File Hall
UCLA
①

ffice of
cademic
omputing

TECHNICAL REPORT TR20
DECEMBER 15, 1979

"AN IBM 360/370 IMPLEMENTATION OF THE INTERNET AND TCP PROTOCOLS -- DESIGN SPECIFICATIONS"

ROBERT T. BRADEN

FINAL TECHNICAL REPORT

ARPA CONTRACT NUMBER MDA903 74C 0083, ORDER 2543/8,
SPONSORED BY THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
OCTOBER 1977 - MARCH 1979
"DEVELOPMENT OF AN ARPANET TCP FOR AN IBM 360"

niversity of
alifornia,
os
ngeles

UNIVERSITY OF CALIFORNIA AT LOS ANGELES

Office of Academic Computing

Technical Report TR20

December 15, 1979

"An IBM 360/370 Implementation of the Internet and TCP Protocols -- Design Specifications"

Robert T. Braden

FINAL TECHNICAL REPORT

ARPA CONTRACT NUMBER MDA903 74C 0083, ORDER 2543/8,
Sponsored by the Defense Advanced Research Projects Agency

October 1977 - March 1979

"Development of an ARPANET TCP for an IBM 360"

Principal Investigator: William B. Kehl
Director, UCLA Office of Academic Computing

-A-

REPORT SUMMARY

A family of "internet" host-to-host protocols has recently been defined to allow computer communications across interconnected packet networks with diverse properties. This internet protocol family is defined in two distinct levels. The lower level, Internetwork Protocol or IP, provides simple datagram service. Transmission Control Protocol or TCP is a "higher-level" internet protocol that uses IP for data transport. TCP provides connections, strong end-to-end error control, flow control, and a form of out-of-band signalling. The IP/TCP combination is intended to be the successor to the original ARPANET Host-to-Host Protocol (AHHP).

Under ARPA contract, UCLA has implemented Version 4 of the IP and TCP protocols for an IBM 360/370 host computer on the ARPANET. This implementation is integrated into the existing Network Control Program for AHHP, and was designed to be compatible at the system-call interface so that existing user-level protocol programs can be used interchangeably with AHHP and IP/TCP. The implementation is layered to match the protocols.

This document gives a techical overview of the UCLA IP/TCP implementation. It describes the NCP software environment, the resolution of compatibility issues, and the design of both the IP and TCP layers.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OAC/TR-20	2. GOVT ACCESSION NO. AD-A155057	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "An IBM 360/370 Implementation of the Internet and TCP Protocols--Design Specifications"		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert T. Braden		8. CONTRACT OR GRANT NUMBER(s) MDA 903-74-C-0083
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Academic Computing 5628 Math Sciences Addition C0012 - UCLA Los Angeles, CA 90024		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Order 2543/8
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE December 15, 1979
		13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ARPANET. Computer Communication Protocols. Network Control Program. Internet Protocol. TCP. Software Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

UCLA has implemented Version 4 of the IP and TCP protocols for an IBM 360/370 host computer on the ARPANET. This implementation is integrated into the existing Network Control Program for AHHP, and was designed to be compatible at the system-call interface so that existing user-level protocol programs can be used interchangeably with AHHP and IP/TCP. The implementation is layered to match the protocols.

(continued)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ILLUSTRATIONS

1. INTRODUCTION

In 1971, the UCLA Office of Academic Computing (OAC) began to implement ARPANET interface software for its IBM 360/91 CPU under the OS/MVT operating system. As described in the paper "A Server Host System on the ARPANET" (Snowbird Data Communications Symposium, September 1977 [Bra77]), this software included:

* a Network Control Program, or NCP;

* support for various user-level protocols; and

* the Exchange, an OS/MVT operating system extension for interprocess communication [BraFe72].

The user-level protocol support required interfacing to server subsystems, principally the TSO timesharing subsystem and the RJS remote batch entry subsystem. Within the NCP itself, server processes implement the ARPANET File Transfer Protocol (FTP) [RivWo77] as well as MSG, the interprocess communication facility for the National Software Works [RivLB77].

The original NCP implemented the standard ARPANET host-host protocol [McKen72]. Under ARPA contract, UCLA has now completed an initial implementation of the new internetwork host-to-host protocol IP/TCP, allowing effective communication with hosts on other packet-switching networks interconnected with the ARPANET. This IP/TCP implementation is currently operating to make UCLA's IBM 3033 mainframe an "internet host".

This document is the Final Technical Report under the IP/TCP contract and describes the design of that implementation in general terms. More complete documentation will be found in the Program Logic and interface manuals [BraTCP,BraIP,Bra79B]. This document assumes familiarity with ARPANET protocols, including AHHP and IP/TCP; however, the next section will summarize aspects of these protocols that are relevant to this report.

1.1. INTERNET PROTOCOLS

The protocols used by hosts on the ARPANET packet-switching network are said to be "layered" [FeinPos]. That is, the protocols are defined in distinct layers or _levels_, with protocols on a given level being defined in terms of an abstract communication model created by the next lower level. For example, the _user-level_ protocols such as Telnet [McKen73] were defined in terms of the model created by the ARPANET host-to-host protocol (_AHHP_), one level lower.

The AHHP model [McKen72] is based on simplex data streams or _connections_ whose ends are labeled with 32-bit numbers called _sockets_. Sockets have an intrinsic parity: odd-numbered sockets send data, while even-numbered ones receive data. Hence a connection always links an odd socket and an even socket. AHHP also provides flow control and out-of-band signalling. AHHP allows messages to be a multiple of any byte size (in practice, byte sizes are usually 8, 32, or 36 bits).

Packet-switching networks have rapidly proliferated in the last few years, and many of them are being interconnected. Networks are generally interconnected by hosts called "gateways" which are common to two (or more) networks [CerKa74]. Since AHHP is inadequate for communicating across interconnected packet networks, ARPA and its contractors have designed a new family of "internetwork" host-host protocols [PosIP,PosTCP]. This internetwork protocol family itself consists of two layers:

(1) a lower level called _Internetwork_ _Protocol_ or _IP_;

(2) a "higher-level" host-to-host protocol.

IP provides datagram service in an internetwork environment, sending "internet packets" between hosts which may be on different networks. An internet packet consists of a _segment_ of data prefixed with an _IP_ _header_.

IP provides the functions: (1) internetwork host addresses and (2) the reassembly of internet packets which have been fragmented by intermediate gateways. IP does not provide error control; depending upon the properties of the networks and gateways, a transmitted packet may be lost, delivered out of order, or delivered in duplicate.

Transmission _Control_ _Protocol_ or _TCP_ [PosTCP] is a particular "higher-level" host-to-host protocol built upon IP; thus, TCP uses IP as a "data transport" service to transmit and receive segments. A TCP segment generally consists of a _TCP_ _header_ possibly followed by data.

TCP provides all the functions of AHHP with the addition of strong end-to-end error control. In particular, TCP provides full-duplex connections whose ends are labeled with 16-bit numbers called ports. Unlike AHHP, TCP allows the same 16-bit port number on a given host to participate in any number of connections whose remote ends have differing (host,port#) pairs. TCP also provides flow control and a facility called urgent that may be considered a form of out-of-band signalling. TCP messages consist of 8-bit bytes or octets.

The user-level protocols defined for AHHP must be changed slightly for use with TCP, due to the significant differences between the two host-host protocols which will now be summarized. The effects of these differences on the UCLA implementation of TCP will be described in later sections.

1.1.1. Datagram vs. Virtual Circuit Services

AHHP provides only "virtual circuit" service, i.e., data is sent over logical paths or 'connections". Two hosts must exchange control messages to establish a connection before they can send data to each other.

TCP also provides connections, and may be used in virtual-circuit mode as a replacement for AHHP. On the other hand, in TCP a single message can open a connection, send data, and close it again, effecting a datagram service mode.

1.1.2. Full-duplex vs. Half-duplex connections:

Under AHHP, the user-level protocols require a pair of simplex connections to obtain full-duplex operation. Under TCP, these protocols can use a single full-duplex TCP connection.

A further complication is the fact that a TCP connection is allowed to be half-open indefinitely. Thus, a close request (<FIN>) only signals the end of data transmission in one direction; the local process can continue to send data in the other direction on that connection. The connection will be fully closed and deleted only by request of the local process, or by the receipt of a <RST> (Reset) message. In contrast, AHHP protocols that use a pair of connections generally expect both to close simultaneously.

1.1.3. Ports vs. Sockets

TCP ports differ from AHHP sockets in their size (16 instead of 32 bits) and in having no odd/even parity. More importantly, a TCP port can participate in multiple simultaneous connections.

Under AHHP, starting a new session requires an initial handshake, the <u>Initial Connection Protocol</u> or <u>ICP</u> [Pos71]. At the server host, ICP begins with a connection to a <u>well-known socket</u>, followed by reconnection to a unique socket (pair); the reconnection leaves the well-known socket free for the next ICP sequence.

Under TCP, a particular server's <u>well-known port</u> can participate in any number of connections, as long as the user's (host,port) pair is unique for each session. Therefore, TCP does not require an ICP sequence.

1.1.4. <u>Urgent vs. Interrupts</u>:

A TCP segment may include a field called the "Urgent pointer" which indicates there is "urgent" data a specified number of bytes ahead in the data stream. This fact is to be communicated to the user-level protocol, which must read ahead to find and interpret the urgent data.

Although the Urgent pointer is "out-of-band" in the sense it is communicated outside the data stream, it is not exactly like the "interrupt" control messages of AHHP; the Urgent pointer is state information rather than a discrete event.

However, TCP's Urgent pointer can be used to achieve the same function as the AHHP interrupt in many contexts. For example, the Telnet protocol needs an out-of-band signal to force control bytes through to the server operating system when the data pipeline is clogged [McKen73]. Under AHHP, the control bytes are followed in the data stream by an identifiable byte called a Data Mark. A matching interrupt is also sent, informing the receiver that by reading ahead to the Data Mark it will pass (and should interpret) some important control bytes. The receiver's Telnet program is required to count interrupts and Data Marks to maintain synchronism. Under TCP, the urgent mechanism obviates the need for a Data Mark; the Urgent pointer identifies the location in the data stream of the urgent control bytes.

The layering of the ARPANET protocols is reflected in message formats; the data defined by a given layer is "wrapped" or <u>embedded</u> within framing control bits defined by the next lower layer. Figure 1 illustrates successive embedding when data is sent using TCP: the data is prefixed with a TCP header, an IP header, and finally a local packet header for transmission over the local packet network. Similarly, AHHP prefixes the data with an AHHP header before the local packet header is prefixed.

In the ARPANET case, the local packet header is a 96-bit leader. The format of a leader is described by the IMP-host protocol, the lowest level protocol seen by an ARPANET host [BBN1822].

Figure 1. Protocol Levels and Embedding

```
                                                          ________..._____
(1) Data from User-level Protocol Process                |                |
                                                         |<----data----->|
                                                         |________..._____|

                                      __________          ________..._____
(2) TCP prefixes its header          |   TCP    |  ->    |                |
                                     |  header  |        |<---data------>|
                                     |__________|        |________..._____|

                           __________        _________  ________..._____
(3) IPP prefixes          |    IP    |  ->  |  TCP    .                 |
        its header        |  header  |      | header  .<----data----->|
                          |__________|      |_________.________..._____|

(4) ARPANET Outgoing Gateway
    prefixes leader
                    _________        _________ _________ ________..._____
                   | ARPANET |  ->  |    IP   .   TCP   .                |
                   | leader  |      |  header .  header .<----data----->|
                   |_________|      |_________._________.________..._____|
```

The term "gateway" was originally chosen for a host which is connected to two or more packet networks in order to forward data from one network to another. The gateway software must strip off the local network framing when an internet packet is received and then re-embed the packet in the framing required by the target network.

Every host implementing IP must similarly strip and embed the internet packets for transmisson over the local packet network; in this sense, every internet host includes a kind of gateway into the local net. Therefore, the modules of the UCLA NCP which handle the IMP-host protocol for the ARPANET will be referred to in this document as the (local) _ARPANET gateway_.

Through the ARPANET gateway, a local host-to-host protocol program has access to two types of ARPANET message service: _standard_ and _uncontrolled_ [BBN1822]:

* Subtype 0 ("Standard")

  The AHHP always uses Subtype 0 messages, which the ARPANET delivers "reliably". That is, the packet-switching subnet will either (1) deliver one correct copy of the original message to the destination host and return an acknowledgment to the source host, or (2) return a negative acknowledgment. The acknowledgment (either positive or negative) will be returned as an IMP-to-host or _irregular_ message, carrying the 12-bit _message-id_ field from the leader of the original message.

  In particular, an irregular message of type "Request for Next Message" (_RFNM_) will be returned when the original message has been successfully reassembled at the destination IMP and placed on its queue for transmission to the destination host. AHHP ensures reliable and ordered delivery of ARPANET messages by requiring the source host to wait for a RFNM before sending another message with the same message-id.

* Subtype 3 ("Uncontrolled")

  A host that sends an uncontrolled message will receive no acknowledgment from the IMP. An uncontrolled message may be lost, duplicated, or reordered by the subnet. However, uncontrolled message may be delivered faster than standard messages and are therefore useful when speed is more important than relibility.

The two message subtypes differ in maximum size. Standard messages (which may be sent on the ARPANET in multiple packets) may contain up to 1007 octets, exclusive of leader; uncontrolled messages may contain at most 113 octets [BBN1822].

The AHHP and IP actually use only the 8 high-order bits of the message-id, called the <u>link number</u>, leaving the low-order 4 bits of the message-id field zero. In particular, AHHP uses link numbers 0-71 for multiplexing the logical message streams to a particular remote host. Internet packets, however, use a single link number (currently 155); logical streams must be demultiplexed by the internet host based on the IP and higher-level protocol headers.

In the Internet Protocol model, the choice of message subtype (and any other network parameters [PosIP]) is based on a field in the IP header called <u>Type of Service</u> (<u>TOS</u>). Generally, each network which is traversed by an internet datagram should interpret the TOS field to select appropriate network parameters. The 8 bits in the TOS field are divided into 5 subfields [PosIP]. For example, for Telnet service in TCP the TOS field could be the catenation of the bits:

00B => Priority= none.

1B => Stream/Datagram Service = Stream.

10B => Reliability= "higher" (or "normal").

1B => Speed over reliability= true.

10B => Speed= "higher" (or "fast").

This is the hex byte X'36'. Similarly, for file transfers TCP might want to use X'31', favoring reliability over speed.

1.2. IP/TCP IMPLEMENTATION STRATEGY

UCLA has implemented the two-layer internet protocol consisting of IP and TCP for an IBM 360/370 system under the OS/MVT operating system. The implementation is written in IBM Assembly Language.

The IP/TCP implementation was integrated into the existing ARPANET NCP, which can now support both the old host-to-host protocol AHHP and the new internet protocols simultaneously. Furthermore, the IP/TCP implementation is (as nearly as possible) compatible with AHHP at the system-call level, so that the AHHP routines which implement user-level protocols such as Telnet and FTP can be converted to TCP with minimal modification.

The IP/TCP implementation is itself divided into two distinct layers to match the protocols:

(1) internet protocol program (IPP), and

(2) higher-level protocol module (HLPM).

The IPP implements the IP protocol layer while the HLPM implements the higher-level protocol layer. For TCP in particular, the HLPM is called TCPMOD. The IPP/HLPM interface is defined so that other higher-level host-host protocols can be added in parallel to TCP without changing the IPP [Bra79B].

The IP/TCP implementation was designed for ease of debugging while the AHHP code is operating for users. This required the new code to be in transient load module(s) rather than linkage edited with the resident AHHP module. Also, the IP/TCP processing must be performed on distinct NCP processes which can block indefinitely or terminate without interfering with AHHP operation.

Fitting the IP/TCP implementation into an existing NCP and providing compatibility with existing protocol modules imposed severe constraints on the design of the new code. The existing NCP did not clearly separate IMP-host and host-host protocol processing, so many of the internal interfaces required by IP/TCP were fuzzy, undocumented, or non-existent. Furthermore, for economy and future compatibility it was desirable to use common code as much as possible.

We adopted the general strategy of adding documented interfaces to the existing NCP modules while disturbing those modules as little as possible. In the future, it will be possible to rewrite the AHHP and other NCP code to use the new interfaces and clearly recognize the protocol boundaries. However, this was not required in order to implement IP/TCP.

The following section of this document describes the software environment of the NCP, after the interface changes for IP/TCP have been added. Thus, it describes both the environment within which IP/TCP code must operate and the common compatible interface to the user-level protocol program that IP/TCP code must match. Later sections describe the actual designs of the IPP and of TCPMOD.

2. NCP SOFTWARE ENVIRONMENT

This section describes the structure of the IBM 360/370 NCP developed at UCLA. Its purpose is to define the execution environment for new additions to the NCP (e.g., support for new user-level protocols), using either the original AHHP or an internet protocol.

The UCLA NCP design has the following general features [Bra77]:

* The NCP executes as a system job rather than as part of the OS/MVT Supervisor, providing an isolated environment for developing and maintaining ARPANET protocol modules. While a buggy module can damage the programs or control blocks of other active ARPANET users, it cannot damage any other part of the host system. OS/MVT allows the NCP to be permanently resident in main memory and to have high-priority access to the CPU. This design is a compromise between efficiency and modifiability.

* The NCP job's region provides a dynamically-sharable memory pool for protocol-dependent transformation modules and ARPANET I/O buffers.

* The NCP executes programs which transform between ARPANET protocols and canonical protocols used internally within the IBM host. The canonical internal protocols are also used for non-ARPANET virtual terminal access to the same user and server subsystems.

* The Exchange is used for all communication between the NCP and the user/server processes within the IBM system. The Exchange provides virtual I/O paths called windows between any two tasks under OS/MVT. As a result, the interaction of these tasks can be defined entirely in terms of the internal protocols used to communicate through the Exchange windows. The Exchange primitives to open and close a window and to transmit data are actually Supervisor Call (SVC) routines.

* The ARPANET-protocol dependence is concentrated in the NCP, thus localizing network protocol changes (e.g., "old" to "new" Telnet). Furthermore, the virtual terminal interfaces to the server subsystems, which often exist in difficult and risky environments, are largely independent of the ARPANET protocol details.

We now describe the internal NCP environment in more detail.

2.1. ICT SUBSYSTEM CONTROLLER

The ARPANET NCP executes as an independent subsystem, i.e., as an unprivileged system job in its own region of main storage. The NCP looks to the operating system OS/MVT like a single task (process), but it multiprograms internally using a general-purpose subsystem controller called ICT [Wolfe74]. The most important functions of ICT are:

* multiprogramming to create internal processes, called psuedo-tasks or ptasks;

* synchronization among these ptasks and between ptasks and real tasks outside the NCP;

* sub-allocation of core memory within the NCP region;

* timing services for the ptasks;

* recovery from failures of individual ptasks;

* maintenance of a dynamic pool of program modules.

The ptasks created by ICT are coroutines, i.e., they always relinquish control to other ptasks voluntarily. This simplifies the design of the NCP, as ptasks can manipulate common data structures without requiring mutual exclusion. ICT is a commutator, that is, it dispatches ready ptasks with a simple round-robin discipline. The state vector for each ptask is saved in a 256 byte control block called a Pseudo-task Area or PTA.

2.1.1. P-Services

ICT provides the ptasks with a set of system calls known as "P-services". The P-services are actually subroutine calls through a transfer vector whose address appears in every PTA, and are invoked via assembly-language macros [Wolfe74]. The most important P-services are:

* PATTACH

  Fork (create) a (sub-)ptask.

  Following the classical process model, the ptask which called PATTACH becomes the "parent" or "superior" of the new ptask. ICT maintains the ptask family tree, and when a ptask terminates ICT forces inferior ptasks to terminate also.

  The PATTACH caller specifies the name of the load module to be loaded and executed by the sub-ptask.

* PEXIT

  Voluntarily terminate the caller's ptask.

* PDETACH

  Force an inferior ptask to terminate (PEXIT).

* PWAIT

  Block the calling ptask (coroutine) until some combination of events occurs. Thus, PWAIT provides process synchronization among ptasks and between ptasks and real OS/MVT tasks, as well as timing services.

* PPOST

  Send a "wakeup" signal to a ptask, by signalling a particular binary semaphore (see below).

* PCORE GET, PCORE FREE

  Obtain or free memory sub-allocated within the NCP region, in 256 byte pages.

* PLOAD, PDELETE

  Load a transient load module from a system library, or delete it. If the module is marked "Reentrant" and "Reusable", it will be shared; ICT maintains a responsibility count to determine when to physically delete a shared module from the region. Modules may also have aliases.

  PATTACH invokes PLOAD to obtain the sub-ptask load module.

* PEXOPEN, PEXCLOSE

  Open, close an Exchange window.

* PSPIE, PSTAE

  Recover from a failure in the calling ptask.

2.1.2. Ptask Synchronization

Using PWAIT, a ptask can wait on any combination of the following kinds of event signals:

(1) a list of real OS/MVT Event Control Blocks (ECB's);

(2) a list of pseudo- (or "internal") ECB's;

(3) any subset of the seven binary semaphores (called "flags") that are associated with each PTA.

(4) a specified time of day or time interval.

Real ECB's are signalled with the normal OS/MVT Supervisor Call (POST SVC), while internal ECB's are posted by another ptask simply setting their "complete" bit. Six of the binary semaphores are assigned particular meanings by the NCP and are named accordingly (see Table 1). However, a ptask may use them for other purposes.

Table 1. Standard ICT Binary Semaphore Names

PWAIT operand	Bit Name	Standard Meaning
OPEN	PTAFCOPN	Remote host has requested open
CLOSE	PTAFCCLS	Remote host has requested close
INPUT	PTAFCINP	Input has arrived from ARPANET
OUTPUT	PTAFCOUT	Output to ARPANET is completed
ATTN	PTAFCATN	Out-of-band signal from ARPANET
CORE	PTAFCCOR	PCORE request is now satisfied

2.1.3. Resources

An ICT ptask is also the owner of resources. There are three kinds of resources:

* load modules, dynamically loaded by PLOAD or PATTACH;

* main storage, obtained with PCORE;

* Exchange windows, opened with PEXOPEN.

ICT will free all resources owned by a ptask when it PEXIT's or is PDETACHed. In particular, ICT will close all open Exchange windows by calling PEXCLOSE implicitly, and it will delete all PLOADed modules by calling PDELETE implicitly.

There are P-service calls that allow a ptask to pass ownership of a resource to another ptask.

2.1.4. A-Services

NCP routines obtain ARPANET-dependent services by issuing local system calls known as "A-services". Macros are provided for coding A-service calls [WolBr79]. The A-services are simply subroutines since the entire NCP operates within the same protection domain, the NCP job.

Because many NCP routines are loaded dynamically, the A-service subroutines must be located via a resident transfer vector whose address is contained within a PTA field (PTAATRV). In general, an NCP routine will have its PTA address in a register (R11 by convention) in order to issue A-service and P-service calls.

Certain A-services operate as extensions of corresponding P-services. For example, an NCP ptask always terminates, whether voluntarily or not, by entering PEXIT. PEXIT in turn calls the A-service routine AEXIT to free ARPANET-specific resources; then PEXIT frees ICT resources as discussed earlier. The exact sequence of events when a ptask terminates is discussed in Appendix D.

For full details on ICT and the P-services, see OAC Systems document "ICT Monitor Services and Macros" [Wolfe74].

2.2. NCP PROCESS STRUCTURE

The basic unit of activity within the NCP is a session. A session which is created as the result of a service request received through the ARPANET is called a server session. Alternatively, a session may be created as the result of a request from a local process, generally to act as a user of a remote server program; this is called a user session. Sessions are designated by a 16-bit integer called the session number.

A session will normally require one or more ARPANET connections (logical data streams) for communication with the remote host. The semantics of sessions and connections and the corresponding control blocks are discussed below.

2.2.1. Dynamic ptasks

Communication is performed by programs executing under ptasks which are dynamically created and destroyed as sessions start and terminate [Bra77]. These communication ptasks are either User Level Protocol Processes (ULPP's) or Host Control pTasks (HCT's).

* ULPP -- User-Level Protocol Process

  For each active ARPANET session, a set of one or more ULPP ptasks will execute programs particular to the user-level protocol(s) used by that session. Some of these programs implement ARPANET service functions (e.g., FTP) entirely within the NCP subsystem. However, most ULPP's relay data between the ARPANET and Exchange connections [BraFe72] to local user and server processes outside the NCP.

  In general, ULPP's are protocol transformers, i.e., they convert between their particular ARPANET user-level protocols and corresponding internal protocols used through the Exchange windows.

  The ULPP modules are loaded dynamically from the NCP load module library by PATTACH. To start a session, an NCP module calls PATTACH to fork a primary ULPP ptask executing the appropriate user-level protocol module. This ULPP may in turn fork inferior ULPP's, forming a ptask sub-tree for the session with the primary ULPP at its root.

* HCT -- Host Control Task (AHHP only)

  There will be an active HCT ptask for every ARPANET host which is currently communicating through the NCP using AHHP. Internet sessions do not have HCT's.

An HCT performs host-specific processing for AHHP. Most importantly, an HCT performs the _outgoing_ _logger_ and _incoming_ _logger_ functions to create user and server sessions (respectively) using AHHP. Specifically, the HCT executes an ICP sequence and then forks the primary ULPP ptask.

2.2.2. Fixed Ptasks

Within the NCP, there are six _fixed_ _ptasks_ which will always be present even when the NCP is completely idle. Figure 2 shows the ptask tree structure of the NCP.

* NCP Ptask

    The NCP ptask decodes the leaders and AHHP headers of messages which are received from the ARPANET, and handles much of the IMP-host protocol. In addition, it handles the receive-side of AHHP.

    NCP includes an intercept mechanism for filtering "raw" packets received from the IMP, as described under "ARPANET GATEWAY" below. In particular, this filter mechanism diverts all internet packets to the IPP.

* IMPIO Ptask

    IMPIO is the I/O driver process for the hardware connection to the IMP. IMPIO builds channel programs, issues the Supervisor Calls (EXCP) to initiate Read and Write operations to the IMP, and analyzes the results upon completion of these operations.

* LOGGER Ptask

    LOGGER handles startup and shutdown of the NCP and/or the IMP. LOGGER also initiates the "outgoing logger" function, creating a new user session in the NCP and causing it to connect to a remote server. For this purpose, LOGGER always has a pending Exchange OPEN with a "well-known" symbolic tag for each user-level protocol. A local process starts a user session by issuing a matching Exchange OPEN request and passing the remote host name and contact socket number through the window to LOGGER.

    For AHHP, LOGGER passes the outgoing logger request to the HCT for the remote host, which then performs the required ICP sequence and forks the primary ULPP. If there is no HCT ptask for that host, LOGGER forks one. For IP, LOGGER starts up a transient INPOLOG ptask (see below) to initiate the outgoing logging function.

Finally, LOGGER is part of the "incoming logger" function for AHHP. The NCP ptask will request LOGGER to fork a new HCT when an ICP request arrives for a local server process and there is no corresponding HCT.

* INPTASK-- IPP ptask

  INPTASK is the primary IPP driver ptask. It handles input, timeouts, and outgoing logging requests for all internetwork protocols, including IP and TCP. A module executed under this ptask issues the PATTACH to fork the primary ULPP for a user or server session using an internet protocol, making the primary ULPP ptask its direct descendant.

  The INPTASK module itself is resident. However, it issues PLOAD to dynamically load the main IPP module, INTMOD. INTMOD will PLOAD the proper higher-level protocol module (e.g., TCPMOD) when needed and PDELETE the module when the protocol becomes idle.

* INTERNET -- IPP control ptask

  This ptask, created by LOGGER when the NCP job starts, starts the internet protocol program IPP by forking INPTASK. If INPTASK ever exits (due to operator action or program failure), INTERNET cleans up and restarts INPTASK.

* MSGMAIN -- MSG ptask

  This ptask, really a very complex ULPP, is the primary controller for the MSG interprocess transaction protocol used by the National Software Works [RivBL77]. The MSGMAIN ptask is created by LOGGER when the NCP starts.

2.2.3. Transient Ptasks

In addition to the session-related HCT and ULPP ptasks and the fixed ptasks, there are transient ptasks which perform particular functions and immediately vanish. Examples of transient ptasks include:

* ARPASRST

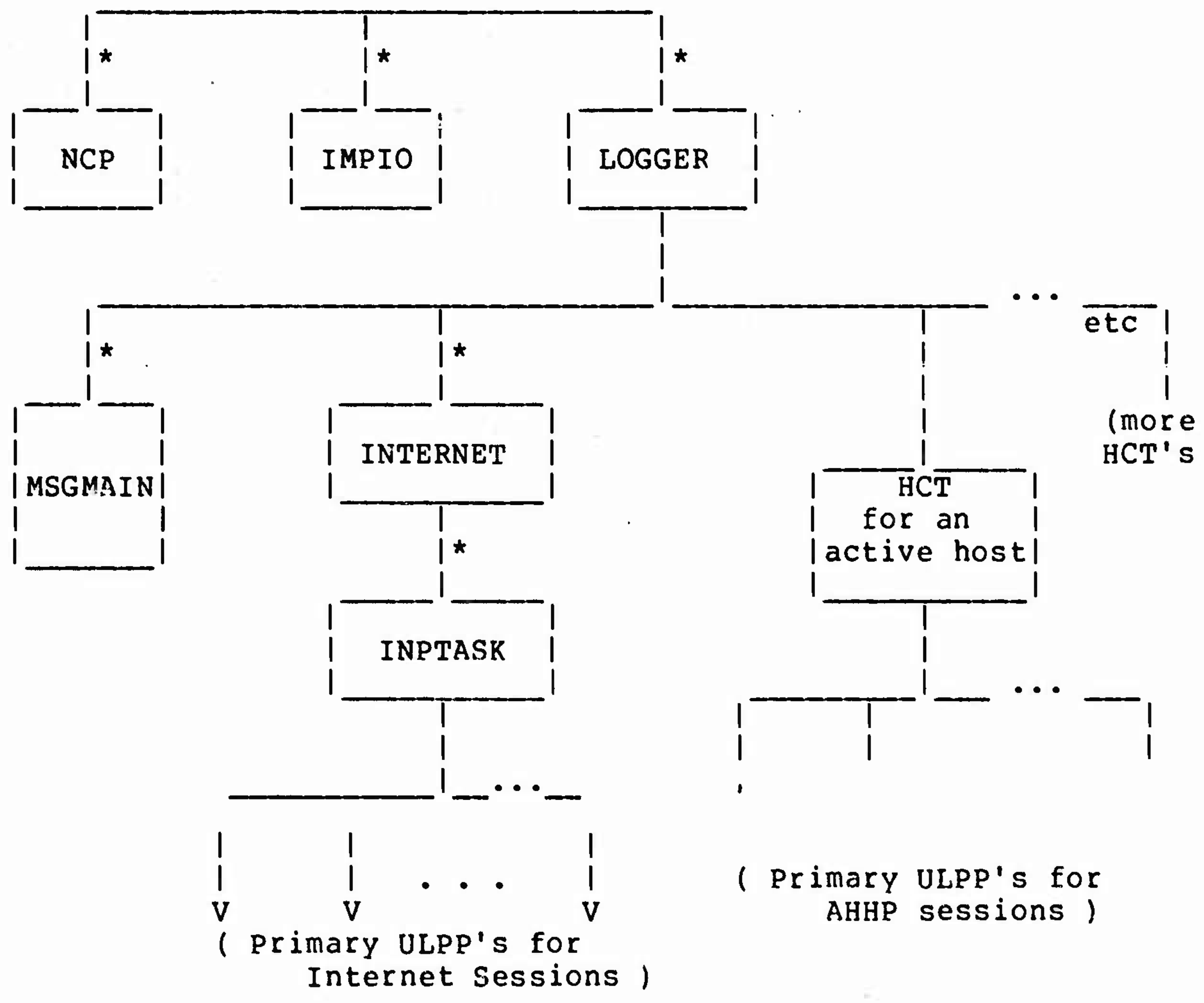
  This send-Reset ptask is forked by NCP initialization to send an AHHP host-to-host RST (Reset) command to every ARPANET host.

* INPOLOG

  This transient routine initiates outgoing-logging for internet sessions, by parsing a character string defining the desired session (see Appendix A). Assuming the parse is successful, INPOLOG creates an "Outlog Queue Element" (OLQE) for the request and enqueues it for IPP, then calls PEXIT and vanishes.

The following figure shows the basic ptask structure of the NCP.

Figure 2. Ptask Tree Structure in NCP



*: Denotes fixed ptask.

2.3. AHHP AND INTERNET ENVIRONMENTS

A single ULPP may use different "higher-level" internet protocols simultaneously, but it may not use both an internet protocol and AHHP. A ULPP for a session using internet protocol operates in an environment which is different from, but nearly compatible with, the environment seen by a ULPP using AHHP. It is convenient to use the terms "internet ULPP" and "AHHP ULPP" to describe ULPP's operating in the specified environments. However, note that the two environments are designed to be essentially compatible from the viewpoint of the ULPP, so that the same ULPP code can be used in either environment.

The A-service system call routines for AHHP and internet (TCP) protocols must therefore implement compatible semantic models for a connection. We say that the internet A-service routines provide a <u>compatibility interface</u> to the ULPP's, i.e., they emulate as nearly as possible the corresponding A-service routines used for AHHP.

The compatibility interface allows only connection-oriented usage of TCP. A new set of A-Services will be required to use TCP as a transaction-oriented or datagram-like service.

Note that:

> the primary ULPP ptask for an AHHP session will be directly inferior to an HCT, while a primary internet ULPP ptask will be directly inferior to INPTASK.

AHHP and internet protocol use <u>different</u> A-service transfer vectors.

> <u>A ULPP is in the AHHP (internet) environment when the PTAATRV field of its PTA points to the AHHP (internet, respectively) transfer vector.</u>

Appendix B contains a list of A-services for both the AHHP and internet environments.

When a ptask is created, the PTAATRV address in the new PTA is set equal to the creator's PTAATRV. The result is to propagate the A-service transfer vector down the ptask tree. Since the INPTASK PTA points to the internet transfer vector, all internet ULPP's will also have the internet A-service vector, for example.

In addition to its A-service transfer vector, the internet environment includes a resident control area called the "P3CB" (explained under "STANDARD ULPP ENVIRONMENT", below).

The IPP design allows the possibility of more than one active IPP instance concurrently, each with its own internet environment. For example, a second environment might be used for testing new IPP versions. A new environment would be created by the INTERNET ptask forking a new INPTASK ptask, and would have its own A-service transfer vector and P3CB.

When a primary ULPP ptask is forked by either a HCT (AHHP) or by INPTASK (internet), the ULPP's PTA contains an ICV (Initial Connection Values) parameter list. The ICV list defines the initial "logging" connection(s), i.e., the initial connection(s) opened as a result of the logger function. The ICV includes the session number and a specification of the remote host.

2.4. NCP LOAD MODULE STRUCTURE

The previous section discussed the NCP structure in terms of its component processes. Now we consider the load modules which are used. It is convenient to divide the NCP program modules into three categories:

* ULPP routines, which are dynamically loaded (usually by the PATTACH P-service) to handle the user-level protocols for active sessions.

* the Telnet access method, a set of resident reentrant subroutines which ULPP's can invoke to handle the Telnet protocol. These subroutines provide a standard Telnet I/O interface, including nearly all Telnet protocol translation and control functions required by any ULPP [Tol77].

  The Telnet access method is invoked with the macros:

  ATOPEN-- open a Telnet connection

  ATCLOSE -- close a Telnet connection

  ATPUT -- send data on Telnet connection

  ATGET -- receive data from Telnet connection

  The routines themselves are located on the A-service transfer vector(s).

* a set of routines collectively called the ARPANET Control Program or ACP, concerned with the host-host and IMP-host protocols.

The ACP includes both resident and dynamically-loaded modules for AHHP and internet protocols. All resident ACP modules are linkage edited into the resident NCP load module ARPAMOD. The TCP code and the bulk of the IPP code are contained in dynamically-loaded modules:

INTMOD for IPP

TCPMOD for TCP

ARPAMOD includes all resident modules, which generally perform the following functions (see Appendix B):

* Commutator Support Routines

  These routines perform NCP-specific functions related to creating and destroying ptasks.

* ULPP Environment Creation and Control

  These modules control the creation of dynamic modules, clean up when a ULPP exits, and create the standard control-block environment for a ULPP (described under "STANDARD ULPP ENVIRONMENT", below).

* ARPANET Gateway Routines

  These routines handle the IMP-host protocol and provide a logical "gateway" to the ARPANET. They include the IMPIO and NCP routines which are executed by the ptasks of the same names, as discussed earlier. See subsection "ARPANET GATEWAY", below.

* AHHP Connection A-Services

  These are the A-service subroutines that AHHP ULPP's call to create and manipulate connections.

* AHHP Protocol Modules

  These are internal ACP subroutines that implement AHHP.

* Resident IPP Code

The functions listed so far belong to the ACP. In addition, ARPAMOD includes:

* Telnet Access Method routines

* Resident Tables

Appendix B includes a list of actual module names within these functional categories; notice that in some cases a single module fits within more than one category.

Most of the modules in ARPAMOD are either executed by fixed ptasks or are called as A-services. The AHHP A-service routines all have names of the form: ARPAxxxx, while the corresponding internet A-service routines in the compatibility interface have names of the form: ARPIxxxx. The ARPAxxxx routines are linkage edited into ARPAMOD. However, the ARPIxxxx routines are part of the dynamically-loaded IPP module INTMOD, as we will describe under "INTERNET LAYER DESIGN", below.

Not all A-service modules differ between the AHHP and internet environments. The A-service routines concerned with environment creation and control as well as the Telnet access method routines can be almost identical in the two cases, differing by only a few instructions. Therefore there is only a single version of these modules. The important ULPP control blocks have a common flag bit which is off in the AHHP environment and on in an internet environment; the common A-services test these bits when necessary to select appropriate environment-dependent instructions.

Within the ACP, there are some standard interfaces which the host-host protocol routines use to invoke gateway functions and to manipulate the control block environment [BRA79A], ensuring compatibility. Most of these internal interfaces appear on an auxiliary transfer vector, called ARPXTRV, which in turn appears on every A-service transfer vector. These interfaces routines are invoked by the ACPX macro, and are listed in Appendix B.

2.5. ARPANET GATEWAY

Those modules of the ACP which handle the lowest protocol layer, the IMP-host protocol, are referred to as the "ARPANET gateway". For explanatory purposes, it is convenient to model the gateway routines by two functions, the Incoming Gateway <u>AGAWI</u> and the Outgoing Gateway <u>AGAWO</u>.

2.5.1. AGAWO -- Outgoing Gateway Function

Given a parameter list defining a message to be sent, a destination host and link number, and the type of service desired, the Outgoing Gateway will prefix an appropriate ARPANET leader and send the resulting packet to the IMP hardware interface. The parameter list is called a <u>Write Request Element</u> or <u>WRE</u>, and the call is coded with the ACPX QUEOUT macro.

An ACPX QUEOUT call adds the WRE to the IMP output queue and signals the OUTPUT semaphore of the IMPIO ptask. When the path to the IMP is free, IMPIO builds an ARPANET leader for the message as well as a channel program containing a Write operation and pointing to the data, and issues an OS/MVT Supervisor Call to start the Write operation.

When the Write operation completes, IMPIO deletes the WRE from the output queue and signals completion of the request. The exact manner of signalling differs for AHHP and IP [Bra79A]. For AHHP, AGAWO simply enqueues the WRE on the DONE Queue. The subsequent receipt of a RFNM (or a negative acknowledgment) for the same link number causes NCP to remove the WRE from the DONE Queue and complete processing of the send request.

The DONE Queue is not used for the IPP, however, because all messages use the same link number and because IPP may use Subtype 3 (Uncontrolled) messages which return no RFNM or other acknowledgment from the subnet. Therefore, if the WRE is marked "Uncontrolled" or "Not AHHP", then AGAWO simply omits the WRE's sojourn on the DONE Queue and marks it "completed" immediately.

Thus, there is no direct signal to IPP that a send request has completed and the WRE is free. The IPP must depend upon being awakened either by the receipt of a host-host acknowledgment message (<ACK>, in the case of TCP) or else by a timeout. It must treat WRE's as a relatively plentiful resource.

AGAWO has an interface entered from AGAWI to send irregular (host-to-IMP) messages using a private pool of WRE's. AGAWO also includes an IMP queue purge function, which is invoked by the ACPX HALTIO macro call. This call searches the AGAWO output and NOW queues for any WRE's pointing to a given CCB (or its internet equivalent), and dequeues them.

2.5.2. AGAWI-- Incoming Gateway

The incoming gateway function is performed by parts of IMPIO and the NCP ptask. IMPIO keeps a hardware Read operation pending to the IMP. This Read completes whenever the IMP sends the last bit of a message to the host interface, and the NCP ptask is awakened as a result. The AGAWI portion of the NCP ptask interprets the ARPANET leader to determine the message type and link number. Irregular messages are in general handled by AGAWI, but some are passed to the AHHP part of the ACP.

AGAWI includes an intercept to filter "raw" packets from the ARPANET; this mechanism is called the "Network Measurement Center intercept" for historical reasons. The leader of each received message is compared with a set of filters. If the leader matches an active filter, AGAWI copies the message into an associated buffer and signals the INPUT semaphore of the corresponding ptask. An intercept buffer is capable of holding more than one message, so each message is preceded by an 8-byte header which contains the message length.

The same message may be intercepted by one or more filters as well as the normal AHHP mechanism. Furthermore, there is a similar mechanism in AGAWO for outgoing packets, so a given filter may select incoming and/or outgoing packets. To establish a filter using the NMC intercept, a ptask calls the NMC-Intercept Open/Close A-service ANMOC. See Reference [Bra79A] for details.

In particular, the IPP ptask INPTASK establishes an incoming filter for the internet link number (currently 155), so that an arriving internet packet will be copied into the buffer and INPTASK awakened. Although the buffer is governed by pointers like a normal input circular buffer, it is not used in a circular manner; therefore, the IPP can assume that a single packet is in contiguous memory. The IPP is expected to process the packet "promptly", moving it from the intercept buffer into a segment reassembly buffer (see "INTERNET LAYER DESIGN").

2.6. STANDARD ULPP ENVIRONMENT

A ULPP is concerned with the basic communication objects: sessions, connections, and Telnet connections. For each of these objects, there is a corresponding control block:

* Session => Account Control Element or ACE;

* Connection => Connection Control Block or CCB;

* Telnet Connection => Telnet Connection Control Block or TCCB.

These control blocks are chained together in a manner to reflect their inter-relationships (see Figure 3). These chains and the control block formats are important aspects of the "environment" seen by any ULPP.

ACE's and TCCB's are used in both the AHHP and the internet environments, with no significant differences. However, the format of a CCB is (partly) dependent upon the particular host-host protocol in use. The CCB-analogs in the internet environment are called "hlpB's", where "hlp" denotes a three-letter mnemonic for the particular higher-level protocol. For example, a TCP connection is controlled by a TCPB.

As discussed previously, the ACP is designed to provide a compatible environment for both AHHP and internet ULPP's. This requirement for compatibility implies the following general conditions:

* The A-service routines for AHHP and internet protocols must implement a "universal" semantic model for a connection (described in Appendix C).

* Those fields of the ACE, TCPB, and CCB (or equivalent hlpB) that are used by a ULPP must be the same in both environments. This implies in particular that certain fields of a TCPB must exactly correspond to fields of a CCB; those fields are listed in Appendix C. The other CCB/TCPB fields, which depend upon the host-host protocol, will be used internally by the ACP but generally may not be used by a (compatible) ULPP.

* It must be possible for a ULPP (or an A-service called by a ULPP) to determine which environment it is operating in. Thus, any control block which differs in the two environments must have a common flag bit. This bit, called the "Not Host-Host" bit, is off in the AHHP environment and on in the internet environment.

* The host and socket parameters passed to the primary ULLP in the ICV list should cause the ULPP to issue a corresponding sequence of AOPEN's for AHHP and TCP, to have the proper connection be completed in either case.

Perfect AHHP/internet compatibility is impossible because of the real protocol differences outlined in the introductory section. For example, there are small but significant differences in the connection states which must be observed if a ULPP is to operate correctly in both environments (see Appendix C). We have attempted to minimize the impact of these differences on the ULPP's.

Many user-level protocols open a pair of (simplex) AHHP connections corresponding to a single (duplex) TCP connection. Fortunately, in most cases such a connection pair uses the Telnet protocol and is manipulated only by the common Telnet access method. This centralizes many of the compatibility problems in the Telnet access method subroutines. These subroutines contain code which tests the environment and executes a few instructions differently for AHHP or internet. The incompatibilities are in two areas: (1) a pair of simplex connections vs. a single full-duplex connection, and (2) "urgent" vs. "interrupt" signalling.

We can now describe the semantics of sessions, connections, and Telnet connections. We will assume the compatibility interface, and will discuss only those fields of a CCB/hlpB that are common to both environments. Therefore, we can speak of a "CCB" and imply either a CCB or any analogous hlpB (in particular, TCPB).

Whenever the HCT exits, the corresponding control CCB is deleted (closed), and as a result the ACE's chained from it are also deleted. This will normally occur as the result of receiving a "Host Down" irregular message for that host.

(5B) In the internet environment there are no HCT's, so all ACE's are chained from the IPP control area. To simplify compatibility in various ACP routines, some fields of this area are formatted to correspond to a control CCB. For this reason, the IPP control area is called a _pseudo control CCB_, abbreviated _P3CB_.

(6) An ACE (and session) is deleted by the A-service ACESELL. ACESELL may be called explicitly by a ULPP (presumably the one that called ACEBUY), or implicitly when:

* the primary ULPP ptask owning the ACE exits; or

* the HCT for the host with which the ACE is associated exits (AHHP only).

(Note: in the (common) case that the session was created by the incoming/outgoing logger, the primary ptask will be directly inferior to the HCT, and these two conditions will be logically equivalent. In the case that the connection is opened by a ptask not in the subtree of the primary session PTA, ACESELL has a more complex effect; see the ACESELL writeup).

Before writing an accounting record and deleting the ACE, ACESELL will close all connections open within the session, thereby freeing all CCB's and TCCB's chained from the ACE.

2.6.1.2. ACE Contents

An ACE includes the following fields:

* Unique session number (ACESESS)

* 10-byte user identification string (ACEUSER)

* User-level protocol name (ACESYS)

* Remote host id (ACEHOST)

* Flags (ACEFLG)

  The flag bit ACEF1NHH will be off for all ACE's in the AHHP environment, and on for all internet ACE's.

Figure 3 illustrates the control block chains involving ACE's, the control CCB, and CCB's. It shows n ACE's chained from the Control CCB. ACE 1 belongs to "PTA 1,0", and has two CCB's chained from it. "CCB 1,1" belongs to "PTA 1,1" and has pointers back to the ACE and to the Control CCB. In the internet environment, the "Control CCB" will be the P3CB, and the "HCT PTA" will be the INPTASK PTA.

Figure 3. Principle Control Blocks in ULPP Environment

```
    CCCB
 __________               ACE 1                     ACE n
|          |            _________  ACE chain      _________
| Control  | --------> |      ACEo--------...--> |      ACEo--*
|  CCB     |           |         |               |         |
|          | <---------oCTRL     |       CCCB<---oCTRL     |
|          |           |         |               |         |
|          o-->HCT     |  o      o-->PTA         | o       o--->P
|__________|  PTA      |__|______| 1,0           |_|_______|    n
                          |                        |
                          |                        |
                    CCB   |                  CCB   |
                    chain |                  chain |
                          V                        V
                     ___________
                    |           |
                    |   CCB     o-->ACE 1
        CCCB<-------o   1,1     |
                    |           o-->PTA 1,1
                    |_____o_____|
                          |
                          |
                          |
                          V
                     ___________
                    |           |
                    |   CCB     o-->ACE 1
         CCCB<------o   1,2     |
                    |           o-->PTA 1,2
                    |_____o_____|
                          |
                          *
```

2.6.2. CONNECTIONS

For each network connection there is a CCB (or TCPB) which contains all the relevant pointers, queues, and state variables. The address of this block is the "handle" used within the NCP for naming the connection.

2.6.2.1. Socket and Port Numbers

A connection is terminated at each end by a "socket". In AHHP, for example, the full name of a socket is the pair:

(<32-bit number>, <host address>)

where <host address> is the address of the remote host on which the connection terminates. At times, the <32-bit number> is itself called "the socket". Similarly, a TCP socket is named by the pair:

(<port number>, <host-address>)

where <port number> is 16 bits.

Within the UCLA NCP, there are 32-bit numbers associated with both the local and remote ends of a connection; these numbers will be called the Local Socket Number and Remote Socket Number, respectively. They obey the rules:

* The Local Socket Number _must have the session number in the high-order 16 bits_.

* For AHHP, each 32-bit Local Socket Number must be unique and different from any TCP Local Socket Number (the session number guarantees the last).

The following table summarizes the assignment of Local Socket and Remote Socket Numbers. For AHHP, the ICP sequence assigns a Local Socket Number subspace of 2**16 socket numbers; the values shown in the table under AHHP are the origins of this subspace. The actual socket used for connections generally have small offsets from these origins.

Figure 4. Local Socket and Remote Socket Numbers

	Local Socket <u>Number</u>	Remote Socket <u>Number</u>
<u>AHHP</u> (Socket subspace from ICP)		
User Session	<sess#, 0>	S-sock
Server Session	<sess#, 0>	U-sock
<u>TCP</u>		
User Session	<sess#, L-port>	<0,WK-port>
Server Session	<sess#, WK-port>	<0,U-port>

Here:

The notation <a,b> represents a 32-bit number, composed of two 16-bit quantities a and b; the high-order part is a.

"sess#" is a 16-bit session number.

"S-sock" is the 32-bit socket number supplied by the remote (Server) host.

"U-sock" is the 32-bit socket number supplied by the remote (User) host.

"L-port" is a unique 16-bit local port number. It will not be in the range of a WK-port.

"WK-port" is a 16-bit well-known (i.e., contact) port, in the range 0-255.

"U-port" is the 16-bit port number supplied by the remote (User) host.

2.6.2.2. Initial Connection Values

The primary (root) ULPP ptask is created by the HCT (for AHHP) or by INPTASK (for an internet protocol). In either case, it is started with a parameter list called the ICV (Initial Connection Values) in the "user" field of its PTA. The ICV format is:

PTAUSER+4:	4 bytes:	Address of the ACE for session.
PTAUSER+10:	1 byte:	Default byte size (AHHP), or Protocol id (internet)
+11:	1 byte:	Remote Host Id
+12:	4 bytes:	Local Socket Number
+16:	4 bytes:	Remote Socket Number
+20:	4 bytes:	ICP contact socket (AHHP server session), or contact port (internet server session), or Exchange Window Id (user session)

The Local Socket Number and Remote Socket Number values are those shown in Figure 4 above.

2.6.2.3. Host Id's

NCP routines seldom deal directly with actual 24-bit ARPANET host addresses or 32-bit internet host addresses. Instead, they use a one-byte "handle" called a host id to refer to a host address.

A host id is mapped into the corresponding host address when a message is sent to the ARPANET and when error messages are composed. The details of creation of a host id differ in the AHHP and internet cases, but generally the host id for the session is passed in the ICV to the primary ULPP by the incoming/outgoing logger function.

2.6.2.4. Connection Semantics

The semantics of a connection are as follows:

(1) A CCB is created with the ALSTN ("Listen") A-Service. The parameter list contains:

* Local Socket Number (32 bits);
* Remote Socket Number (32 bits);
* Remote Host id (8 bits);
* Byte Size (AHHP only), or Higher-level protocol id (internet only); (8 bits).

ALSTN uses the session number from the high-order 16 bits of the Local Socket Number together with the remote host id to locate the session ACE to which the connection is to belong. ALSTN chains the CCB from this ACE (see Figure 3), and stores pointers to the ACE and the corresponding control CCB in the new CCB.

The ULPP PTA which issued the ALSTN will "own" the CCB.

Under AHHP, ALSTN can be called only once for a given connection; under TCP, there is no such restriction. In any case, a successful ALSTN call returns the address of the CCB/TCPB as a "handle" for the connection.

(3) To open a connection, the connection handle is passed to the AOPEN A-service. Under AHHP, AOPEN may need to be called twice for an active open (i.e., an open request that is initiated locally). See Appendix C for details.

(2) The connection will be closed and the CCB deleted when:

* a ptask calls the ACLOSE A-service (in some cases, two separate calls are necessary); or
* the owner ptask exits, causing the ACP to call ACLOSE implicitly.

In the AHHP environment, the owning ptask will be forced to exit if the HCT (pointed to by the control CCB) exits, e.g., if the host goes down. Normally, the ptask owning the connection will be subordinate to the HCT ptask, so this would happen necessarily; however, it is possible for a

ULPP to open a connection for a HCT which is not its superior. Thus, in Figure 3 "PTA 1,0" is normally, but not necessarily, subordinate to "HCT PTA"; in any case, if the HCT ptask exits, the control CCB will be deleted and the ptasks of all CCB's that point to the Control CCB will be forced to PEXIT. This in turn will ACLOSE all CCB's that point to the control CCB.

(3) In the AHHP environment, a connection will receive (send) data if Local Socket is even (odd, respectively). An internet connection is inherently full-duplex, allowing both send and receive operations.

(4) A ULPP sends data to the ARPANET by building a parameter list called a Write Request Element (WRE) which points to the CCB and to the data to be sent, and passing the WRE address to the ASEND A-service.

When the data has been sent and acknowledged by the remote host, the "Completed" flag will be turned on in the WRE and the ULPP ptask's OUTPUT semaphore will be signalled.

(5) Data is received in a circular buffer owned by the ULPP. Whenever data arrives, the ULPP ptask's INPUT semaphore will be signalled.

The ULPP may use the ARECV A-service to remove data from this buffer (preferable), or may itself manipulate the buffer pointers in the CCB. In either case, the ULPP invokes the ARLSE ("Release") A-Service to inform the ACP that data has been consumed from the buffer.

(6) When an out-of-band signal arrives, a field in the CCB/TCPB is updated and the ATTN (Attention) semaphore is signalled. The specific mechanism differs for AHHP and for TCP:

* For AHHP, receipt of an interrupt (INS or INR) command increments a count (CCBINC) in the CCB by 1 and signals ATTN.

* For TCP, there is an Urgent Data Count field (TCPRURGN) in the TCPB. This is the number of bytes which the ULPP needs to remove from the buffer to read all urgent data (and may exceed the bytes currently in the buffer). Whenever this value advances, the ATTN semaphore is signalled.

(7) Occurrence of a connection-related event causes the appropriate semaphore (OPEN, CLOSE, INPUT, OUTPUT, ATTN) of the owning ptask to be signalled. There is a single set of semaphores for each ptask, shared by all connections it owns; therefore, it is generally necessary to test state flags in each CCB to determine which connection had a state change. The state flags are discussed below, and the rules for coding system calls for both AHHP and TCP connections are contained in Appendix C.

2.6.2.5. CCB Contents

Appendix C contains a list of the CCB/hlpB fields which must correspond. Included in these CCB/hlpB fields are:

* The Open/Closed state bits (CCBLOG).

* The address of the PTA under which ALSTN was called, and which therefore owns the connection CCBPTA).

* For AHHP, the address of the appropriate "control CCB"; for TCP, the address of the P3CB (CCBCTRL).

* The 32-bit Local Socket Number used to label the CCB/hlpB; the high-order 16 bits must be the session number (CCBLSCK).

* Pointers used to control the circular receive buffer.

* ACE Address (CCBACE)

  This is the address of the ACE for the session under which this connection was opened.

* ACE Chain Word (CCBCHA)

  This word is used for the ACE chain of all CCB's for this session.

As noted earlier, the out-of-band signalling mechanism differs in AHHP and TCP, resulting in incompatible fields in the CCB and TCPB.

2.6.2.6. Connection States

The A-services assume a standard state diagram for connections. State changes are signalled to the ULPP by signalling the OPEN or CLOSE semaphore and by the value of the "LOG" (CCBLOG) state bits. These bits have the values:

* 00B: Not yet open.

* 01B: Open connection.

* 10B: Pseudo-CCB (see below).

* 11B: Closing or closed.

The "Closing" flag bits LOG= 11B are turned on when a close request (i.e., a CLS command for AHHP, or a <FIN> bit for TCP) is received from the remote site. At the same time, the CLOSE semaphore for the ULPP that opened the connection is signalled. The semantics of this value are as follows:

* For an AHHP send connection, LOG=11B indicates that no more ASEND's may be issued; however, ACLOSE will wait for completion of any ASEND's which are pending.

* For a TCP connection, the ULPP may continue to call ASEND for this connection indefinitely after LOG=11B is set and the CLOSE semaphore is signalled; the connection is <u>half-open</u>.

* For an AHHP receive connection or a TCP connection, LOG=11B indicates that no more data will be received; however, there may still be new data for the ULPP in the circular buffer, so the ULPP should issue ARECV and/or ARLSE calls until the circular buffer is empty.

* If a <RST> ("Reset") message is received for a TCP connection, the CLOSE semaphore will be signalled (perhaps for the second time), LOG=11B will be set, and an additional "Reset Received" (TCPFLRST) will be turned on; no data may be sent <u>or</u> received after this.

The ACLOSE call has both blocking and non-blocking forms. Under TCP, the blocking form ("TYPE=WAIT") returns to its caller only after the complete 3-way <FIN> handshake has occurred (or a timeout). If a blocking ACLOSE is used for a simplex receive connection, and if the user-level protocol allows the TCP connection to be half open, the remote process may ignore the <FIN>, causing a deadlock.

Except for this deadlock problem, the handling of LOG=11B for a simplex receive connection is normally compatible between AHHP and TCP. However, the logic may differ for a simplex send connection, due to the possibility of a half-open TCP connection. See

Appendix C for details.

A pseudo-CCB is a control block which has the shape of a CCB and is chained in the environment like a CCB, but is not associated with a real ARPANET connection. When a ptask owning a pseudo-CCB exits, ACLOSE is called to free the pseudo-CCB and any associated circular buffer. Pseudo-CCB's are used, for example, to control NMC intercept filters and for trace buffers (discussed later). See Appendix C for more information.

2.6.3. TELNET CONNECTIONS

The reentrant Telnet access method routines [Tol77] use a "Telnet Connection Control Block" or TCCB to store all state information relevant to a particular Telnet connection. The TCCB address is used as a handle to name the Telnet connection.

2.6.3.1. Telnet Connection Semantics

The semantics of a Telnet connection in the NCP are as follows:

(1) A Telnet connection is a full-duplex path which uses the user-level protocol Telnet.

(2) Telnet uses two (simplex) AHHP connections or one (full-duplex) TCP connection.

(3) A Telnet connection is created by an "ATOPEN" call of the form: ATOPEN(L, R).

Here the parameters L and R are Local Socket Number and Remote Socket Number, respectively; see Figure 4. L and R are usually obtained from the corresponding elements of the ICV by adding small integer offsets.

* AHHP: L and R are both even; ATOPEN opens two connections:

```
SendCCB := ALSTN( L+1, R [illegible]

RecvCCB := ALSTN( L, [illegible]
```

* TCP: ATOPEN opens a single TCP connection:

```
SendCCB := RecvCCB := ALSTN( L, R) ;
```

ATOPEN obtains and initializes the TCCB, and saves in it the addresses "SendCCB" and "RecvCCB" (in the TCP case, these addresses are the same).

(3) To send data over an open Telnet connection, the ULPP uses the ATPUT macro; to receive data, it uses the ATGET macro. In either case, the call may be blocking or non-blocking. A non-blocking call causes the INPUT (OUTPUT) semaphore to be signalled when input is received (output is sent, respectively); a blocking call issues an internal PWAIT on the appropriate semaphore.

(4) Under AHHP, ATPUT and ATGET will fail with return code 12 if the remote site has closed the data stream(s); ATGET will return 12 in the first call after the circular buffer is emptied. The ATPUT/ATGET calls can also specify an "end-of-file exit" routine which will be called in the same circumstances.

Under TCP, ATGET signals a closed data stream exactly as it is signalled under AHHP. However, the ULPP may call ATPUT even after the receive data stream has been closed; ATPUT will signal a closed data stream only if a <RST> (Reset) segment is received. Thus, the user-level protocol can choose whether or not to allow the Telnet connection to remain half open.

(5) The two ARPANET connection(s) composing a Telnet connection will be closed and the TCCB freed when:

* A ULPP issues the Telnet close macro ATCLOSE;

* or the ptask that issued ATOPEN exits.

Note: in the UCLA implementation, a ULPP cannot half-close a Telnet connection under TCP; ATCLOSE always closes both send and receive paths, and is a blocking call. If the remote host has not sent a <FIN> and does not send one within a reasonable period, ATCLOSE will timeout and delete the connection; any subsequent messages from the remote host will invoke a <RST>.

2.6.3.2. TCCB Contents

A TCCB includes the following types of information:

* Addresses of the send and receive CCB's (for TCP, both point to the same TCPB).

* Parameter area for ATGET and ATPUT calls.

* Parameters that control the details of translation to be performed on the data. There are complex options for handling Ascii and Telnet control characters. There is an escape sequence which allows the ULPP to specify a number of these options symbolically with an ATPUT call.

* State information on the connections and translation.

* A save area for calling the A-services to manipulate the ARPANET connection(s).

3. INTERNET LAYER DESIGN

The internet protocol program or IPP implements the Internet Protocol (IP) to send and receive internet datagrams [PosIP]. This section will describe some of the design features of the IPP implementation in the UCLA NCP.

IPP must support a number of different higher-level protocols, each of which is implemented by a corresponding higher-level protocol module (HLPM). IPP accepts from the HLPM's segments of data to be sent to internet hosts, and passes to the HLPM's complete segments which have been received.

The datagram service of IP is "unreliable": a datagram may be delivered out of order, lost, or duplicated. A "higher-level" internet protocol (e.g., TCP) will provide error detection and correction if desired. The data transport functions which IP does support, and which IPP must implement, are [PosIP]:

* internet addressing;

* routing transmitted datagrams;

* demultiplexing received datagrams;

* fragmenting and reassembling internet datagrams.

In addition to these IP functions, IPP includes the following control functions:

* Dynamically loading and deleting HLPM load modules and their resource pools.

* Providing a timing service for HLPM's.

* Creating new ULPP's (user-level protocol processes) in response to incoming and outgoing logger requests.

* Controlling the startup and shutdown of all internet protocol operation.

3.1. OVERVIEW

We will now give a brief overview of the IPP functions, expanding upon the discussion in later subsections. We will sometimes use the term "packet" for "internet datagram".

3.1.1. Higher-Level Protocol Control and IPB's

Generally, the IPP supports and controls the operation of the HLPM's. For each higher-level protocol that is supported, IPP has a fixed data structure called an _IPB_ (_Internet Protocol Block_). An IPB contains all the parameters that IPP needs to control the corresponding protocol. For example, IPP uses information in the IPB to generate a HLPM module name when it is necessary to load the module, and then keeps the loaded module's address in the IPB. IPB's are often referenced by a one-byte handle called a _protocol id_ or _PID_.

The IPP must maintain a pool of buffers for reassembling incoming segments and passing these segments to the HLPM's. IPP provides a separate buffer pool for each HLPM, and each pool grows and shrinks with activity in a manner to be described later. Each IPB contains parameters controlling the dynamic size of its buffer pool as well as the size of each buffer.

3.1.2. Associations and ICB's

Each IPP has its own _internet host address_, composed of 3 parts:

(<network number>, <host number>, <logical host number>)

The <logical host number> may be used to distinguish different internet protocol programs operating on the same physical host.

IP provides only datagram service; thus, it does not define "connections" of any type. However, when it sends or receives datagrams, IPP must be aware of the path to the remote IP program. Thus, there is an implicit and perhaps transient _association_ [CerKa74] between the local IPP and the remote internet host. It is convenient for IPP implementation to introduce an explicit control block for an active association: an _Internet Control Block_ or _ICB_.

A particular ICB is involved in sending or receiving any segment. For example, to demultiplex a packet received from the ARPANET, IPP locates or creates an ICB for its association. To request that the IPP send a segment as an internet datagram, a HLPM specifies the segment address(es) and the address of an ICB.

When the IPP receives a packet, it must demultiplex on: (1) the internet host address of the source (in order to reassemble fragmented segments), and (2) the higher-level protocol number (to select the HLPM to receive the segment). If strict protocol layering were obeyed, any further demultiplexing of logical streams would be delegated to the HLPM.

However, the UCLA IP/TCP implementation was designed to provide an efficient interface to stream-oriented higher-level protocols such as TCP. This is achieved by a mechanism, described below, that extends the association definition to specify a particular logical stream to a remote internet host. The choice of logical stream is specific to a higher-level protocol. We expect that higher-level protocol implementations using this IPP will define logical streams in such a way that IP associations are in one-to-one correspondence to higher-level connections, as TCP does. This will create a one-to-one correspondence between ICB's and the hlpB's used for the connections.

All ICB's associated with a given IPB (hence higher-level protocol) are chained together and in turn point to the IPB. An ICB is initialized from the corresponding IPB. In general, a HLPM is permitted to read values from the ICB's for its associations, but it is not permitted to store into them.

3.2. IPP INTERFACES

The IPP has interfaces both to the ARPANET gateway and to the HLPM's. In addition, the IPP includes (part of) the A-service compatibility subroutines used by ULPP's operating in the internet environment. We will now describe these interfaces briefly.

3.2.1. IPP - Gateway Interface [Bra79A]

The IPP accesses the ARPANET gateway by issuing the appropriate calls:

* ACPX QUEOUT to send messages to the IMP.

* ACPX HALTIO to purge the AGAWO send queues.

* ANMOC to establish an NMC Intercept filter that will select all messages with the internet link number.

3.2.2. IPP - HLPM Interfaces [Bra79B]

The IPP provides a set of <u>INTERNET</u> services to the HLPM's. Since the HLPM's are loaded dynamically, INTERNET services must be called via a transfer vector, INTNETRV. The HLPM's obtain the address of INTNETRV from the A-service transfer vector used in the internet environment. The transfer vector INTNETRV and the internet routines are link-edited into a single module named INTMOD.

The INTERNET macro is used to code calls of the internet services provided by IPP. The major INTERNET services are:

* INTERNET OPEN

  Locate or create an ICB for a specified association.

* INTERNET CLOSE

  Delete an association, freeing the ICB.

* INTERNET OUTPUT

  Send a segment on a given association.

* INTERNET START

  Create a new session by forking a primary ULPP.

* INTERNET TIMER

  Request interval timing service.

On the other hand, the IPP calls certain HLPM routines:

* HLPM INPUT

  Process a reassembled input segment.

* HLPM TIMEOUT

  The time interval requested by the last INTERNET TIMER call has expired.

* HLPM OUTLOG

  An outgoing logger request has been received and parsed.

* HLPM DEMUX

  Generate logical stream number (see below).

These routines are located at canonical offsets in a HLPM transfer vector whose address is found in the IPB. The HLPM macros are documented in "Interface Specifications for Programming a Higher-Level Host-Host Protocol using Internet Protocol" [Bra79B].

Figure 5, below, shows the major IPP/HLPM interfaces.

3.2.3. ULPP - IPP A-service compatibility interface

The A-service compatibility interface implementation must be particular to a higher-level protocol (e.g., TCP). However, it was designed in two layers, an IPP layer and a HLPM layer. Thus, the following sequence occurs when a ULPP issues a connection-oriented A-service call:

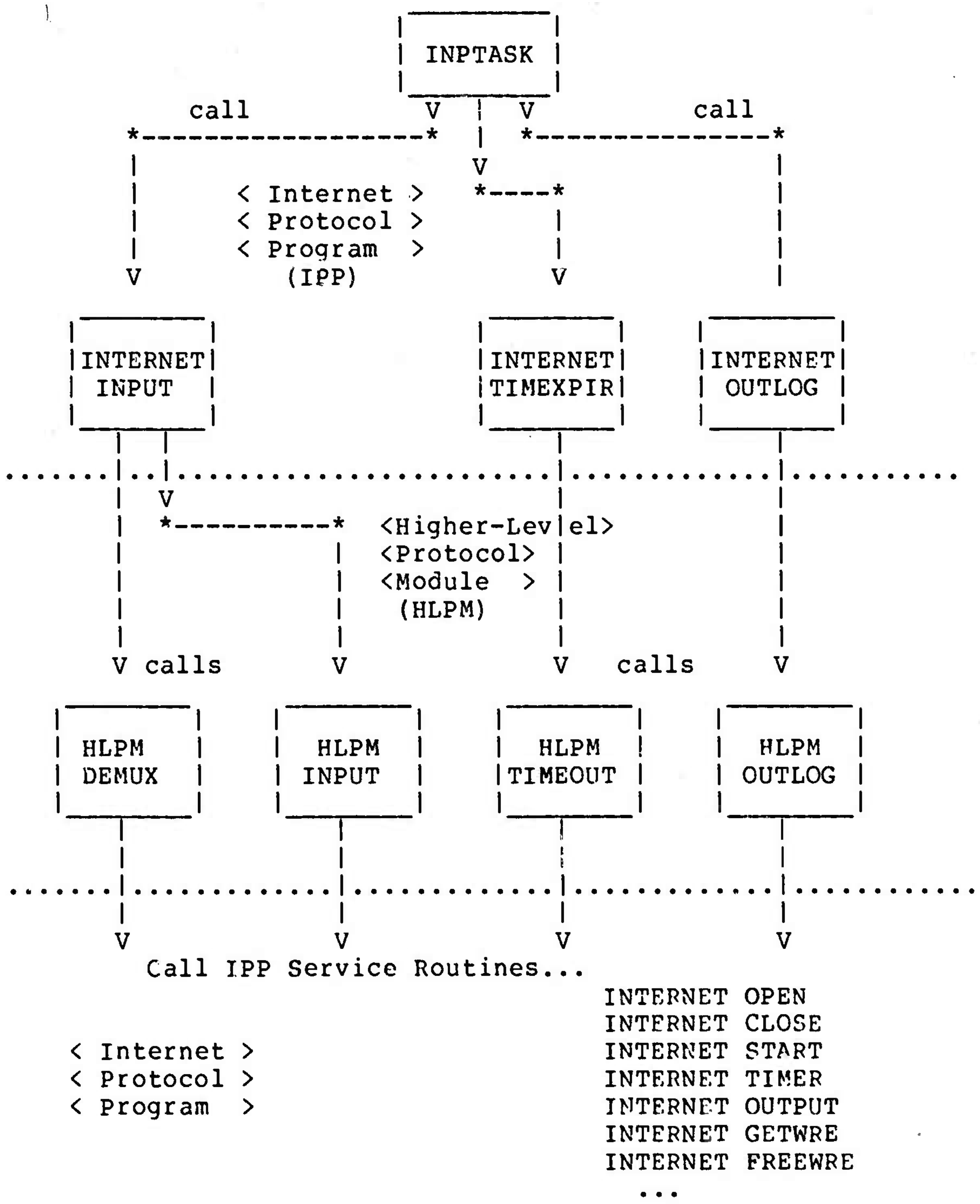
(1) An IPP subroutine with a name of the form "ARPIxxxx" is called. This subroutine is linkage edited into INTMOD.

(2) The ARPIxxxx subroutine in turn calls the corresponding subroutine in the appropriate HLPM, via the HLPM transfer vector. In the case of TCPMOD, the subroutine that is called will have the name "TCxxxx".

The ULPP's call of the ARPIxxxx subroutine traverses two transfer vectors -- from the resident A-service vector to the dynamically-loaded INTNETRV -- to reach the subroutine entry point. This technique added three instructions to the path-length of every call, but significantly eased development and maintenance of IPP. Before this double-linkage was developed, the ARPIxxxx subroutines were linkage edited with ARPAMOD; this meant that the production NCP had to be restarted every time the ARPIxxxx code was changed.

Further discussion of the compatibility interface is deferred to the section "TCP LAYER DESIGN", below.

Figure 5. IPP/HLPM Interfaces



3.3. IPP PROCESS STRUCTURE

INPTASK is the controlling ptask for the internet layer; in addition, some of the HLPM functions execute under INPTASK. The INPTASK ptask executes a driver module, also named INPTASK, to perform the following functions (see Figure 5):

* Load INTMOD

  INPTASK issues a PLOAD to load the main IPP module INTMOD dynamically the first time that an internet packet arrives or an outgoing logger request is made for an internet protocol. The entry point address, which is the address of INTNETRV, is stored into the A-service transfer vector.

  INPTASK could delete INTMOD whenever the internet protocols are completely idle, but it does not in the present implementation.

* Obtain Input

  When it is forked by the INTERNET ptask, INPTASK calls the ANMOC A-Service to establish a filter for the internet link (155). Whenever a packet arrives on this link, the incoming ARPANET gateway moves the packet into a buffer associated with the filter and signals the INPUT semaphore of INPTASK. For each message in the buffer, INPTASK calls INTERNET INPUT. If it is able to reassemble an entire segment, INTERNET INPUT in turn calls HLPM INPUT to process the segment.

* Detect timeouts

  When it issues a PWAIT, INPTASK includes an interval timer for the top request on the its timer queue. When this interval expires, INPTASK calls INTERNET TIMEXPIR. TIMEXPIR will generally call HLPM TIMEOUT to inform the higher-level protocol of the event. However, it may also mark the expiration of the 30-second "watch-dog" timer for the IPP itself. The timing function is discussed further below in the subsection entitled "Timing".

* Handle outgoing logging

  An outgoing logger request is described by an Outgoing Logger Queue Element (OLQE). The OLQE's are queued and the ATTN ("Attention") semaphore of INPTASK is signalled. Finding an OLQE on its outgoing logger request queue, INPTASK dequeues it and passes its address to the HLPM OUTLOG routine.

The INPTASK module is resident, linkage edited into ARPAMOD. This allows it to respond to an input packet or to an outgoing logger request and dynamically load the main IPP routine INTMOD. Note that the INTERNET INPUT and TIMEXPIR calls are interfaces internal to IPP, although they have the same format as IPP services for the HLPM. The INPUT and TIMEXPIR calls (as well as ERLOG) are interfaces from the resident code of INPTASK to INTMOD, via the internet transfer vector INTNETRV.

Figure 5 shows that the INTERNET INPUT, TIMEXPIR, and OUTLOG routines in turn call HLPM routines; these routines all execute under the INPTASK ptask. In general, the other major IPP routines are services which are used by both the HLPM and the IPP itself (e.g., the INTERNET INPUT routine calls INTERNET OPEN). These IPP service routines are executed under the ptask of the caller, which may be INPTASK or may be a ULPP.

The ptask structure determines the ownership of resources under ICT. For this reason, control blocks obtained by the IPP service routines (e.g., ICB's) cannot be obtained with PCORE, because they might belong to the wrong ptask; instead, the OS/MVT GETMAIN service must be used directly. This in turn presents the problem of freeing all storage GETMAINed in the NCP region by a (buggy) HLPM when it terminates. This problem is solved by using a separate storage subpool (zone) for the GETMAIN's from each HLPM; the subpool number is contained in the IPB. The IPP (INPTASK) frees the entire subpool collectively when the HLPM becomes idle, and INTERNET frees the subpools of all HLPM's if INPTASK ever terminates.

3.4. IPP FUNCTIONS

We will now describe in more detail the manner in which the IPP performs its key functions.

3.4.1. Sending Segments

The parameter list to INTERNET OUTPUT is a Write Request Element or WRE, which includes a pointer to the ICB for the association on which this segment is to be sent. The ICB points to an entry in the Internet Host Table (IHT) which includes a specification of the remote gateway on the ARPANET to which packets must be routed.

A WRE also specifies the data to be sent, by means of a list of (address, length) pairs; each pair is called an _extent_. The WRE may have any number of extents, but the first extent must be unused. The IPP OUTPUT routine builds an IP header and points the first extent at it, and then passes the WRE to the outgoing ARPANET gateway.

In addition to the WRE, two data areas are needed: a 4-byte leader parameter area for AGAWO, and an area for building the IP header. Furthermore, most higher-level protocols will require an area for building their headers. All three areas are provided at once, in a control block called an _IWRE_. An IWRE begins with space for a WRE, followed by the leader parameter area, the IP header area, and a HLPM header area.

As a service to the HLPM's, the IPP maintains a pool of available IWRE's and will supply one for a particular ICB when the INTERNET GETWRE service is called. INTERNET FREEWRE will return a WRE to the pool. The HLPM is required to return all IWRE's for an ICB before calling INTERNET CLOSE to delete that ICB.

3.4.2. Fragmentation

In principle, the IPP is responsible for fragmenting segments as necessary to fit into the constraints imposed by the local packet network, the ARPANET. However, the preeminent higher-level protocol, TCP, must packetize the data stream and can itself produce segments of any desired maximum size. As a simplification, the initial IPP implementation therefore leaves fragmentation entirely to the HLPM, which learns the maximum send segment size from the ICB. The IPP simply sets this maximum to an appropriate value (depending upon which Subtype will be used; see below), making allowance for the IP header and for the ARPANET leader.

Note that the maximum segment size is obtained from the ICB, not the IPB, so that each ICB (association) could have a different value. This is to accomodate future definition of an IP mechanism for negotiating the maximum segment size up or down. The initial value is obtained from the corresponding IPB when the ICB is created (by INTERNET OPEN). In the absence of a negotiation mechanism, all ICB's currently have the same value.

Further discussion of the fragmentation mechanism will be found below under "AREAS FOR FUTURE WORK".

3.4.3. Segment Id's

Each IP header must contain a 16-bit segment id field to identify the fragments of the segment at the ultimate destination. A segment id must be unique for a given stination host and higher-level protocol, within the maximum lifetime of a segment in the internet transmission system. However, since 2**16 is a very ,e id space, we have chosen to use a single global segment id counter for all associations. This choice is discussed below under "AREAS FOR FUTURE WORK".

3.4.4. Demultiplexing Received Packets

For efficient support of connection-oriented protocols such as TCP, the IPP is designed to do the complete demultiplexing of a received packet with a single hash-table lookup. This is accomplished in the following manner:

(1) We have introduced into the demultiplexing decision ~n additional parameter, the logical stream number; this is a 32-bit number whose computation is dependent upon the appropriate HLPM. Thus, the IPP demultiplexes an incoming packet using the triplet:

(<internet host address (source host)>,

<higher-level protocol>,

<logical stream number>).

(2) When an incoming packet arrives, the IPP input routine uses the higher-level protocol number from the IP header to locate the corresponding HLPM. The IPP then passes the address of the segment to a "DEMUX" subroutine in that HLPM. The DEMUX routine generates an appropriate 32-bit logical stream number (by looking at the header for its protocol) and returns the value to IPP. IPP finally performs the full demultiplexing for the message, using a single hashed lookup.

The demultplexing triplet is called an association and corresponds to an active ICB. An ICB is created with a call to INTERNET OPEN, using the parameters:

(<host id>, <PID>, <logical stream number>).

Here <host id> is a one-byte handle for the internet host, and <PID> is the protocol id, i.e., the one-byte handle for the IPB. INTERNET CLOSE will delete the ICB.

The hash table uses the familiar chained-overflow scheme. That is, the hash table itself consists of a set of fullwords, each of which is the head of a chain of ICB's that hash into the same bucket. This scheme is simple and efficient, and allows ICB's to be easily deleted from the hash table in INTERNET CLOSE.

It is expected that HLPM's will choose logical stream numbers so that associations will be in one-to-one correspondence with connections.

> For example, TCP's logical stream number is composed of the two 16-bit numbers defining the source and destination ports. As a result, each TCP connection (TCPB) has its corresponding ICB.

For each connection, there will be a "higher-level protocol block", or hlpB; for example, TCP uses a TCPB. Therefore, we expect to always have a hlpB dualed with each ICB.

To simplify the HLPM implementation, INTERNET OPEN is prepared to obtain main storage for the dual hlpB at the same time it obtains an ICB, making the two blocks contiguous. However, note that neither the IPP nor the HLPM's assume contiguity; instead, they use the fact that each control block points to the other. The space to reserve for the hlpB is a parameter in the IPB. Calling INTERNET CLOSE will free both the ICB and the hlpB.

3.4.5. Recursion and ICB Deletion.

As explained earlier, the INTERNET INPUT and TIMEXPIR routines are called from INPTASK, and in turn call HLPM routines. Suppose one of the latter decides to close the connection being processed, i.e., it calls INTERNET CLOSE for the corresponding ICB. There is the danger of a logic error arising when the INTERNET routine, upon regaining control, attempts to access an ICB that has been deleted by INTERNET CLOSE.

The solution to this synchronizing problem uses a "Lock" bit and a "Delete Deferred" bit in every ICB.

(1) Before calling the HLPM INPUT or TIMEOUT routine, the IPP will turn on the Lock bit in the ICB.

(2) Finding the Lock bit on, INTERNET CLOSE will not delete the the ICB, only turn on the "Delete Deferred" flag bit.

(3) Upon regaining control, the IPP turns off the Lock bit and, finding the "Delete Deferred" bit on, calls INTERNET CLOSE again to actually delete the ICB.

3.4.6. Reassembly

When a packet is received, the demultiplexing process just described chooses an IPB and an ICB. Next, the IPP must move this packet into its place in a <u>segment reassembly buffer</u>, called an <u>RAB</u>. The first 16 bytes of an RAB are a buffer header, used for controlling and chaining the buffer.

Each ICB contains the head of a chain of active segment reassembly buffers for that association. The IPP searches this chain for a matching segment id, and obtains a new RAB if no match is found. Then the new packet is moved into its place in the proper buffer, as determined from the fragment offset field in the IP header.

RAB's on the active chain may be in one of three states, as determined by a flag byte in the buffer header.

* <u>Filling</u> -- contains at least one fragment, but not completely reassembled.

* <u>Full</u> -- fully reassembled, and passed to HLPM.

* <u>Emptied</u> -- marked processed by HLPM, may be freed.

Fragments of a given segment may arrive in any order, may be duplicated, and may overlap in an arbitrary manner. Although there is no error check on the data, there is no reason to prefer the earliest over the latest version of a given byte. Therefore, the reassembly routine can simply move each fragment into its place in the buffer, possibly overlaying some earlier fragments.

However, in order to determine whether the segment has been fully received, the IPP must create an auxiliary data structure for "bookkeeping" on the bytes in the buffer. The IPP uses a linked list of 8-byte <u>Reassembly</u>

Control Elements or RCE's for this bookkeeping. Each RCE contains the first and last address of a contiguous block of data. Inserting a new fragment may add an RCE, modify an existing RCE, or coalesce two existing RCE's and delete one. It is believed that this algorithm works well for the most probable case, a few large fragments; however, a detailed efficiency comparison with the bit map algorithm has not been made.

The current IP protocol has a fixed maximum segment size for all internet connections, 576 bytes including the internet header. Therefore all RAB's have the same fixed size, 576-20+16= 572 bytes. Possible extensions to allow varying segment sizes are discussed below under "AREAS FOR FUTURE WORK".

3.4.7. Reassembly Timeout

Normally, the INTERNET INPUT routine (INTNETI) will reassemble the fragments of a segment and pass the reassembly buffer to the higher-level protocol input routine. However, because of bit-errors in the transmission or lost packets in the networks, reassembly of a particular segment may never be completed. The IPP must time out such never-to-be-completed buffers.

It is undesirable to pay the overhead cost of keeping a logical timer on every RAB, since the timeout is to protect against a situation which is expected to appear only rarely, and which need not be corrected instantly. Therefore, a "watch dog" timing scheme was implemented. The present scheme scans all the RAB's on all ICB's roughly every 30 seconds. A one-bit timeout counter in the RAB flag field is used. The bit is set in each scan and unset when reassembly is completed. If a scan finds the bit set, the buffer has remained for 30 seconds without completion of reassembly, and IPP returns it to the available chain on the IPB.

3.4.8. Reassembly Deadlock

Reassembly deadlock is a possibility in any IPP, due to a finite supply of reassembly buffers. At the IPP level, the timeout of partially-reassembled buffers prevents an absolute deadlock. However, once a segment has been fully reassembled, the HLPM is permitted to keep it (i.e., to not mark it emptied) until the order required by the higher-level protocol is satisfied. This can easily lead to deadlock, even in the absence of any fragmentation, if segments arrive sufficiently out of order.

The solution is to include among the RAB header flags a "Potential Deadlock" bit. The IPP turns on this bit when the RAB it is handing to the HLPM INPUT routine is the last one allowed by the active buffer limit. The HLPM is required to examine this bit, and finding it on, to empty at least one RAB before returning to the IPP. This may require discarding a segment that has been received earlier.

3.4.9. Buffer Pool Management

In general, the segment reassembly buffers will represent one of the critical resources for internet operation, so the algorithm used to manage them is very important. This algorithm should have the following properties:

* There should be a pool of RAB's shared by all associations under the same IPB (higher-level protocol).

* The size of a buffer pool needs to grow dynamically with the requirement for reassembly buffers.

* No single association should be able to monopolize the RAB's in a pool or cause it tc grow unreasonably large.

* A pool should shrink as associations are deleted, (roughly) in proportion to the amount of the pool used by that association.

These desiderata are met by the following scheme, which is used in the UCLA IPP implementation:

(1) Each IPB contains the head of a chain of available RAB's. The pool consists of the available RAB's, plus all the active RAB's which are chained on the ICB's.

(2) The pool size is limited indirectly by a limit on the number of RAB's that an individual ICB can have on its active chain. If this limit is reached, the INTERNET INPUT routine will fail to get a new RAB for a packet and may have to discard it. (This leads to a reassembly deadlock problem, whose solution was described earlier).

This per-ICB RAB limit is carried in each ICB, although it is initialized from the IPB. This would allow the IPP or a HLPM to adaptively modify the limit. For example, if there is a satellite link in a particular conversation, a greater depth of reassembly buffering is required for high bandwidth. However, no such adaptive mechanism has been built yet, so all ICB's for a given IPB have the same active RAB limit.

(3) When an RAB is requested for an association which has not reached its limit, but there are none on the IPB available chain, a GETMAIN is executed to add an RAB to the pool. In this way, the pool expands upon demand.

However, the expansion is "charged" to ICB whose request forced it. That is, an "expansion count", or count of the number of times its request forced a GETMAIN, is kept in each ICB. This expansion count will be a rough average of the amount of the pool that exists because of the particular association.

(4) After an association is deleted (by INTERNET CLOSE), the pool will be reduced or "trimmed" by a number of RAB's equal to the expansion count of the deleted ICB.

It may not be possible to delete all of them immediately, since an RAB cannot be deleted from the pool unless it is on the available chain. Therefore, a "trim-needed" count is maintained in the IPB; as RAB's are subsequently made available they are deleted until the "trim-needed" count is reduced to zero.

This algorithm has several nice properties. First, it adds little overhead, requiring only two counters in each ICB and a "trim-needed" count in each IPB, and trivial CPU processing. Also, at any moment the sum over all ICB's of the expansion counts will be equal to the number of RAB's in the pool less the "trim-needed" count. This means that as the last ICB is deleted, the pool will exactly vanish.

3.4.10. IWRE Pool

We noted earlier that the IPP maintains a pool of IWRE's for the use of the HLPM's in sending data. Although the IWRE problems are not as severe, management of the IWRE pool has the same characteristics as management of the RAB pool, and therefore the RAB pool algorithms are used for the IWRE's as well.

3.4.11. Starting a ULPP

The HLPM may request the creation of a user or server session by calling the INTERNET START subroutine, passing a parameter list which contains:

* The contact port (complemented for user session);

* PID and host id;

* Local Socket Number and Remote Socket Number.

* Exchange Window Id (for user session).

Note that, except for the first item, this list defines the contents of the ICV parameters to be passed to the primary ULPP.

The contact port and PID are used as keys to obtain the primary ULPP module name from an "Internet Logger Table", or ILOGTAB. ILOGTAB must be generated with entries for all user-level protocols which are support IP. This table also specifies a small signed integer called the "socket offset", which is designed to simplify compatibility between AHHP sockets and TCP ports. INTERNET START adds the socket offset from the table to the Local Socket (Remote Socket) Number in the ICV, for a user session (server session, respectively). This is intended to compensate for the small integer offsets that AHHP uses in its socket subspaces.

The INTERNET START routine performs the following operations:

* Assign a new session number;

* Locate an ILOGTAB entry for the (contact port, PID) pair;

* Create an ACE for the session;

* Issue PATTACH for the primary ULPP;

* Apply the socket offset from ILOGTAB to the appropriate ICV socket.

* Pass the ICV parameters to the primary ULPP.

If INTERNET START returns a code that indicates success, then the ULPP ptask has been created and will eventually go through AEXIT, freeing its NCP resources. This is true even if the ULPP module cannot be loaded. IPP has a timeout to ensure that the primary ULPP does start properly. If the ULPP does not issue an ALSTN call within the timeout period, the IPP will call PDETACH to force it through PEXIT (hence, AEXIT).

3.4.12. Timing

At an early stage of the design of the UCLA implementation, we planned to have a separate timing ptask for each higher-level protocol that needed timeouts. Since INPTASK must provide a watch-dog timer for the IPP, a simpler design resulted from having INPTASK provide all timing services. It was also natural to attach a time interval to the ICB, since that is the control block known directly to the IPP, and would require no additional control blocks.

The timer service that resulted operates in the following manner:

(1) There is an IPP service, INTERNET TIMER, that the HLPM can call to define, change, or cancel a time interval for a given ICB.

(2) When the time interval expires, the IPP calls the HLPM TIMEOUT subroutine.

(3) INPTASK also keeps track of its own watch-dog timeout interval, and when it expires calls its internal watch-dog timeout routine INPIMEO.

The preeminent higher-level protocol, TCP, is timer-driven. It is very important to keep timing overhead from becoming overwhelming as the number of connections increases. To aid this problem, the IPP timing algorithms include a minimum timing resolution called <u>fuzz</u>; its value, in units of 0.01 seconds, may be found in the P3CB (P3FUZZ). The rules for timing are as follows:

(1) The HLPM cannot set a timeout interval less than the fuzz; if it attempts to do so, the actual interval will be equal to the fuzz.

(2) INPTASK will consider any request for a timeout earlier than <current_time>+<fuzz> to be expired. Thus, if there are several timeout requests for different ICB's on the queue, all expiring within the fuzz, all will be timed out (HLPM TIMEOUT called) before INPTASK calls PWAIT again. Once it gets control, the HLPM TIMEOUT routine must consider the <fuzz> if it tests to ensure that an interval actually expired. The P3CB contains both a pointer to <current time> and the value of <fuzz>.

(3) Setting a new timeout interval (e.g., with INTERNET TIMER call) is guaranteed to force a PWAIT before the HLPM TIMEOUT routine is dispatched again. This is to prevent inadvertant infinite loops when the

HLPM resets the timer for a very small interval.

IPP also maintains a 30 second watchdog timer for the internet layer. When this timer expires, IPP scans all RAB's and times out any "stale" ones, as discussed earlier under "Reassembly Timeout". If the corresponding ICB has never had any segments successfully reassembled, the ICB is deleted.

Finally, if there is an idle protocol (no ICB's chained from an IPB), INPTASK issues PDELETE for the corresponding HLPM and FREEMAIN for its control block subpool.

3.4.13. Error Logging and Tracing

The internet layer includes an error logging routine, invoked by the ERLOG macro [Bra79B]. This routine calls the ATRACE service to record the error in an appropriate log file.

ATRACE is an A-Service for creating and using a trace buffer with variable-length entries. An internet trace can be enabled in IPP, to maintain a history of all internet segments sent and/or received. In addition, a HLPM can associate a trace buffer with every connection; however, there are some special provisions in the internet environment for this use of ATRACE; see the section below entitled "Tracing TCP Transactions".

3.4.14. Statistics

The IPP has provisions for gathering three classes of statistics.

(1) In the P3CB, it keeps statistics on the number of packets received and the number discarded with bad checksum, expired lifetime, or other serious defect which prevents demultiplexing the packet.

(2) In each ICB, the IPP keeps statistics on the performance of the IP layer. Specifically, it keeps the total count of segments sent, packets received, and segments reassembled, as well as the total bytes sent and reassembled. INTERNET CLOSE accumulates these five values in the IPB before deleting the ICB.

(3) Each IPB has space for accumulating statistics which depend upon the higher-level protocol. The HLPM should call the INTERNET STATSUM macro to perform this accumulation before the hlpB is deleted.

3.5. IPP DATA STRUCTURES

3.5.1. P3CB

The "P3CB", or "pseudo-control CCB", is the primary work and control area for INPTASK, hence for a particular IPP instance. As indicated by its name, the P3CB has a role in the environmental control block chains which is generally equivalent to the role of a control CCB under AHHP. For compatibility with AHHP, therefore, certain of the P3CB fields are fixed to match those of a (control) CCB. For example, P3ACE is the anchor of a chain of all internet ACE's, and P3CPTA (matching CCBPTA) is the INPTASK PTA address.

There is an important difference between the P3CB and control CCB's: the P3CB is not obtained dynamically but is resident and linkage edited into ARPAMOD.

The P3CB contains global IPP information, such as:

* <internet host address> for this IPP;

* global segment id counter;

* anchor of a chain of all IPB's (IPBLIST);

* timer chain anchor;

* value of the timing "fuzz";

* startup delay time for IPP (to allow old packets to disappear);

* ANMOC parameters to set up the internet packet filter.

The P3CB also contains the outgoing logger interface, needed by the transient ptask INPOLOG to enqueue a request for INPTASK. In particular, the P3CB contains the address of an enqueue routine, the anchor of the OLQE queue, and the INPTASK PTA address.

If there are multiple IPP's within the NCP, there must be a distinct P3CB (as well as A-Service transfer vector and IPBLIST) for each IPP instance.

3.5.2. IPB (LIST)

For each higher-level protocol, there is an assembled-in IPB which contains (1) the information common to all active ICB's for that protocol, (2) the default values needed to initialize a new ICB, and (3) control information for the protocol. IPB's are used by IPP but not by the HLPM's.

For example, an IPB includes:

* Higher-level protocol number for IP header.

* Character string (e.g., 'TCP') needed to construct the HLPM module name.

* Head of a chain of all ICB's for associations using this higher-level protocol.

* Address of the transfer vector for the HLPM, once loaded.

* Heads of chains of available RAB's and IWRE's.

* Summary statistics for both the IPP level and the higher-level protocol level.

* Parameters used to initialize the following ICB fields:

  Type of Service

  Internet options

  Maximum send segment size

  Maximum number of RAB's per ICB

  Maximum number of IWRE's per ICB

The IPB's are resident, assembled and linkage edited into ARPAMOD. They are chained together in a module called IPBLIST, and the head of this chain appears in the P3CB.

3.5.3. ICB

An ICB includes:

* Address of a companion higher-level protocol block ("hlpB") for the association (and, generally, the corresponding connection). For TCP, in particular, this will be a TCPB.

* Address of the corresponding IPB.

* (a pointer to) the <internet host address> of the remote host in the IHT, and the corresponding internet host id.

* Logical stream number.

* Chain of active RAB's for this association.

* Hash table chain pointer.

* Type-of-Service and option flags for sending segments on this association.

* Maximum segment lengths for sending and receiving.

* Pool control parameters: maximum numbers of RAB's and IWRE's for this ICB.

* Timing control and queue fields.

* Statistics kept by the IPP on this association.

3.5.4. IHT

The "host id" is a one-byte handle used to designate a particular internet host address and associated routing information. A host id is an index to a dynamically-created table of internet hosts currently communicating with the local host; this table is called the "Internet Host Table", or IHT.

An IHT entry contains the internet host address plus routing information to locate the ARPANET gateway to reach that internet host. The routing information currently includes only the ARPANET host address, link number, and gateway-supports-Subtype 3 flag for a single gateway.

3.5.5. INAMTBL

Since the table of internet host names and addresses has the potential of growing very large, it is contained in a separate load module which can be PLOAD'ed when needed. Fortunately, the names of internet hosts are required only for two purposes:

* The outgoing logger maps a host name into its internet address and gateway address;

* The name may be required for display, e.g., in a error message.

In either case, the delay and cost of loading the table are tolerable.

The INAMTBL is designed to supplement but not replace the existing ARPANET host tables within the NCP. Therefore, INAMTBL references ARPANET hosts by name rather than by number. INAMTBL contains entries for all named objects: internet hosts, networks, gateways, and higher-level protocols. Specifically, its entries make the following transformations:

Internet Host Name =>

(Network Name, 24-bit address, Default higher-level protocol name)

Network Name =>

(Network Number, Gateway Name)

Higher-level protocol name => PID

Gateway Name =>

(Link number, Accepts-Subtype 3 Flag)

3.6. OUTGOING LOGGER FUNCTION

The outgoing logger function is driven by a process outside the NCP and must accomodate a user at a terminal. It therefore accepts and parses a character string which defines the initial connection to be established and the host-host as well as the user-level protocol to be used.

In the internet environment, this information may be specified in a variety of ways. For example:

(a) User specifies: <internet host name>, and

<internet host name> implies network,

which implies gateway.

(b) User specifies: <internet host address> and network,

and network implies gateway.

(c) User specifies: <internet host address>, network, and gateway.

The syntax of the outgoing logger parameter string is therefore quite rich; see Appendix A. For ease of maintainance and future development, the code to parse this string was packaged in a transient module, INPOLOG. The syntax of the logger parameter string was designed to be compatible with the AHHP outgoing logger, so that INPOLOG can eventually replace the existing AHHP parsing code in LOGGER. The interface to AHHP has not been completed, however.

INPOLOG builds a control block called an Outlog Queue Element (OLQE) describing the request; the OLQE contains no text, only numbers. INPOLOG enqueues the OLQE for INPTASK, calls PPOST to signal INPTASK's ATTN semaphore, and vanishes.

Finding an OLQE in its outgoing logger queue, INPTASK passes the OLQE to the INTERNET OUTLOG routine, which in turn passes it to the outgoing logger routine of the appropriate HLPM.

Notice that the INPOLOG transient ptask is directly inferior to LOGGER and operates in the AHHP environment, not the internet environment. INPOLOG must be able to find the P3CB in order to enqueue an OLQE; for this reason, the address of the P3CB appears in the AHHP A-service transfer vector.

3.7. AREAS FOR FUTURE WORK

3.7.1. Segment Id Assignment

The current implementation assigns segment id's using a global 16-bit counter. This will be adequate for an internet host on the ARPANET with a modest number of active connections. The minimum packet size (IP header plus ARPANET leader) is 256= 2**8 bits, so one can send at least 2**24 = 16 million bits before the segment id recycles. With average bit rates of less than 10**5 bits per second, maximum packet lifetimes must be less than 160 seconds. ARPANET packets have a lifetime under this limit.

One could conceive of circumstances which use up segment id's too fast. For example, two internet hosts might be connected via a link capable of 10**7 bits per second. However, such high bandwidths do not appear to be feasible within the present hardware/software context of the IBM implementation of an ARPANET IP/TCP. If the implementation were adapted to such a high-bandwidth application, attention would need to be paid to the segment id assignment.

It would be trivial to have a separate segment id counter for each higher-level protocol, in the IPB's. At the present time, it appears that there are not likely to be more than a few higher-level protocols, and even fewer that consume many segment id's, so a separate counter per higher-level protocol would not conserve segment id's significantly.

A much more useful alternative would be to associate a segment id counter with each active internet host, storing it in the IHT. This would be an easy extension of the current code.

Finally, one might hope the worst case would not arise. If one were to send a very large amount of data so rapidly as to make the packet lifetimes comparable to the cycle time for the segment id space, one would hope that the user-level protocol and the IBM system would send maximal-sized segments (576 bytes). This would increase the average packet size towards 4800 bits, an order-of-magnitude change from 256.

3.7.2. Gateway Link Numbers

The present implementation makes the presumption that IP will use a fixed ARPANET link number, accepted by all ARPANET gateways and IPP's. However, the internet name table (INAMTBL) does specify a link number for every gateway, and the outgoing logger inserts this value in IHT for use by the session. This will allow the UCLA IPP to contact an experimental IPP which uses a different link. However, the incoming logger has no corresponding mechanism to map the gateway host number into a link number. Such a mechanism could easily be added, but there is no requirement for it at present.

3.7.3. Type of Service

The IP Header contains a type of service (TOS) field, which is intended to be interpreted in an appropriate manner by each packet network which the segment traverses. On the ARPANET, the TOS field must select either Subtype 0 or Subtype 3 packets.

The current specification for TCP [PosTCP] is incomplete in describing the use of TOS, and this is an area in which further protocol developments are likely. Furthermore, a number of the current IPP implementations on the ARPANET do not support Subtype 3 packets, but all support Subtype 0. Therefore, the UCLA IPP implements TOS in the following simple manner:

* The information kept in IHT for an ARPANET gateway includes, in addition to the internet link number and 24-bit host address, a flag bit which indicates whether this host can accept Subtype 3 packets. In the case of a connection initiated by the outgoing logger, this information is obtained from the permanent Internet Name Table (NAMTBL). For a session initiated remotely, if the first packet arrives with Subtype 3, then the Gateway from which it came is assumed to accept Subtype 3 packets.

* The TOS byte in the ICB is defaulted to X'36', speed-over-reliability.

* INTERNET OUTPUT sends a segment with Subtype 3 if the IHT flag indicates that the gateway can accept Type 3 packets and if the TOS bit indicating speed-over-reliability is on in the ICB; otherwise, the segment is sent with Subtype 0.

It would be useful in the future to define a new IPP service to allow a ULPP to change the default TOS. Note that it is not sufficient to simply change the TOS field in the ICB; the maximum send segment length must also be computed, since Subtype 0 and Subtype 3 packets have different limits. Furthermore, notice that the receive segment length is not affected by using Subtype 3; a remote IP may send segments of up to 576 bytes, fragmented to fit into the 113 byte limit of Subtype 3. A reassembly buffer must accomodate the maximum segment, regardless of the subtype.

Ideally, a ULPP should have a parameter to specify the TOS when it opens a new connection. The current ALSTN parameter list, constrained by a requirement for compatibility with AHHP, has no provision for such information. A reasonable solution would be to use the PID (protocol ID) to specify the TOS variables as well as the higher-level protocol. Each higher-level protocol will probably use only a few different values of the TOS byte; the TOS space is much richer than is currently useful. Hence one byte should in principle be sufficient to specify both.

We considered using a separate IPB for each (TOS, higher-level protocol) pair, so that different TOS classes could have different reassembly buffer pool parameters. On the other hand, the different TOS classes could not share buffer pools if they used separate IPB's, and a given connection could not change its TOS after it was opened. This approach was therefore rejected.

3.7.4. Fragmentation by the HLPM

The present IPP implementation has no mechanism for fragmenting packets, leaving this task to the HLPM. As a result, the higher-level protocol header must be duplicated in each "fragment" (segment). This leads to a bandwidth penalty which becomes significant when TCP segments are sent using Subtype 3 packets.

A Subtype 3 packet may contain 113 octets exclusive of the ARPANET leader, and the internet header normally consumes 20 octets out of the 113. If the IPP were fragmenting 576-octet TCP segments into Subtype 3 packets, the efficiency would be approximately 93/113 = 82%, since the TCP header length of 20 is negligible compared to 576. If we consider the fact that fragmentation takes place on 8-octet boundaries, a more accurate efficiency figure is 88/113= 77%. On the other hand, the present implementation will have an efficiency of only 73/113 = 64%.

We conclude that Subtype 0 (standard) messages should be used for applications like file transfer in which high efficiency is important. Alternatively, the IPP could be extended to fragment segments. Using an internal pool of IWRE's, INTERNET OUTPUT would generate and send to AGAWO all fragments of a segment, and return to its caller. Either the IPP would need to set a timer to poll for completion, or the HLPM would have to call a new INTERNET CHECK service to test for completion of its output request.

3.7.5. Reassembly Buffer Sizes

Every IPP is required by the protocol definition to be able to reassemble segments of 576 bytes (including the IP header). There is currently no protocol mechanism defined in either IP or TCP to negotiate any larger, or smaller, segment size. We believe this to be a significant omission. For applications like Telnet, 576 byte buffers will often be mostly empty, while higher-bandwidth operations like file transfer will benefit from larger segments.

Furthermore, much of the internet traffic will not require reassembly, in which case the segment could be moved into a buffer which is just large enough for the actual segment. Therefore, a mechanism which handled variable segment sizes would save buffer space even in the absence of a negotiation protocol.

In the present UCLA implementation of IP/TCP, all reassembly buffers in the pool for a given higher-level protocol (IPB) must have the same size. There are three possible ways to provide for varying segment sizes:

(1) Multiple fixed-size buffers per segment;

(2) Varying buffer sizes within a pool;

(3) Multiple pools per IPB.

Either would require modifications and extensions to the IPP. Further design work is necessary choose the best approach and to develop efficient algorithms.

3.7.6. Time to Live

The present reassembly timeout scheme uses a marker bit to time out a buffer in 30 to 60 seconds. The timeout period should not be fixed, but should be tied to the Time-to-live field of the IP Header. At present, the Time-to-live field is not treated very seriosly by most IP implementations; however, it is potentially useful for controlling packet lifetimes. Packet lifetimes are in turn related to the segment id space, as discussed earlier.

The marker bit could be thought of as a 1-bit counter. There is room in the flag byte to make this a 4-bit counter. This would allow us to use the Time-to-live value for buffer timeout, in units of 16 seconds.

3.7.7. Internet Routing

The problem of routing packets through multiple networks is still an area for research. As general solutions are found, the UCLA implementation of IP will need to incorporate them.

The present implementation keeps the simplest routing information for an active internet host: a single ARPANET gateway address. When a session is initiated by a remote host, the source ARPANET host address of the first packet is taken as the gateway. The outgoing logger depends upon INAMTBL or explicit definition of the gateway.

In many cases, there will be two or more gateways which can reach a given host. If a gateway host which is being used goes down, the IPP will receive a DEAD HOST message from the ARPANET. This could be used as a signal to choose an alternate gateway.

Any extension of the routing facility would begin with a significant extension to the IHT data structure. In addition, a new fixed table would be defined to map network numbers into lists of possible gateways.

3.7.8. Internet Name User

As pointed out earlier, the Internet Name Table (INAMTBL) is included in a transient module because it is expected to grow large. In the future, it will probably be useful to employ the Internet Name Server protocol [PosINS], to consult a centralized directory of internet hosts. It would be natural to extend the INAMTBL lookup routines to contact an Internet Name Server when a local search fails. Alternatively, an Internet Name Server could be implemented locally.

3.7.9. Miscellaneous Unimplemented Features

There are several planned features of IPP which have not yet been implemented.

* IP Error Options

  IPP currently discards an erroneous internet packet without reporting the error to the remote host. The error option [PosIP] has not been implemented.

* Partially-Specified Associations

  It should be possible for a HLPM to request a partially-specified association. For example, a TCP user may want to "listen" for a connection with any remote port number on a given internet host. At present, the primary hash table mechanism used for demultiplexing in IPP requires that the association be fully specified.

* Multiple IPP's

As we pointed out earlier, the IPP design allows multiple concurrent IPP's with different logical host numbers. However, the necessary code to start multiple IPP's has not been added to the INTERNET ptask.

4. TCP LAYER DESIGN

TCP is an internet host-host protocol that provides reliable connection-oriented communication paths between processes [PosTCP]. TCP assumes the existence of the Internet Protocol IP for data transport [PosIP].

The UCLA implementation of TCP is contained in the load module TCPMOD, which is a particular instance of a HLPM (higher-level protocol module). This section describes the design of TCPMOD. We assume a general knowledge of the design of the internet protocol program (IPP).

4.1. TCPMOD FUNCTIONS

To implement TCP, TCPMOD must provide the following functions:

* Data Transfer--

packetize data to be sent to a remote internet host, i.e., split the data stream into blocks called segments. TCPMOD must build a suitable TCP header in each segment and request the IPP to send the segment as a datagram.

The segments which TCPMOD receives must be ordered and duplicate data must be deleted before the data can be made available to the appropriate User Level Protocol Process (ULPP) for the connection.

* Reliable Communication--

provide reliable communication by means of sequence numbers and acknowledgments (ACK's), protected by a checksum over the entire segment. TCP provides full-duplex communication paths, and TCPMOD attempts to "piggy-back" the acknowledgments on data segments going in the reverse direction. TCPMOD is timer-driven to retransmit data which has not been acknowledged within a suitable time interval.

* Flow Control--

provide flow control by means of windows in the sequence number space. TCPMOD must set its receive window suitably, and it must obey the send window set by the remote TCP.

* Connections--

create logical data streams called connections. For reliable operation, TCP uses a "three-way handshake" (i.e., 3 messages) during both establishment and termination of a connection.

create logical data streams called connections. For reliable operation, TCP uses a "three-way handshake" (i.e., 3 messages) during both establishment and termination of a connection.

The connection states of TCP are reflected to the internet ULPP's in a manner which is essentially compatible with AHHP connection logic.

* Logger--

  perform the final steps in the incoming logger and outgoing logger functions, creating new sessions in response to remote and local requests.

* Urgent--

  provide an out-of-band signalling mechanism called "urgent". TCPMOD must be able to send and receive "urgent" data.

For each active TCP connection, there is a corresponding internet association; as a result, there is an ICB dualed with each TCPB. In practice, the (ICB, TCPB) pair will be contiguous, but no routine depends upon contiguity. The structure of a TCPB is constrained to be compatible with a CCB, as described in Appendix C. The IPP has no knowledge of the internal structure of the TCPB (other than its total length); on the other hand, the TCPMOD routines may read but generally not change the contents of the ICB.

A TCPMOD routine is always invoked to operate on a particular connection, denoted by the address of its TCPB or equivalent ICB. TCPMOD may be considered to be a reentrant finite-state machine, driven by the state of the given connection using a (conceptual) transition matrix.

TCPMOD provides a Network I/O interface to the ULPP's like that provided by AHHP:

* Output is transmitted by reference. That is, the ULPP specifies the addresses and lengths of data chunks in its buffers. These data pointers are passed through successive protocol program layers -- TCP, IP, and AGAWO -- and finally inserted into hardware channel programs which send data to the IMP.

* Input is provided in a circular buffer associated with the connection. The ULPP moves data from this buffer, and then calls ARLSE (the "Release" A-service) to indicate consumption of the data.

4.2. TCPMOD INTERFACES

It is helpful to review the interfaces between TCPMOD and the rest of the NCP.

4.2.1. INTERNET Services

TCPMOD may invoke any of the IPP ("INTERNET") services [Bra79B] discussed in the section "INTERNET LAYER DESIGN". Note that these IPP service routines are strictly synchronous; that is, they never issue a PWAIT call and therefore do not give up control to another NCP coroutine.

4.2.2. HLPM calls from IPP

As discussed previously, the IPP uses the HLPM macro with the options: INPUT, TIMEOUT, OUTLOG, DEMUX, and PURGE to call the corresponding TCPMOD subroutines; see Figure 5. These calls assume that the corresponding TCPMOD subroutines appear at canonical offsets on a transfer vector, TCPTRV; TCPTRV is linkage edited into TCPMOD and is the entry point of the module.

4.2.3. ULPP Interface

The ULPP's interface to TCPMOD through the A-services, which form the compatibility interface. As described in the section "INTERNET LAYER DESIGN", the compatibility interface includes two layers, the ARPIxxxx routines which are considered part of the IPP and are included in INTMOD, and the corresponding HLPM routines. The compatibility interface includes the following chains of calls for TCP (here "->" means "calls"):

* ALSTN macro -> ARPILSTN -> TCLSTN

  "Listen", i.e., create TCPB and initiate passive open.

* AOPEN macro -> ARPIOPEN -> TCOPEN

  Initiate active open, or complete passive open of TCP connection.

* ACLOSE macro -> ARPICLSE -> TCCLSE

  Close or abort specified TCP connection.

* ASEND macro -> ARPISEND -> TCSEND

  Send data on specified TCP connection.

* ARLSE macro -> ARPIRLSE -> TCRLSE

  Release data from circular buffer for specified connection.

* AINT macro -> ARPIINT -> TCAINT

  Mark last data sent as (end of) urgent; approximately simulates sending the out-of-band interrupt signal of AHHP.

The interfaces between these ARPIxxxx routines and the corresponding TCxxxx routines do not have the same degree of intellectual credibility or stability as the rest of the IPP/HLPM interface [Bra79B]. Thus, the division of function between the IPP level and the TCP level of the compatibility interface has changed a number of times during the development of TCPMOD; it may change further when and if some other connection-oriented higher-level protocol is implemented, or when a different (non-compatible) user interface to TCP is designed.

The minimal function of an ARPIxxxx routine is to locate the corresponding HLPM routine by following the control block chain from the TCPB (whose address is a parameter to most connection-oriented A-services) to the ICB to the IPB, to obtain the address of the HLPM transfer vector. The exception is ARPILSTN, which maps a given protocol id into an IPB and then issues INTERNET LOAD to PLOAD the corresponding HLPM if necessary.

Since the ARPIxxxx layer will be the same for all higher-level protocols, it is tempting to assign further function to the ARPIxxxx routines. This approach would attempt to model the semantics of the problem -- the ARPIxxxx routines would perform those functions which related to the control block environment, leaving to the HLPM layer all functions related to the higher-level protocol. This approach came asunder a number of times, when the particular manipulations of the environment were found to depend upon information specific to TCP. This required either moving those manipulations to the HLPM layer of the compatibility interface, or providing more complex interactions between the two layers.

Another, and sometimes conflicting, design approach is to use the IPP layer only to economize on code -- factor out of the TCPMOD routines those functions which (we imagine) every connection-oriented HLPM would need. This would include standard validation of parameters.

The current ARPIxxxx routines generally validate parameters, locate the HLPM, and call the corresponding HLPM (TCxxxx) routines. However, some of them (e.g., ARPICLSE) do perform significant manipulations of the environment. A single clear model for designing these interfaces is still lacking. Therefore, in the following we will discuss the compatibility A-services without making a distinction between the IPP and HLPM parts of each.

4.2.4. P-Services and A-Services

TCPMOD routines are permitted to issue PWAIT calls and bypassed SVC operations, giving up the commutator. TCPMOD also uses some A-services, including ABUF (get/free a circular buffer), ACLOSE, and APURGE. Notice that the last two actually call other TCPMOD routines through the compatibility interface; TCPMOD must avoid recursion from these calls.

4.3. TCPMOD FUNCTIONS

We will now describe in more detail the algorithms that TCPMOD uses to perform its functions.

4.3.1. Sending Data

To send data on a particular TCP connection, a ULPP issues the ASEND macro, calling ARPISEND which calls TCSEND. The parameter list to this call is a Write Request Element (WRE) that specifies:

* The address of the TCPB for the connection.

* A list of one or more buffer extents, i.e., (address,length) pairs whose catenation defines the data area(s) to be sent; and

* An "Urgent" bit and a "Not-EOL" bit.

This WRE must be compatible with AHHP; the only fields that differ are the two TCP-specific control bits Urgent and Not-EOL (Not-End-of-Letter). The corresponding bits will always be zero in the AHHP environment, so the default for compatibility is not-Urgent and EOL (i.e., each ASEND call sends a letter). To break a letter into several system calls, a ULPP must have TCP-dependent code to turn on the Not-EOL bit.

The send routines (ARPISEND, TCSEND) basically enqueue the WRE on the tail of the Send Queue, whose queue pointers (TCPSENDQ) are in the TCPB, and then return to the caller via TCPACKT, the packetizer subroutine.

4.3.2. Packetizing Output

Output is "packetized", i.e., divided into segments for transmission (and possible retransmission) by the TCPACKT routine in TCPMOD. As illustrated in Figure 6, TCPACKT is primarily concerned with two queues of WRE's: the Send Queue and the Segment Queue. The Send Queue contains the WRE's defining the data to be sent. The Segment Queue contains (I)WRE's for segments that have been sent at least once on the ARPANET but have not yet been fully acknowledged by the receiver; thus, the Segment Queue functions as the "retransmission queue".

TCPACKT divides the data in the Send Queue into maximal-size segments which will fit into the current send window. Each new segment is described by an IWRE, which is a WRE extended to include space for a TCP header and an Internet Protocol header. The IWRE is used as the parameter list and queueing element for sending the segment originally and, if necessary, for subsequent retransmissions.

TCPACKT appends each IWRE representing a new segment on the Segment Queue and then calls a subroutine (TCSEGOUT) to send it to the remote TCP. See Figure 7 for the major call paths for sending data. TCSEGOUT forms a TCP header containing the latest ACK and urgent information and a checksum, and then calls INTERNET OUTPUT to send the segment as a datagram.

TCPACKT continues this process until it exhausts the data in the Send Queue or reaches the right edge of the send window or is unable to obtain another IWRE. As discussed under "INTERNET LAYER DESIGN", the IPP does not fragment segments to satisfy the ARPANET constraints. Instead, it sets the limit in the ICB, and TCPACKT uses this value as the maximum segment size.

TCPACKT has a number of auxiliary functions, including:

* Send <SYN> on first segment.

* Send <FIN> bit on last segment.

* Send <RST> segment.

* Send empty <ACK> segment.

* Special processing if the send window is zero (see below).

* Mark a packetized segment with "end-of-letter" when appropriate.

TCPACKT is entered:

* By TCSEND when a new WRE has been appended to the Send Queue;

* By various TCPMOD routines to send a control message specifying <SYN>, <RST>, or <FIN>, or to send an empty <ACK> segment;

* By the HLPM INPUT routine TCPIN whenever a segment is received (and the connection is in a state that allows data to be sent). The <ACK> and window fields of the segment will have been used to update the corresponding TCPB fields before TCPACKT is called.

Segments which contain no data (e.g., empty <ACK> segments, and <RST> segments) must be handled specially, since they are never acknowledged by the remote host and are not retransmitted. Such segments are not placed in the Segment Queue; instead they are placed on the No-ACK list. When the Gateway has completed sending a segment to the IMP, it marks the WRE "Completed" but does not signal TCP. Therefore, TCP must use a timeout mechanism to inspect IWRE's on the No-ACK list and free all which are marked "Completed". TCPACKT looks first on the No-ACK list for an IWRE. This optimization is likely to succeed when a sequence of empty <ACK> segments are being sent.

Figure 6 -- Queues Manipulated by TCPACKT

```
                  SEND QUEUE                     SEGMENT QUEUE
                 ______________                  ______________
                |__o__|__o__|                   |__o__|__o__|
                   |     |                         |     |
      WRE's        |     |          Packetiz|ed          |
      from         |     |          Segments|            |
      ULPP         |     |                  |            |
                 __V__   |                __V__          |
                | WRE |  |               | IWRE |        |
                |.......|  |             |.......|       |
                |.......|  |             |.......|       |
                |80_____|  |             |_______|       |
                   |       |             |       |       |
                   |       |             ...   ...       |
 TCPNXTWR-->     __V__     |             |_______|       |
                | WRE |    |                 |           |
 (Next WRE      |.......|  |                 |           |
 to pkt'ize)    |80_____|  |               __V__    <--o
                   |       |             | IWRE |
                   |       |             |.......|
                   |       |             |.......|  Extents
                 __V__   <-o             |.......|
                | WRE |                  |_______|
                |.......|                |       |  TCP hdr,
                |.......|                ...   ...  IP hdr,
                |.......|                |_______|  etc.
                |80_____|
```

Figure 7 -- Major TCPMOD Modules

```
         HLPM INPUT     HLPM SEND      HLPM TIMEOUT
              V              V              V
              *              *              *
              *              *              *
 INTMOD       *              *              *
----------------------------------------------------
 TCPMOD       *              *              *
              *   calls      *              *
              *              *              *
         _____V____      ____V_____         *
        |          |    |          |        *
        |  TCPIN   |    |  TCSEND  |        *
        |__o____o__|    |____o_____|        *
           *    *            *              *
           v    v            v              *
      *<<<<*    *>>>>*    *<<*              *
      *    calls     *    *          calls  *
      *              *    *                 *
 _____V____        __V____V__           ____V_____
|          |      |          |         |          |
| TCGOTACK |      | TCPACKT  |         | TCTIMEO  |
|_____o____|      |__o____o__|         |____o_____|
      *              *    *                 *
      *              v    *    calls        v
      *    *<<<<<<<<<*    *    *<<<<<<<<<<<<*
      *    *              *    *
      *    *              *    *
      *    *            __V____V__
      *    *           |          |
      *    *           | TCSEGOUT |
      *    *           |____o_____|
      *    *                *
      *    *                *
      *    *                *          TCPMOD
----------------------------------------------------
      *    *                *          INTMOD
      *    *                *
      V    V                V
   INTERNET TIMER       INTERNET OUTPUT
```

4.3.3. Receiving Acknowledgments

TCP segments are received by the IPP, reassembled, and passed to the HLPM INPUT routine TCPIN. A segment generally includes an <ACK> field to acknowledge data sent by the local host. To effect the acknowledgment, TCPIN calls the TCGOTACK subroutine; see Figure 7.

TCGOTACK compares the latest <ACK> information with the sequence number fields of the WRE's on the Send Queue and the IWRE's on the Segment Queue, and dequeues all that are fully acknowledged. Each dequeued IWRE is freed by a call to INTERNET FREEWRE. When a complete WRE on the Send Queue has been acknowledged, TCGOTACK marks the WRE "Complete" and then calls PPOST to signal the OUTPUT semaphore of the ULPP that called ASEND.

4.3.4. Retransmission

Retransmission timeout is under control of the IPP timer service. When one or more IWRE's are enqueued on the Segment Queue of any active TCPB, TCPMOD will have scheduled a retransmission timeout interval. The IPP calls the HLPM TIMEOUT routine (TCTIMEO) when this interval expires.

TCTIMEO checks the Segment Queue and calls TCSEGOUT to retransmit each segment that has expired. TCSEGOUT builds a new TCP header for each segment, to send the latest <ACK> and window information. Finally, TCTIMEO frees all completed IWRE's from the No-ACK list, and arranges to reschedule the timer for the next timeout.

Notice that we do not generally re-packetize the data for retransmission (except in one special case, described later), although the queue organization would allow us to do so. The original WRE's from which segments were formed by TCPACKT are still in the Send Queue; TCTIMEO could empty the Segment Queue and call TCPACKT to repacketize the Send Queue.

The retransmission scheme operates in the following manner.

(1) Generally, the retrasnmission timeout interval is computed as f(R,N), where:

   * R is a measure of the "round-trip delay", including both network delay and host processing time.

* N is a count of the number of times the segment has been retransmitted while no <ACK> has arrived. Generally, f increases with N, the "backoff count".

The function which is currently implemented is:

```
if window = 0 then SLOWTIME else

      max( min(R, FASTTIME) * 2**(N+a),

                          SLOWTIME)
```

Here SLOWTIME provides a lower limit on the measured round-trip time, while FASTTIME is an upper limit on the retransmission time. The constant "a" is a small positive integer.

Thus, this formula provides "exponential backoff" for retransmission. The first retransmission will be larger than the measured R by a factor of 2**a.

(2) The round-trip time R is measured by maintaining an exponential average of the round-trip times of individual segments. We chose to define the round-trip time as the time interval from packetizing the segment until it is fully acknowledged; however, if more than 1 retransmission is required, the time interval is omitted from the average.

The exponential weighting factor has the form: 2**-b, where b is a positive integer generated in the P3CB. It would be useful to try different values for b experimentally.

(3) Whenever a new segment is packetized (presumably reflecting new window information), TCPACKT will retransmit any segments already on the Segment Queue that have already been retransmitted at least once. The fact that the remote TCP has enlarged the window without acknowledging all previous data is taken as evidence that an earlier segment was lost in transmission or discarded by the remote TCP. This provision removes a possible long delay in recovering when the remote TCP comes alive after being very slow, given the exponential backoff.

(4) We do not maintain a separate timer for each segment in the Segment Queue; instead, the first segment in the queue controls the retransmission timeout interval for all in the queue.

Suppose that the first segment does time out after an interval Q and is retransmitted; all segments below it in the Segment Queue which have been waiting at least Q since their last transmission are also retransmitted.

These rules deserve further comment. The decision to include the time for (one) retransmission in measuring R means that retransmissions tend to lengthen the timeout period. The assumption here is that retransmissions due to network losses will be at a low, relatively constant rate. However, as the timeout interval decreases, congestion in host processing will become dominant and retransmissions will rise rapidly. The scheme described here attempts to back off from such host congestion.

The formula shown above depends upon the assumption that the distribution of delay times is fairly narrow, and is roughly proportional to the delay time. In fact, current use of the UCLA TCP has been confined to networks with low delay, so that the host processing time is probably dominant; in this case, FASTTIME should dominate the formula.

Suppose that segment "A" has been packetized and transmitted once, and the next segment "B" is packetized before "A" is acknowledged or times out. Then "B" will not be timed out and retransmitted until the <u>second</u> retransmission of "A". After that, "A" and "B" will be retransmitted together, until "A" is finally acknowledged. At that time, "B" will revert to fast retransmission, since the <ACK> will clear N.

4.3.5. Zero Send Window

The TCP protocol requires special action when the send window is zero -- retransmit one byte of data "slowly" [PosTCP]. Finding data in the Send Queue, no IWRE's in the Segment Queue, and a zero window, TCPACKT packetizes 1 byte; however, this segment is not sent, but is left on the Segment Queue for transmittal after a long timeout period by the normal retransmission mechanism.

If the window opens before the 1-byte segment times out, TCPACKT never sends it; instead, it effectively backs up the window and repacketizes the byte. This is the only case in which data is repacketized. If the window opens after the 1-byte segment has been sent, it is retransmitted again immediately before the new segment.

4.3.6. Purging Network Sends

The APURGE service in the compatibility interface is used to "purge" TCP send operations for a given TCPB. ARPIPRGE purges the outgoing Gateway queues by calling ACPX HALTIO, then calls TCPRGE. TCPRGE purges the TCPB output queues: the Send Queue, Segment Queue, and No-ACK list. (Note that in this case the semantic layering of the compatibility interface is clean).

Unlike its AHHP cousin, APURGE under TCP does not affect the receive side of the connection.

4.3.7. Receiving Input

The HLPM INPUT routine TCPIN is called by IPP when a TCP segment is received. The parameters in this call are:

* the address of the reassembly buffer (RAB) containing the segment;

* the address of the association's ICB (which points to the TCPB for the connection); and

* a pointer to the IP header (required for the TCP checksum).

TCPIN checksums the segment and discards the segment if the checksum fails. Further processing depends upon the state of the connection. If the connection is in other than the Established state, special processing may be required for opening or closing the connection.

In the Established state, TCPIN checks the Packet Sequence number and length against the current receive window, to determine whether the segment is acceptable. To be acceptable, a segment must overlap the receive window in some manner (this is a more general definition than is required by the protocol [PosTCP]). An unacceptable segment is discarded. An acceptable segment is first truncated on the left to the current left window edge, and then TCPIN attempts to move it into the ULPP's circular receive buffer.

4.3.8. Reassembling Input

A segment may arrive out of order. TCPIN could move the data into the circular buffer and then use a bookkeeping mechanism (e.g., linked lists of RCE's or a bit map) to keep track of "holes". In the interest of simplicity, however, TCPIN simply queues any out-of-order RAB's internally, until they can be moved in order into the user's circular buffer. This approach has the disadvantage of possibly holding unnecessary buffer space in the case of frequent out-of-order transmission with small segments. If experience shows this to be a serious resource problem, a more elaborate reassembly mechanism can be added to TCPIN.

Thus, given an acceptable segment containing data, TCPIN tests whether the data is contiguous with the last information placed in the user's circular buffer. If so, the data is moved into the buffer, and the RAB is marked "emptied". If the data is out of order, however, the RAB is placed on an out-of-order list, in order of initial sequence number. This queueing uses an available field in the RAB header.

Whenever data is moved into the circular buffer, the top RAB in the out-of-order list, if any, is truncated on the left, and if it is now contiguous it is removed from the out-of-order list and its data is moved into the circular buffer.

This algorithm handles overlapping as well as misordered segments.

Reassembly deadlock must be avoided. When the count of buffers queued internally by TCP reaches the limit on reassembly buffers per connection, IPP marks the last RAB with a "Deadlock Possible" bit. When this bit is on, TCPIN must return at least one RAB, even if one must be discarded. It takes care to return the one with largest sequence number. The sending TCP will eventually retransmit the segment in the discarded buffer.

Note that this mechanism will quite happily queue a segment which is partly beyond the space in the circular buffer. In fact, if the segment passes the "acceptability" test, the out-of-order queueing algorithm would happily queue data which is totally beyond the right window edge (although the remote TCP is not supposed to send such data).

4.3.9. The Receive Window

TCPMOD exercises flow control over the input data stream by specifying a receive window size to the remote TCP. The present TCPMOD implementation uses the conservative windowing strategy, i.e., it "advertises" a window which is exactly equal to the available space in the circular buffer.

In order to get high bandwidth, it may be useful in some cases to advertise a larger window than is currently available. The out-of-order queueing mechanism, described earlier, could be extended to provide the additional buffering necessary to avoid occasional retransmissions with a "liberal" buffering strategy.

To simplify the implementation of a different windowing strategy, TCPMOD centralizes all manipulation of the receive window in a single subroutine, the Receive Window Strategy Module (TCRWSM). Whenever TCPIN moves data into the ULPP's circular buffer, it calls TCPRWSM to update the window and then turns on the "ACK Needed" flag. The result will be to send at least an empty <ACK> segment containing the revised (reduced, for the conservative strategy) receive window.

As the ULPP processes data from the circular buffer, it calls ARLSE (directly, or implicitly from ARECV MOVE) to "release" the space. ARLSE calls (ARPIRLSE which calls) TCRLSE. TCPRLSE again calls TCPRWSM to update (increase) the window size.

The remote TCP will need to be informed of an increase in window size. When data is flowing prodominantly in only one direction, this will require spontaneous generation of empty <ACK> segments. However, the ULPP may consume the input data in very small chunks, which would create a large number of empty <ACK> segments containing new small window updates. Therefore, TCPRWSM implements an algorithm to optimize the window updating and consequent spontaneous generation of empty <ACK> segments.

Specifically, TCPRWSM increases the window and sends an empty <ACK> segment if:

(1) the circular buffer is more than half empty, and

(2) the new window size exceeds the last size reported to the remote host by at least 1/8 of the buffer.

The receipt of a segment always triggers the creation of at least an empty <ACK> segment containing the full current window, so the remote TCP's send window will be updated as he continues to send. This algorithm significantly reduces the network traffic when there is a constant stream of sma`l messages.

4.3.10. Buffer Size Option

Since TCPMOD always passes received data to the user (ULPP) in a circular buffer, its buffering grain is 1 byte. Therefore, TCPMOD needs no mechanism for specifying the Buffer Size option.

On the other hand, the remote TCP may specify a buffer size, and TCPACKT must make appropriate adjustments in the send sequence number when the end of a letter is reached.

4.3.11. Urgent

For TCP, the ASEND call used to send data includes an Urgent bit. Turning this bit on indicates that the data being sent is "urgent". The sending TCP marks it as urgent by including an Urgent pointer in the TCP header; this pointer contains a sequence number one greater than the last byte of urgent data. The ASEND call may specify Urgent but no data; in that case, the urgent pointer will point to the next sequence number to be packetized.

The principal use of the AHHP interrupt mechanism has been in the Telnet protocol [McKen73], where the coincidence of an out-of-band interrupt and a Data Mark in the stream mark the end of urgent data characters. ATPUT was modified to send those characters (including, redundantly, the Data Mark) in TCP as "Urgent" data.

Unfortunately, the pipeline can be so clogged that ATPUT cannot even issue ASEND. This problem was solved by including the "send interrupt" (AINT) routine in the A-service compatibility interface. The TCP version of the AINT service simply sends a zero-length data segment marked "urgent". This should cause the receiving user-level protocol to unclog the pipeline looking for the urgent information; as the pipeline empties out, the real urgent data and the Data Mark can be sent, marked "urgent". This will advance the Urgent pointer past the real urgent data. The Urgent pointer is sent until the send left window edge passes it.

It is possible that the send window is zero, so no data can be sent. Therefore, calling ASEND with the Urgent bit on turns on the "ACK needed" flag in the TCPB. This flag will cause TCPACKT to send at least an empty <ACK> segment, which will contain the current Urgent pointer.

On the receive side, the ULPP is notified of urgent data in two ways:

* When the Urgent pointer advances in the data stream, the ULPP's ATTN (Attention) semaphore is signalled.

* There is a field in the TCPB which records the number of bytes which the ULPP must remove from the circular buffer to reach the end of the urgent information. The ULPP should consume data from the buffer until this Urgent Data Count field is reduced to zero.

In general, TCPIN increases the Urgent Data Count field and signals ATTN when the urgent pointer advances, and TCRLSE decreases the Urgent Data Count field as bytes are released from the circular buffer.

4.3.12. Connection States

The required states for a TCP connection are basically defined in the TCP protocol specification [PosTCP]. However, some minor variations were forced by particular features of the implementation. The actual states and their numerical representations are as follows:

* <u>Null</u> = 0

  The TCPB has been created but not initialized.

* <u>Listen</u> = 1

* <u>SYN Sent</u> = 2

* <u>SYN Received</u> = 3

* <u>Established</u> = 4

* <u>Close Wait</u> = 5

* <u>FIN Wait</u> = 6

  Note: there is a bit "Fin ACK'd" which may be turned on while in this state; this effectively creates a second FIN Wait state, in agreement with the current TCP document [PosTCP].

* <u>Closing</u> = 7

  <FIN>'s have been sent and received, so connection is awaiting acknowledgment of a <FIN> (or timeout).

* <u>Remote Abort Wait</u> = 8

  A <RST> has been received to abort the connection. The local ULPP needs to call ACLOSE to delete the TCPB.

* <u>RST/ACK Delay</u> = 9

  When a connection is being closed, the last segment to be sent will generally be a <RST> or an empty <ACK>. The TCPB must not be deleted until the segment has been sent to the IMP; unfortunately, such a segment is not subject to acknowledgment, so it must be removed from the No-ACK List by a timeout mechanism.

  We chose to hide this mechanism from the ULPP, in the following manner. ACLOSE will indicate successful close (return code = 0) as soon as the sement is sent. The TCPB will be removed from the control block environment (so AEXIT won't find it), and its state will be "RST/ACK Delay". The normal TCTIMEO mechanism will delete a TCPB in this state when its No-ACK List is emptied.

Figure 8 shows a state diagram for the implementation.

It is also necessary to form a correspondence between the TCP states and the effective states seen by a ULPP under the universal connection state model (see Figure 9). Although Figures 8 and 9 are superficially similar, there were a number of serious issues to be resolved.

(a) Good <SYN>'s and bad <SYN>'s

Under AHHP (for which the universal state model was originally designed), a process which has issued a passive listen for a connection has the option of "refusing" an open command ("RFC") that it doesn't like, by calling ACLOSE instead of AOPEN when the OPEN semaphore is signalled. An obvious mapping of TCP states into universal states would provide the "refusal" capability in TCP: basically, receiving the initial <SYN> would merely signal OPEN; the process would then call AOPEN to send <SYN,ACK>, or ACLOSE to send <RST>.

Unfortunately, this approach would force the ULPP to recover from an "old duplicate <SYN>" segment [PosTCP]. We feel that TCP should hide from the ULPP all artifacts of unreliable communication, including old duplicate <SYN> segments. Therefore, in the case of a passive open, the OPEN semaphore must not be signalled until the handshake is completed.

(b) One Call of AOPEN

Under AHHP, two calls of AOPEN are required for an active open. This allowed the allocation of a circular buffer to be deferred until the open handshake was complete, and satisfied the system requirement that the circular buffer be obtained by a routine executing under the ULPP ptask (so the storage obtained by PCORE would belong to the proper ptask).

Under TCP, it is desirable to obtain the circular buffer as early as possible, so that the first <SYN> segment can specify an initial receive window. As a result, we chose to obviate the second AOPEN call, although it is allowed for compatibility.

As a result of these considerations, a ULPP sees an effective TCP state diagram like that sketched in Figure 10. Calling AOPEN to buy a circular buffer effectively creates two new states from the

SYN-Received and Established states. Comparing this diagram with the universal states of Figure 9, we see that:

* "Established-1" state of TCP corresponds to the universal "Remote Open" state.

* "SYN-Received-1" state of TCP is hidden from the ULPP.

* "SYN-Received-2" and "SYN-Sent" states of TCP together correspond to the universal "Local Open" state.

(c) Implicit ALSTN

For reasons explained later, the incoming logger function issues an ALSTN call for the logging connection, in behalf of the ULPP that is being started. The ULPP will later issue ALSTN for the same connection (using the ICV list as parameter), and proceed with the open sequence. This makes several slight modifications in the universal state diagram for TCP:

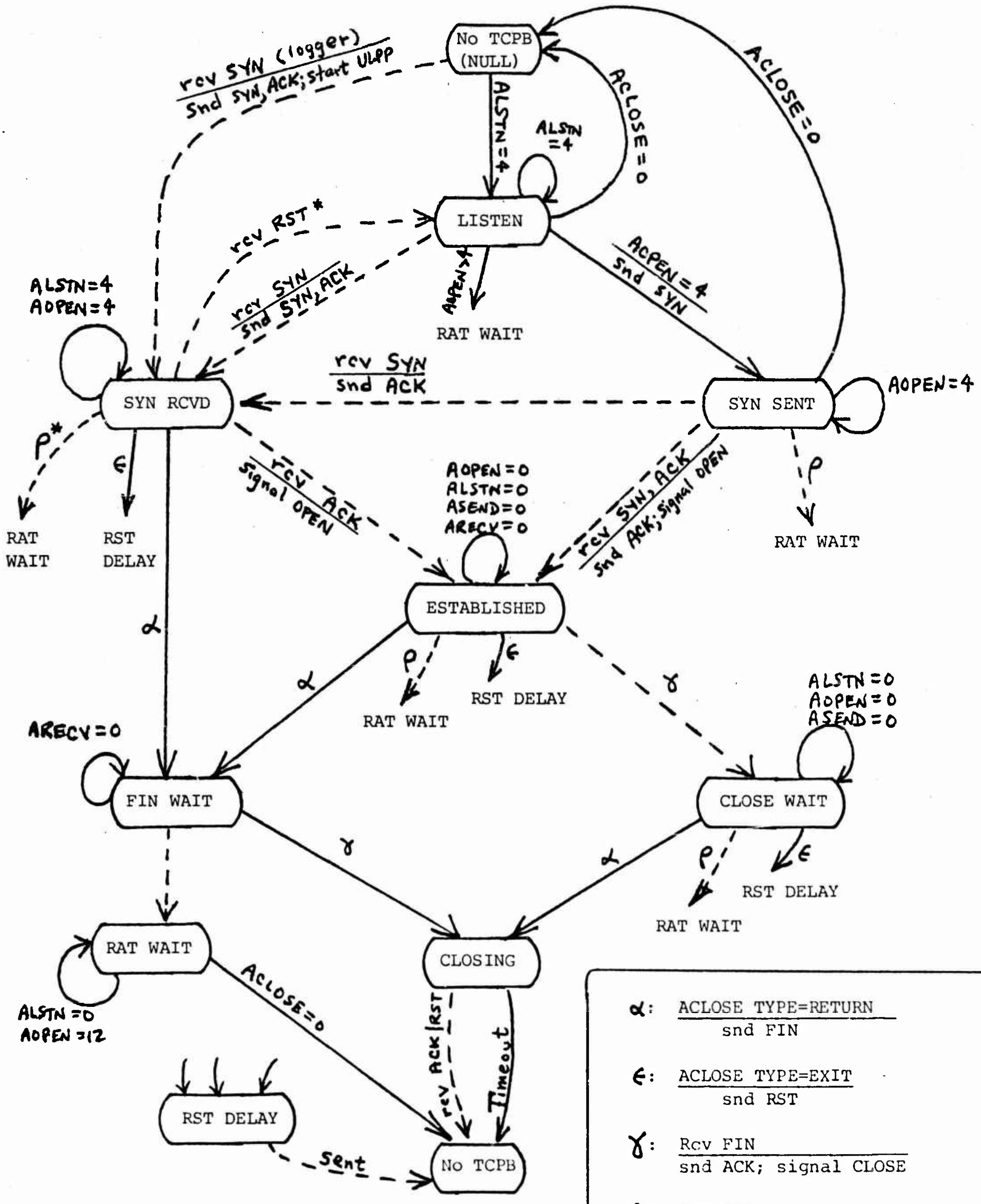
* ALSTN can be called more than once for the same connection (AHHP will not allow this).

* When ALSTN is called, the connection may already be open, and in fact it might have closed again. To preserve the universal state diagram, ALTSN will give a return code of 0 in either case.

(d) Half-Open Connection

Under AHHP, the universal "Remote Close" state is a (hopefully brief) intermediate state during the closing handskake. Under TCP, this state may last indefinitely, with the local ULPP continuing to send data even after it has removed all received data from the circular buffer.

Figure 8 -- TCP Connection States



α: ACLOSE TYPE=RETURN / snd FIN

ε: ACLOSE TYPE=EXIT / snd RST

γ: Rcv FIN / snd ACK; signal CLOSE

ρ: Rcv RST / signal CLOSE;

Figure 9 -- