send packet queue: 16 bits

Head of Packetized buffer queue: 16 bits

Tail of Packetized buffer queue: 16 bits

Head of Retransmit packet queue: 16 bits

Tail of Retransmit packet queue: 16 bits

Head of Receive buffer queue [queue of buffers given by user to RECEIVE letters, but unfilled]: 16 bits

Tail of Receive buffer queue: 16 bits

Head of Receive packet queue: 16 bits

Tail of receive packet queue: 16 bits

Pointer to last octet filled in receive buffer: 16 bits

Pointer to next octet to read from partly emptied packet: 16 bits

> Note: The above two pointers refer to the head of the receive buffer and receive packet queues respectively.

Forward TCB pointer: 16 bits

Backward TCB pointer: 16 bits

4.4 Structure of the TCP

4.4.1 Introduction

Any particular TCP could be viewed in a number of ways. It could be implemented as an independent process, servicing many user processes. It could be viewed as a set of re-entrant library routines which share a common interface to the local PSN, and common buffer storage. It could even be viewed as a set of processes, some handling the user, some the input of packets from the net, and some the output of packets to the net.

We offer one conceptual framework in which to view the various algorithms that make up the TCP design. In our view, the TCP is written in two parts, an interrupt or signal driven part (consisting of five processes), and a reentrant library of subroutines or system calls which interface the user process to the TCP. The subroutines communicate with the interrupt part through shared data structures (TCBs, shared buffer queues etc.). The five processes are the Output Packet Handler which sends packets to the packet switch; the Packetizer which formats letters into internet packets; the Input Packet Handler which processes incoming packets; the Reassembler which builds letters for users; and the Retransmitter which retransmits unacknowledged packets.

> *NOTA BENE:* This model is purely conceptual and not recommended for any conventional operating system with process switch times on the order of 1 ms. Examples of such systems are: Multics, TENEX, UNIX, and ELF.

As an example, we can consider what happens when a user executes a SEND call to the TCP service routines. The buffer to be sent is placed on a send buffer queue associated with the user's TCB. A “Packetizer” process is awakened to create one or more output packets from the buffer. The Packetizer attempts to maintain a non-empty queue of output packets so that the output handler will not fall idle waiting for the packetizing operation.

A major implementation issue is whether to use TCP resources or user resources for incoming and outgoing packets. If the former, there is a fairness issue, both among competing connections and between the sending and receiving sides of the TCP.

When a packet is created, it is placed on a FIFO send packet queue associated with its TCB. The Packetizer wakes the Output Packet Handler and then continues to packetize a few more buffers, perhaps, before going to sleep. The Output Packet Handler is awakened either by a “hungry” packet switch or by the Packetizer. The send packet queue can be used as a “work queue” for the Output Packet Handler. After a packet has been sent, but usually before an ACK is returned, the Output Packet Handler moves the packet to a retransmission queue associated with each TCB.

Retransmission timeouts can refer to specific packets, or the retransmission queue can be periodically searched for the timed-out packets. If an ACK is received, the retransmission entry can be removed from the retransmit queue. The send packet queue contains only packets waiting to be sent for the first time.

Simultaneous reading and writing of the TCB queue pointers must be inhibited through some sort of semaphore or lockout mechanism. When the Packetizer wants to serve the next send buffer queue, it must lock out all other access to the queue, remove the head of the queue (assuming of course that there are enough buffers for packetization), advance the head of the queue, and then unlock access to the queue.

Incoming packets are examined by the Input Packet Handler. Here they are checked for valid connection sockets and acknowledgements are processed, causing packets to be removed, possibly, from the retransmit packet queue, as needed.

Packets which should be reassembled into buffers and sent to users are queued by the Input Packet Handler, on the receive packet queue, for processing by the reassembly process. The Reassembler looks at its FIFO work queue and tries to move packets into user buffers which are queued up in an input buffer queue on each TCB. If a packet has arrived out of order, it can be queued for processing in the correct sequence. Each time a packet is moved into a user buffer, the left window edge of the receiving TCB is moved to the right so that outgoing packets can carry the correct ACK information. If the send buffer queue for the connection in question is empty, then the Reassembler creates a packet to carry the ACK.

As packets are moved into buffers and there are filled, the buffers are dequeued from the receive buffer queue and passed to the user. The Reassembler can also be awakened by the RECEIVE user call should it have a non-empty receive packet queue with an empty receive buffer queue.

4.4.2 Input Packet Handler

The Input Packet Handler is awakened when a packet arrives from the network. It first verifies that the packet is for an existing TCB (i.e. the local and foreign socket numbers are matched with those of existing TCBs). If this fails, a “reset” message is constructed and sent to the point of origin.

The Input Packet Handler looks out for control or error information and acts appropriately. As an example, if the incoming packet is a RST (reset) request, and is “believable”, then the input packet handler clears out the related TCB, empties the associated send and receive packet queues, and prepares error returns for outstanding user SEND(s) and RECEIVE(s) on the reset TCB. The TCB is marked unused and returned to storage. If the RST refers to an unknown connection, it is ignored.

