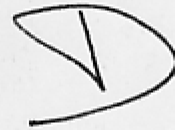
September 1978

IEN: 55
Section: 2.4.2.1



SPECIFICATION OF INTERNETWORK TRANSMISSION CONTROL PROTOCOL

TCP

Version 4

Jonathan B. Postel

September 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 iii

1. INTRODUCTION 1

1.1 History 1
1.2 Scope 2
1.3 Documentation 2
1.4 Interfaces 3
1.5 Operation 3

2. PHILOSOPHY 7

2.1 Related Work 7
2.2 Mechanisms Explained 7
2.3 Functional Specification of Interfaces 10
2.4 Problems Remaining 10
2.5 Lessons Learned 11
2.6 Future Directions 11

3. SPECIFICATION 13

3.1 Formalism Explained 13
3.2 Formal Specification 13
3.3 Internet Header Format 33
3.4 Discussion 38
3.5 Examples & Scenarios 53
3.6 Interfaces 53

4. VERIFICATION 65

5. IMPLEMENTATION 67

5.1 What Not To Leave Out 67
5.2 User Interfaces 67
5.3 Mechanisms 67
5.4 Data Structures 72
5.5 Program Sizes, Performance Data 74
5.6 Test Sequences, Procedures, Exerciser 74
5.7 Parameter Values 75
5.8 Debugging 78

GLOSSARY 79

BIBLIOGRAPHY 85

APPENDICES 99

A. Reconnection Procedure 99

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 four previous TCP specifications as described in the introduction. The present text draws borrows heavily from them.

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

The version 3 specification was influenced by many people, but special mention should be made of 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 were first described by Ray Tomlinson of BBN [Tomlinson77]. The final document benefited from the comments of the following reviewers: Michael Padlipsky, Carl Sunshine, John Day, Gary Grossman, and Ray Tomlinson.

This revised edition of the version 4 specification was influenced by the comments of the following: Vint Cerf, Dick Watson, Carl Sunshine, Danny Cohen, Dave Clark, John Day, Gary Grossman, Jim Mathis, Bill Plummer, Jack Haverty, and the whole TCP Working Group.

September 1978
IEN:55
Section: 2.4.2.1
Replaces: IENs 44, 40, 27, 21, 5

Transmission Control Protocol

Version 4

1. INTRODUCTION

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

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

1.1. History

There have been four 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 fourth [CP78] defined version 3, a refinement of version 2.

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 first internet version (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, segment speech services, diagnostic timestamping, and so on.

Version 2 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 TCP further evolved. The primary changes from version 2 were as follows:

The resynchronization mechanism was eliminated in favor of a quiet period on initialization of the TCP.

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

The interrupt signal was eliminated in favor of an urgent pointer.

A further separation of the internet and TCP specific information in the segment format was achieved.

Version 4 TCP specified here has the following changes:

The TCP now expects to call on a lower level protocol module in the host for certain functions; in the general internetwork case, TCP expects the lower level module to implement the ARPA Internetwork Protocol [Postel78d] or something functionally equivalent to it.

Addressing information necessary to reach a specific TCP implementation is expected to be carried on the lower level protocol.

Fragmentation and reassembly have been eliminated from TCP and made the responsibility of the lower level protocol.

1.2. Scope

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

1.3. Other Documentation

For other documentation, see the items cited in the History Section (1.1) and the items listed in the Bibliography.

1.4. Interfaces

The TCP interfaces on one side to application processes and on the other side to a transmission protocol such as Internetwork Protocol [Postel78d].

The interface between an application process and the TCP is illustrated in reasonable detail. This interface consists of a set of calls much like the calls an operating system provides to application process for manipulating files. For example, there are calls to open and close connections and to send and receive letters on established connections. It is also expected that the TCP can asynchronously communicate with application programs via events.

The interface between TCP and a transmission protocol is essentially unspecified except that it is assumed there is a mechanism whereby the two can pass information to each other such as events.

1.5. Operation

Several basic assumptions are made about process-to-process communication; these are listed here without further justification. The interested reader is referred to [CK74, Tomlinson74, Belsnes74, Dalal74, Dalal75, Sunshine76a, 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

00001010-00010110-0000000000000001-0000000000000011

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.

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 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).

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 TCP segment in an INTERNET SEGMENT. Each internet segment 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 segment is destined for a TCP in a foreign PSN, the local packet is routed to a gateway which connects the originating PSN with an intermediate PSN or with the destination PSN. Routing of internetwork segment to the gateway may be the responsibility of the source TCP or the local PSN, depending upon the PSN services available.

The model of TCP operation is that there is a basic gateway (or internet protocol module) associated with each TCP which provides an interface to the local network. This basic gateway performs routing and segment 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 internet segment is "unwrapped" from its local packet format and examined to determine through which network the internet segment should travel next. The internet segment 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 an internet segment into smaller FRAGMENTS if this is necessary for transmission through the next network. To do this, the gateway produces a set of internet packets, each carrying a fragment. Fragments may be broken into smaller ones at intermediate gateways. The internet packet format is designed so that the destination gateway can reassemble fragments into internet segments. Segments, of course, can be reassembled into letters by the destination TCP.

The TCP is responsible for regulating the flow of TCP segments 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 segments and for detecting duplicates. A consequence of this error detection and 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.

2. PHILOSOPHY

2.1. Related Work

Some work that at one point was closely related is the definition of a Transport Protocol by the Internation Network Working Group [CMSZ78].

2.2. Mechanisms Explained

The key idea of TCP is that *processes* exchange *letters* via *connections.* TCP utilizes many mechanisms to provide this service.

Processes are supported by the host operating system.

Letters are supported by the reliable transmission of TCP segments containing beginning and end of letter flags. Transmission is made reliable via the use of sequence numbers and acknowledgments.

Connections are supported via procedures to establish and clear connection. These procedures utilize the synchronize (SYN) and finis (FIN) control flags and involve a three-way hand shake. Connections are identified by pairs of addresses that include port identifiers. Such address are called sockets to stress that fact.

Letters

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 segment. The end of a letter is marked by the appearance of an EOL control flag in a segment. 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; 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.

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 acknowledgment, and error control services.

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, echoer, and sink (the last 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.

Sockets and Addressing

We have borrowed the term SOCKET from the ARPANET terminology [CCC70, FP78]. In general, a socket is an internetwork ADDRESS including 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 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 below. To simplify passing a connection within a single TCP "invisible" switches are be allowed.

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 segment and a waiting Transmission Control Block (TCB) created by a user OPEN, SEND, URGENT, or RECEIVE command. 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.

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 segments 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 segments which match with existing TCB's. Generally, both the local socket field of the TCB and the destination socket field of the arriving segment must match, and the foreign field of the TCB and the source socket field of the arriving segment must match.

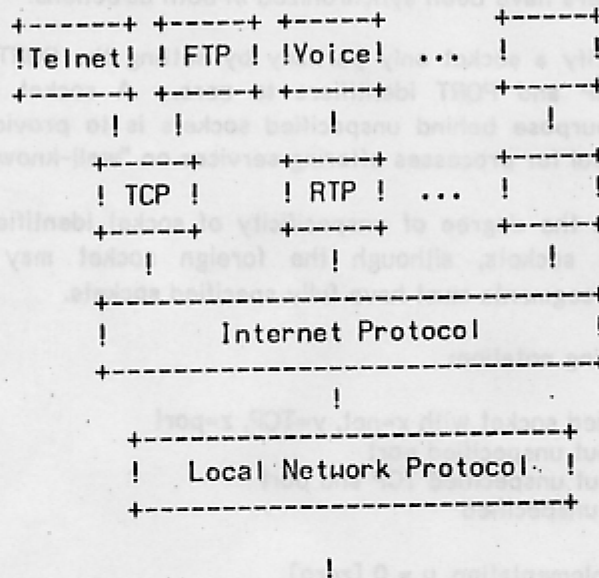
	TCB		segment	
CASE	local	foreign	source	dest
(a)	a.b.c	e.f.g	e.f.g	a.b.c
(b)	a.b.c	e.f.u	e.f.g	a.b.c
(c)	a.b.c	e.u.u	e.f.g	a.b.c
(d)	a.b.c	u.u.u	e.f.g	a.b.c

~~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. Functional Specification of Interfaces

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



Protocol Relationships

Figure 1.

