A Prototypical Implementation of the NCP

While involved in attempting to specify the formal protocol, we also attempted to formulate a prototypical NCP in an Algol-like language. After some weeks of concentrated effort, the project was abandoned as we realized that the code was becoming unreadable. We still, however, felt the need to demonstrate our conception of how an NCP might be implemented; we felt that this would help suggest solutions for problems that might arise in trying to mold the formal specifications into an existing system. This document is that attempt to specify in a prose format what an NCP could look like.

There are obvious limitations on a project of this nature. We do not, and cannot, know all of the quirks of the various systems that must write an NCP. We are forced to make some assumptions about the environment, system calls, and the like. We have tried to be as general as possible, but no doubt many sites will have completely different ways of conceptualizing the NCP. There is great difficulty involved in conveying our concepts and the mechanisms that deal with these concepts to people who have wholly different ways of looking at things. We have, however, benefited greatly by trying to actually code this program for our fictitious machine. Many unforeseen problems surfaced during the coding, and we hope that by issuing this document we can help to alleviate similar problems which may arise in individual cases.

There is, of course, absolutely no requirement to implement anything which is contained in this document. The only rigid rules which an NCP _must_ conform to are stated in NWG/RFC#54. This description is intended only as an example, _not_ as a model.

In the discussion which follows we first describe the environment to be assumed and postulate a set of system calls. We discuss the overall architecture of the NCP and the tables that will be used to hold relevant information. Narratives of network operations follow. A state diagram is then presented as a convenient method for conceptualizing the cause-effect sequencing of events. The detailed processing of each type of network event (system calls or incoming network messages) is then discussed.

II. Environment

We assume that the host will have a time-sharing operating system in which the CPU is shared by processes.

We envision that each process is tagged with a user number. There may be more than one process with the same user number; if so, they should all be cooperating with respect to using the network.

We envision that each process contains a set of ports which are unique to the process. These ports are used for input to or output from the process, from or to files, devices, or other processes.

We also envision that a process is not put to sleep (i.e., blocked or dismissed) when it attempts to LISTEN or CONNECT. Instead it is informed when some action is complete. Of course, a process may dismiss itself so that it wakes up only on some external event.

To engage in network activity, a process attaches a local socket to one of its ports. Sockets are identified by user number, host and AEN; a socket is local to a process if the user numbers of the two match and they are in the same host. Thus, a process need only specify an AEN when it is referring to a local socket.

Each port has a status which is modified by system calls and concurrent events outside the process (e.g., a 'close connection' command from a foreign host). The process may look at a port's status as any time (via the STATUS system call).

We assume a one-to-one correspondence between ports and sockets.

III. System Calls

These are typical system calls which a user process might execute.

We use the notation

SYSCALL (ARG1, ARG2....)

where
SYSCALL is the name of the system call
and
ARGk, etc. are the parameters of the system call.

    CONNECT (P, AEN, FS, CR)

        P   specifies a port of the process
        AEN   specifies a local socket; the user number and host are
                  implicit
        FS   specifies a socket with any user number in any hose,
                  and with any AEN
        CR   the condition code returned

    CONNECT attempts to attach the local socket specified by AEN to
    the port P and to initiate a connection with a specific foreign
    socket, FS. Possible values of CR are:

        CR=OK   The CONNECT was legal and the socket FS is being
                           contacted. When the connection is established
                           or refused the status will be updated.

        CR = BUSY   The local socket is in use (illegal command
                           sequence).

        CR = BADSKT   The socket specification was illegal.

        CR = NOROOM   Local host's resources are exhausted.

        CR = HOMOSEX   Incorrect send/receive pair

        CR = IMP DEAD   Our imp has died

        CR = LINK DEAD   The link to the foreign host is dead because:
                           1. the foreign Imp is dead,
                           2. the foreign host is dead, or
                           3. the foreign NCP does not respond.

    LISTEN (P, AEN, CR)

        P   specifies a port of the process
        AEN   specifies a local socket
        CR   the condition code returned

    The local socket specified by AEN is attached to port P. If there
    is a pending call, it is processed; otherwise, no action is taken.
    When a call comes in, the user will be notified. After examining
    the call, he may either accept or refuse it. Possible values of
    CR are:

        CR = OK   Connection begun, listening

        CR = BUSY

            CR = NOROOM

            CR = IMP DEAD

            CR = LINK DEAD

    ACCEPT (P, CR)

            P   specifies a port of the process
            CR   the condition code returned

        Accept implies that the user process has inspected the foreign
        socket to determine who is calling and will accept the call.
        (Note: an interesting alternative defines ACCEPT as the implicit
        default condition. Thus any incoming RFC automatically satisfies
        a LISTEN.) Possible values of CR are:

            CR = BADSKT

            CR = NOROOM

            CR = IMP DEAD

            CR = LINK DEAD

            CR = BADCOMM   Illegal command sequence. (E.g., Accept issued
                            before a LISTEN.

            CR = PREMCLS   Foreign user aborted connection after RFC was
                            locally received but before Accept was executed.

