Network Working Group
Request for Comments: 46

Edwin E. Meyer, Jr.
Massachusetts Institute of Technology
17 April 1970

ARPA Network Protocol Notes

The attached document contains comments and suggestions of the Network Working Group at Project MAC. It is based upon the protocol outlined in NWG/RFC 33, 36, and later documents.

This proposal is intended as a contribution to the dialog leading to a protocol specification to be accepted by the entire Network Working Group.

We solicit your comments.

I - INTRODUCTION

In this document the Network Working Group at MIT Project MAC suggest modifications and extensions to the protocol specified by Carr, Crocker, and Cerf in a preprint of their 1970 SJCC paper and extended by Crocker in NWG/RFC 36. This document broadly outlines our proposal but does not attempt to be a complete specification. It is intended to be an indication of the type and extent of the protocol we think should be initially implemented.

We agree with the basic concept of simplex communication between sockets having unique identifiers. We propose the implementation of a slightly modified subset of the network commands specified in NWG/RFC36 plus the ERR command as specified by Harslem and Heafner in NWG/RFC 40.

Given the basic objective of getting all ARPA contractors onto the network and talking to each other at the earliest possible date, we think that it is important to implement an initial protocol that is reasonably simple yet extendable while providing for the major initial uses of the network. It should be a simple protocol so as to elicit the broadest possible support and to be easily implementable at all installations with a minimum of added software.

While the protocol will evolve, the fundamentals of a protocol accepted and implemented by all installations are likely to prove very resistant to change. Thus it is very important to make the initial protocol open-ended and flexible. A simple basic protocol is more likely to succeed in this respect than a complicated one. This

does not preclude the existence of additional layers of protocol between several installations so long as the basic protocol remains supported.

We feel that three facilities must be provided for in the initial protocol:

1. Multi-path communication between two existing processes which know how to connect to each other.

2. A standard way for a process to connect to the logger (logging process at a HOST) at a foreign HOST and request the creation of a user process. (The login ritual may or may not be standardized.)

3. A standard way for a newly created process to initiate pseudo-typewriter communication with the foreign process which requested its creation.

The major differences between the protocol as proposed by Carr, Crocker, and Cerf and this proposal are the following:

1. The dynamic reconnection strategy specified in Crocker's NWG/RFC 36 is reserved for future implementation. We feel that its inclusion would unduly complicate the initial implementation of the protocol. We outline a strategy for foreign process creation that does not require dynamic reconnection. Nothing in this proposal precludes the implementation of dynamic reconnection at a later date.

2. We propose that an "instance tag" be added to the socket identifier so as to separate sockets belonging to different processes of the same user coexisting at one HOST.

3. The following NCP commands have been added:

   a. The ERR command specified in NWG/RFC 40 is included.

   b. BLK and RSM commands are presented as possible alternatives to the "cease on link" IMP command and SPD and RSM commands set forth in NWG/RFC 36. Because these commands operate on socket connections rather than link numbers, they do not impede the implementation of socket connection multiplexing over a single link number, should that later prove desirable.

   c. An INT command that interrupts a process is specified. We feel that it is highly important to be able to interrupt a process that may be engaged in unwanted computation or output. To implement the interrupt as a special format within a normal

message raises severe difficulties: the connection may be blocked when the interrupt is needed, and the NCP must scan each incoming message for an interrupt signal.

d. An ECO echoing command to test communications between NCPs is included.

4. Sockets are conceptualized as having several states, and these are related to conditions under which network requests may be queued. This differs from the unlimited queuing feature, which presents certain implementation difficulties.

5. The protocol regarding creation of a foreign process and communication with it is removed to a separate User Control and Communication (UCC) protocol level and is more fully specified.

II - A HIERARCHY OF PROTOCOLS

It seems convenient and useful to view the network as consisting of a hierarchy of protocol and implementation levels. In addition to aiding independent software and hardware development, provisions for a layered protocol allow additions and substitution of certain levels in experimental or special purpose systems.

We view the initial network communications system as a hierarchy of three systems of increasing generality and decreasing privilege level. These are:

1. IMP Network - The network of IMPs and physical communication lines is the basic resource which higher level systems convert into more generalized communication facilities. The IMP network acts as a "wholesaler" of message transmission facilities to a highly privileged module within each HOST.

2. Network Control Program - Each HOST contains a module called the Network Control Program (NCP) which has sole control over communications between its HOST and the IMP network. It acts as a "retailer" of the wholesale communications facilities provided by the IMP network. The network of NCPs can be viewed as a higher level communications system surrounding the IMP network which factors raw message transmission capabilities between HOSTs into communication facilities between ordinary unprivileged processes.