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 [FP78] and FTP [FP78] to the TCP.

2.4. Problems Remaining

Major Points

A real formal specification is needed.

The protocol must be verified.

Technical Issues

2.5. Lessons Learned

???

2.6. Future Directions

???

3. SPECIFICATION

3.1. Formalisms Explained

The material in the following section by no means qualifies as a formal specification, but it is a close as we have on hand. First is a state diagram of the connection opening and closing sequences, followed by a description of the action to be taken when is state x and event y occurs for the expected combinations of x and y.

3.2. Formal Specification

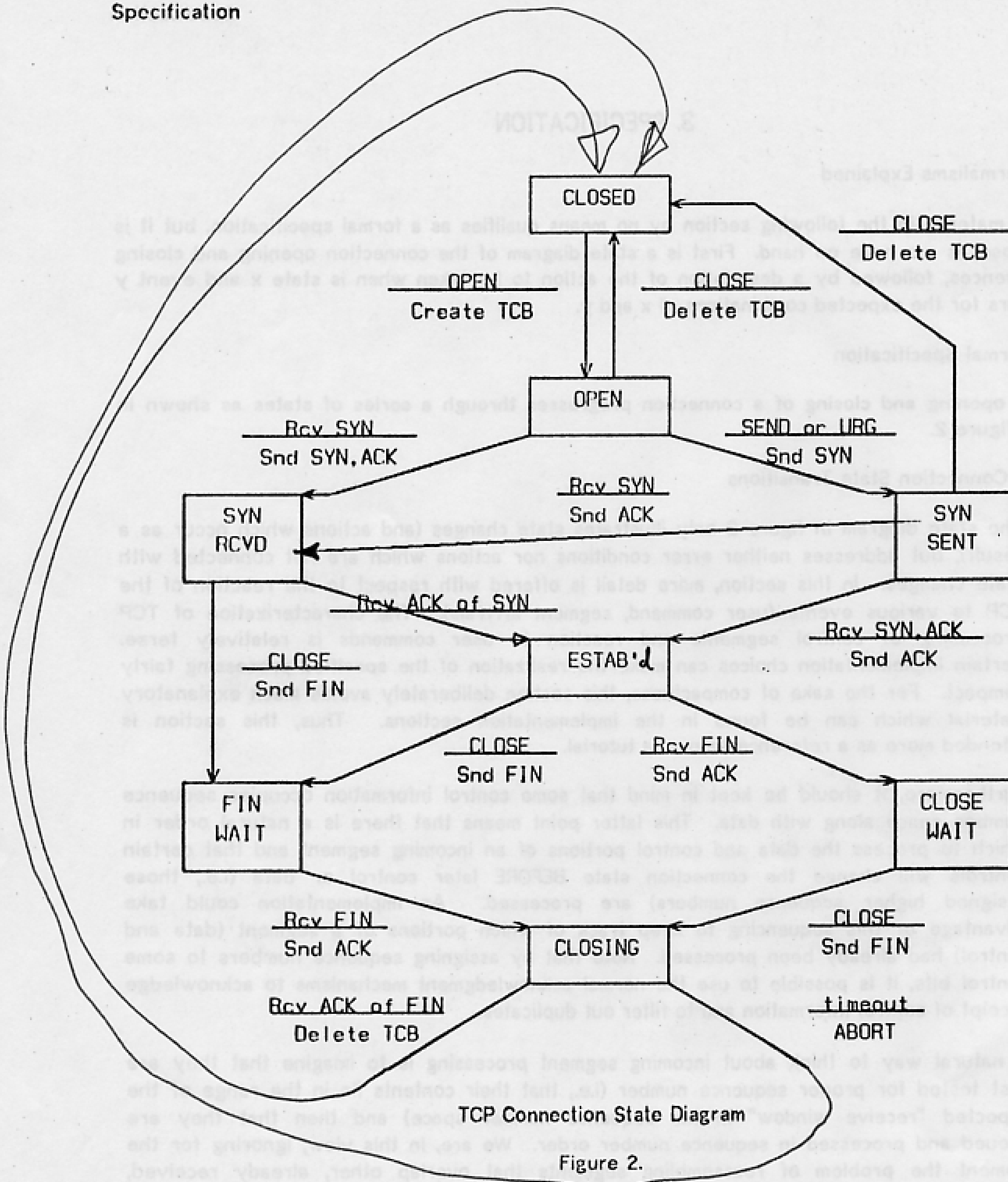
The opening and closing of a connection progresses through a series of states as shown in the figure 2.

TCP Connection State Transitions

The state diagram in figure 2 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, segment arrivals). The characterization of TCP processing of control segments and reaction to user commands is relatively terse. Certain implementation choices can make the realization of the specified processing fairly compact. 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.

Furthermore, it should be kept in mind that some control information occupies sequence number space along with data. This latter point means that there is a natural order in which to process the data and control portions of an incoming segment and that certain controls will change the connection state BEFORE later control or data (i.e., those assigned higher sequence numbers) are processed. An implementation could take advantage of this sequencing to keep track of which portions of a segment (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 segment 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 overlap other, already received, segments.



TCP Connection State Diagram

Figure 2.

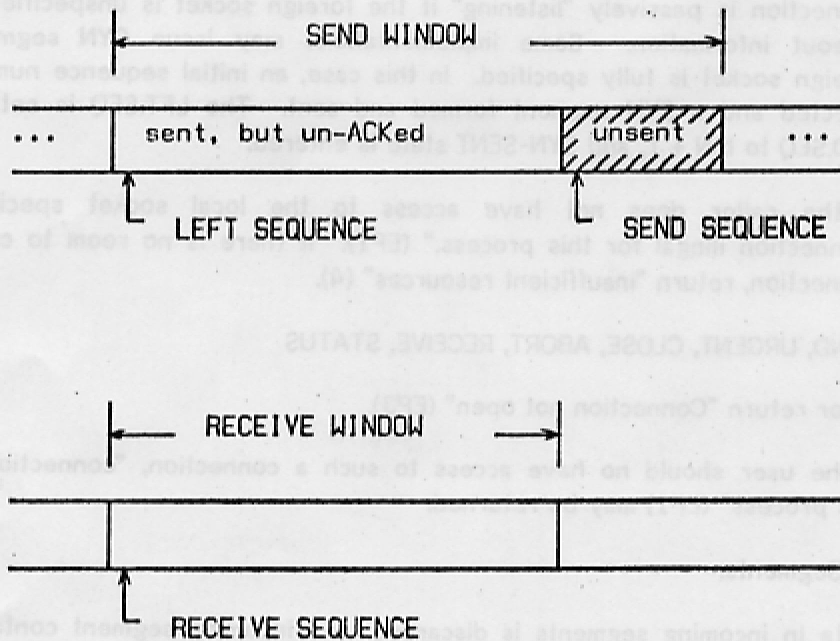
We have chosen to organize the description according to the connection state, to key the description to figure 2. In the following specifications the user events are mutually exclusive, while the incoming segment may call for some or all of the steps described to be carried out. When a segment causes 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 segment's acknowledgment field or sequence number field should not be repeated (lest a segment 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 segments. The following abbreviations are used in the action summaries below:

BUF.SIZ - buffer size
LFT.SEQ - left sequence
RCV.SEQ - receive sequence
RCV.WND - receive window
SEG.ACK - segment acknowledgement
SEG.LEN - segment length
SEG.SEQ - segment sequence
SND.SEQ - send sequence
SND.WND - send window
URG.PTR - urgent pointer

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



Sequence Number Management

Figure 3.

For concreteness error responses are given in terms of the model user interface described in detail in section 3.6, where the error information is reported via TCP-to-user messages called events. 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 result in a "foreign socket unspecified" (E5) response.

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

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 segments if the foreign socket is fully specified. In this case, an initial sequence number (ISN) is selected and a SYN segment formed and sent. The LFT.SEQ is set to ISN, the SND.SEQ 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 Segments.

All data in incoming segments is discarded. An incoming segment containing a RST is discarded. An incoming segment containing an ACK (and no RST) causes a RST to be sent in response.

Such a RST response has its sequence number set to the value of the acknowledgment field of the incoming segment; its acknowledgment field is set to

the sum of sequence number and the segment length of the incoming segment, and its RST and ACK control bits are set. See figure 11 for an example.

