IEN-69

Jon Postel
ISI
9 October 1978

TCP Meeting Notes - 18 & 19 September 1978

Editors Note

This meeting was billed as a TCP testing session. The first hour or so of the meeting was taken up with a discussion of what the various implementors present expected their programs to be able to accomplish with the programs of the other implementors.

It became apparent that almost no useful demonstrations could be performed due to the differences between the implementations.

The planned agenda was discarded, and the meeting turned primarily into a discussion of the specifications of TCP with a few remarks about testing procedures.

Much of the detail of the discussion of the TCP specifications will be omitted from these notes, but the information will be conveyed to the specification editor for use in the next edition.

Discussion of the State of TCP Implementations

The TCP implementations available for testing at the meeting were:

SRI:	TCP 2.5
BBN Tenex/Tops 20:	TCP 2.X
UCLA:	TCP 4
BBN Unix:	TCP 2.5
MIT Multics:	not ready

The state of BBN's Tenex and Tops 20 versions was explored in some detail:

Version	Comments
2.5-e	Tested; BCPL, running at ISI; "old faithful"; has resynchronization ARQ, RSN, INT, etc.
2.5	Tested; BCPL & Monitor versions; running at SRI-KA; INT, RSN,ARQ removed, code cleaned up.
2.5+e	Tested; Hand coded; running at BBND, BBNC; bugs fixed.

2.5+2e	Untested; fixes installed to attempt cures for all reported bugs.
4.0	Untested; will have rubber EOL, fragmentation and reassembling (in internet module).

A schedule of when the represented sites would have TCP 4's ready for testing was made up:

SRI:	1 Oct	(Mathis)
UCLA:	now	(Braden)
BBN Unix:	9 Oct	(Haverty)
BBN Unix:	1 Nov	(Wingfield)
MIT-Multics:	16 Oct	(Clark)
BBN-Tenex:	?	(Plummer)

Information was collected on the parameters for testing the various TCP's

UCLA: ARPANET, IMP 01, HOST 1, LINK 155
port 1 talk program
contact: BRADEN@UCLA-CCN (213)825-7518

SRI: ARPANET, IMP 1, HOST 1, LINK 155
port 1 either a manually started traffic generator or a telnet server
contact: MATHIS@SRI-KL (415)326-6200

BBN-Unix: ARPANET, IMP 63, HOST 0, LINK 155
port 1 talk program or sever telnet
Echo Server
traffic generator run from exec after login from ARPANET via NCP

BBN-Tenex/Tops 20: ARPANET, IMP 49, HOST 3, LINK 155

port 23	server telnet
port 9	sink
port 19	source
port 7	echo
port 13	daytime

A traffic generator available in exec via login from NCP ARPANET connection.

- Status report from BBN Tenex/Tops 20, BBN Unix, MIT-Multics, UCLA, and CCA were circulated (see appendices).

Vint stressed the importance of the work on TCP and noted that TCP is a candidate for a DOD standard host-to-host protocol.

Early tests of TOPS 20 release 3 indicate that it is slower than release 1 by as much as 25% to 30%. That is, the system is saturated with about 3/4 as many jobs.

Arrangements will be made for login directories at BBNC, SRI-KA, and MIT-Multics for testing of TCPs. The login name will be "TCP-TEST." The password may be obtained from Vint.

Each implementor should prepare and circulate a one or two page "users guide" for the other implementors.

Discussion of the TCP Specification

Checksum

There was a good deal of discussion of what should be checksummed. It was concluded that for end-to-end reliability, the address fields, the protocol field, and the length of the TCP portion should be included in the checksum.

One way to think of this is to imagine a pseudo header prefixed to the TCP segment.

```
+------------------------+
!     SOURCE ADDRESS     !
+------------------------+
!   DESTINATION ADDRESS  !
+------------------------+
! ZERO ! PTCL ! TCP LENGTH !
+------------------------+
```

Note that the TCP length is a computed quantity which is not carried in any header field. It is the Internet Header total length field value (octets) minus the Internet Header IHL field (32 bit units) when the IHL is adjusted to octets. The TCP length does not include the length of the pseudo header.

There was some discussion of explicitly including a data length field as a hook to later provide a multiplexing function, but this was put off for consideration at the next Internet meeting.

Retransmission

There was a good deal of discussion of retransmission strategy and measures on which to base a dynamic strategy.

Various "backoff" parameters were described. The most elaborate is described in Plummer's memo on the TCP interface (see Appendix E).

There are two problems that have the same symptoms--losing segments due to poor communications media, and delayed segments due to congestion. These two problems call for distinctly different solutions. In a lossy network, one would retransmit more frequently; in a congested network, one would retransmit less frequently.

Refusing Connection

It is not ever made clear in the specification how to refuse a connection, or tell if a connection attempt was refused.

The following strategy is proposed:

If the connection is passive (i.e., listening)

When SYN arrives, send a SYN in response.

If a RST arrives, go back to listening.

If the connection is active (i.e., SYN-SENT)

If a RST arrives, the connection was refused.

To refuse a connection send a RST.