    TRANSMIT (P, BUFF, BITSRQST, BITSACC, CR)

            P   specifies a port of the process
            BUFF   specifies the text buffer for transmission
            BITSRQST specifies the length to be transmitted in bits
            BITSACC returns the number of bits actually transmitted
            CR   the condition code returned

        Transmission takes place.   Possible values for CR are:

            CR = OK

            CR = IMP DEAD

            CR = LINK DEAD

        CR = NOT OPEN Connection is not open (illegal command
                      sequence).

        CR = BAD BOUND BITSRQST out of bounds (e.g., for a receive
                      socket BUFF was shorter than BITSRQST
                      indicated).

INT (P, CR)

    P   specifies the local socket of this process
    CR   the condition code returned

The process on the other (foreign) side of this port is to be
interrupted. Possible values of CR are:

    CR = OK

    CR = BADSKT

    CR = BADCOMM

    CR = IMP DEAD

    CR = LINK DEAD

STATUS (P, RTAB, CR)

    P   specifies a port of this process
    RTAB   the returned rendezvous table entry
    CR   the condition code returned

The relevant fields of the rendezvous table entry associated with
this port are returned in RTAB. This is the mechanism a user
process employs for monitoring the state of a connection.
Possible values of CR are:

    CR = OK

    CR = BADSKT

CLOSE (P, CR)

> P   specifies a port of this process
> CR   the condition code returned

Activity on the connection attached to this port stops, the
connection is broken and the port becomes free for other use.
Possible values of CR are:

CR = OK

CR = BADSKT

CR = BADCOMM

CR = IMP DEAD

CR = LINK DEAD

IV. The NCP - Gross Structure

We view the NCP as having five component programs, several
associative tables, and some queues and buffers.

The Component Programs (see Fig. 4.1)

1. The Input Handler

   This is an interrupt-driven routine. It initiates Imp-to-Host
   transmission into a resident buffer and wakes up the input
   interpreter when transmission is complete.

2. The Output Handler

   This is an interrupt-driven output routine. It initiates Host-
   to-Imp transmission out of a resident buffer and wakes up the
   output scheduler when transmission is complete.

3. The Input Interpreter

   This program decides whether the input is a regular message
   intended for a user, a network control message, an Imp-to Host
   message, or an error. For each class of message this program
   invokes a subroutine to take the appropriate action.

4. The Output Scheduler

   Three classes of messages are sent to the Imp

        (a) Host-to-Imp messages
        (b) Control messages
        (c) Regular messages

   We believe that a priority should be imposed among these
   classes. The priority we suggest is the ordering above. The
   output scheduler selects the highest priority message and
   passes it to the output handler.

   Host-to-Imp messages are processed first come first served.
   Control messages are processed individually by host, each host
   being taken in turn. A control message queue for each foreign
   host is provided. When any particular host is scheduled for
   output, as many control commands for that host as will fit are
   concatenated into a single message. Regular messages are
   processed in groups by host and link, each unique combination
   being taken in turn.
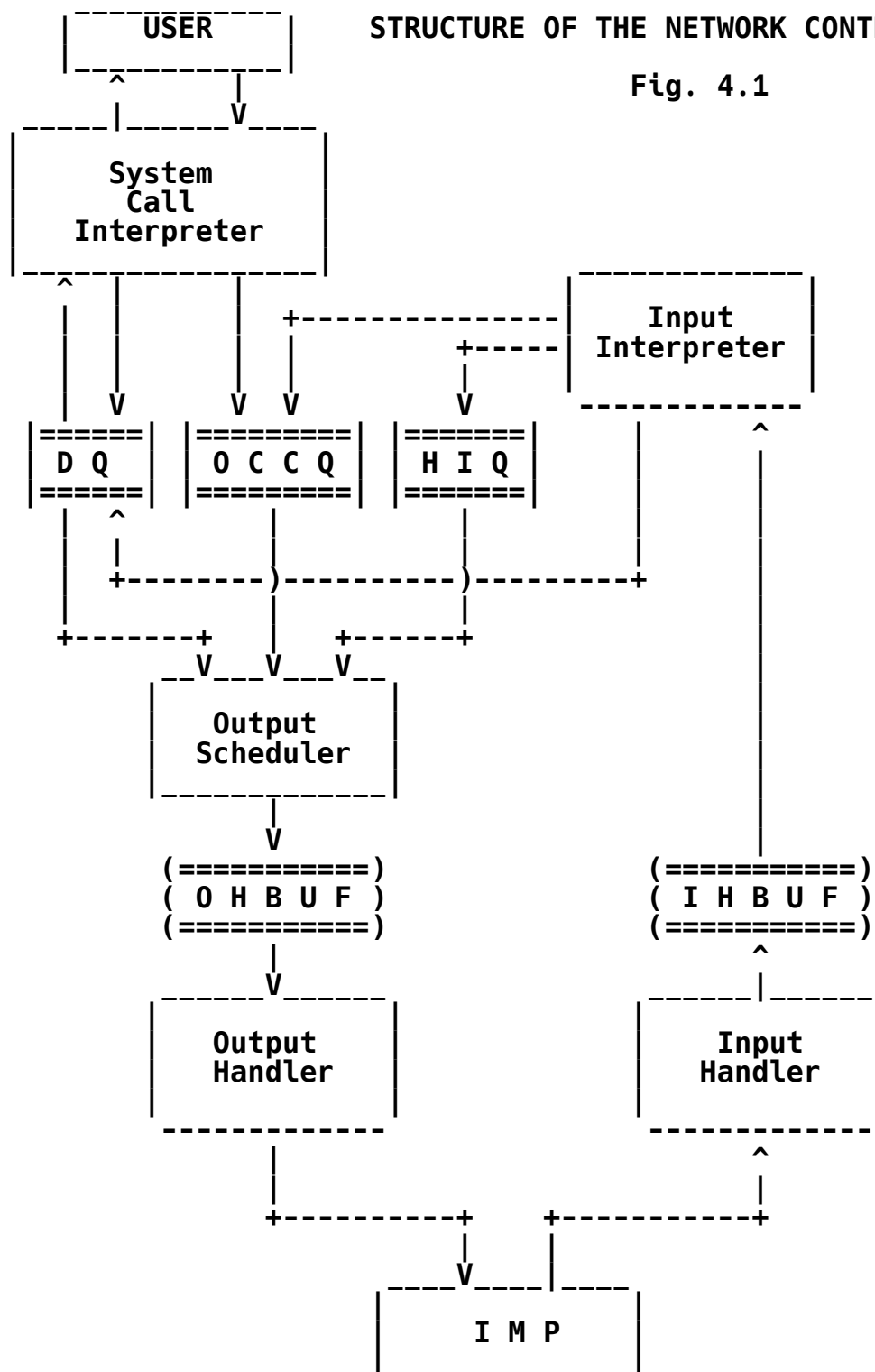
5. The System Call Interpreter

   This program interprets requests from the user. Each system
   call has a corresponding routine which takes the appropriate
   action.

The two interesting components are the input interpreter and the
system call interpreter. These are similar in that the input
interpreter services foreign requests and the system call
interpreter services local requests.

The diagram in Figure 4.1 is our conception of the Network
Control Program. Squishy amoeba-like objects represent component
programs, cylinders represent queues, and the arrows represent
data paths. In this simplified diagram tables are not shown.
["Amoeba-like" objects in original hand drawing are now firm
rectangular boxes: Ed.]

The abbreviated labels in the figure have the following meanings:

        HIQ   -   Host-to-Imp Queue
        OCCQ   -   Output Control Command Queue
        DQ   -   Data Queue
        IHBUF   -   Input Handler Buffer
        OHBUF   -   Output Handler Buffer

```
 _____          STRUCTURE OF THE NETWORK CONTROL PROGRAM
|    USER     |
|_____|                        Fig. 4.1
   |^     |
 __|_|____|V_____
|                 |
|     System      |
|      Call       |              _____
|   Interpreter   |             |              |
|_____|             |    Input     |
  |^   |    |    |  +--------------|  Interpreter |
  | |  |    |    | +-----|        |              |
  | |  |    |    | |     |        |_____|
  | |  |    |    | |     |              ^
  | V  V    V    V V     V              |
 |======| |========| |======|          |
 |  D Q | | O C C Q| | H I Q|          |
 |======| |========| |======|          |
   |       ^     |      |               |
   |       |     |      |               |
   |    +--------)------)---------+
   |    |        |      |         |
   |    |        |      |         |
 +-------+       | +------+
 |       |       V V V  V__
 |    ___V___V___V__
 |   |              |
 |   |    Output     |
 |   |   Scheduler   |
 |   |_____|
 |          |
 |          V
 |   (==========)         (==========)
 |   ( O H B U F )        ( I H B U F )
 |   (==========)         (==========)
 |          |                  ^
 |          V                  |
 |   _____     _____
 |  |              |   |             |
 |  |    Output     |   |    Input    |
 |  |   Handler     |   |   Handler   |
 |  |_____|   |_____|
 |          |                  ^
 |          |                  |
 |   +---------+    +----------+
 |   |         |    |          |
          V____|____|
         |_____|
         |             |
         |    I M P     |
         |_____|
```

V. Tables in the NCP

   We envision that the bulk of the NCP's data base is in associative
   tables. By "associative" we mean that there is some lookup routine
   which is presented with a key and either returns successfully with a
   pointer to the corresponding entry, or fails if no entry corresponds
   to the key. The major tables are as follows:

   1. The Rendezvous Table

      This table holds the attributes of a connection. The table is
      accessed by the local socket, but other tables may have
      pointers to existing entries.

      The components of an entry are:

         (a) Local Socket
         (b) Foreign Socket
         (c) Link
         (d) Connection State
         (e) Flow State
         (f) Data Queue
         (g) Call Queue
         (h) Port Pointer
         (i) Their Buffer Size (only needed on the send side)
         (j) Error State

      An entry is created when either a CONNECT or a LISTEN system
      call is executed or when a request for connection is received.
      Various fields remain unused until after the connection is
      established.