Any ACKs contained in incoming packets are used to update the send left window edge, and to remove the ACKed packets from the TCB retransmit packet queue. If the packet being removed was the end of a user buffer, then the buffer must be dequeued from the packetized buffer queue, and the user informed.

The packet sequence number, the current receive window size, and the receive left window edge determine whether the packet lies within the window or outside of it.

Let W = window size
S = size of sequence number space
L = left window edge
R = L + W = right window edge
x = sequence number to be tested

For any sequence number, x, if

$$0 <= (x – L) \bmod S < (R – L) \bmod S = W \quad (4.4\text{-}1)$$

then x is within the window.

A packet should be rejected only if all of it lies outside the window. This is easily tested by letting x be, first the packet sequence number, and then the sum of packet sequence number and packet length, less one in equation 4.4-1 above.

The other case to be checked occurs when the packet has both head and tail outside of the receive window, but includes the window.

Let PL = packet length

L,R are as before

H = first sequence number in packet

T = H + PL – 1 = last sequence number in packet

For any packet ranging over sequence numbers [H,T], if

$$0 <= L – H < PL$$

and

$$0 <= R – H < PL \quad (4.4\text{-}2)$$

then the packet includes the receive window.

If the packet length is zero (e.g., an ACK packet), tests should be performed as if the packet length were one to accommodate the case when the receive window is zero.

If the packet lies outside the window, and there are no packets waiting to be sent, then the Input Packet Handler should construct an ACK of the current receive left window edge and queue it for output on the send packet queue, and signal the Output Packet Handler. Successfully received packets are placed on the receive packet queue in the appropriate sequence order, and the Reassembler is signalled.

The packet window check can not be made if the associated TCB has not received a SYN, so care must be taken to check for control and TCB state before doing the window check.

4.4.3 Reassembler

It is possible that fragmentation of segments may be removed from responsibility of TCP and placed at the gateway level only. That decision has not been made as yet so we include the following discussion of fragment reassembly.

The Reassembler process is activated by both the Input Packet Handler and the RECEIVE user command. When the Reassembler is awakened it looks at the receive packet queue for the associated TCB. If there are some packets there, then it sees whether the receive buffer queue is empty. If it is, then the Reassembler gives up on this TCB and goes back to sleep; otherwise, if the first packet matches the left window edge, then the packet can be moved into the user's buffer. The Reassembler keeps transferring packets into the user's buffer until the packet is empty or the buffer is full. Note that a buffer may be partly filled and then a sequence "hole" be encountered in the receive packet queue. The Reassembler must mark progress so that the buffer can be filled up starting at the right place when the "hole" is filled. Similarly a packet might be only partially emptied when a buffer is filled, so progress in the packet must be marked.

If a letter was successfully transferred to a user buffer, then the Reassembler signals the user that a letter has arrived and dequeues the buffer associated with it from the TCB receive buffer queue. If the buffer is filled, then the user is signaled and the buffer dequeued as before. The event code indicates whether the buffer contains all or part of a letter, as described in section 2.4.

> Of course, the sequence number processing is adjusted to take into account the EOL as indicated in section 4.2.6.

In every case, when a packet is delivered to a buffer, the receive left window edge is updated, and the Packetizer is signaled. If the send packet queue is empty, then the Reassembler must create a packet to carry the ACK and place it on the send packet queue.

Reassembly of incoming packets containing both the beginning and end-of-segment marks (BOS, EOS; see section 4.3.2) is straightforward. The packet checksum is intact in the packet header and can be used to validate the end-to-end correctness of the data.

Arriving packets with only one or neither bit set are fragments created at a gateway. The intent behind the TCP design is to preserve the end-to-end nature of the checksum and acknowledgement procedure, even in the presence of fragmentation. To achieve this goal, fragments must be reassembled into segments and checksummed. This means, in particular, that the original segment header must be reconstructed.

Gateway fragmentation is straightforward. For instance, a packet consisting of sequence numbers 100–599 can be fragmented into two packets of 250 octets each (including control). The gateway uses figure 4.1-3 to determine which sequence-consuming control flags to set in each fragment header. In the worst case, suppose both sequence-bearing control bits are set (i.e., SYN and FIN), leaving 498 octets of data. A gateway could produce two fragments, the first beginning with sequence number 100 and including SYN, and up to and including data sequence 349. BOS would be set, along with ACK and the window field. The checksum field would be zero.

The second packet would contain data sequences 350–598 and control FIN, as well as EOS, and a checksum (for the original segment—it is not recomputed). The ACK and window fields are duplicates of those in the first fragment.