Maximum Segment Size

What is the maximum segment that a TCP or Internet module must be prepared to accept? This was discussed some, then put off to the Internet Meeting. TCP implementors are encouraged to go on record what the size they believe to be a reasonable upper bound. There was some call for a mechanism to dynamically negotiate the upper bound on segment size.

Urgent

Well, it was a TCP meeting so about half of the time was spent discussing URGENT.

The following rules seem to be the outcome:

1. The user must send data with the URGENT.

2. An EOL with URGENT ensures prompt delivery of the data to the user, but is not required.

3. The user is notified at the arrival of urgent data even if the user has no pending receive calls.

The urgent pointer points numerically to the octet just beyond the urgent data.

```
---+---+---+---+---+---+---+---
 U ! U ! U ! U ! R ! R ! R ! R
---+---+---+---+-o-+---+---+---
                  \
                   urgent pointer
```

Discussion of Higher Level Protocols

1. Buffer size - no higher level protocol shall be designed that requires a particular buffer size.

2. Telnet - same as with NCP (new telnet).
SYNCH is <IAC><DM> and urgent pointer.

3. FTP - issue for 3rd party transfers requires use of PASV and SOCK commands and passing socket number in response to PASV.

If user U wants to transmit a file from A to B

```
U->A   connect to socket 3
U->B   connect to socket 3
U->A   PASV
A->U   OK, my data socket is X
U->B   SOCK A,X
B->U   OK
U->A   STOR file
U->B   RETR file
B->A   connect my Y to your X
```

X should not be a well-known socket and should not have been previously used by the server choosing it during this session.

Tests to be conducted before the next TCP meeting:

1. Single connection. Open & close a single connection many times.

2. Multi connections. Open connections several simultaneously. Two connections to some socket (i.e., a-b and a-c) check proper separation of data.

3. Half Open Connection. Open a connection, crash local TCP and attempt to open same connection again.

4. Piggy-back Loop. Open connections via Telnet.

```
user telnet--->TCP--->TCP--->server telnet
                                  !
                                  V
server telnet<---TCP<---TCP<---user telnet
    !
    V
user telnet--->...
```

5. Maximum connections. Open connections between a pair of TCP until refused or worse.

6. Refused connection. Open a connection to a non-accepting socket, does it get refused?

7. Zero Window. Try to send data to a TCP that is presenting a zero window.

8. Fire Hose. Make many connections to data source ports (e.g., TTYTST at TENEX)

9. Urgent Test. Try to send data to a user program that only receives data when in urgent mode.

10. Kamikazi Segment. Send and Receive NASTYGRAMS. A NASTYGRAM is a segment with SYN, EOL, URG, and FIN on and carrying one octet of data.

11. Sequence Wraparound. Test proper functioning when sequence numbers (a) pass 2**31 (i.e., go from plus to "minus") and (b) pass 2**32 (i.e., go from 2**32-1 to 0).

12. Buffer size. With buffer size not equal to one, send data in letters of various sizes, use urgent occasionally.

13. Send a NASTYGRAM into a half open connection when the sequence number is about to wrap around.

Next Meeting:

Next meeting will be December 11 and 12, Monday and Tuesday, at ARPA.

Action Items:

Dave Clark will write something about URGENT.

Vint Cerf will review the TCP specification.

The implementors will provide documents on TCPs, including internet address, port number-function map, maximum segment size accepted, point of contact with telephone number and sndmsg address. Plummer, Wingfield, Mathis, Haverty, Grossman, Kou Mei, Braden, Clark.

Bill Plummer will provide old scenarios on SYN, etc.

Memos Distributed:

UCLA 360 TCP Status Report - Braden

MIT-Multics TCP Status Report - Clark

CCA TCP Status Report - Kou Mei

BBN-Unix TCP Status Report - Haverty

BBN-Tenex TCP Status Report - Plummer

BBN-Tenex TCP JSYS Calling - Plummer

Internet Type of Service to Arpanet Parameter Mappings - Postel

Internet Notebook Table of Contents - Postel

IEN Index - Postel

Attendees:

Name	Affiliation	Mailbox
Vint Cerf	ARPA	Cerf@ISIA
Jack Haverty	BBN	JHaverty@BBND
Bill Plummer	BBN	Plummer@BBNA
Mike Wingfield	BBN	Wingfield@BBNE
Ed Cain	DCEC	DECE-R850@BBNB
Ray McFarland	DOD	McFarland@ISIA
Danny Cohen	ISI	Cohen@ISIB
Jon Postel	ISI	Postel@ISIB
Dave Clark	MIT	Clark@MIT-Multics
Geoff Goodfellow	SRI	Geoff@SRI-KA
Jim Mathis	SRI	Mathis@SRI-KL
Bob Braden	UCLA	Braden@UCLA-CCN
Denis De La Roca	UCLA	Delaroca@UCLA-CCN
Yogen Dalal	XEROX	Dalal@PARC

APPENDICES

APPENDIX A

Braden

Implementation Status -- IBM 360 TCP
September 17, 1978

1. Philosophical Remarks

This implementation of the Internet and TCP protocols is designed to meet the following general objectives:

(a) operate within the existing NCP system job, sharing code and control-block formats wherever possible;

(b) be compatible at the system-call level with the existing user-level protocol modules;

(c) Implement the Internet protocol as a distinct layer, with interfaces designed to expedite the implementation of other higher-level internet protocols in addition to TCP;

(d) require minimum NCP resources when internet protocol is not in use.