   2. The Input Link Table

      The input interpreter uses the concatenation of the foreign
      host and link as a key into the input table. The table is used
      in processing a user-destined message on an incoming link by
      providing a pointer into the rendezvous table.

   3. The Output Link Table

      The input interpreter uses the output link table to access the
      flow state as RFNM's return from transmitted messages. The
      output link table is keyed by host and link and provides a
      pointer into the rendezvous table.

4. The Port Table

   The system call interpreter uses the concatenation of the
   process identification and the port identification as a key to
   obtain a pointer into the rendezvous table.

5. The Output Control Command Table

   The system call interpreter and the input interpreter use this
   table to make entries in the appropriate output control command
   queues. Commands are queued in separate table entries
   corresponding to foreign hosts. Before output the contents of
   the queue are concatenated into a large control message. The
   components of an entry are:

       (a) Host
       (b) Output Control Command Queue

6. The Output Request Queue

   This queue contains an entry for each connection which has data
   requiring transmission to the net. There is only one entry per
   connection, which is deleted when the last packet of data is
   transmitted and is entered whenever a user makes a system
   request for data transmission.

   The entry is re-inserted if transmission is not completed
   (message too long) or is prevented by the flow control
   mechanism. The only component of an entry is a local socket.

7. The Host Live Table

   This is a simple table listing the hosts which are alive. This
   table is checked before establishing a connection and before
   sending any data to ensure that the destination host actually
   exists. At present the protocol does not define the procedure
   to be followed for the Host up/Host down conditions. See
   NWG/RFC#57.

8. The Link Assignment Table

   Link numbers are assigned by the receiver. This table records
   which links are free and can, therefore, be assigned.

VI. Informal Description of Network Operations

   We present here narratives describing the operation conducted during
   the three major phases of network usage: opening, flow control, and
   closing.

   A. Opening

      In order to establish a connection for data transmission, a pair
      of RFC's must be exchanged. An RTS must go from the receive-side
      to the send-side, and an STR must be issued by the send-side to
      the receive-side. In addition, the receive-side, in its RTS, must
      specify a link number. These RFC's (RFC is a generic term
      encompassing RTS and STR) may be issued in any time sequence. A
      provision must also be made for queuing pending calls (i.e., RFC's
      which have not been dealt with by the user program). Thus, when a
      user is finished with a connection, he may choose to examine the
      next pending call from another process and decide to either accept
      or refuse the request for connection. A problem develops because
      the user may not choose to examine his pending calls; thus they
      will merely serve to occupy queue space in the NCP. Several
      alternative solutions to this problem will be mentioned later.

      Utilizing the framework of the prototype system calls described
      above, we envision at least four temporal sequences for obtaining
      a successfully opened connection:

         1. The user may issue a LISTEN, indicating he is willing to
            consider connecting to anyone who sends him an RFC. When an
            RFC comes in the user is notified. The user then decides
            whether he wishes to connect to this socket and issues an
            ACCEPT or a CLOSE on the basis of that decision. A CLOSE '
            refuses' the connection, as discussed under "Closing." An
            ACCEPT indicates he is willing to connect; an RFC is issued,
            and the connection becomes fully opened.

         2. Upon processing a user request for a LISTEN, the NCP
            discovers that a pending call exists for that local socket.
            The user is immediately notified, and he may ACCEPT or
            CLOSE, as above.

         3. The user issues a CONNECT, specifying a particular foreign
            socket that he would like to connect to. An RFC is issued.
            If the foreign process accepts the request, it answers by
            returning an RFC. When this acknowledging RFC is received,
            the connection is opened.

4. When presented with a CONNECT, the NCP may discover that a
   pending call exists from the specified foreign socket to the
   local socket in question. An acknowledging RFC is issued
   and the connection is opened.

In all of the above cases the user is notified when the connection
is opened, but data flow cannot begin until buffer space is
allocated and an ALL command is transmitted.

Any of these connection scenarios will be interrupted if a CLS
comes in, as discussed under "Closing."

1. Pending Call Queues

   It is essential that some form of queuing for pending RFC's
   be implemented. A simple way to see this is to examine a
   typical LISTEN-CONNECT sequence. One side issues a LISTEN,
   the other a CONNECT. If the LISTEN is issued before the RFC
   coming from the remote CONNECT arrives, all is fine.
   However, due to the asynchronous nature of the net, we can
   never guarantee that this sequence of events will occur. If
   calls are not queued, and the RFC comes in before the LISTEN
   is issued, it will be refused; if it arrives later, it will
   be accepted. Thus we have an extremely ambiguous situation.

   Unless one has infinite queue space, it is desirable that
   some mechanism for purging the queues of old RFC's which the
   user never bothered to examine. An obvious but informal
   method is to note the time when each RFC is entered into the
   queue, and then periodically refuse all RFC's which have
   exceeded some arbitrary time limit. Another thought, which
   probably should be included within the context of any
   scheme, is for the NCP to send a CLS on all outstanding
   connections or pending calls when a user logs out or blows
   up.

   The scheme which is utilized in this description may seem at
   first blush to be non-intuitive; but we feel it is more
   realistic than other proposals. Basically, when a CONNECT
   is issued, the NCP assumes that this socket wishes to talk
   to the specified foreign socket and to that socket only. It
   therefore purges from the pending call queue all non-
   matching RFC's by sending back CLS's. Similarly, when the
   connection is in the RFC-SEND state (a CONNECT has been
   issued), all non-matching RFC's are refused. If a LISTEN-
   ACCEPT or LISTEN- CLOSE sequence is executed, the remainder

of the pending calls are not removed from the queue, in the
expectation that the user may wish to accept these requests
in the future.

Although the latter method may seem to be arbitrary and/or
unnecessarily restrictive, we have not yet concocted a
scenario which would be prohibited by this method, assuming
that we are dealing with a competent programmer (i.e., one
who is wary of race conditions and the asynchronous nature
of the net). Of course whatever scheme or schemes a
particular site chooses is highly implementation dependent;
we suggest that some provision for the queuing of RFC's be
provided for a period of time at least of the order of
magnitude that they are retained in the CONNECT-clear scheme
mentioned above.

B. Flow Control

   Meaningful data can only flow on a connection when it is fully
   opened (i.e., two RFC's have been exchanged and closing has not
   begun). We assume that the NCP's have a buffer for receiving
   incoming data and that there is some meaningful quantity which
   they can advertise (on a per connection basis) indicating the size
   message they can handle. We further assume that the sending side
   regulates its transmission according to the advertisements of that
   size.

   When a connection is opened, a cell (called 'Their Size') is set
   to zero. The receive-side will decide how much space it can
   allocate and send an ALL message specifying that space. The
   send-side will increment 'Their Size' by the allocated space and
   will then be able to send messages of length less than or equal to
   'Their Size' When messages are transmitted, the length of the
   message is subtracted from 'Their Size'. When the receive-side
   allocates more buffer space (e.g. when a message is taken by the
   user, thus freeing some system buffer space), the number of bits
   released is sent to the send-side via an ALL message.

   Thus, 'Their Size' is never allowed to become negative and no
   transmission can take place if 'Their Size' equals zero.

   Notice that the lengths specified in ALL messages are increments
   not the absolute size of the receiving buffer. This is
   necessitated by the full duplex nature of the flow control
   protocol. The length field of the ALL message can be 32 bits long
   (note: this is an unsigned integer), thus providing the facility
   for essentially an infinite "bit sink", if that may ever be
   desired.

C. Closing

   Just as two RFC's are required to open a connection, two CLS's are
   required to close a connection. Closing occurs under various
   circumstances and serves several purposes. To simplify the
   analysis of race conditions, we distinguish four cases: aborting,
   refusing, termination by receiver, termination by sender.

   A user "aborts" a connection when he issues a CONNECT and then a
   CLOSE before the CONNECT is acknowledged. Typically a user will
   abort following an extended wait for the acknowledgment; his
   system may also abort for him if he blows up.

   A user "refuses" a connection when he issues a LISTEN and, after
   being notified of a prospective caller, issues a CLOSE. Any
   requests for connection to a socket which is expecting a call from
   a particular socket are also refused.

   After a connection is established, either side may terminate. The
   required sequence of events suggests that attempts to CLOSE by the
   receive-side should be viewed as "requests" which are always
   honored as soon as possible by the send-side. Any data which has
   not yet been passed to the user, or which continues over the
   network, is discarded. Requests to CLOSE by the send-side are
   honored as soon as all data transmission is complete.

   1. Aborting

      We may distinguish three cases:

      a) In the simplest case, we send an RFC followed later by a
         CLS. The other side responds with a CLS and the attempt
         to connect ends.

      b) The foreign process may accept the connection
         concurrently with the local process aborting it. In this
         case, the foreign process will believe the local process
         is terminating an open connection.

      c) The foreign process may refuse the connection
         concurrently with the local process aborting it. In this
         case, the foreign process will believe the local process
         is acknowledging its refusal.

2. Refusing

   After an RFC is received, the local host may respond with an
   RFC or a CLS, or it may fail to respond. (The local host
   may have already sent its own RFC, etc.) If the local host
   sends a CLS, the local host is said to be "refusing" the
   request for connection.

   We require that CLS commands be exchanged to close a
   connection, so it is necessary for the local host to
   maintain the rendezvous table entry until an acknowledging
   CLS is returned.

3. Terminating by the Sender

   When the user on the send side issues a CLOSE system call,
   his NCP must accept it immediately, but may not send out a
   CLS command until all the data in the local buffers has been
   passed to the foreign host. It is thus necessary to test
   for both 'buffer-empty' and
   'RFNM-received' before sending the CLS command. As usual,
   the CLS must be acknowledged before the entry may be
   deleted.

4. Terminating by the Receiver

   When the user on the receive side issues a CLOSE system
   call, his NCP accepts and sends the CLS command immediately.
   Data may still arrive, however, and this data should be
   discarded. The send side, upon receiving the CLS, should
   immediately terminate the data flow.

VII. Connection Status

   An excellent mechanism for describing the sequence of events required
   to establish and terminate a connection involves a state diagram. We
   may assume that each socket can be associated with a state machine,
   and that this state machine may, at any time, be in one of ten
   possible states. In any state, certain network events cause the
   connection status to enter another state; other events are ignored;
   still others are error. A transition may also involve the local NCP
   performing some action. Figure 7.1 depicts the state machine.
   Circles [now boxes: Ed] represent states (described below); arrows
   show legal transitions between states. The labels on the arrows
   identify the event which caused them (note that CLOSE is a system
   call, CLS is a control command). Phrases after slashes denote the
   action which should be performed while traveling over that arrow.
   The arrow labeled '[E]RFC' (found between states 0 and 1) represents

the condition that whenever a connection enters the CLOSED state, the
pending call queue for that connection is checked [Original was
backwards "E": Ed.]

If any pending calls exist in the queue, the connection moves to the
PENDING state. If an RFC is received for a socket in the CLOSED
state, it is also moved along this path to the PENDING state. Events
and the actions they cause are described in sections VIII and IX
below. Descriptions of the ten states follow:

   (0) CLOSED

      The local socket is not attached to any port and no user has
      requested a connection with it. (The table entry is non-
      existent).

   (1) PENDING CALL

      The socket is not attached to any port but one or more
      requests for connection have been received. A LISTEN system
      call will be satisfied immediately by the first entry in the
      pending call queue for a matching request; all other pending
      calls are deleted.

   (2) LISTENING

      The socket is attached to a port. We are waiting for a user
      to request connection with this socket.

   (3) RFC-RCVD

      We are listening and an RFC was received. The local user has
      been informed of the pending call. He must respond with
      either a CLOSE or an ACCEPT.

   (4) ABORT

      We have notified the user that his LISTEN has been satisfied
      but he has not yet responded; if during this time the foreign
      user aborts the connection by sending a CLS, we send a CLS to
      acknowledge the abort and mark the fact with this state. When
      the user accepts or refuses the call, we can inform him the
      connection has been prematurely terminated.

(5) RFC-SENT

    This state is entered when:

    a) The local user has attached this socket to a port by
        issuing a CONNECT.
    b) An RFC has been sent, and
    c) No reply has been received.

    When the user issues a CONNECT the pending call queue is
    searched.

    If a matching RFC is not found, the queue is deleted and this
    state is entered. As new RFC's arrive they are compared with
    our user's request. If they do not match, the RFC is
    immediately refused. If the RFC matches, it completes the
    initialization process and the connection enters the OPEN
    state.

(6) OPEN

    RFC's have been exchanged and the connection is securely
    established. Transmission may begin following receipt of an
    ALL command from the receive side, and will then proceed
    subject to flow control.

(7) CLS-WAIT

    After the local user has executed a CLOSE, and we have issued
    a CLS, we must wait for an acknowledging CLS before the
    connection can be completely closed.   If the appropriate CLS
    has not already been received, this state is entered.

(8) DATA-WAIT

    If we are on the send side and the local user executes a CLOSE
    system call, a CLS cannot be issued if our data buffer is not
    empty or if a RFNM for the last data message is outstanding.
    The connection enters this state to wait for these conditions
    to be fulfilled. Upon completion and acknowledgement of
    output a CLS may be issued and the connection enters the CLS-
    WAIT state, waiting for the acknowledging CLS.   If a CLS
    arrives while in the DATA-WAIT state we clear our buffer (the
    CLS came from a receive socket, indicating it is no longer
    interested in our data) and enter the RFNM-WAIT state to wait
    for the network to clear.

(9) RFNM-WAIT

     If we are on the send side and a CLS command arrives, we
     cannot issue an acknowledging CLS if we have not received the
     RFNM for our last data message. We enter this state to await
     the RFNM, and cease all further data transmission. When the
     RFNM comes in, a CLS may then be issued, and the connection
     will be closed.

```
                              _____
     CONN/                   |   CLOSED  |         CLOSE
     send RFC                |    (0)    |<--------------------------+
     +-------------------    |           |  LISTEN                   |
     |                       |_____|  -------------------+     |
     |                          [E]RFC |  ^  CLS/send CLS       |     |
     |              non-matching       V  |                     V     |
     |              CONN/send RFC  _____            _____
     |              +------------ |  PENDING  |  LISTEN  | LISTENING |
     |              |             |    (1)    | ---------+   RFC     |    (2)    |
     |              |             |           |          +----|            |
     |              |             |_____|               |   |_____|
     V     V        matching        |                         V   V
  _____     CONN/send RFC    _____           _____  CLS/
 | RFC-SENT  |    RFC              ACCEPT/              | RFC-RECD  |  send CLS
 |    (5)    | ----------+         send RFC             |    (3)    | ---------+
 |           |           |         +----------|         |           |         |
 |_____|           |         |          |         |_____|         |
    |    |               V___V___V___   SND&CLOSE  |                |          |
    | RCV&CLS/          |   OPEN    | -----------)->  _____   |          |
    | send CLS          |    (6)    |  SND&CLS      | DATA-WAIT |   |          |
    +--------- |         |           |  --------+   |    (8)    |   |          |
    |    RCV&CLOSE/      |_____|          |   |_____|   |          |
    |    send CLS           |                   |         |         |          |
    |                       V                   |        CLS        |          |
    | CLOSE/            _____   CLOSE/     |         |         |          |
    | send CLS         | CLS-WAIT  |  send CLS   |         |         |          |
    +---)--------->    |    (8)    | <--------)--+         V         V          V
    |                  |_____|                  _____   _____
    |                       |                        | RFNM-WAIT |  |   ABORT   |
    |                      CLS                       |    (9)    |  |    (4)    |
    |                       |                        |_____|  |_____|
    |                       V             RFNM/          |              |
    |                   _____      send CLS         |              |
 CLS/ +--------->      |   CLOSED  | <----------+         |              |
 send CLS              |    (0)    |                ACCEPT|CLOSE         |
 +-------------------> |           | <--------------------------+
                       |_____|
```