OPEN STATE

User Commands

1. OPEN

Return "already OPEN" (EP6)

2. SEND or URGENT

Select an ISN, send a SYN segment, set LFT.SEQ to ISN and SND.SEQ to ISN + 1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. URGENT can be sent as a combination SYN, URG segment. 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 Segments

1. ACK

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

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

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

2. SYN

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

<SEQ ISN><SYN><ACK RCV.SEQ>

SND.SEQ should be set to ISN + 1 and LFT.SEQ 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 segment (not containing SYN) will have an ACK and thus will be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection.

SYN-SENT STATE

User Commands

1. OPEN

Return "already OPEN" (EP6)

2. SEND or URGENT

Queue for processing after the connection is ESTABLISHED or segmentize, starting with the current SND.SEQ 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; SND.SEQ, RCV.WND.

Incoming segments

1. ACK

   If LFT.SEQ < SEG.ACK <= SND.SEQ then the ACK is acceptable. LFT.SEQ should be advanced to equal SEG.ACK, and any segment(s) on the retransmission queue which are thereby acknowledged should be removed.

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

2. SYN

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

   <SEQ SND.SEQ><ACK RCV.SEQ>

   and send it.

3. RST

   Notify user, delete TCB, enter CLOSED state.

4. Other text or control.

   Incoming segments with other control or text combined with SYN will be processed in SYN-RECEIVED or ESTABLISHED state. Arriving segments which do not contain SYN are old duplicates. 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 segment 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 segmentize 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 segmentize and send FIN segment. If the latter, enter FIN-WAIT state.

5. ABORT

Delete TCB, send a RST of the form:

<SEQ SND.SEQ><RST><ACK RCV.SEQ>

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

6. STATUS

Return state = SYN-RECEIVED, LFT.SEQ, SND.SEQ, SND.WND, RCV.SEQ, RCV.WND, and other desired statistics number of (SEND, RECEIVE buffers queued), segments queued for reassembly, for retransmission, etc.

Incoming Segments

1. Check sequence number

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

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

   If an incoming segment is not acceptable, form a reset (RST) segment:

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

   If the incoming segment 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 segment lies in the receive-window. The special "MAX" operation makes certain that empty ACK segments, whose length are 0, will be accepted. If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs.

   There is one other case: If the incoming segment is a SYN and it has the same value as the SYN which started this connection, then it is a delayed duplicate and should be acknowledged even if it fails the normal test.

2. ACK

   If LFT.SEQ < SEG.ACK <= SND.SEQ then set LFT.SEQ = SEG.ACK, remove any acknowledged segments from the retransmission queue, and enter ESTABLISHED state.

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

3. RST

   If the segment has passed sequence and acknowledgment tests, it is valid. Return this connection to OPEN state. The user need not be informed. All segments on the retransmission queue should be removed. All segmentized

buffers must be assigned new sequence numbers, so they should be requeued for re-segmentizing.

4. Other text or control

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

5. User Timeout

If the user timeout expires on any segment 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

Segmentize the buffer, send or queue it for output, send a piggy-backed acknowledgment (acknowledgment value = LFT.SEQ) with the data (this is not required, but there is no advantage in not doing so). If there is insufficient space to remember this buffer, simply respond with "insufficient resources" (4).

If remote buffer size is not one octet; then, if this is the end of a letter, do end-of-letter/buffer-size adjustment processing.

Let ISS be the initial send sequence number used on this connection, OSS be the send sequence before sending this segment, NSS the send sequence after sending this segment, RB be the remote buffer size, and L the number of octets in this segment. Then:

```
if EOL = 0 then NSS = OSS + L

if EOL = 1 then NSS = ISS + i * RB

where i is the minimum integer that satisfies:

ISS + (i-1)*RB < OSS + L <= ISS + i*RB
```

Set SND.SEQ = NSS.

3. RECEIVE

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

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).

When data is delivered to the user that fact must be communicated to the sender via an acknowledgment.

Let E be the sequence number of the last octet of data delivered into the users buffer. Then an acknowledgment with <SEQ SND.SEQ><ACK E>. This does not have to be sent as a separate segment, but rather should be piggy-backed on a data segment if possible without incuring undue delay.

If the data delivered to the use included an end of letter, and the buffer size is not 1; then the acknowledgment sent should take into account the end-of-letter/buffer-size sequence number adjustment.

Let IRS be the initial receive sequence number on this connection, LB be the local buffer size, and NES be the next expected sequrnce number.

```
if not EOL then NES = E + 1

if EOL then NES = IRS + i*LB

where i is the minimum integer that satisfies:

  IRS + (i-1)*LB < E + 1 <= IRS + i*LB
```

Send the acknowledgment <SEQ SND.SEQ><ACK NES>.

4. CLOSE

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

5. ABORT

Delete TCB and send a reset segment:

<SEQ SND.SEQ><RST><ACK RCV.SEQ>

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

6. STATUS

Return state = ESTABLISHED; SND.SEQ, LFT.SEQ, SND.WND, RCV.SEQ, RCV.WND, and other statistics, as desired.

Incoming Segments

1. Check sequence number

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

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

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

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

<SEQ SND.SEQ><ACK RCV.SEQ>

In any case, unacceptable segments should be discarded.

2. ACK

If LFT.SEQ < SEG.ACK <= SND.SEQ then set LFT.SEQ = SEG.ACK. Any segments on the retransmission queue which are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers which have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "OK" (0) response). If the ACK is a duplicate, it can be ignored.

If the remote buffer size is not 1, then the end-of-letter/buffer-size adjustment to sequence numbers may have an effect on the next expected sequence number to be acknowledged. It is possible that the remote TCP will acknowledge with a SEG.ACK equal to a sequence number of an octet that was

skipped over at the end of a letter. This a mild error on the remote TCPs part, but not cause for alarm.

3. RST

All pending RECEIVEs, SENDs, and URGENTs receive "reset" (P14) responses. All segment queues are flushed. User also receives an unsolicited general "reset" signal (P14). Enter the CLOSED state.

4. SYN

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

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

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

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

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. Segment text

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

If buffer size is not one octet, then do end-of-letter/buffer-size adjustment processing.

Let IRS be the initial receive sequence number used on this connection, ORS be the receive sequence before receiving this segment, NRS the receive sequence after receiving this segment, LB be the local buffer size, and L the number of octets in this segment. Then:

```
if EOL = 0 then NRS = ORS + L
```

if EOL = 1 then NRS = IRS + i * LB

where i is the minimum integer that satisfies:

IRS + (i-1)*LB < ORS + L <= IRS + i*LB

Set RCV.SEQ = NRS.

7. FIN

An ACK segment 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. Note that FIN implies EOL for processing any segment text not yet delivered to the user.

8. User Timeout

If the user timeout expires on a segment 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 segment (RST) should be formed and sent:

<SEQ SND.SEQ><RST><ACK RCV.SEQ>

Outstanding SENDs, URGENTs, RECEIVEs, CLOSEs, and/or segments queued for retransmission, or segmentizing, should be flushed, with appropriate "connection reset" (P12).

6. STATUS

Respond with state = FIN-WAIT, SND.SEQ, LFT.SEQ, SND.WND, RCV.SEQ, RCV.WND, and other statistical information, as desired.

Incoming segments

1. Check sequence number

If RCV.SEQ <= SEG.SEQ + MAX(SEG.LEN-1,0) < RCV.SEQ + RCV.WND then segment sequence is acceptable. Otherwise, if SEG.LEN is non-zero, an ACK segment should be sent:

<SEQ SND.SEQ><ACK RCV.SEQ>

In any case, an unacceptable segment should be discarded.

2. ACK

If LFT.SEQ < SEG.ACK <= SND.SEQ, then LFT.SEQ should be advanced appropriately and any acknowledged segments deleted from the retransmission queue. SENDs or URGENTs which are thereby completed can also be acknowledged to the user. ACK's outside of the SND.WND 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 segment 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 SEG.SEQ processing. Other

SYN's could not have passed the SEG.SEQ 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. Segment Text

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

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 segment 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

Segmentize any text to be sent and queue for output. If there is insufficient space to remember the SEND or URGENT, return "insufficient resouces" (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 segment 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 segments until they have been delivered into user buffers, the FIN segment which led to the CLOSE-WAIT state will not be processed until all preceding segment 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 segmentized; then send a FIN segment, enter CLOSING state.

5. ABORT

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

<SEQ SND.SEQ><RST><ACK RCV.SEQ>

Flush all segment queues and delete the TCB.

6. STATUS

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

Incoming Segments

1. Check sequence number

If RCV.SEQ <= SEG.SEQ + MAX(SEG.LEN-1,0) < RCV.SEQ + RCV.WND then the segment sequence is acceptable. Otherwise, if SEG.LEN is non-zero, an ACK should be sent:

<SEQ SND.SEQ><ACK RCV.SEQ>

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

2. ACK

   If LFT.SEQ < SEG.ACK <= SND.SEQ, then LFT.SEQ should be advanced appropriately and any acknowledged segments 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. Segment 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 SEG.SEQ 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. Segment text

   This should not occur, since a FIN has been received from the remote side. Ignore the segment 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 segment 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 segment queues. If a CLOSE command is still pending, respond "connection reset" (P14).

6. STATUS

   Return State = CLOSING along with other TCP parameters.

Incoming segments

1. Check sequence number

   If RCV.SEQ <= SEG.SEQ + MAX(SEG.LEN-1,0) < RCV.SEQ + RCV.WND then segment sequence is acceptable. Otherwise, if SEG.LEN is non-zero, an ACK segment should be formed and sent:

   <SEQ SND.SEQ><ACK RCV.SEQ>

   In any case, an unacceptable segment should be discarded.

2. ACK

   If LFT.SEQ < SEG.ACK <= SND.SEQ, then LFT.SEQ should be advanced and any acknowledged segments deleted from the retransmission queue. SENDs or URGENTs which are thereby completed can also be acknowledged to the user. ACK's outside of the SND.WND can be ignored.

3. RST

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

4. Segment text or control

   No other control or text should be sent by the remote side, so segments containing non-zero SEG.LEN should be ignored.

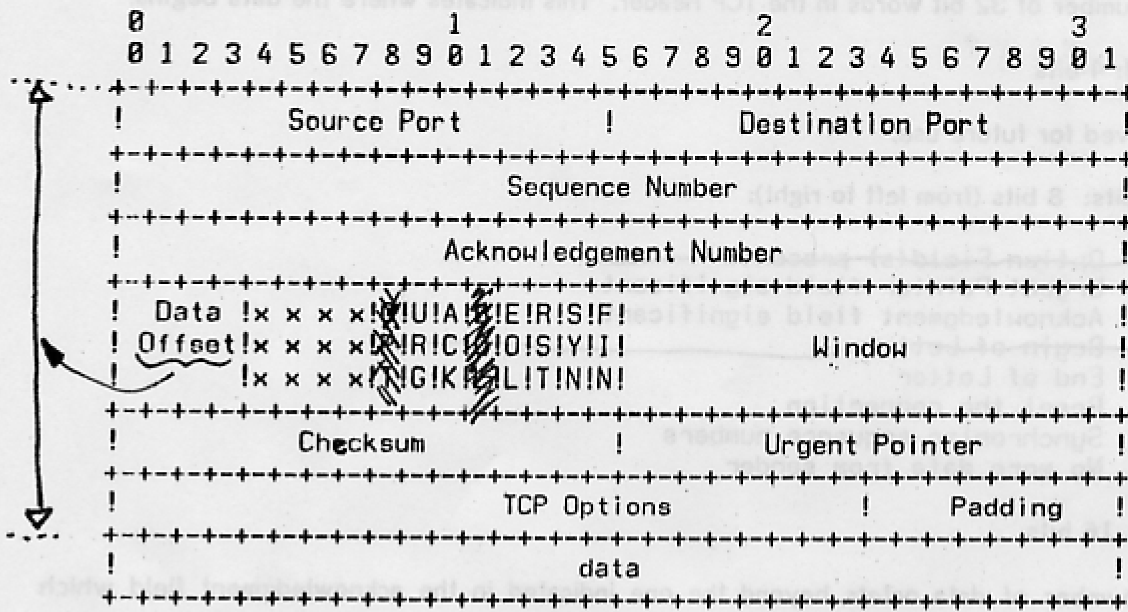
5. User Timeout

   If the user timeout expires on a segment 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.

3.3. Header Format

All internetwork segments (TCP and otherwise) have a basic internet header consisting of source and destination addresses, and header and total length fields, among others [Postel78d]. 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.

TCP Header Format



Example TCP Header

Note that one tick mark represents one bit position.

Figure 4.

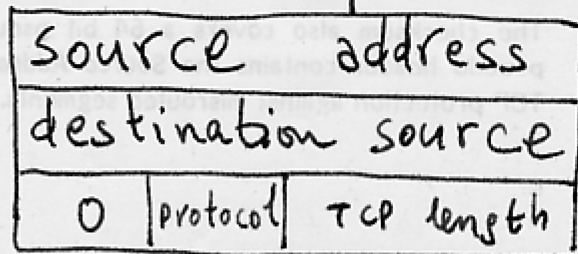
the pseudo header is assumed to precede the real header for checksum purposes.

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.



TCP pseudo header

Sequence Number: 32 bits

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

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 segment is expecting to receive.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins.

Reserved: 4 bits

Reserved for future use.

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

~~OPT: Option Field(s) present~~
URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
~~BOL: Begin of Letter~~
EOL: End of Letter
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

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

Checksum: 16 bits

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

The checksum also covers a 64 bit pseudo header prefixed to the TCP header. This pseudo header contains the Source Address and the Destination Address. This give the TCP protection against misrouted segments.

Urgent Pointer: 16 bits

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

~~If the high order bit is set this option is not included in the checksum calculation.~~
Options are ~~ordinarily~~ always included in the checksum.

There are two special cases for options.

The first is the 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 the option whose length field is one. This option serves as ~~padding~~ NOP-OPTION 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. The content of the header beyond the end-of-option mark should be header TCP padding (i.e., zero). The two special options are included in the checksum of the segment.

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
-	1	End of option list.
-	1	~~Padding.~~ NOPTION
100	-	Reserved.
~~101~~	~~4~~	~~Segment Label=sequence number for debugging purposes.~~
102	~~6~~ 4	Secure Open - used by TCP's communicating with BCR security system.
103	~~6~~ 4	Secure Close-used by TCP's communicating with BCR security system.
105	4	Buffer size, in octets.
~~204~~	~~6~~	~~TCP timestamp for diagnostics (not checksummed).~~

Specific Option Definitions

End of Option List

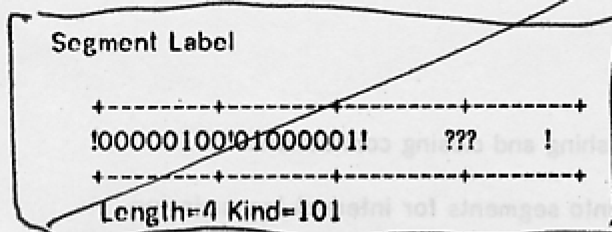
```
+--------+
!00000000!
+--------+
 Length=0
```

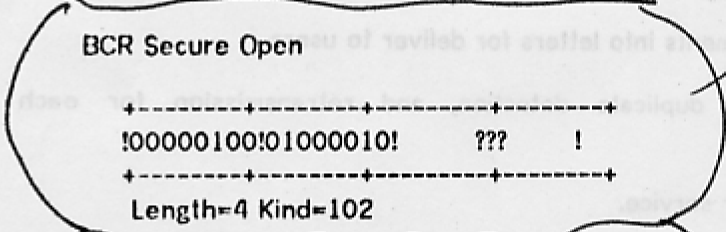
This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end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

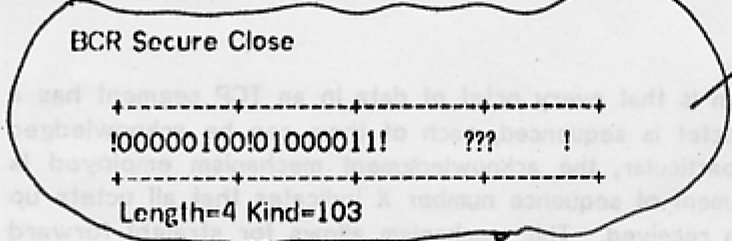
~~Padding~~ NOPTION

```
+--------+
!00000001!
+--------+
 Length=1
```

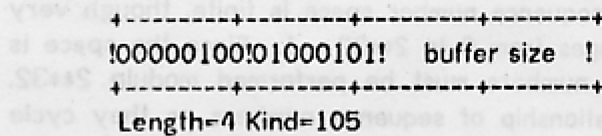
This option code may be used between options, for example, to align the begining of a subsequent option on a word boundary.