2. Machine

IBM 360/370, with a Santa Barbara interface to the IMP.

3. Operating System

OS/MVT, Release 21.8, with the addition of several user-written Supervisor-call routines (including the Exchange program).

4. Implementation Language

BAL (IBM's macro assembly language)

5. Code Size

Internet Protocol Layer:	4K bytes.
TCP Protocol Layer:	8K bytes.
User-level protocol system-call interface routines:	1K bytes.

6. Buffer Space

   Receive: segment reassembly buffer=
   max segment size + 16 bytes.

   Send: 128 bytes per unacknowledged segment. Note: the actual data being sent is not counted here, as it occupies buffer space belonging to the appropriate user-level protocol module.

7. Connections Supported:

   Limited only by memory available in NCP job.

8. Cost Per Connection

   Approximately 200 bytes fixed (excluding per-session control blocks and user-level protocol work areas).

9. Delay Per Packet

   Unknown

10. Bandwidth

    Unknown

11. CPU Used

    Unknown

APPENDIX B

Clark

MULTICS TCP

Preliminary Summary
15 September 1978
David D. Clark

Philosophical Remarks

1. This version is not designed to optimize efficiency. It is intended as a preliminary realization to debug our understanding of TCP. (Efficiency should not be incompatible with this version though.)

2. This version has been designed to exist outside the kernel (to reduce our interaction with Honeywell). It can be moved in later if efficiency would be enhanced thereby.

Machine

Honeywell 68/80

Operating System

Multics

Implementation Language

PL/1

Code Size

(Preliminary) about 2K lines PL/1 (rough guess 10K words code)

Buffer Space Used

? (Dynamic allocation)

Number of Connections Supported

No significant limit I know of

Cost Per Connection

?

Delay Per Packet

?? (I will not measure this version)

Bandwidth

? (Same Comment)

CPU Utilization

? (Same Comment)

For TCP Testing

David D. Clark

Clark at MIT-Multics

(617)253-6003

User Interface Is Changing

APPENDIX C

Haverty

BBN Unix TCP (Arpa Version)

Contact: Jack Haverty
Bolt, Beranek, and Newman, Inc.
50 Moulton Street
Cambridge, Mass. 02138

JHAVERTY@BBND
-- or --
jfh@BBN-UNIX (host 63, octal 77)

(617) 491-1850 ext 133

I. Philosophical Remarks

This note describes two TCP implementations. The TCP 2.5 implementation has been operational for over 6 months. It consists of a core protocol-handling module based on the TCP11 program by Jim Mathis, encapsulated in a layer of code which interfaces to Unix. The TCP 4 implementation is based on the same structure as the 2.5 implementation, i.e. a core protocol handler of TCP11 descent, encapsulated in a Unix-interface layer. The user interface is architecturally similar, changed in detail for compatibility with TCP 4 and to improve performance.

The TCP 4 implementation of the Unix-specific layer is complete, up to the point of integration with the core protocol module. This will be done when the TCP11 code has been checked out during testing.

Testing of TCP 2.5 revealed several problems in performance, which have been traced to inefficiencies within the Unix system itself. In parallel with the TCP 4 implementation, the Unix resources which are used by the TCP are being reworked to improve performance. Only minimal tests on TCP performance have been done to date, pending the changes to the system to make such tests more realistic. The test results for the 2.5 implementation are discussed below.

The TCP implementations are fully described in the 'Unix TCP User's Guide', BBN Report # 3724, along with a definition of the user interface. The latter will be updated slightly to reflect the differences with TCP 4. The current version is available on request. An extract of the spec appears later.

The TCP is a single user-level process, which handles all TCP traffic for the system. It is logically a part of the operating system, but is implemented as a user-level process. Standard Unix was modified to support additional inter-process communication and control mechanisms to make this possible.

The interface to user processes is via 'ports', which are serial I/O paths between processes. This structure is somewhat different than the one assumed by the TCP design, where a user process passes 'buffers' to and from the TCP. The interface to the TCP is strictly speaking defined by the formats of the data passed on the ports. A library module for user processes, written in C, is included to present a subroutine-level interface to application programs.

The TCP 2.5 implementation has been reworked for TCP 4 to change the structure somewhat. Two basic areas were changed, primarily because of the 'buffer-passing' design of TCP11. The 2.5 implementation was built on top of this interface, and converted the buffers into a serial-stream model. The TCP 4 implementation is somewhat different, in that it can use the serial data stream itself as a kind of reassembly buffer. In this implementation, incoming data is often able to be passed directly to the user process by transferring it into the port, when the packets arrive in order. Reassembly per se is done only when needed because packets arrive out of order, and is handled in a fashion similar to fragment reassembly.

This latter change should have two major effects on the TCP 4 implementation. The avoidance of reassembly when packets arrive in order reduces processing time as well as delay in presenting data to the user. Out-of-order data is queued pending arrival of the missing pieces. The first missing piece, being in order, is passed directly to the user process, and then the other queued pieces are processed. The effect is that the user process gets the 'next' data as fast as possible, since it bypasses most of the processing in the 2.5 implementation which was building buffers of data.

II. Machine

The TCP is currently runnable on an 11/40, running a modified Unix. Development work is done on the BBN 11/70. The network interface code for the Unix system is being enhanced to permit non-NCP use of the network simultaneously with normal TCP operation. This addition will permit operation of the TCP for testing without disrupting normal system operation. We also intend to test the TCP in communication with the TCP being developed for the EDN (autodin testbed) network, by running both TCPs simultaneously on the same machine.

The 11/40 processor is not on the Arpanet directly, and can only be accessed through a gateway, which must be modified to handle the new internet header formats. The 11/70 processor will be directly accessible as Arpanet host 63.

III. Operating System

The system being used at BBN on the 11/40 is a descendant of a Rand version of Unix, modified at BBN to include new interprocess communication mechanisms which are required for the TCP operation.

The 11/40 system has been integrated with an NCP unix system, which has all of the enhancements, and is now running on the 11/70 system. TCP 4 is intended to run on this system, which will replace the current 11/40 system.

IV. Implementation Language

The TCP process is written in Macro-11. It uses the Harvard assembler, and a converter program to make the object files conform to Unix formats. The library module is written in C.

V. Code Size

The TCP process will use up to an entire 32K address space if desired. The basic code size of the 2.5 implementation is 3876 words. An additional 246 words of data is used by the raw TCP for buffers, etc. The remainder of a 32K address space can be split up into packet buffers and TCBs (1 per connection). Each TCB uses 1082 words, primarily because each contains 1024 words devoted to reassembly and retransmission buffers.

The TCP 4 implementation is not yet integrated with the new TCP11 code, so the exact code size is unknown. If the TCP11 code is comparable in size to the old implementation, the TCP will also require approximately 4K of code space. The remainder is available for buffers. In this implementation, no reassembly or retransmission buffers exist per se, but all dynamically used data is obtained from the buffer pool. Experimentation will be required to determine the appropriate ratio of buffers to connections in any particular. The TCP process will function with as few as 2 or 3 buffers, in the extreme case where only packets which arrive in order are accepted. More buffers can be specified when the TCP is started, to provide resources for reassembly of out-of-order arrivals.

VI. Buffer Space Used

As mentioned above, when th TCP acts simply as a pipeline, it will read a net packet, and immediately transfer its contents to the user process, using only the buffer(s) needed to hold the net message. More typically, the system should probably allocate at least 3 or 4 buffers per connection. Buffer sizes are a parameter, currently set fairly small (100 bytes) to handle interactive types of traffic. Bulk transfers are handled by use of chained multiple buffers. This can also be altered by parameter settings when the TCP is assembled.

VII. Number of Connections

TCP 2.5 was limited primarily in its use of reassembly and retransmission buffers. The number of connections supported could not exceed approximately 25, due to address space limitations.

The TCP 4 implementation uses a buffer pool, and hence does not reserve any buffers for each connection. The primary limitation here is the number of interprocess ports which the Unix system will permit between the TCP and its customer processes. A typical system might allow a similar number (25) of connections, but this is also a system parameter.

The implementation is designed to make addition of a feature which permits multiplexing of several TCP connections for a process on a single port relatively easy, should this ever be desirable.

VIII. Delay per Packet, Bandwidth, CPU Utilization

These tests have been performed only for the 2.5 implementation, and then only to the point of recognizing the problems inherent in the Unix support. The packet delay has been on the order of 300 to 500 milliseconds, or more, depending on the vagaries of the scheduler. (Note these times are all for the 11/40 based system). Bandwidth seems to top out at 5 kilobits/sec.

We performed one analysis of the system performance using the port mechanism, which the TCP uses to interface to user processes. These tests indicated a typical 125 millisecond delay in sending through a port, and a per-read/write overhead of 4.5 milliseconds, which is deadly to character-at-a-time I/O. The report compares use of ports, pipes, and disk I/O, for the 11/40. It is available on request.

Tests were also done with a typical 'user' process using the TCP to support several connections simultaneously, communicating with several other processors via the network. The CPU time spent in the two processes, in 'user code', and 'system code' is in the table below,. for two different test runs.

#	User	System
TCP	0.5	4.9
User	2.9	19.8
TCP	0.2	3.8
User	3.2	18.4

The ratios of user to system time clearly indicate the bottleneck in the system, which is now being reworked to improve the efficiency.

Extract from BBN report 3724, Unix TCP User's Guide

3.1 TCP←listen

The TCP←listen routine is used to establish contact with the TCP process. It sets up the command and response ports between the user process and the TCP process, without opening any connections. If called while the user process is already actively utilizing the TCP, it acts as a reset command, aborting any open connections before attempting to re-establish contact with the TCP process. TCP←listen must be successfully called before any TCP connections may be opened.

Calling Sequence:
TCP←Listen(tm);
where:
int tm;   /* timeout, in seconds */

Function:
Establishes contact between the user process and the TCP process. If called while TCP activity is in progress, resets by aborting any open connections.

Arguments:
tm -- timeout value, i.e. how long in seconds TCP←listen should wait for the TCP process to reply

Returns:
TRUE if successful, FALSE on error

Possible Error Codes:

EEARPRT[CXI]	-- open of TCP ear port failed
	-- message send failed
ERSPPRT[CXIV]	-- couldn't create response port
	-- error in read from response port
	-- bad response received
ENOBUFS[I]	-- couldn't allocate buffer
ECMDPRT[C]	-- unable to open command port
ETCPNRDY[X]	-- TCP not responding within time specified

3.2 TCP←quit

The inverse operation to TCP←listen is performed by the TCP←quit routine. TCP←quit will cleanly break the communication paths to the TCP. If any connections were still open, they will be aborted (see TCP←abort). This routine should be called when TCP activity is complete, in order to release the resources allocated within both the user process and the TCP process. Killing the user process will have the same effect as calling TCP←quit.

Calling sequence:
TCP←quit();

Function:
Finishes usage of the TCP by the user process. Releases all resources, and aborts any remaining connections.

Arguments:
None

Returns:
TRUE

Possible Error Codes:
None

3.3 TCP←open

The TCP←open routine is used to create a TCP connection. The parameters of the connection are specified by a connection address block structure, or cab. The contents of the structure specify the address of the foreign socket as well as the local address components which the user is permited to specify. A mask is also supplied to specify which events will trigger the issuance of change-notices, as discussed with the TCP←receive specifications.

If the connection is accepted by the TCP process, TCP←open will return a pointer to a structure termed a user transmission control block, or utcb, which is used to identify the connection for other routine calls. The connection itself is not necessarily 'established' in the TCP sense, since the call to TCP←open returns before the TCP process begins the handshake procedure with the foreign TCP.

```
Calling Sequence:
pt=TCP←Open(c, tm, f)
where:
struct cab *c ;          /* specifies connection */
int tm ;                 /* in seconds */
char f ;                 /* mode of opening */
struct utcb *pt ;        /* if successful */
```

Function:
Creates a TCP connection, obtaining a pointer to a utcb to be used for subsequent data transfers. Optionally begins establishment of the connection with the foreign TCP.

Arguments:
c -- a cab structure, as defined below
tm -- time to allow for the TCP to reply to the open request
f -- mode bits. Assigned bits are:
    01 -- open as a listen-only connection

Returns:
TRUE, pointer to utcb, FALSE on error

Possible Error Codes:

ETCPNRDY[X]	-- TCP took too long to respond
ECMDPRT[C]	-- error in use of command-port
ERSPPRT[CV]	-- error in use of response-port
ETCPBAD[V]	-- illegal message from TCP process (not conforming to the protocol)
EUNKR[V]	-- unknown type of message from TCP
ENOBUFS[IX]	-- either the user process or TCP process is short of buffer space
ESNDPRT[CIX]	-- couldn't open send-port
ERCVPRT[CIXV]	-- couldn't open receive-port
	-- error in reading from receive-port
	-- message received violates protocol
EUSRC[I]	-- too many connections in this process
ESYSC[X]	-- too many connections system-wide
ESYSF[X]	-- too many processes using TCP
EINUS[X]	-- connection already open by someone
EILLS[U]	-- illegal local port specification

The connection to be opened is specified by a cab (connection address block) structure, defined as follows.

```
struct  cab {
char    c←lph ;    /* local port, high byte */
char    c←xx1 ;    /* used internally */
int     c←lpm ;    /* local port, low 2 bytes */
char    c←fnid ;   /* destination net */
char    c←ftidh ;  /* destination TCP id high byte */
int     c←ftid ;   /* destination TCP id low 2 bytes */
char    c←fph ;    /* destination port high byte */
char    c←xx2 ;    /* used internally */
int     c←fpm ;    /* destination port low 2 bytes */
int     c←cnmask ; /* change-notice mask */
};
```

3.4 TCP←close

The TCP←close routine is used to initiate a close procedure. It will cause the TCP process to transmit a close request to the foreign TCP. The TCP←close routine will return as soon as the close command is accepted by the TCP process. The connection will remain open until the close procedure has been completed. The user process may wait for a change-notice to arrive which signals the completion of the close procedure. Attempts to send data will be ignored.

Calling Sequence:

```
TCP←close(pt,tm)
where
struct utcb *pt ;        /* from TCP←open */
int tm ;                 /* timeout */
```

Function:
Begins the close procedure.

Arguments:
pt -- pointer returned by TCP←open
tm -- timeout limit in seconds

Returns:
TRUE if successful, FALSE on error

Possible Error Codes:

ETCPNRDY[X]	--	TCP took too long to respond
ECMDPRT[C]	--	error in use of command-port
ERSPPRT[CV]	--	error in use of response-port
ETCPBAD[V]	--	illegal message from TCP process (not conforming to the protocol)
EUNKR[V]	--	unknown type of message from TCP
ENOBUFS[IX]	--	buffer shortage

3.5 TCP←abort

The TCP←abort routine, if called before a close procedure has completed, serves to destroy the specified TCP connection, without waiting for the foreign TCP to acknowledge. It also serves the purpose of cleaning up the tables and buffers which were allocated to the connection.

Calling Sequence:

```
TCP←abort(pt,tm)
where
struct utcb *pt ;        /* from TCP←open */
int tm ;                 /* timeout */
```

Function:
Destroys a TCP connection, and releases all resources assigned to the connection.

Arguments:
pt -- pointer obtained from TCP←open
tm -- timeout limit in seconds

Returns:
TRUE if successful, FALSE on error

Possible error Codes:

```
ETCPNRDY[X]   -- TCP took too long to respond
ECMDPRT[C]    -- error in use of command-port
ERSPPRT[CV]   -- error in use of response-port
ETCPBAD[V]    -- illegal message from TCP process
                 (not conforming to the protocol)
EUNKR[V]      -- unknown type of message from TCP
ENOBUFS[X]    -- buffer shortage
```

3.6 TCP←send

The TCP←send routine is used to send data over a TCP connection. It requires as an argument a pointer to a utcb, as returned by TCP←open. Data given to TCP←send is buffered internally for transmission to the TCP process. Therefore, TCP←send will actually do I/O to transfer data to the TCP process only when the internal buffer fills, or when it is instructed to send immediately. Large data transfers will be split into smaller chunks for transfer to the TCP process. The TCP←status routine can be used to obtain information concerning the buffer sizes being used.

Two switch parameters to the TCP←send routine are used to control initiation of data transfer. Either the 'go' or the 'eol' flag will cause the transfer of the current data buffer within the user process to the TCP process even if the buffer is not full. The 'eol' flag is also used to indicate that the last byte of the supplied data should also be marked as end-of-letter.

```
Calling Sequence:
TCP←send(pt, b, l, go, eol);
where
struct utcb *pt ;   /* from TCP←open */
char *b ;           /* pointer to data */
int l ;             /* length of data in bytes */
int go ;            /* 'go' flag */
int eol ;           /* 'eol' flag */
```

Function:
Send a data stream over a TCP connection.

Arguments:
pt -- pointer from TCP←open
b -- pointer to byte stream to be sent
l -- number of bytes to send
go -- boolean, if true, send immediately
eol -- boolean, if true, indicate eol
(also send immediately)

Returns:
TRUE if successful, FALSE on error

Possible error codes:
ESNDPRT[C] -- The communication path to the TCP process for this connection is failing.
ENOBUFS[I] -- No internal buffers are available within the user process.

The flag arguments follow standard UNIX conventions for boolean values. TRUE implies that the flag is set.

3.7 TCP←receive

The TCP←receive routine performs the inverse of TCP←send. The user supplies TCP←receive with a buffer area, into which the incoming data stream will be placed. TCP←receive accepts data from the TCP process in packages, transfers it into the user-supplied area until either it is filled, or an incoming end-of-letter is encountered, or no more data is available.

In addition to data, TCP←receive processes change-notices received from the TCP process, and passes these on to the user. A change notice is simply a word of data which encodes the occurrence of several possible state changes of the connection, such as connection-established, network-down, remote-close-received, and so on. The user process may use this information as desired. One possible usage would be to trigger calls to TCP←status when some change-notice is received, to obtain more detailed information. An enumeration of the possible change-notices is in Appendix D. The set of state changes which to TCP←open for the connection in question.

Calling Sequence:

```
cnt=TCP←receive(pt, b, l, eol, flg)
where
struct utcb *pt ;  /* from TCP←open */
char *b ;          /* buffer area */
int l ;            /* maximum buffer length */
int *eol ;         /* set to indicate e-o-l */
int *flg ;         /* set to indicate type */
int cnt ;          /* bytes received, if flg=0 */
```

Function:
Receive a data stream from a TCP connection, or a change-notice concerning the connection state.

Arguments:
pt -- pointer from TCP←open
b -- pointer to area in which to place data bytes
l -- maximum number of bytes to be transferred
eol -- pointer to integer, to be set to indicate
end-of-letter.
flg -- pointer to integer, to be set to indicate whether
data
        or a change notice is returned
cnt -- number of bytes transferred if flg indicates data was
received

Returns:
TRUE if something received:
if flg is FALSE, data was received, cnt is count of bytes placed into the user buffer
if flg is TRUE, a change notice was received, and flg is the change-notice value. No data was placed in the buffer.
FALSE if error, or if no data was received. In the latter case, the error code will be zero.

Possible error codes:
ERCVPRT[C] -- i/o error on port from TCP
ETCPBAD[V] -- unknown type of message from the TCP

APPENDIX D

Plummer

TCP for TENEX and TOPS20

There are two TCP implementations for TENEX. One is written in BCPL and runs as a normal user mode program. The other is hand coded in MACRO-10 assembly language and is part of the monitor. The hand coded version may be run as part of either the TENEX monitor or TOPS20 monitor on 1090T, Model A or Model B KL20 processors running release 101B or 3A of the TOPS20 monitor, or the 2020 running TOPS20. The BCPL version of the TCP runs only under TENEX, however.

The hand coded TCP was written in order to obtain higher performance. Code produced by the BCPL compiler was estimated to be roughly five times as large as well structured hand code. With fewer instructions to execute, the assembly language version should see a comparable decrease in execution time. As it turned out, the BCPL TCP has more than 24,000 (decimal) instructions and the hand coded version has fewer than 5000. The execution time of a standard benchmark program saw a reduction of a factor of eight in execution time. Some of this improvement was due to the fact that the overhead associated with JSYS traps and the network interface are not present in the monitor version.

The benchmark program is TCPTST, a program which sends 1000, single-byte messages to itself. Since this program keeps eight buffers outstanding on both the send and receive sides, somewhere between 125 and 1000 acknowledgment packets will be processed in the reverse direction. Three hardware configurations are available: sending to self via the IMP, via a loop back plug on a special "Raw Packet Interface" (an 1822 meant to connect to a remote gateway machine), or using a software bypass which avoids interfaces for packets destined for "this" host. The benchmark timings on BBND (1090T, TOPS20 101B) for these three paths are 45, 30, and 11 seconds respectively. On SRI-KA TENEX the Raw Packet Interface is not available, but using the IMP required 55 seconds and the internal bypass 38 seconds. The BCPL TCP always uses the IMP and took 438 seconds to complete on SRI-KA TENEX.

Informal measurements in the past have indicated that amount of data in the packets does not strongly affect the processing time. Rather, it is the processing of the headers which consumes most of the per-packet processing time. Some of the functions performed in this respect are assigning and releasing storage used for the packet, composing the headers, inserting and checking the checksum, sequencing the packets when received, and generating and processing acknowledgements. The actual data transfer does not seem to be a limiting factor at present.

There is speculation that there is still more speed up to be had. While the test program is running, the lights on the computer seem

to indicate that it is not being fully consumed. Thus, the TCP is occasionally waiting for some event. It might be that packets are being thrown away for internal reasons and that a retransmission is required to restart the packet flow. More thorough metering is planned so that efficiency issues such as the above may be investigated.

The amount of buffer space available to the TCP is an assembly time parameter. The BCPL version is typically generated with 16,000 (decimal) words of free storage. The BBN-TENEX assembly specifies the same amount while the version on SRI-KA has somewhat more than 6000 words. BBND has only 1000 words due to the shortage of monitor address space in the 101B release of TOPS20. (The address space is consumed by tables to keep track of the relatively large disk system and the 512K memory system.) The release 3A monitor will permit the TCP to have almost all of a full 256K address space for its free area.

Roughly, the amount of storage S required to support C connections is S = 500 + 1000*C. This estimate assumes several conditions: First, only TELNET connections are being supported; that there is only one listening (server) connection; that the connections are fully active in the "normal" pattern; and that the process is not the limiting factor. Thus, a larger number of connections can be supported if only sporadic typing is involved, but a fewer number if there are large file transfers in progress. The TCP will permit more connections than the formula predicts but performance will be degraded if they simultaneously demand maximum resources.

Contact for TCP Testing:

William W. Plummer (PLUMMER@BBN) Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, MA. 02138

(617) 491-1850 ext. 234

APPENDIX E

Plummer

In the following, a "JCN" may be thought of much as a JFN is for TENEX files. A "Connection Block" (referred to below) is currently a 3-word block:

Word-0:	24-bit Local Port
Word-1:	8-bit Foreign Network and 24-bit Foreign Host
Word-2:	24-bit Foreign Port

(N.B. Version 3 TCP may have difference in addresses.)

All JSYS's take flags in the left half of AC 1.
Not all JSYS's look at all of the flags. Flag bits are:

Bit-0: RH has JCN rather than pointer to connection block
Bit-1: Wait for the JSYS to complete.
Bit-5: ForceSync -- cause SYN to be sent when OPEN executed.
Bit-6: Persist -- keep resending SYN packet
Bit-7: Return statistics (STAT call only)

Some JSYSs take a "Retransmission Parameters" word. This is controls the retransmission function. The right half is the initial retransmission interval which is to be used. If the right half is 0, the initial interval will be computed based on the measured round trip time. The left half of the parameters control word has two 9-bit quantities. In computing the next retransmission interval from the previous one, the TCP multiplies by the number in the leftmost 9 bits and then divides by the number in the next 9-bit byte. Common backoff functions are:

SRI PR demo: Numerator=1, Denominator=1, Initial Interval=3.

(3 seconds constant retransmission interval with no backoff)

BBN (vanilla): Numerator=3, Denominator=2, Initial interval=0.

(Used in "average" conditions involving congested gateways and few dropped packets. 150% backoff from best guess initial interval).

BBN (old): Same as above but 200% backoff.