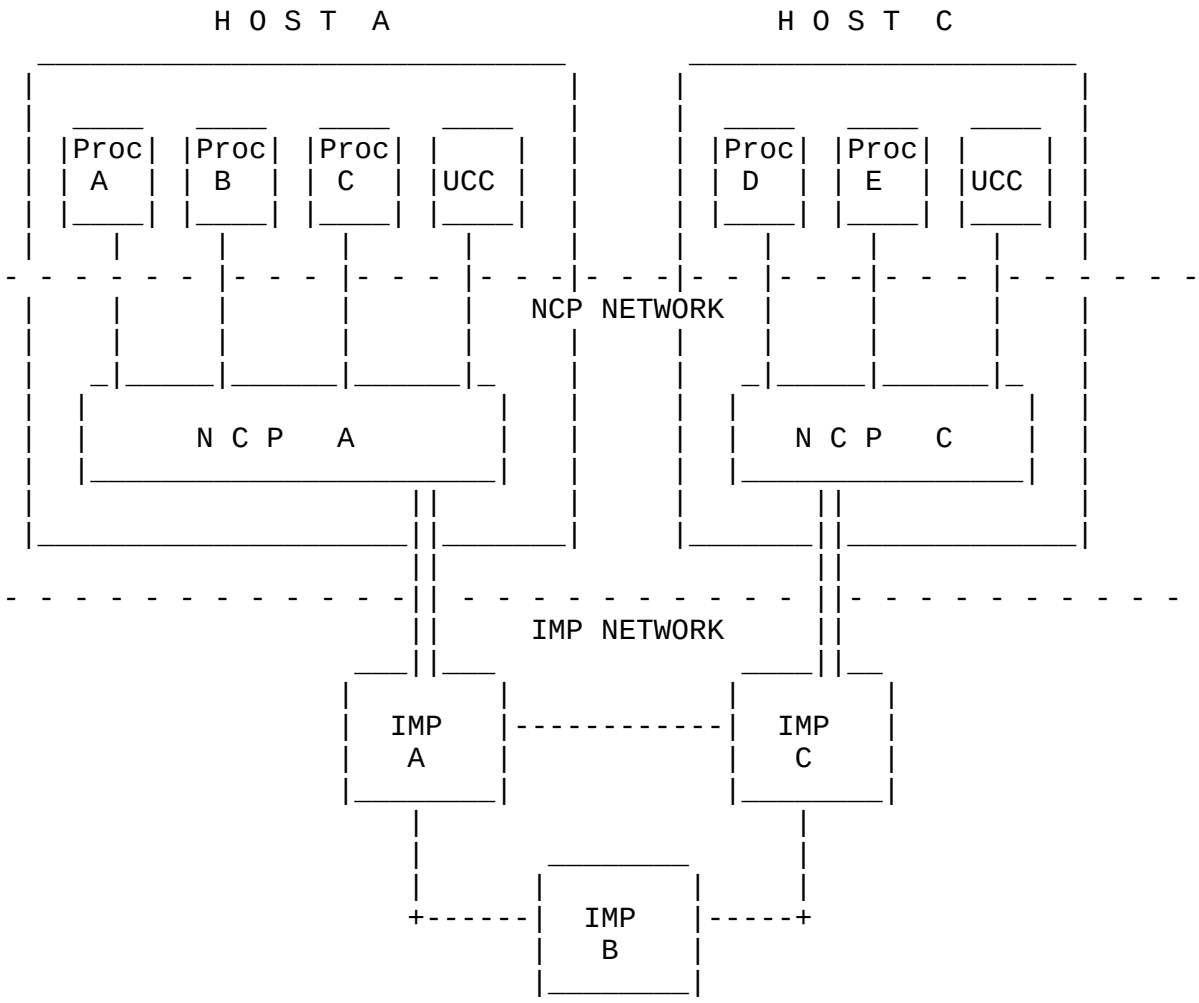

FIG 1. Modular View Of Network

3. User Control and Communication Module - The preceding two communication systems are sufficient to permit communication between unprivileged processes that already exist. However, one of the primary initial uses of the network is thought to involve the creation of a foreign user process through interaction with the foreign HOST's logger. The User Control and Communication Module (UCC) implements protocol sufficient for a process to communicate with a foreign HOST's logger and to make initial control communication with a created process. Such a process is to have the same privileges (subject to administrative control) as a local (to the foreign HOST) user process. The UCC module communicates through the NCP in a manner similar to an ordinary process. Except for the ability to close connections to a dead

process, the UCC module has no special network privileges. The UCC protocol is only one of several third-level protocols that could be implemented. For example, a set of batch processing systems connected through the NCP system might implement a load-sharing protocol, but not a UCC.

III - NETWORK CONTROL PROGRAM

Each HOST implements a module called the Network Control Program (NCP) which controls all network communications involving that HOST. The network of NCPs forms a distributed communication system that implements communication paths between individual processes. The NCP protocol issues involve: (i) the definition of these communication paths, and (ii) a system for coordinating the distributed NCP system in maintaining these communication paths. These are discussed below.

Sockets

Communication between two processes is made through a simplex connection between two sockets: a send socket attached to one process and a receive socket attached to another process. Sockets have the following characteristics:

Socket Identifier - A socket identifier is used throughout the network to uniquely identify a socket. It consists of 48 bits, having the following components:

a. User Number (24 bits) - A socket attached to a process is identified as belonging to that process by a user number consisting of 8 bits of "home" HOST code plus 16 bits of user code assigned by the home HOST. This user number is the same for all sockets attached to any of his processes in any HOST.

b. Instance Tag (8 bits) - More than one process belonging to a user may simultaneously exist within a single HOST. The instance tag identifies the particular process to which a socket belongs. A user’s first process at a HOST to use the network receives instance tag = 0 by convention.

c. HOST Number (8 bits) - This is the code of the HOST on which the attached process exists.

d. Socket Code (8 bits) - This code provides for 128 send and 128 receive sockets in each process. The low order bit determines whether this is a "send" (= 1) or "receive" (= 0) socket.

States of Sockets - Each socket has an associated state. The NCP may implement more transitory states of a socket, but the three following are of conceptual importance.

a. Inactive - there is no currently existing process which has told the NCP that it wishes to listen to this socket. No other process can successfully communicate with an inactive socket.

b. Open - Some process has agreed to listen to events concerning this socket but it is not yet connected.

c. Connected - This socket is currently connected to another socket.

Socket Event Queue - A queue of events to be disclosed to the owning process is maintained for each open or connected socket. It consists of a chronologically ordered list of certain events generated by the action of one or more foreign processes trying to connect or disconnect this socket. An entry in the event queue consists of the event type plus the identifier of the foreign socket concerned. The following event types are defined:

a. "request" - a foreign socket requests connection. (not queued if local socket is already connected)

b. "accept" - a foreign socket accepts requested connection.

c. "reject" - a foreign socket rejects requested connection.

d. "close" - a foreign socket disconnects an existing connection.

A "request" event is removed from the queue when it is accepted or rejected. The other events are removed from the queue as they are disclosed to the owning process.

Some events are intended to be transparent to the process owning the socket, and they do not generate entries in the event queue.

Although an event queue is conceptually unlimited, it seems necessary to place some practical limit on its length. When an event queue for a socket is full, any incoming event that would add to the queue should be discarded and the sending NCP notified (via ERR command described below).

NCP Control Communications

The NCP network coordinates its activities through control commands passed between its individual components. These commands generally concern the creation and manipulation of socket connections controlled by the NCP receiving the command. A control command is directed to a particular NCP by being sent to its HOST as a message over link number 1 (designated as the control link), which is reserved for that purpose. The IMP network does not distinguish between these messages and regular data messages implementing communication through a socket connection.

The following NCP control commands are defined:

a. Request for Connection

RFC <local socket> <foreign socket> [<link no.>]

An NCP directs this command to a foreign NCP to attempt to initiate a connection between a local socket and a foreign socket. If the foreign socket is open, the foreign NCP places a "request" event into the socket's event queue for disclosure to the process that owns it. If the foreign process accepts, the foreign NCP returns a positive acknowledgement in the form of another RFC. It rejects connection by issuing the CLS command (see below). An RFC is automatically rejected without consulting the owning process if the foreign socket is not open (inactive or connected). Multiple RFCs to the same socket are placed into its event queue in order of receipt. Any queued RFCs are automatically rejected by the NCP once the owning process decides to accept a connection. The NCP which has control of the "receive" socket of the potentially connected pair designates a link number over which messages are to flow.

b. Close a Connection

CLS <local socket> <foreign socket>

An NCP issues this network command to disconnect an existing connection or to negatively acknowledge an RFC. There is a potential race problem if an NCP closes a local send socket in that the CLS command may reach the foreign NCP prior to the last message over that socket connection. This race is prevented by adhering to two standards: (i) A CLS command for a local send socket is not transmitted until the RFNM for the last message to the foreign socket comes back, and (ii) the foreign NCP processes all incoming messages in the order received.

c. Block Output over a Connection

BLK <foreign send socket>

A process may read data through a receive socket slower than messages are coming in and thus the NCP's buffers may tend to clog up. The NCP issues this command to a foreign NCP to block further transmission over the socket pair until the receiving process catches up.

d. Resume Output over a Blocked Connection

RSM <foreign send socket>

An NCP issues this command to unblock a previously blocked connection.

e. Interrupt the Process Attached to a Connection

INT <foreign socket>

Receipt of this message causes the foreign NCP to immediately interrupt the foreign process attached to <foreign socket> if it is connected to a local socket. Data already in transit within the NCP network over the interrupted connection will be transmitted to the destination socket. The meaning of "interrupt" is that the process will immediately break off its current execution and execute some standard procedure. That procedure is not defined at this protocol level.

f. Report an Erroneous Command to a Foreign NCP

ERR <code> <command length> <command in error>

This command is used to report spurious network commands or messages, or overload conditions that prevent processing of the command. <code> specifies the error type. If <code> specifies an erroneous network command, <command in error> is that command (not including IMP header) and <command length> is an integer specifying its length in bits. If <code> specifies an erroneous message, <command in error> contains only the link number over which the erroneous message was transmitted. (This is slightly modified from the specification in NWG/RFC 40.)

g. Network Test Command

ECO <48 bit code> <echo switch>

An NCP may test the quality of communications between it and a foreign NCP by directing to it an ECO command with an arbitrary <48 bit code> (of the same length as a socket identifier) and <echo switch> 'on'. An NCP receiving such a ECO command should immediately send an acknowledging ECO with the same <48 bit code> and <echo switch> 'off' to the originating NCP. An NCP does not acknowledge an ECO with <echo switch> 'off'. We feel that this command will be of considerable aid in the initial shakedown of the entire network.

h. No Operation Command

NOP

An NCP discards this command upon receipt.

User Interface to the NCP

The NCP at each HOST has an interface through which a local process can exercise the network, subject to the control of the NCP. The exact specification of this interface is not a network protocol issue, since each installation will have its own interface keyed to its particular requirements. The protocol requirements for the user interface to an NCP are that it provide all intended network functions and no illegal privileges. Examples of such illegal privileges include the ability to masquerade as another process, eavesdrop on communications not intended for it, or to induce the NCP to send out spurious network commands or messages.

We outline here an interface based on the Carr, Crocker, and Cerf proposal that is sufficient to fully utilize the network. While this particular set of calls is intended mainly for illustrative purposes, it indicates the types of functions necessary.

The following calls to the NCP are available:

a. LISTEN <my 8 bit socket code>

A user opens this socket, creating an empty event queue for it. This LISTEN call may block waiting for the first "request" event, or it may return immediately.

b. INIT <my socket code> <foreign socket>

A user attempts to connect <my socket> to <foreign socket>. The local NCP sends an RFC to the foreign NCP requesting that the connection be created. The returned acknowledgemnet is either an RFC (request accepted) or CLS (request rejected). At the caller's option, the INIT call blocks on the expected "accept" or "reject" event, or it can return immediately without waiting. In this case the user must call STATUS (see below) at a later time to determine the action by the foreign NCP. When a blocked INIT call returns, the "accept" or "reject" event is removed from the event queue.

c. STATUS <my socket code>

This call reports out the earliest previously unreported event in the queue of <my socket>. The STATUS call deletes the event from the queue if that type of event is deleteable by disclosure.

d. ACCEPT <my socket code>

The user accepts connection with the foreign socket whose "request" event is earliest in the event queue for <my socket>. An acknowledging RFC is sent to the accepted foreign socket, and the "request" event is deleted from the event queue. Should any other "request" event exist in the queue, the NCP automatically denies connection by sending out a CLS command and deleting the event.

e. REJECT <my socket code>

The user rejects connection with the foreign socket whose "request" event is earliest in the event queue for <my socket>. The NCP sends out a CLS command and deletes the "request" event from the queue.

f. CLOSE <my socket code>

The user directs the NCP to disconnect any active connection to this socket and to deactivate the socket. The NCP sends out a CLS command to the foreign socket if a connection has existed. The status of the foreign socket also becomes closed once the "close" event is disclosed to the foreign process.

g. INTERRUPT <my socket code>

The user directs the NCP to send out an INT command to the foreign socket connected to <my socket>.

h. TRANSMIT <my socket code> <pointer> <nbits>

The user wishes to read (<my socket> is receive) or write (<my socket> is send) <nbits> of data into or out of an area pointed to by <pointer>. A call to write returns immediately after the NCP has queued the data to send a message over the connection. The call to write blocks only if the connection is blocked or if the local NCP is too loaded to process the request immediately. Data to be transmitted over a connection is formatted into one or more IMP messages of maximum length 8095 bits and transmitted to the foreign HOST over the link number specified in the RFC sent by the NCP controlling the receiving connection. A "close" event in the event queue for <my socket> is disclosed through the action of TRANSMIT. A call to write discloses the "close" event immediately. A read call discloses it when all data has been read.

The History of a Connection From a User View

An Illustrative Example

Assume that process ’a’ on HOST A wishes to establish connection with process ’b’ on HOST B. Before communication can take place, two conditions must be fulfilled:

a. process ’a’ must be able to specify to its NCP a socket in ’b’s socket space to which it wants to connect.

b. process ’b’ must already be LISTENing to this socket.

1. Establishing the Connection

a. process ’b’ LISTENs to socket ’Bb9’.

b. process ’a’ INITs ’Bb9’ to its ’Aa12’. The NCP at A generates an RFC specifying link number = 47, which it chooses from its available set of links. This is the link over which it will receive messages if the connection is ACCEPTed by process ’b’.

c. process ’b’ is informed of A’s INIT request. He may REJECT connection (NCP B sends back a CLS) or ACCEPT (NCP B sends back an RFC).

d. If process ’b’ ACCEPTs, the confirming RFC establishes the connection, and messages can now flow.

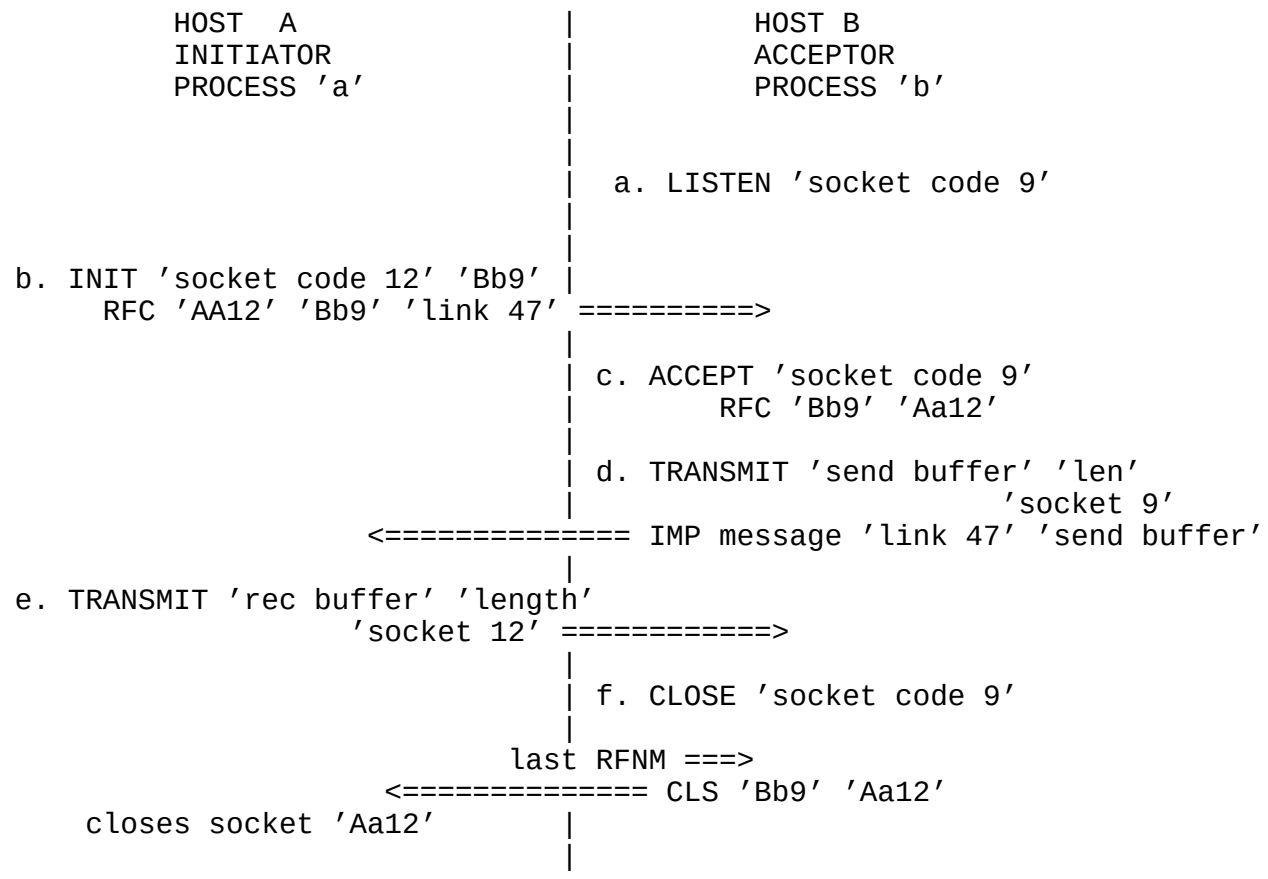

FIG 2. Establishing and Communicating over a Socket Connection

2. Sending Messages over a Connection.

a. Process 'b' issues a TRANSMIT call to send data through the connection. NCP B formats this into an IMP message and sends it to NCP A with link number = 47 as specified by A's RFC.

b. NCP A receives the raw message from NCP B with link number = 47. NCP A uses this link number in deciding who the intended recipient is, and stores the message in a buffer for the recipient process.

c. Process 'a' may issue a read (TRANSMIT) call for socket code 12 at any arbitrary time. The read call blocks if there is no data pending for the socket. The read call picks up the specified number of bits transmitted over socket code 12, perhaps across an IMP message boundary. The boundaries of the IMP messages are invisible to the read call.

d. Should process ’b’ send data over the connection at a faster rate than process ’a’ picks it up, NCP A can issue a BLK command to NCP B if A’s buffers start filling. Later, when process ’a’ catches up NCP A can tell B to resume transmission via an RSM command.

3. Process ’b’ Closes the Connection

a. Process ’b’ decides to close the connection, and it issues the CLOSE call to NCP B. To avoid race problems B waits for the RFNM from the previous message over this connection, then sends the CLS command to NCP A. When the RFNM from the CLS command message returns, NCP B flushes socket ’Bb9’ from its tables, effecting the close at its end and deactivating ’Bb9’.

b. Because of sequential processing within NCP A, the last message to socket ’Aa12’ is guaranteed to have been directed to a process before the CLS from NCP B comes through. Upon receipt of the CLS from B, NCP A marks socket ’Aa12’ as "close pending" and places a "close" event into the event queue of ’Aa12’.

c. Process ’a’ can still issue read calls for socket ’Aa12’ while there is buffered data pending. When ’a’ issues a read call after the buffer has been emptied, the "close" event is disclosed to inform ’a’ of the closure, and socket ’Aa12’ is flushed from the tables of NCP A.

4. Process ’a’ Closes the Connection

a. Let us return to step 2. and assume that process ’a’ wants to close the connection from its end. There is no race problem because we assume that once ’a’ issues a CLOSE call, it no longer wants to read messages over that socket.

b. Assume that process ’a’ issues a CLOSE call on socket ’Aa12’. NCP A immediately sends out a CLS command to NCP B and marks socket ’Aa12’ as "close pending". Any data buffered for read on ’Aa12’ is discarded. To allow remaining messages already in transit from process ’b’ to percolate through the IMP network to NCP A and be discarded without error comments, NCP A retains ’Aa12’ in its tables for a suitable period of time after receiving the RFNM from the CLS command. During this period NCP A discards all messages received over the closing connection. After allowing a reasonable amount of time for these dead messages to come in, NCP A flushes ’Aa12’ from its tables, effectively closing the connection and deactivating ’Aa12’. Further messages to socket ’Aa12’ result in NCP A sending an ERR "erroneous command" to the originating NCP.

c. When NCP B receives the CLS command, socket ’Bb9’ is marked as "close pending", and the CLS event is placed into the event queue of ’Bb9’. The next time process ’b’ wishes to write over that socket, the CLS event is disclosed to inform him of the closure, and socket ’Bb9’ is removed from NCP B’s tables.

IV - USER CONTROL AND COMMUNICATION PROTOCOL

Some process must exist which agrees to listen to anybody in the network and create a process for him upon proper identification. This process is called the logger and interacts through the NCP via the network-related User Control and Communication (UCC) module, which implements the necessary protocol. Except for one instance (CLOSEing connections of dead processes), the process operating the UCC module does not have special network privileges.

Under the UCC protocol a "requestor" process which has directed the creation of a "foreign" process maintains two full-duplex pseudo-typewriter connections: one to the foreign logger, and one to the created process. The duplex connection to the foreign logger is used to identify the requestor process to the logger, and after login to return to the requestor process basic information concerning the health of the created process. The duplex connection to the created process is used for control communication to it.

Maintaining two full-duplex connections avoids reconnection problems both when the logger transfers communication to the created process and when it needs to regain control. This is at the modest expense of requiring the requestor process to switch typewriter communications between two sets of connections.

The way that communication is established is essentially as follows: the requestor process first reserves four of its sockets having contiguous socket codes. Then it "signals" the UCC, specifying one of these sockets. From the "signal" the UCC knows which process is calling, and by protocol, on which requestor socket pair the UCC is to communicate with the requestor process, and which requestor socket pair the created process is to use for its communications. This is specified below in more detail.

Establishing and Operating a Remote Process

The UCC at each HOST always keeps a send socket with user number = 0, instance tag = 0 open (active and unconnected) as a "signal" socket, and periodically checks for INITs to this socket. Processes wishing to create a process at this HOST must first signal the UCC by issuing an INIT to this socket.

The requesting process must have four free sockets with contiguous socket codes: <base_socket> (receive) through <base_socket+3> (send). The high numbered send/receive set of sockets is used for typewriter communication with the foreign UCC, the low numbered set for typewriter communication with the created process.

1. The "requestor" process calls LISTEN twice to open the <base_socket+2> and <base_socket+3> receive/send pair over which it will talk to the foreign UCC. Then it sends out a "signalling" INIT call on <base_socket> to the UCC "signal" socket. The only thing that the UCC does with this "signalling" INIT call is to note down the socket number <base_socket> from which it originated. The UCC immediately rejects this request so as to keep its "signal" socket open for other signals.

2. After receiving the expected REJECT on its initial INIT call to the UCC's signal socket, the requestor process issues LISTENs for <base_socket> and <base_socket+1>. (The created process will INIT these sockets to establish control communication with the requestor process.) The requestor process then blocks by calling STATUS <base_socket+2> .

3. The UCC INITs a free send/receive socket pair to the requestor's <base_socket+2> and <base_socket+3> on which the requestor process is presumably LISTENing. The requestor process has called STATUS <base_socket+2> with block option after LISTENing for the two sockets, so that when the INIT from the foreign UCC reaches the requestor process, STATUS returns with the INIT indication. The requestor process verifies that the UCC is the process that is calling, then it ACCEPTs the call. The requestor process then calls STATUS <base_socket+3> and returns when the INIT for that socket reaches it. It does a similar verify and ACCEPT. (There is an arbitrary choice as for which socket the requestor process first calls STATUS.) Two way communication is established when the requestor process has ACCEPTed both INITs from the UCC. This connection is maintained during the login ritual and throughout the life of the created process. Should the requestor process fail to respond properly within a limited amount of time to the INITs of the UCC, the UCC abandons the connection attempt.

4. The requestor process must then perform the login ritual with the UCC. (The initial protocol might standardize the login ritual.) If the logger is not satisfied and wishes to cut off the requestor, the UCC module CLOSEs both <base_socket+2> and <base_socket+3>, perhaps after the logger has sent a suitable message.

5. If satisfied, the logger creates a process for the user. The UCC maintains direct communication with the requestor, but this connection is now used only to report basic information concerning the created process.

6. The first task of a created process is to establish a dual pseudo-typewriter control connection with its requestor process. The created process INITs one of its send/receive socket pairs to the requestor's <base_socket> and <base_socket+1>. If both requests are ACCEPTed, the created process sends an initial message over this connection. Then it goes to command level, in which it awaits a typewriter command message over the connection. If the created process is unable to establish duplex communication with the requestor process, it should destroy itself. The UCC will either CLOSE its own connections with the requestor or make arrangements for another process to be created.

7. When a created process is logged-out, the UCC uses a privileged entry to the NCP to CLOSE all connections between the dead process and other processes, and to deactivate all open sockets of the dead process. The UCC transmits a message back to the requestor process, then CLOSEs the dual connections between it and the requestor process.

8. The INTERRUPT call has a standard "quit" meaning when sent from a requestor process to a created process over the requestor's receive socket <base_socket>. All pending output from the created process is aborted, and the it enters "command level" where it awaits a command over the typewriter connection to the requestor process. The interrupted processing is resumable by issuing a "start" command to the created process. (Note that the rule about pending output is more restrictive than that implemented by the INT NCP command.)

This document was prepared through the use of the MULTICS "runoff" command. A source file consisting of intermixed text and "runoff" requests was created using the "qed" text editor. This file was then compiled by the "runoff" command to produce a finished copy. The latest version of this document exists online in MULTICS in the segment

>udd>Multics>Meyer>network_protocol.runoff

(END)

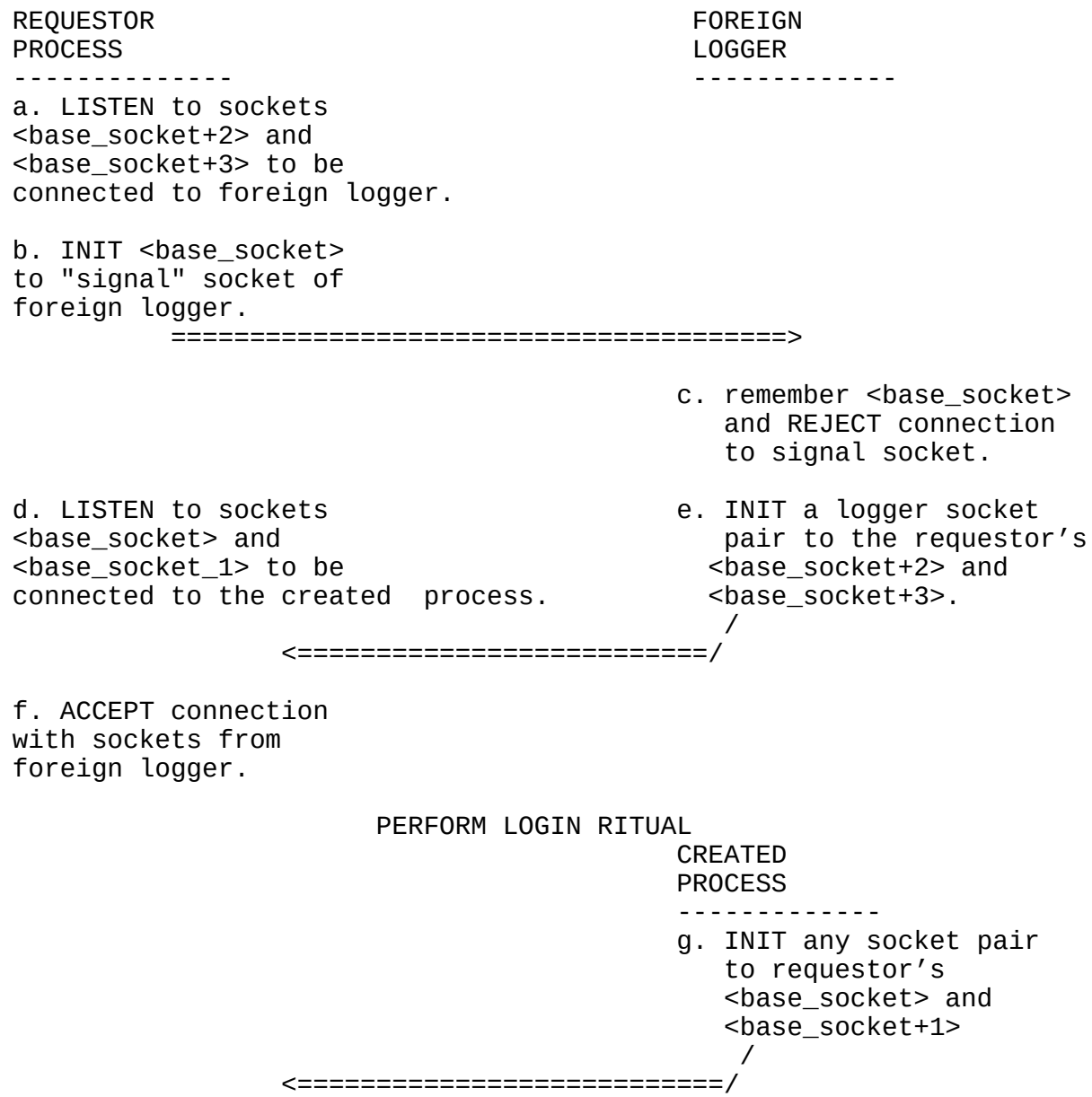

FIG. 4 Establishing a Process at a Foreign HOST

[This RFC was put into machine readable form for entry]
[into the online RFC archives by Miles McCredie 11/99]