If EOL is present in the original packet, it is carried only in the last fragment produced. Note that a segment can be divided into more than two fragments, and that a fragment can also be divided. The BOS bit stays with the first fragment, even if that fragment is subdivided later. The EOS and EOL bits stay with the last fragment. Intermediate fragments may not carry any of BOS, EOS, or EOL.

During reassembly of a segment, it may happen that fragments arrive with sequence number extents which overlap (due to alternate gateway routing and different fragmentation). This makes the job of reassembling fragments more difficult, but not impossible. Although it is not part of the current specification, it may be useful for gateways to produce a fragment checksum in addition to passing the segment checksum intact. In this way, a bad fragment is less likely to mess up reassembly of a segment.

Gateway fragmentation rules may require modification or augmentation to deal with option fields in packet headers. It is generally true that options tend to stay with the fragment marked “BOS”.

The rules of packet retransmission require that retransmitted packets contain the latest ACK and window information available. This means that a duplicate of a segment, if fragmented, may have a different checksum than earlier copies. To assure that segment reassembly is not frustrated by this effect, the ACK and window information used to validate the reassembled checksum should be taken from the packet containing the checksum (i.e., the fragment marked “EOS”).

4.4.4 Packetizer

The Packetizer process gets work from both the Input Packet Handler and the SEND user call. The signal from the SEND user call indicates that there is something new to send, while the one from the Input Packet Handler indicates that more TCP buffers may be available from delivered packets.

When the Packetizer is awakened it looks at the send buffer queue for the associated TCB. If there is a new or partial letter awaiting packetization, it tries to packetize the letter, TCP buffers and window permitting. For every packet produced it signals the Output Packet Handler (to prevent deadlock in a time sliced scheduling scheme). If a ‘run to completion’ scheme is used then one signal only need be produced, the first time a packet is produced since awakening. If packetization is not possible the Packetizer goes to sleep.

If a partial buffer was transferred then the Packetizer must mark progress in the send buffer queue. Completely packetized buffers are dequeued from the send buffer queue and placed on a packetized buffer queue, when an ACK for the last bit is received the send buffer is returned to the user.

A SYN must logically precede the first data transmitted on a connection. When the Packetizer packetizes a letter it must see whether it is the first piece of data being sent on the connection, in which case it must include the SYN bit, or cause a SYN packet to be sent before the data packet. Some implementations may choose not to send data with SYN, and some may choose to discard any data received with SYN.

4.4.5 Output Packet Handler

When activated by the Packetizer, or the Input Packet Handler, or some of the user call routines, the Output Packet Handler attempts to transmit packets to the network (this may involve going through some other network interface program). Transmitted packets are dequeued from the send packet queue and put on the retransmit queue along with the time when they should be retransmitted.

All data packets that are (re)transmitted have the latest receive left window edge in the ACK field. Some error messages may set the ACK field to refer to a received packet's sequence number.

4.4.6 Retransmitter

This process can either be viewed as a separate process, or as part of the Output Packet Handler. Its implementation can vary; it could either perform its function by being awakened at regular intervals, or when the retransmission time occurs for every packet put on the retransmit queue. In the first case the retransmit queue for each TCB is examined to see if there is anything to retransmit. If there is, a packet is placed on the send packet queue of the corresponding TCB. The Output Packet Handler is also signaled.

A "demon" process monitors all user send buffers and retransmittable control messages sent on each connection, but not yet acknowledged. If the global retransmission timeout is exceeded for any of these, the user is notified and the connection aborted.

Note that, since retransmitted packets carry the latest receive left window edge and acknowledgement information, the checksum may require recomputation.

4.5 Buffer and Window Allocation

4.5.1 Introduction

The TCP manages buffer and window allocation on connections for two main purposes: equitably sharing limited TCP buffer space among all connections (multiplexing function), and limiting attempts to send packets, so that the receiver is not swamped (flow control function). For further details on the operation and advantages of the window mechanism see [CK74].

Good allocation schemes are one of the hardest problems of TCP design, and much experimentation must be done to develop efficient end effective algorithms. Hence the following suggestions are merely initial thoughts. Different implementations are encouraged with the hope that results can be compared and better schemes developed. For comments on some allocation policies and other factors effecting communication performance see [GRP77, Sunshine77c].

4.5.2 The SEND Side

The window is determined by the receiver. Currently the sender has no control over the send window size, and never transmits beyond the right window edge. An exception is made in the case of a zero send window when it is necessary to periodically retransmit to poll for a window opening ACK.

Buffers must be allocated for outgoing packets from a TCP buffer pool. The sending TCP may not be willing to allocate a full receiver's window's worth of buffers, so buffer space for a connection may be less than what the window would permit. No deadlocks are possible even if there is insufficient buffer or window space for one letter, since the receiver will ACK parts of letters as they are put into its user's buffer, thus advancing the window and freeing buffers for the remainder of the letter.

It is not mandatory that the TCP buffer outgoing packets until acknowledgements for them are received, since it is possible to reconstruct them from the actual buffers sent by the user. However, for purposes of retransmission and processing efficiency it is very convenient to do.

4.5.3 The RECEIVE Side

At the receiving side there are two requirements for buffering:

(1) Rate Discrepancy:
If the sender produces data much faster or much slower than the receiver consumes it, little buffering is needed to maintain the receiver at near maximum rate of operation. Simple queueing analysis indicates that when the production and consumption (arrival and service) rates are similar in magnitude, more buffering is needed to reduce the effect of stochastic or bursty arrivals and to keep the receiver busy.

(2) Disorderly Arrivals:
When packets arrive out of order, they must be buffered until the missing packets arrive so that packets (or letters) are delivered in sequence. We do not advocate the philosophy

that they be discarded, unless they have to be, lest a poor effective bandwidth may be observed. Path length, packet size, traffic level, routing, timeouts, window size, and other factors may affect the degree to which packets arrive out of order.

The considerations for choosing an appropriate window are as follows:

Suppose that the receiver knows the sender’s retransmission timeout, K. This is usually close to the round trip transmission time. Suppose also that the receiver’s acceptance rate is U bits/sec, and the window size is W bits. Ignoring line errors and other traffic, the sender transmits at a rate between W/K and the maximum line rate. The sender is permitted by the protocol to send at most a window’s worth of data each timeout period.

If W/K is greater than U the difference must be retransmissions, which are undesirable, so the window should be reduced to W', such that W'/K is approximately equal to U. This may mean that the entire bandwidth of the transmission channel is not being used, but it is the fastest rate at which the receiver is accepting data, and the line capacity is free for other users. This is exactly the same as the case where the rates of the sender and receiver are almost equal, and so more buffering is needed. Thus we see that line utilization and retransmissions can be traded off against buffering.

If the receiver does not accept data fast enough (by not performing sufficient RECEIVEs) the sender may continue retransmitting since the unaccepted data will not be ACKed. In this case the receiver should reduce the window size to “throttle” the sender and inhibit useless retransmissions.

Limited experimentation, simulation, and analysis with buffering and window allocation suggests that the receiver should set aside buffer space to accommodate any window sent to the sender. Any attempts at optimistically setting a large window with inadequate buffer appears to lead to poor bandwidth owing to occasional (or frequent) discarding of arriving packets for which no buffers are available. Theoretically, selection of the ratio of window size granted to buffer store reserved should be equivalent to the selection of a buffer size for a statistical multiplexor.

If the user at the receiving side is not accepting data, the window should be reduced to zero. In particular, if all TCP incoming packet buffers for a connection are filled with received packets, the window must go to zero to prevent retransmissions until the user accepts some packets.

Setting the receive window to zero can have some interesting side effects. In particular, it is not enough to merely send an empty ACK packet with the newly non-zero window, when the window is re-opened. If the ACK is lost, the other TCP may never transmit again. (ACKs cannot be retransmitted since they cannot, themselves, be ACKed as we would not know when to stop retransmitting). A TCP should therefore continue to send data (retransmissions) even when faced with a zero window, albeit at a low rate. Design and discussion of several mechanisms have led to the belief that this is the simplest and least costly solution to the zero window problem.

BIBLIOGRAPHY

Notes of Working Group 6.1 of the International Federation of Information Processing, [also known as the International Network Working Group or INWG], are available through its chairman,

Mr. Derek L. A. Barber,
Project EIN,
National Physical Laboratory,
Teddington, Middlesex, England.

Readers interested in a rich source of reference to the literature on resource sharing networks are urged to consult NBS special publication 384:

Helen M. Wood, Shirley Ward Watkins, Ira W. Cotton
Annotated Bibliography of the Literature on Resource Sharing Networks
National Bureau of Standards Special Publication 384
Institute for Computer Sciences and Technology
Revised 1976

available from

Superintendent of Documents
U.S. Government Printing Office
Washington, D.C. 20402
order by SD Catalog No. C13.10.384/rev
Stock No. 003-003-01670-5, $2.45

Special collections of papers on related subjects may be found in:

1. Wesley Chu (Ed.), Advances in Computer Communications, Artech House, 1976 (revised).

2. Robert Blanc and Ira Cotton (Eds.), Computer Networking, IEEE Press, New York, 1976.

AR76

D. Aitwyver, A. M. Rybczynski, “Datapac Subscriber Interfaces,” Proceedings of ICCC76, p. 143–149.

Barber76

Derek L. A. Barber, “A European Informatics Network,” Proceedings of ICCC76, p. 44–50.

BBN1822

Bolt Beranek and Newman, “Specification for the Interconnection of a Host and an IMP,” BBN technical Report 1822, January 1976 (Revised).

Belsnes74

Dag Belsnes, “Note on Single Message Communication,” INWG Protocol Note 3, IFIP Working Group 6.1, September 1974.