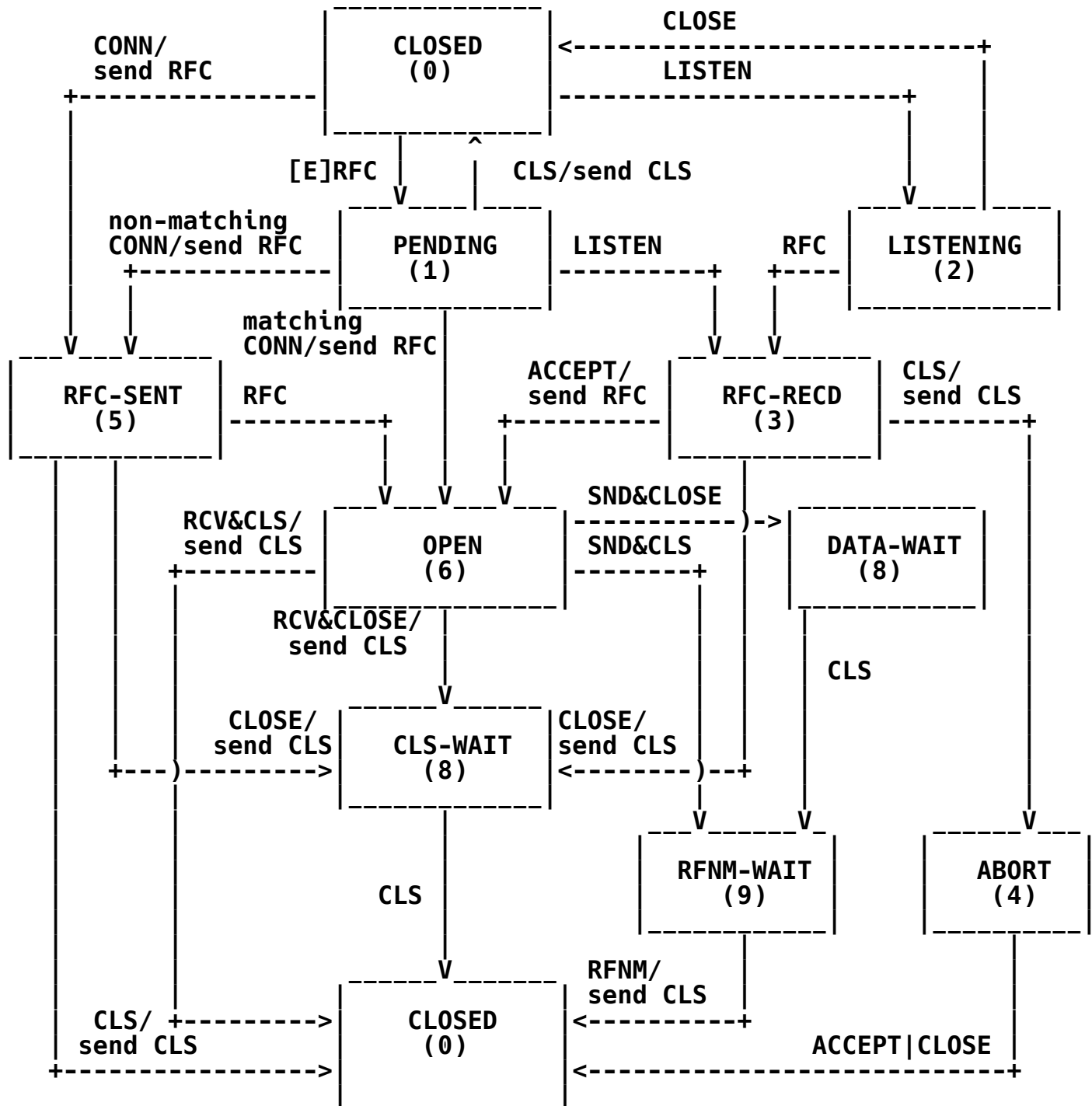   Figure 7.1
                       Connection State Diagram

VIII. Algorithms for the Input Interpreter

   The following is a concise description of the NCP's responses to
   incoming network commands. CS always indicates Connection State.
   Note, CLOSE is a system call executed by the local user process, and
   CLS is a network command.

   NOP

      Discard.

   RFC (RTS or STR)

      If no entry exists, create one with status = PENDING CALL, and
      queue the message.

      If CS = LISTENING, then queue the entry, enter the RFC-RCVD state,
      and inform the user of the request.

      If CS = RFC-SENT but the new RFC does not match the request,
      refuse the RFC.

      In all other cases, check the RFC for a match. If none exists,
      queue the RFC. If the RFC matches, then if:

         CS = RFC-SENT, we enter the OPEN state.

         CS = CLOSE-WAIT, the RFC is ignored.

         otherwise, the request is illegal in all states which indicate
         it has already been received (these states are 1,3,4,6,8,9).

      In any case, if processing the RFC causes an overflow condition
      (resources are exhausted), refuse the connection (send a CLS).

   CLS

      The pending call queue is searched. If the CLS doesn't match the
      current request, but does match some other request, then delete
      that request and issue a CLS. If there is no match, the CLS is
      ignored.

      If the CLS matches the current request, and CS =

         PENDING, then delete the current request. If the request queue
            is empty, delete the entry; otherwise, leave the entry
            alone.

RFC-RCVD, Issue a CLS and enter the ABORT state.
ABORT, ignore.

RFC-SENT, issue a CLS. If the pending call queue is empty
delete the entry, else enter the PENDING state.

OPEN, If we are on the receive side, response is identical to
the response for RFC-SENT. If we are on the send side,
clear the data queue, and if a RFNM is still pending enter
the RFNM-WAIT state. Otherwise response is identical to the
response for RFC-SENT.

CLS-WAIT, Issue a CLS and if the pending call queue is empty,
delete the entry, otherwise CS = PENDING.

DATA-WAIT, clear the data queue and enter the RFNM-WAIT state.
A matching CLS cannot occur in the CLOSED or LISTENING
states.

ERR

Errors are queued for later attention by system programmers, and
are considered to be a system error in the host that originated
the exchange. (Not associated with any state).

ECO

The op code is changed to ERP and retransmitted (Not associated
with any state).

ERP

Upon receipt of an ERP, the system passes the text of the command
back to the process which issued the ECO.

INR, INS

These commands are enabled only in the OPEN state. Upon receiving
an INTERRUPT, the system causes an event to be sent to the
associated process. An INTERRUPT is ignored in the CLS-WAIT,
DATA-WAIT, and RFNM-WAIT states. In any other state it is an
error.

ALL

    ALLOCATE is valid only in the OPEN state, and may be sent only to
    a send socket. The NCP increments the 'Their Size' field in the
    associated rendezvous table entry by the size specified in the
    ALLOCATE command.

    In the CLS-WAIT and DATA-WAIT states this command is ignored; in
    any other state it is an error.

Data-RFNM

    If in the OPEN state, mark the Flow Control Status field in the
      appropriate rendezvous table entry as RFNM-RECVD, and send more
      data if required.

    If in the DATA-WAIT state, maintenance the Flow Control Status.
      If the data queue is empty issue a CLS and enter the CLS-WAIT
      state; otherwise, transmit the next message.

    If in the RFNM-WAIT state, maintenance the Flow Control Status and
      issue a CLS. If the Pending Call queue is empty delete the
      rendezvous table entry, otherwise CS = PENDING.

    A Data-RFNM is an error in all other states.

IX. Algorithms for the System Call Interpreter

Each System Call is discussed, giving the state changes it may
effect:

CONNECT

    If there is no entry, create one, issue an RFC, and enter the
      RFC-SENT state.

    If CS = PENDING, search the queue and reject all non-matching
      requests. If no match is found issue an RFC and enter the
      RFC-SENT state. If a match is found, issue an RFC and enter
      the OPEN state. Transmission can commence as soon as buffer
      space has been allocated.

    In any other state this command is illegal.

LISTEN

    If an entry doesn't exist, create one, and enter the LISTENING
      state.

If CS = PENDING, inform the user and enter the RFC-RCVD state.

In any other state this command is illegal.

ACCEPT

If CS = RFC-RCVD, then issue an RFC and enter the OPEN state.
Data transmission can occur as soon as buffer space is
allocated.

If CS = ABORT, inform the user of the premature termination of the
connection. If the pending call queue is empty, delete the
entry; otherwise, enter the PENDING state.

This command cannot be legally executed in any other state.

CLOSE

If CS =

LISTENING, then delete the entry.

RFC-RCVD, then issue a CLS and enter the CLS-WAIT state.

ABORT, inform the user of the premature termination of the
connection. If the pending call queue is empty, delete the
entry; otherwise, enter the PENDING state.

RFC-SENT, then issue a CLS and enter the CLS-WAIT state.

OPEN, if we are on the send side, and the data queue is not empty,
or if a Data-RFNM is still outstanding, enter the DATA-WAIT
state; otherwise, issue a CLS and enter the CLS-WAIT state.

CLS-WAIT, issuing a CLOSE in this state is a USER ERROR.

DATA-WAIT, issuing a CLOSE in this state is also an illegal
sequence.

RFNM-WAIT, ignore the CLOSE.

A valid CLOSE cannot be issued if an entry does not exist, or if a
socket is in the PENDING state.