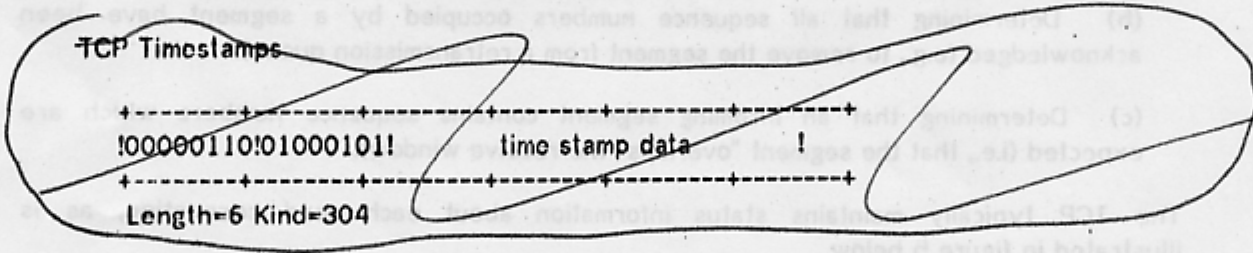


Buffer Size



Buffer Size Option Data: 16 bits

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



Padding:

The Padding field is used to ensure that the data begins on 32 bit word boundary. The padding is composed of zeros.

3.4. Discussion

The main jobs of the TCP are:

a. Connection management, the establishing and closing connections.

b. Packaging of outgoing user letters into segments for internet transmission.

c. Reassembly of incoming segments into letters for deliver to users.

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

e. Processing user requests for service.

Sequence Numbers

A fundamental notion in the design is that every octet of data in an TCP segment 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 straight-forward duplicate detection in the presence of retransmission.

It is essential to remember that the actual sequence number space is finite, though very large. 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. There are some subtlies to computer modulo arithmetic so great care should be taken in programming these tests. The typical kinds of sequence number comparisons which the TCP must perform include:

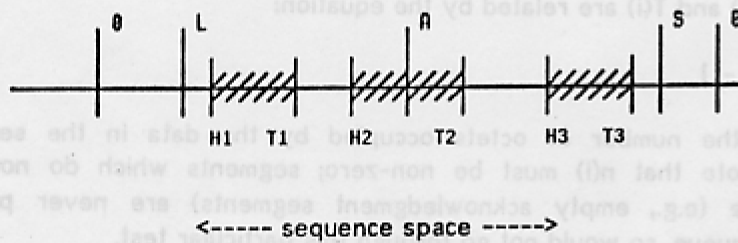
(a) Determining that an acknowledgment refers to some sequence number sent but not yet acknowledged.

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

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

The TCP typically maintains status information about each send connection, as is illustrated in figure 5 below.

older sequence numbers newer sequence numbers



TCP State Information for Sending Sequence Space

Figure 5.

L = oldest, unacknowledged sequence number (LFT.SEQ)

S = next sequence number to be sent (SND.SEQ)

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

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

T(i) = last sequence number of the i-th segment (SEG.SEQ + MAX(0,SEG.LEN-1))

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

0 < (A - L) <= (S - L) (1)

0 < (SEG.ACK - LFT.SEQ) <= (SND.SEQ - LFT.SEQ)

We will often write inequality (1) in the form below:

L < A <= S (2)

LFT.SEQ < SEG.ACK <= SND.SEQ

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

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

0 < (T(i) - L) < (A - L) (3)

0 < ((SEG.SEQ+SEG.LEN-1) - LFT.SEQ) < (SEG.ACK - LFT.SEQ)

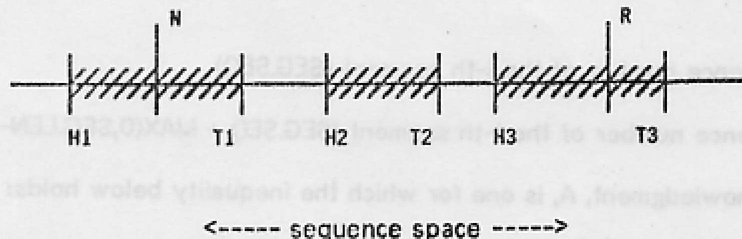
In this instance, H(i) and T(i) are related by the equation:

T(i) = H(i) + n(i) - 1 (4)

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

For each receive connection the following information is needed:

older sequence numbers newer sequence numbers



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

TCP State Information for Receiving Sequence Space

Figure 6.

N = next sequence number expected on incoming segments (RCV.SEQ)

R = last sequence number expected on incoming segments, plus one (RCV.SEQ+RCV.WND)

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

T(i) = last sequence number occupied by the i-th incoming segment (SEG.SEQ+MAX(0,SEG.LEN-1))

R and N in figure 6 are related by the equation:

R = N + W (5)

RCV.SEQ+RCV.WND = RCV.SEQ + RCV.WND

Where W = the receive window size

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

0 <= (T - N) < (R - N) (6)

where T is the last sequence number occupied by the segment, N is the next sequence number expected on an incoming segment, and R is the right edge of the receive window, as shown in figure 6.

Actually, it is a little more complicated than inequality 6 indicates. Due to zero windows and zero length segments, we have four cases for the acceptability test for an incoming segment:

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

Note that the acceptance test for a segment, since it requires the end of a segment to lie in the window, is somewhat more restrictive than is absolutely necessary. If at least the first sequence number of the segment lies in the receive window, or if some part of the segment lies in the receive window, then the segment might be judged acceptable. Thus, in figure 6, at least segments 1 (H(1)-T(1)) and 2 (H(2)-T(2)) are acceptable by the strict rule and segment 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 segments should be acceptable except ACK segments. Thus, it should be possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs 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 segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length includes both data and sequence-space-occupying controls.

Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. New instances of a connection will be referred to as incarnations of the connection. The problem that arises owing to this is -- "how does the TCP identify duplicate segments from previous incarnations of the connection?" This problem becomes 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 segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN.

The "three way handshake" and the advantages of a "clock-driven" scheme are discussed in [Tomlinson74]. More on the subject and algorithms for implementing the clock-driven scheme can be found in [Dalal74, Dalal75, Cerf76b].

Knowing When to Keep Quiet

A basic goal of the TCP design is to prevent segments from being emitted with sequence numbers which duplicate those which are still in the network. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds.

Thus, the ISN cycles approximately every 4.55 hours. Since we assume that segments will stay in the network no more than tens of seconds or minutes, at worst, we can reasonably assume that ISN's will be unique.

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

Establishing a connection

The "three-way handshake" is essentially a unidirectional attempt to establish a connection; i.e., there is an initiator and a responder. The TCP can also establish a connection when a simultaneous initiation occurs. A simultaneous attempt occurs when one TCP receives a "SYN" segment which carries no acknowledgment after having sent a "SYN" earlier. Of course, the arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases. Several examples of connection initiation are offered below, using a notation due to Tomlinson. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the user until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state (see figure 2)).

The simplest three-way handshake is shown in figure 7 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states are keyed to those in figure 3 and represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out, 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

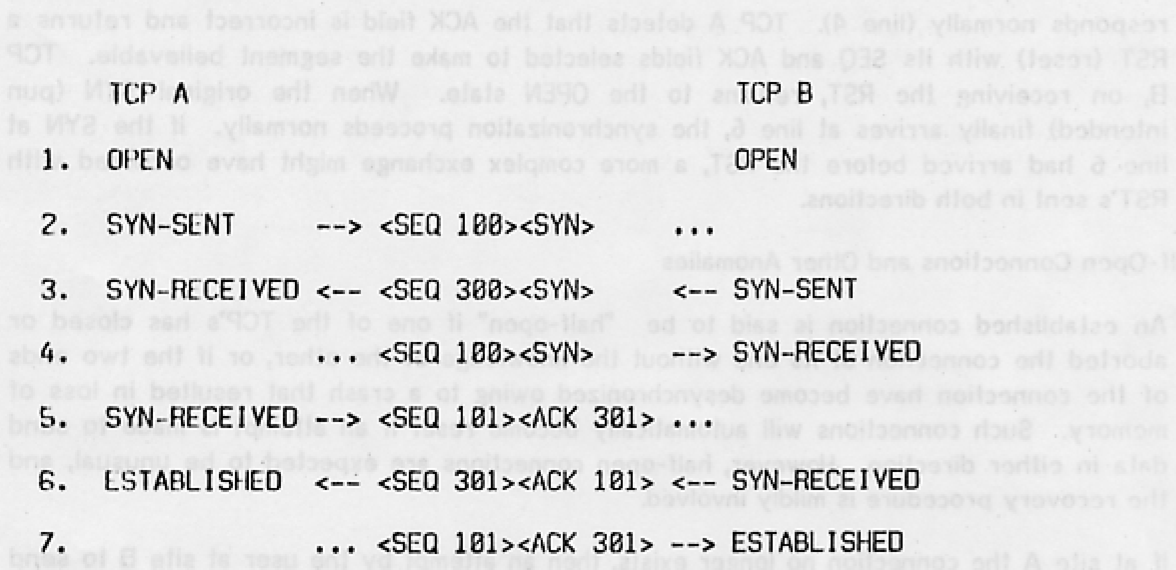
Figure 7.

In line 2 of figure 7, TCP A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP B sends a SYN and acknowledges the SYN it received from TCP A. Note that the acknowledgment field indicates TCP B is now expecting to hear sequence 101, implicitly acknowledging the SYN which occupied sequence 100.

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

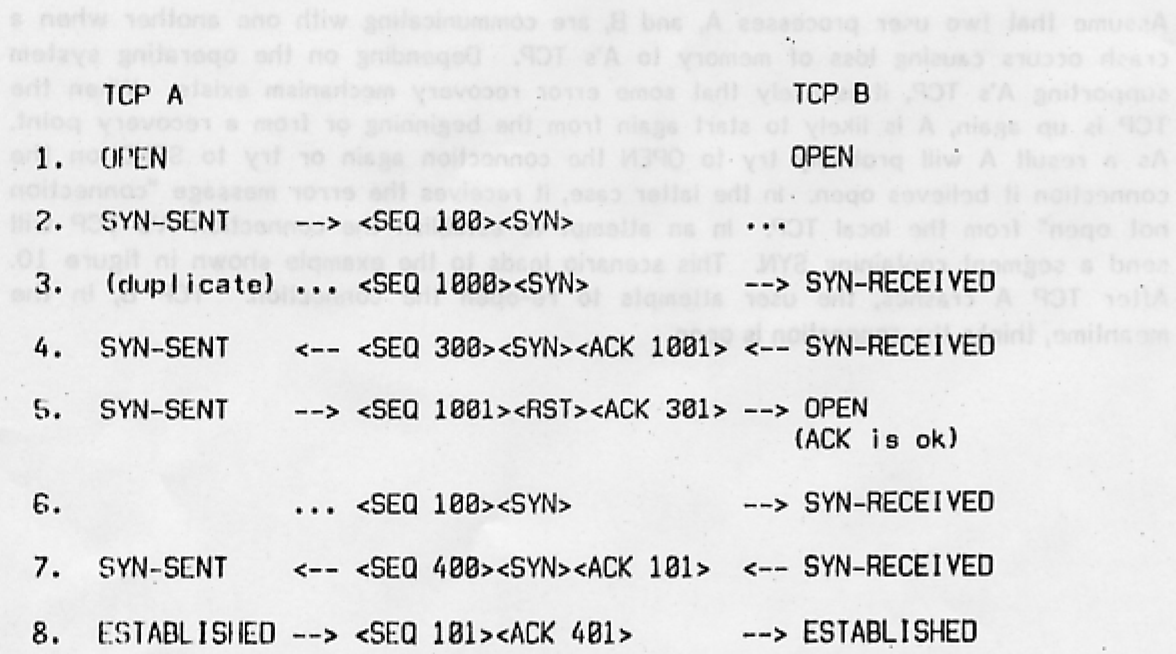
Simultaneous initiation is only slightly more complex, as is shown in figure 8. Each TCP cycles from OPEN to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

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" connections below.



Simultaneous Connection Synchronization

Figure 8.



Recovery from Old Duplicate SYN

Figure 9.

As a simple example of recovery from old duplicates, consider figure 9. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it

responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ and ACK fields selected to make the segment believable. TCP B, on receiving the RST, returns to the 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.

Half-Open Connections and Other Anomalies

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

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

Assume that two user processes A, and B, are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, A is likely to start again from the beginning or from a recovery point. As a result A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local TCP. In an attempt to establish the connection A's TCP will send a segment containing SYN. This scenario leads to the example shown in figure 10. 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 10.

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

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in figure 11. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

```
       TCP A                                                   TCP B

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

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

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

Active Side Causes Half-Open Connection Discovery

Figure 11.

In figure 12, 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.  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 12.

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

Reset Generation

As a general rule, reset (RST) should be sent whenever a segment arrives which apparently is not intended for the current or a future incarnation of the connection. A reset should not be sent if it is not clear that this is the case. Thus, if any segment arrives for a nonexistant connection, a reset should be sent. If a segment 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 segment that acknowledges something not yet sent. The RST should take its SEQ field from the ACK field of the offending segment (if the ACK control bit was set), and its ACK field should acknowledge all data and control in the offending segment. This is done to make the segment believable to the remote TCP. The sequence field will contain the next sequence the remote TCP expects, and the acknowledgment field will acknowledge everything the remote TCP claims to have sent.

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

Reset Processing

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

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). If the receiver was in a synchronized state (ESTABLISHED, FIN-WAIT, CLOSE-WAIT, CLOSING), it aborts the connection and advises the user.

Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The notion of closing a full-duplex connection is subject to ambiguous interpretation, of course, since it may not be obvious how to treat the receiving side of the connection. We have chosen to treat CLOSE in a simplex fashion. The user who CLOSES may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that the TCP will unilaterally inform a user, even if no RECEIVEs are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP.

There are essentially three cases:

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

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

3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT state. RECEIVES are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN and delete the connection (see figure 2). 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. The user should respond with a CLOSE, upon which the TCP can send a FIN to the other TCP. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after a timeout the connection is aborted and the user is told.

Case 3: both users close simultaneously

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

Close also implies an end of letter.

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 always equals 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-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.

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 segment that contains the 16 bit buffer size data in an option field in the TCP header, the presence of the field being signaled by the OPT 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 $im > n > (i-1)m$.

The effect of all this is that each EOL advances the sequence number (SN) to

SN = IS + i*B

where IS is the initial sequence number, B is the buffer size, and i is the smallest integer such that

IS + (i-1)*B < y <= IS +i*B

where y = SEG.SEQ+SEG.LEN

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

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 urgent field which is carried in all segments transmitted. A control bit (URG) indicates that the 16-bit field is meaningful and should be added to the segment 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.

The objective of the TCP urgent mechanism is to allow the sending user to stimulate the receiving user to accept some urgent data and to permit the receiving TCP to indicate to the receiving user which octet in the received data is the last of the currently known urgent data.

The assumption made in providing this service is that the higher level will always transmit new data when urgent is to be asserted. Typically, the higher level protocol may employ a special method to distinguish the urgent data from ordinary data, e.g., by special format or coding conventions, but this need not be necessarily be the case.

The basic urgent service can be described as follows:

When the user hands a buffer of data to the TCP to be sent, and asserts that it is urgent, the TCP assumes that the last octet of the urgent data coincides with the last octet of the buffer. Successive transmission of new urgent data causes the "end of urgent data" to extend farther into the data stream.

If the sending user asserts EOL when sending the urgent data, then the receiving TCP will attempt to deliver the data to the receiving user even if the buffer into which data is being assembled is not full. This is not unique to urgent data since EOL is the mechanism for the user to assert to the receiving TCP "deliver this without waiting for more data".

In any case, the receiving TCP will indicate to the receiving user precisely which octet of data is the last of the urgent octets. This is accomplished by associating with newly delivered data a pointer to the "end of urgent data".

The urgent mechanism provides an out-of-band signal which the sending user can employ to alert the receiving user to enter an "urgent" state. No semantics are assumed for this signal. Furthermore, there is no intent that every urgent data transmission result in an urgent signal to the receiving user. Instead, it is guaranteed that the receiving user will be signalled at least once when he should enter the urgent state and will later be told when he has received that last known (to the receiving TCP) urgent data.

The precise form of the urgent signal is an implementation decision but it must be "out-of-band" with respect to delivery of normal data.

It is assumed that the user always provides new data to send when asserting urgent. However, the TCP may not always be able to accept any new data to transmit (which is one reason for trying to assert urgent). Then sending TCP will attempt to signal urgent to the receiving TCP even if it cannot actually accept new data for transmission. To be consistent with the design of the urgent mechanism, users which have attempted to send urgent data must continue to attempt to send this data until it is accepted or the connection is otherwise closed or aborted.

A consequence of the TCP's attempt to signal urgent even when it cannot accept the new data for transmission is the receiving user may enter and leave the urgent state more than once before the desired urgent data is actually delivered.

Managing the Window

The mechanisms provided would allow a TCP to advertize a large window and to subsequently advertize a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged. For more on window management, see the section on implementation.

The sending TCP must be prepared to accept and send at least one octet of new data

even if the send window is zero. This is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other.

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

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

3.5. Examples & Scenarios

Examples are needed.

3.6. Interfaces

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

User/TCP Interface

The functional description of user commands to the TCP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that 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.

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 succcess or various types of failure.

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

Open

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

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the local network and TCP identifiers for the source address will either be supplied by the TCP or by the processes that serve it (e.g., the program which interfaces the TCP to its segment switch or the segment 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 constitititutes a LISTENING local socket which can accept communication from any foreign socket. Provision is also made for partial specification of foreign sockets.

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 5.4.

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; most likely, it will be related to the measured time for responses from the remote TCP.

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

Send

Format: SEND(local connection name, buffer address, byte count, EOL flag [, 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 segment created from the buffer. 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 segment arriving for the local socket]) then the designated buffer is sent to the implied foreign socket. In general, users who make use of OPEN with an unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

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

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

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. However, this simple method is both highly subject to deadlocks (for example, both sides of the connection might try to do SENDs before doing any

RECEIVEs) and offers poor performance, so it is not recommended. A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

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

NOTA BENE: In order for the process to distinguish among error or success indications for different SENDs, it might be appropriate for the buffer address to be returned along with the coded response to the SEND request. We will offer an example event code format below, showing the information which should be returned to the calling process.

Receive

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

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

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

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

The 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.

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

Close

Format: CLOSE(local connection name)

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

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

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

Close also implies end of letter.

Urgent

Format: URGENT(local connection name)

Special control information is sent to the destination indicating that urgent processing is appropriate. This facility, for example, can be used to simulate "break" signals from terminals or error or completion codes from I/O devices. The semantics of this signal to the receiving process are unspecified. The receiving TCP will signal the urgent condition to the receiving process as long as the urgent pointer indicates 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.

Status

Format: STATUS(local connection name)

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

This command returns a data block containing the following information:

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

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

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs, 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.

TCP-to-User Messages

Type Codes

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

```
 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
```

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
Number of buffers awaiting acknowledgment
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.

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 acceptible
	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.

```
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
```

TCP/Network Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. One case is that of the ARPA internetwork system where the lower level module is the Internet Protocol [Postel78d]. In most cases the following simple interface would be adaquate.

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

SEND (dest, BufPTR, len)

where:

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

Response:

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

RECV (BufPTR)

Reponse:

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

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

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

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

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

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

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

time to live = one minute, or 00111100.

4. VERIFICATION

Requires further research.

5. IMPLEMENTATION

5.1. What Not to Leave Out

???

5.2. User Interfaces

???

5.3. Mechanisms

Structure of the TCP

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 segments from the net, and some the output of segments 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 Segment Handler which sends segments to the segment switch; the Segmentizer which formats letters into internet segments; the Input Segment Handler which processes incoming segments; the Reassembler which builds letters for users; and the Retransmitter which retransmits unacknowledged segments.

> *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 "Segmentizer" process is awakened to create one or more output segments from the buffer. The Segmentizer attempts to maintain a non-empty queue of output segments so that the output handler will not fall idle waiting for the segmentizing operation.

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

When a segment is created, it is placed on a FIFO send segment queue associated with its TCB. The Segmentizer wakes the Output Segment Handler and then continues to

segmentize a few more buffers, perhaps, before going to sleep. The Output Segment Handler is awakened either by a "hungry" segment switch or by the Segmentizer. The send segment queue can be used as a "work queue" for the Output Segment Handler. After a segment has been sent, but usually before an ACK is returned, the Output Segment Handler moves the segment to a retransmission queue associated with each TCB.

Retransmission timeouts can refer to specific segments, or the retransmission queue can be periodically searched for the timed-out segments. If an ACK is received, the retransmission entry can be removed from the retransmit queue. The send segment queue contains only segments 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 Segmentizer 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 segmentization), advance the head of the queue, and then unlock access to the queue.

Incoming segments are examined by the Input Segment Handler. Here they are checked for valid connection sockets and acknowledgments are processed, causing segments to be removed, possibly, from the retransmit segment queue, as needed.

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

As data is moved from segments into buffers, filled 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 segment queue with an empty receive buffer queue.

Input Segment Handler

The Input Segment Handler is awakened when a segment arrives from the network. It first verifies that the segment 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 Segment Handler looks out for control or error information and acts appropriately. For example, if the incoming segment is a RST (reset) request, and is

"believable", then the input segment handler clears out the related TCB, empties the associated send and receive segment 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 segments are used to update the send left window edge and to remove the ACKed segments from the TCB retransmit segment queue. If the segment being removed was the end of a user buffer, then the buffer must be dequeued from the segmentized buffer queue, and the user informed.

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

Let

```
RCV.WND = window size
S       = size of sequence number space
RCV.SEQ = left window edge
R       = RCV.SEQ + RCV.WND = right window edge
x       = sequence number to be tested
```

For any sequence number, x, if

$$0 <= (x-RCV.SEQ) \bmod S < (R-RCV.SEQ) \bmod S \quad (7)$$

then x is within the window.

A segment should be rejected only if all of it lies outside the window. This is easily tested by letting x be, first the segment sequence number, and then the sum of segment sequence number and segment length, less one in inequality 7 above.

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

Let