Belsnes74A

D. Belsnes, “Flow control in packet switching networks,” INWG Note 63, IFIP Working Group 6.1, October 1974.

BLSS

Jerry D. Burchfiel, Eisie M. Leavitt, Sonya Shapiro, Theodore R. Strollo, TENEX USERS’ GUIDE, Bolt Beranek and Newman, Inc., Cambridge, MA, January 1975.

BLW74

Richard Binder, Wai Sum Lai, Morris Wilson, “The Alohanet Menehuna—Version II,” The Aloha System Technical Report B74-6, University of Hawaii, September 1974.

BPT76

Jerry D. Burchfiel, William W. Plummer, Raymond S. Tomlinson, “Proposed Revision to the TCP,” INWG Protocol Note 43, IFIP W.G. 6.1, September 1976.

Bright75

Roy D. Bright, “Experimental Packet Switch Project of the UK Post Office,” in Computer Communication Networks, Grimsdale and Kuo, Editors, NATO Advanced Studies Institute Series, 15-4, Noordhoff International, Leyden, Netherlands, 1975, pp. 435–444.

BTB

Jerry D. Burchfiel, Raymond S. Tomlinson, Michael Beeler, “Functions and Structure of a Packet Radio Station,” AFIPS Proceedings, volume 44, 1975, National Computer

Conference, (Anaheim, CA, May 19–22, 1975), AFIPS Press, Montvale, NJ, 1975, p. 245–251.

BW72

Robert Bressler and David C. Walden, “A proposed Experiment with a Message Switching Protocol,” ARPA RFC 333, NIC 9926, Augmentation Research Center, Stanford Research Institute, Menlo Park, CA, May 1972.

Cashin76

P. M. Cashin, “Datapac Network Protocols,” Proceedings of ICCC76, p. 150.

CCC70

Stephen Carr, Stephen D. Crocker and Vinton G. Cerf, “Host-Host Communication Protocol in the ARPA Network,” AFIPS Proceedings, 1970 Spring Joint Computer Conference, volume 36, (Atlantic City, NJ, May 5–7, 1970), AFIPS Press, Montvale, NJ, 1970, p. 589–598.

CDS74

Vinton G. Cerf, Yogen K. Dalal, Carl Sunshine, “Specification of Internet Transmission Control Program,” INWG General Note 72, IFIP Working Group 6.1, December 1974.

CEHKKS77

Vinton G. Cerf, Stephen Edge, Andrew Hinchley, Richard Karp, Peter T. Kirstein, Paal Spilling, “Final Report of the Internetwork TCP Project,” to appear.

Cerf74

Vinton G. Cerf, “An Assessment of ARPANET Protocols,” The Second Jerusalem Conference on Information Technology, (Jerusalem, Israel, July 29–August 1, 1974), p. 653–664 (also, INWG General Note 70, IFIP W.G. 6.1, July 1974 and in Network Systems and Software Infotech State of the Art Report 24, Infotech Information, Ltd., Nicholson House, Maidenhead, Berkshire, England, 1975).

Cerf76

Vinton G. Cerf, “SCCU/MCCU Characteristics for AUTODIN II,” Digital Systems Laboratory Technical Note 92, Stanford University, July 1976.

Cerf76a

Vinton G. Cerf, “TCP Resynchronization,” Digital Systems Lab Technical Note 79, Stanford University, January 1976.

Cerf76b

Vinton G. Cerf, “ARPA Internetwork Protocols Projects, Status Report, for the period November 15, 1975–February 15, 1976,” Digital Systems Laboratory Technical Note 83, Stanford University, February 1976.

Cerf77

Vinton G. Cerf, “Specification of Internet Transmission Control Program—TCP (Version 2),” March 1977.

CGN76

W. W. Clipsham, F. E. Glave, M. L. Narraway, “Datapac Network Overview,” Proceedings of ICCC76, p. 131–136.

CHMP72

Stephen D. Crocker, John F. Heafner, Robert Metcalfe and Jonathan B. Postel, “Function-Oriented Protocols for the ARPA Computer Network,” AFIPS Proceedings, 1972 Spring Joint Computer Conference, volume 40, (Atlantic City, NJ, May 16–18, 1972), AFIPS Press, Montvale, NJ, 1972, p. 271–279.

CK74

Vinton G. Cerf and Robert E. Kahn, “A Protocol for Packet Network Intercommunication,” IEEE Transactions on Communications, volume COM-22, No. 5, May 1974, p. 637–648. (An early version of this paper appeared as INWG General Note 39, IFIP Working Group 6.1, September 1973).

CMSZ75

Vinton G. Cerf, Alexander McKenzie, Roger Scantlebury, Hubert Zimmermann, “Proposal for an Internetwork End to End Protocol,” INWG General Note 96, IFIP W.G. 6.1, September 1975 (also in ACM SIGCOMM Quarterly Review Vol. 6, No. 1, Jan. 1976) p. 63–89.