Quickly hits the 1 minute maximum interval and turns into slow, constant period retransmission).

OPEN (JSYS 742)

1/ Flags,,Pointer-to-Connection-Block
2/ Persistence in seconds
3/ Retransmission parameters

OPEN

R+1: failure, code in AC1
R+2: OK, useable handle (a JCN, Job Connection Number) in 1

Flags:

ForceSync: On to force synchronization without any data having been sent.

Wait: Don't return until connection is opened.

Persistent: Keep trying by sending SYN packets periodically.

CLOSE (JSYS 743)

1/ Flags,,JCN-or-Pointer

CLOSE

R+1: failure, code in AC1
R+2: OK, connection fully closed.

Flags:

JCNSupplied: On if RH of 1 has a JCN. Off if RH has Pointer-to-Connection Blk.

Wait: Wait for close to happen in both directions.

SEND (JSYS 740)

1/ Flags,,JCN or Pointer-to-Connection-Block
2/ 0,,Pointer-to-Data-Ring
3/ TimeOut in Seconds (0 for infinite)
4/ Retransmission parameters

SEND

R+1: failure, error code in 1
R+2: OK, JCN in 1

Flags:

JCNSupplied: (see above)

Wait: (see above)

Data Buffer Ring Format (SEND, RECV):

Word-0: Flags,,unused (typically ptr to next buffer header)
Word-1: 0,,Address of data buffer
Word-2: Word/Byte count for this buffer

Flags:

Done: Cleared when TCP receives this buffer. Set when TCP has finished with it.

Error: Buffer has an error in it.

EOL: Send an end-of-letter with this buffer. Or, end-of-letter received with this buffer.

WordMode: Buffer is formatted as 36-bit bytes. Off if buffer has four 8-bit bytes per word.

RECV (JSYS 741)

(call is same as SEND, but AC3 and AC4 are ignored).

STAT (JSYS 745)

1/ Flags,,JCN or Pointer-to-Connection-Block
2/ -N,,Offset into TCB
3/ -M,,Address in user's space

STAT

R+1:	failure, error code in 1
R+2:	OK. Min(M, N) words have been transferred from the TCB to the caller's space. The TCB offset identifies where the transfer starts and the Address in user space identifies the start of the destination area.

Flags:

JCNSupplied (see above)
Returns statitics: This flag causes the TCP to dump words from the statistics area rather than a specific TCB. Thus, the JCN is irrelevant. The Source and Destination ACs are updated as if a TCB were being dumped.

CHANL (JSYS 746)

1/ Flags,,JCN or Pointer-to-Connection-Block
2/ Six 6-bit bytes (channel numbers)

CHANL

R+1:	failure, error code in 1
R+2:	OK. This fork will receive TCP PSIs.

Flags:

JCNSupplied	(See above)

Each of the 6-bit bytes may be 77 (octal) if no PSIs are desired for the corresponding event.

Bits 0- 5:	INTRP channel
Bits 6-11:	RECV buffer done
Bits 12-17:	SEND buffer done
Bits 18-23:	Error
Bits 24-29:	State change (open or close)
Bits 30-35:	EOL acknowledged. (Not implemented)

Note: PSIs for the above may be dropped or be VERY tardy. Some serious defensive programming is required to guard against these problems. See TCPTST.MAC.

ABORT (JSYS 747)

1/ Flags,,JCN or Pointer-to-Connection-Block

ABORT

R+1: error, code in 1
R+2: OK, connection deleted

Flags:

JCNSupplied: (see above)

The local end of the connection is forgotten. An attempt to notify the remote end is made by sending a RST packet. Should this not be delivered, the other end will discover its half open connection the next time it attempts to use it.

Differences between "Monitor" TCPs on TENEX and TOPS20 and user mode (BCPL) TCP on TENEX.

1. ABORT JSYS

ABORT is JSYS 733 with the user mode TCP. JSYS 747 does something different.

2. STAT

The ability to dump the main statistics area is not implemented with the user mode TCP. Also, the STAT call takes different arguments: AC3 is ignored while AC2 should contain an AOBJN pointer (-N,,address) describing the space being provided by the user for accepting a copy of the TCB (starting at relative 0 in that TCB).

3. Retransmission control

The old TCP provides no facility for controlling the initial retransmission interval or the backoff function.

4. No INTRP

The Monitor TCP does not have the INTRP (interrupt) facility. This was never checked fully on the older TCP since it was never used and will be replaced by the URGENT scheme in version 3 of TCP.

APPENDIX F

Chuang

CCA's plans are to convert the SRI TCP11 version 4 to operate under RSX11M. Preliminary strategy was mapped out using version 2.5 as a guide, but actual conversion will commence when Jim Mathis is satisfied with his implementation of version 4.

CCA is also installing a "read and relay" facility on its PSMF on the ARPAnet. This facility will permit the recording of a bidirectional, raw, time-stamped packet stream between two hosts using any protocols. The recording may later be retrieved or analyzed. The Record and Relay facility will be described more fully at the October Internet meeting.

Kou-Mei Chuang