```
SEG.LEN = segment length
RCV.SEQ = left window edge
R       = RCV.SEQ + RCV.WND = right window edge
H       = SEG.SEQ = first sequence number in segment
T       = SEG.SEQ + SEG.LEN - 1 = last sequence number in segment
```

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

$0 <= RCV.SEQ - H < SEG.LEN$

and

$0 <= R - H < SEG.LEN$ (8)

then the segment includes the receive window.

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

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

The segment 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.

Reassembler

The Reassembler process is activated by both the Input Segment Handler and the RECEIVE user command. When the Reassembler is awakened, it looks at the receive segment queue for the associated TCB. If there are some segments 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 segment matches the left window edge, then the segment can be moved into the user's buffer. The Reassembler keeps transferring segments into the user's buffer until the segment 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 segment 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 segment might be only partially emptied when a buffer is filled, so progress in the segment 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.

Of course, the sequence number processsing is adjusted to take into account the EOL.

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

Segmentizer

The Segmentizer process gets work from both the Input Segment 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 Segment Handler indicates that more TCP buffers may be available from delivered segments.

When the Segmentizer is awakened it looks at the send buffer queue for the associated TCB. If there is a new or partial letter awaiting segmentization, it tries to segmentize the letter, TCP buffers and window permitting. For every segment produced, it signals the Output Segment 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 segment is produced since awakening. If segmentization is not possible, the Segmentizer goes to sleep.

If a partial buffer was transferred, then the Segmentizer must mark progress in the send buffer queue. Completely segmentized buffers are dequeued from the send buffer queue and placed on a segmentized 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 Segmentizer segmentizes 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 segment to be sent before the data segment. Some implementations may choose not to send data with SYN, and some may choose to discard any data received with SYN.

Output Segment Handler

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

All data segments 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 segment's sequence number.

Retransmitter

This process can either be viewed as a separate process or as part of the Output Segment 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 segment 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 segment is placed on the send segment queue of the corresponding TCB. The Output Segment 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 segments carry the latest receive left window edge and acknowledgment information, the checksum may require recomputation.

In some cases it may be useful to repackage segments on the retransmission queue into new segments more in keeping with current window information, or to take advantage of an partial acknowledgment of a segment.

5.4. Data Structures

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: 48 bits

Local address: 32 bits

Local port: 16 bits

Foreign socket: 48 bits

Foreign address: 32 bits

Foreign port: 16 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 segment sequence number to send: 32 bits

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

Initial Send Sequence Number: 32 bits

Initial Receive Sequence Number: 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 2 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. 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 segmentized]: 16 bits

Tail of Send buffer queue: 16 bits

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

Head of Send segment queue: 16 bits

Tail of Send segment queue: 16 bits

Head of Segmentized buffer queue: 16 bits

Tail of Segmentized buffer queue: 16 bits

Head of Retransmit segment queue: 16 bits

Tail of Retransmit segment 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 segment queue: 16 bits

Tail of receive segment queue: 16 bits

Pointer to last octet filled in receive buffer: 16 bits

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

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

Forward TCB pointer: 16 bits

Backward TCB pointer: 16 bits

5.5. Program Sizes, Performance Data

???

5.6. Test Sequences, Procedures, Exerciser

???

5.7. Parameter Values: Timeouts, Segment Sizes, Buffer Strategies

Much work needs to be done in this area still, but we do have the following discussion on buffers and windows, and the interaction thereof; and some comments on retransmission packaging.

Buffer and Window Allocation

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 segments, 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 and 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].

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 segments 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 it's 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 segments until acknowledgments 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.

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 queuing 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 segments arrive out of order, they must be buffered until the missing segments arrive so that segments (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, segment size, traffic level, routing, timeouts, window size, and other factors may affect the degree to which segments 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 suggest 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 segments 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 segment buffers for a connection are filled with received segments, the window must go to zero to prevent retransmissions until the user accepts some segments.

Setting the receive window to zero can have some interesting side effects. In particular, it is not enough to merely send an empty ACK segment 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.

Retransmission Packaging

Insisting that SEG.SEQ (i.e., the first sequence number occupied by the segment) lie in the RCV.WND could lead to deadlock in the case of alternate gateway routing and different fragmentation.

A Scenario:

Assume the receivers RCV.SEQ is 1.

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

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

Segment p2 arrives at the receiver and is found acceptable. The receiver sets the RCV.SEQ to 5.

Gateway A breaks.

The sender times out and retransmits p1 as p4.

The receiver finds p3 afflicted with errors and discards it.

Gateway B fragments p4 into three new segments, 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 RCV.SEQ.

When p6 arrives at the receiver it is acknowledged then if the special MAX function were not used it would be discarded since it's SEG.SEQ is below the RCV.SEQ.

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

5.8. Debugging

???

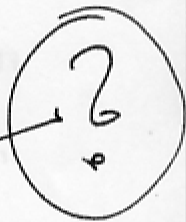
GLOSSARY

1822

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

ACK

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



Address

An address is a variable length quantity (in multiples of octets).

ARPANET message

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

ARPANET packet

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

BOL

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

BUF.SIZ

buffer size

buffer size

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

connection

A logical communication path identified by a pair of sockets.

Destination

The destination address, an internet header field.

DF

The Don't Fragment bit carried in the internet header type of service field.

DGP

DataGram Protocol: A host-to-host protocol for communication of raw data.

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

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

Flags
An internet header field carrying various control flags.

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

Fragment Offset
This internet header field indicates where in the internet segment this fragment belongs.

FTP
A file transfer protocol.

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

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

Identification
An internet header field identifying value assigned by the sender to aid in assembling the fragments of a segment.

IHL
The internet header field Internet Header Length is the length of the internet header measured in 32 bit words.

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

internet fragment
A portion of the data of an internet segment with an internet header.

internet packet
Either an internet segment or an internet fragment.

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

ISN
The Initial Sequence Number.

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

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

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

LFT.SEQ
left sequence

local packet
The unit of transmission within a local network.

MF
The More-Fragments Flag carried in the internet header Flags field.

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

more-fragments-flag
A flag indicating whether or not this internet packet contains the end of an internet segment, carried in the internet header Flags field.

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

NFB
The Number of Fragment Blocks in a portion of an internet segment. That is, the length of a portion of data measured in 8 octet units.

octet
An eight bit byte.

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

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

Padding
A Padding field is used to ensure that the data begins on 32 bit word boundary. The padding is zero. Both the Internetwork Protocol and TCP provide for padding fields.

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

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

PS
A Packet Switch. For example, an IMP.

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

RCV.SEQ
receive sequence

RCV.WND
receive window

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 segments overlapping the range RCV.SEQ to RCV.SEQ + RCV.WND - 1 carry acceptable data or control. segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.

RST

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

RTP

Real Time Protocol: A host-to-host protocol for communication of time critical information. For details do not ask.

SEG.ACK

segment acknowledgment

SEG.LEN

segment length

SEG.SEQ

segment sequence

segment

A logical unit of data, in particular an internet segment is the unit of data transfered between a pair of internet modules, and a TCP segment is the unit of data transfered between a pair of TCP modules. In this document the word segment when used without qualification means a TCP segment.

segment acknowledgment

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

segment length

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

segment sequence

The number in the sequence field of the arriving segment.

send sequence

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

send window

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

SND.SEQ

send sequence

SND.WND
send window

socket
An address which specifically includes a port identifier.

Source
The source address, an internet header field.

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

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

Total Length
The internet header field Total Length is the length of the internet packet in octets including internet header and data.

TOS

Type of Service
An internet header field which indicates the type of service for this internet fragment.

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

URG.PTR
urgent pointer

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

Version
The Version field indicates the format of the internet header.

XNET
A cross-net debugging protocol.

BIBLIOGRAPHY = $\int_{-\infty}^{1978+}$ (was relevant at time t) dt.

Notes of Working Group 6.1 of the International Federation of Information Processing (IFIP), [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. Altwyver, 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, May 1978 (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, Elsie 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 Menehune - 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)," IEN 5, March 1977.

Cerf78

Vinton G. Cerf, "A Proposed New Internet Header Format," Advanced Research Projects Agency, IEN 26, February 1978.

Cerf78a

Vinton G. Cerf, "A Proposal for TCP Version 3.1 Header Format," Advanced Research Projects Agency, IEN 27, February 1978.

CGN76

W. W. Clipsham, F. E. Glave, M. L. Narraway, "Datapac Network Overview," Proceedings of ICCC76, p. 131-136.

CHMMW74

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).

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

CMSZ78

Vinton G. Cerf, Alexander McKenzie, Roger Scantlebury, Hubert Zimmermann, "Proposal for an Internetwork End to End Transport Protocol," Revision editors: A. Danthine, M. Gien, G. Grossman, and C. Sunshine; INWG General Note 96.1, IFIP W.G. 6.1, February 1978.

CP78

Vinton G. Cerf and Jonathan B. Postel, "Specification of Internetwork Transmission Control Program - TCP Version 3," Information Sciences Institute, IEN 21, January 1978.

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 Cat -71C59-C, p. 46-49.

DCA75

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

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.

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.

FP78

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

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.

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.

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.

INWG-P91

International Network Working Group, "Proposal for a Standard Virtual Terminal Protocol," INWG Protocol Note 91, IFIP W.G. 6.1, February 1978.

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, Ill., 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, Berhshire, 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, Information Sciences Institute, Marina del Rey, California, 11 November 1977.

Postel78a

Jonathan B. Postel, "Draft Internetwork Protocol Specification - Version 2," Information Sciences Institute, IEN 28, February 1978.

Postel78b

Jonathan B. Postel, "Draft Specification at Internetwork Transmission Control Protocol - TCP Version 4," Information Sciences Institute, IEN 40, June 1978.

Postel78c

Jonathan B. Postel, "Draft Internetwork Protocol Specification - Version 4," Information Sciences Institute, IEN 41, June 1978.

Postel78d

Jonathan B. Postel, "Internetwork Protocol Specification - Version 4," Information Sciences Institute, IEN 54, September 1978.

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.

APPENDICES

A. Reconnection Procedure

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 identifier's 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 procedure (called the Initial Connection Protocol or ICP [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 N1.T1.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 (N1.T1.Q,N3.T3.S). The serving process can SEND on (N3.T3.S,N1.T1.Q). There are many variations on this scheme, some involving the user process doing a RECEIVE on a different socket (e.g., (N1.T1.X,U.U.U)) with the server doing SEND on (N3.T3.S,N1.T1.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 Procedure Example

Figure 13.

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 (N1.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.