CS74

Vinton G. Cerf and Carl Sunshine, “Protocols and Gateways for the Interconnection of Packet Switching Networks,” The Aloha System Technical Report CN 74-22, Proceedings of the Seventh Hawaii International Conference on Systems Sciences, University of Hawaii, (Honolulu, Hawaii, January 8–10, 1974).

Dalal74

Yogen K. Dalal, “More on Selecting Sequence Numbers,” INWG Protocol Note 4, IFIP Working Group 6.1, August 1974. Also in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, (Santa Monica, CA, March 24–25, 1975), and ACM Operating Systems Review, Volume 9, Number 3, July 1975, Association for Computer Machinery, New York, 1975.

Dalal75

Yogen K. Dalal, “Establishing a Connection,” INWG Protocol Note 14, IFIP Working Group 6.1, March 1975.

Danthine75

Andre Danthine and E. Eschenauer, “Influence on the Node Behavior of the Node-to-Node Protocol,” Proceedings, Fourth Data Comm. p. 7-1 to 7-8.

Davies71

Donald W. Davies, “The Control of Congestion in Packet Switching Networks,” Peter E. Jackson, proceedings, ACM/IEEE Second Symposium on Problems in the Optimization of Data Communication Systems, (Palo Alto, CA, October 20–22, 1971), IEEE (at -71C59-C, p. 46–49.

DCA75

System Performance Specification for Autodin II, Phase 1, Defense Communications Agency, Defense Communication Engineering Center, November 1975.

DCA76

Elizabeth Feinler and Jonathan B. Postel, ARPANET Protocol Handbook, Network Information Center, Stanford Research Institute, Menlo Park, CA, April 1976.

DDLPR76

A. Danet, R. Despres, A. LeRest, G. Pichon, S. Ritzenthaler, “The French Public Packet Switching Service: the TRANSPAC Network,” Proceedings of ICCC76, p. 251–260.

DHMMW74

W. Crowther, F. Heart, A. McKenzie, J. McQuillan, D. Walden, Network Design Issues, Bolt Beranek and Newman, Inc. Technical Report No. 2918, November 1974 (also, INWG General Note 64, IFIP Working Group 6.1, October 1974; ARPA Network Measurement Note 26, Network Measurement Group, October 1974).

FG75

Stanley C. Fralick and James C. Garrett, “Technological Considerations for Packet Radio Networks,” AFIPS Proceedings, volume 44, 1975, National Computer Conference, (Anaheim, CA, May 19–22,1975), AFIPS Press, Montvale, NJ, 1975, p. 233–243.

FGS75

Howard Frank, Israel Gitman, Richard van Slyke, “Packet Radio System—Network Considerations,” AFIPS Proceedings, volume 44, 1975, National Computer Conference, (Anaheim, CA, May 19–22, 1975), AFIPS Press, Montvale, NJ, 1975, p. 217–231.

GS75

M. Gien and R. Scantlebury, “Interconnection of Packet Switched Networks, Theory and Practice,” proceedings of European Computing Conference on Communication Networks, EUROCOMP, pp. 441–260, Brunel University, Online Conferences Ltd., Uxbridge, England, September 1975.

GRP77

L. Garlick, R. Rom, and J. Postel, “Reliable Host-to-Host Protocols: Problems and Techniques,” proceedings of the Fifth Data Communications Symposium, (Snowbird, Utah), ACM and IEEE, pp. 4.58–4.65, September 1977.

HKOCW70

Frank E. Heart, Robert E. Kahn, S. M. Ornstein, William R. Crowther, and David C. Walden, “The Interface Message Processor for the ARPA Computer Network,” AFIPS Proceedings, 1970 Spring Joint Computer Conference, volume 36, (Atlantic City, NJ, May 5–7, 1970), AFIPS Press, Montvale, NJ, 1970, p. 551–567.

Kahn73

Robert E. Kahn, “Status and Plans for the ARPANET,” Martin Greenberger, Julius Aronofsky, James L. McKenney, William F. Massy, Networks for Research and Education: Sharing Computer and Information Resources Nationwide, MIT Press, Cambridge, MA, 1973, p. 51–54.

Kahn75

Robert E. Kahn, “The Organization of Computer Resources into a Packet Radio Network,” AFIPS Proceedings, volume 44, 1975, National Computer Conference, (Anaheim, CA, May 19–22, 1975), AFIPS Press, Montvale, NJ, 1975, p. 179–186.

Karp73

Peggy M. Karp, “Origin, Development and Current Status of the ARPANET,” COMPCON73—Seventh Annual IEEE Computer Society International Conference, Digest of Papers, ‘Computing Networks from Mini’s to Maxi’s—Are They for Real?’ (San Francisco, CA, February 27–28, March 1, 1973), Institute of Electrical and Electronic Engineers, Inc., New York, 1973, p. 49–52.

KC71

Robert E. Kahn, William R. Crowther, “Flow Control in a Resource-Sharing Computer Network,” Peter E. Jackson, Proceedings, ACM/IEEE Second Symposium on Problems in the Optimization of Data Communication Systems, (Palo Alto, CA, October 20–22, 1971), 1971, IEEE (AT-71C59-C, p. 108–116.

Kleinrock74

Leonard Kleinrock and William E. Naylor, “On Measured Behavior of the ARPA Network, AFIPS Proceedings, National Computer Conference, Volume 43, (Chicago, IL, May 6–10, 1974), AFIPS Press, Montvale, NJ, p. 767–780.

Kleinrock75

Leonard Kleinrock and Holger Opderbeck, “Throughput in the ARPANET—Protocols and Measurement,” Proceedings, Fourth Data Communications Symposium, (Quebec City, Canada, 7–9 October 1975), p. 6-1 to 6-11.

Kleinrock76

Leonard Kleinrock, William E. Naylor, Holger Opderbeck, “A Study of Line Overhead in the ARPANET,” Communications of the ACM, Vol. 19, No. 1, p. 3.

LGK75

David Lloyd, Martine Galland, Peter T. Kirstein, “Aim and Objectives of Internetwork Experiments,” INWG Experiments Note 3, IFIP Working Group 6.1, February 1975.

Mathis76

James E. Mathis, “Single-Connection TCP Specification,” Digital Systems Laboratory Technical Note 75, Stanford University, January 25, 1976.

MB76

Robert M. Metcalfe and David R. Boggs, “Ethernet: Distributed Packet Switching for Local Computer Networks,” Communications of the ACM, Volume 19, No. 7, July 1976, p. 395–404.

MCCW72

John M. McQuillan, William R. Crowther, Bernard P. Cosell, David C. Walden, Frank E. Heart, “Improvements in the Design and Performance of the ARPA Network,” AFIPS Proceedings, Fall Joint Computer Conference, Volume 41, p. 741–754.

McKenzie73

A. McKenzie, “Host-Host Protocol for the ARPANET,” NIC 8246, Stanford Research Institute [also in ARPANET Protocols Notebook NIC 7104].

McKenzie74a

Alexander McKenzie, “Some Computer Network Interconnection Issues,” AFIPS Proceedings, National Computer Conference, Volume 43, (Chicago, IL, May 6–10, 1974), AFIPS Press, Montvale, NJ, p. 857–859.

McKenzie74b

Alexander McKenzie, “Internetwork Host-to-Host Protocol,” INWG General Note 74, IFIP Working Group 6.1, December, 1974.

McQuillan75

John M. McQuillan, “The Evolution of Message Processing Techniques in the ARPA Network,” Network Systems and Software, Infotech State of the Art Report 24, Infotech Information, Ltd., Nicholson House, Maidenhead, Berkshire, England, 1975.

MPT74

Eric R. Mader, William R. Plummer, Raymond S. Tomlinson, “A Protocol Experiment,” INWG Experiment Note 1, IFIP Working Group 6.1, August 1974.

NAC73

Network Analysis Corporation, ARPANET: Design, Operation, Management and Performance, Network Analysis Corporation, Glen Cove, NY, April 1973.

OK74

Holger Opderbeck and Leonard Kleinrock, “The Influence of Control Procedures on the Performance of Packet-Switched Networks,” National Telecommunications Conference, San Diego, California, December 1974.

PGR76a

Jonathan B. Postel, Larry L. Garlick, Raphael Rom, Transmission Control Protocol Specification, Augmentation Research Center, Stanford Research Institute, Menlo Park, CA, 15 July 1976.

PGR76b

Jonathan B. Postel, Larry L. Garlick, Raphael Rom, Terminal-to-Host Protocol Specification, Augmentation Research Center, Stanford Research Institute, Menlo Park, CA, 15 July 1976.

Postel72

J. Postel, “Official Initial Connection Protocol,” Current Network Protocols, Network Information Center, Stanford Research Institute, Menlo Park, California, January 1972 (NIC 7101).

Postel77

J. Postel, “Assigned Numbers,” RFC 739, NIC 42341, USC—Information Sciences Institute, Marina del Rey, California, 11 November 77.

Pouzin73

Louis Pouzin, “Interconnection of Packet Switching Networks,” INWG General Note 42, IFIP Working Group 6.1, October 1973.

Pouzin73a

Louis Pouzin, “Presentation and major design aspects of the CYCLADES Computer Network,” Data Networks: Analysis and Design, Third Data Communications Symposium, St. Petersburg, Florida, November 1973, pp. 80–87. Also in: Grimsdale and F. Kuo eds., Computer Communication Networks, NATO Advanced Studies Institute Series, E-4, Noordhoff, Leyden, Netherlands, 1975, pp. 415–434.

Pouzin74a

Louis Pouzin, “A Proposal for Interconnecting Packet Switching Networks,” INWG General Note 60, IFIP W.G. 6.1, March 1974 (also in proceedings of EUROCOMP, Brunel University, May 1974, p. 1023–1036).

Pouzin74b

Louis Pouzin, “Cigale, the Packet Switching Machine on the CYCLADES Computer Network,” Jack L. Rosenfeld, Information Processing 74, proceedings of the IFIP Congress 1974, Computer Hardware and Architecture Volume, (Stockholm, Sweden, August 5–10, 1974), American Elsevier Publishing Co., Inc., New York, 1974, p. 2155–159.

Retz75

David L. Retz, “ELF—A System for Network Access,” 1975 IEEE Intercon Conference Record, (New York, April 8–10, 1975), Institute of Electrical and Electronic Engineers, Inc., New York, 1975, p. 25-2-1 to 25-2-5.

Roberts76

Lawrence G. Roberts, “International Interconnection of Public Packet Networks,” Proceedings, International Conference on Computer Communication, (Toronto, Ontario, Canada, August 1976), p. 239–245.

RW70

Lawrence G. Roberts and Barry D. Wessler, “Computer Network Development to Achieve Resource Sharing,” AFIPS Proceedings, 1970 Spring Joint Computer Conference, volume 36, (Atlantic City, NJ, May 5–7, 1970), AFIPS Press, Montvale, NJ, 1970, p. 543–549.

RW73

Lawrence G. Roberts and Barry D. Wessler, “The ARPA Net,” Norman Abramson and Franklin F. Kuo, Computer-Communication Networks, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

Schantz74

R. Schantz, “Reconnection Protocol”, private communication; available from Schantz at BBN.

SH75

Adrian V. Stokes and Peter L. Higginson, “The Problems of Connecting Hosts into ARPANET,” Proceedings of the European Conference on Communication Networks, September 1975, On-line Conferences, Ltd., Oxbridge, England, p. 25–34.

Sunshine75

Carl Sunshine, “Issues in Communication Protocol Design—Formal Correctness,” INWG Protocol Note 5, IFIP Working Group 6.1, October 1975. Also in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, (Santa Monica, CA, March 24–25, 1975).

Sunshine76a

Carl Sunshine, Interprocess Communication Protocols for Computer Networks, Stanford University (Ph.D. Dissertation), 1976.

Sunshine76b

Carl Sunshine, “Interconnection of Computer Networks,” Computer Networks, Vol. 1, No. 3, January 1977, pp. 175–195.

Sunshine76c

Carl Sunshine, “Efficiency of Interprocess Communication Protocols for Computer Networks,” Transactions of the IEEE on Communications, February 1977, pp. 287–293.

SW71

R. Scantlebury and P. T. Wilkinson, “The Design of a Switching System to allow remote Access to Computer Services by other computers and Terminal Devices,” Second Symposium on Problems in the Optimization of Data Communication Systems Proceedings, Palo Alto, California, October 1971, pp. 160–167.

Tomlinson74

Raymond S. Tomlinson, “Selecting Sequence Numbers,” INWG Protocol Note 2, IFIP Working Group 6.1, August 1974. Also in Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, (Santa Monica, CA, March 24–25, 1975), and ACM Operating Systems Review, Volume 9, Number 3, July 1975, Association for Computer Machinery, New York, 1975.

Tomlinson77

Raymond S. Tomlinson, “Proposal for TCP 3,” ARPANET message number <[BBN-TENEXA]12-Oct-77 11:59.Tomlinson>, October 1977.

Walden72

David C. Walden, “A System for Interprocess Communication in a Resource Sharing Computer Network,” Communications of the ACM, Volume 15, Issue 4, April 1972, p. 221–230.

WR75

D. C. Walden and R. C. Rettberg, “Gateway Design for Computer Network Interconnection,” Proceedings, European Computing Conference on Communication Networks, September 1975, On-line Conferences, Ltd., Oxbridge, England, p. 113–128.

YM76

S. C. K. Young, C. I. McGibbon, “The Control System of the Datapac Network,” Proceedings of ICCC76, p. 137–142.

ZE73

Hubert Zimmermann and Michele Elie, “Proposed Standard Host-Host Protocol for Heterogeneous Computer Networks: Transport Protocol,” INWG General Note 43, IFIP Working Group 6.1, December 1973 (also Institute Recherche d’Informatique et d’Automatique [IRIA] Project CYCLADES report SCM 519).

ZE74

Hubert Zimmermann and Michele Elie, “Transport Protocol Standard Host/Host Protocol for Heterogeneous Computer Networks,” INWG General Note 61, IFIP Working Group 6.1, April 1974 (also IRIA Project CYCLADES Report SCH 519.1).

Zimmermann75

Hubert Zimmermann, “The CYCLADES End to End Protocol,” Proceedings, Fourth Data Communication Symposium, (Quebec City, Canada, October 7–9, 1975), p. 7-21 to 7-26.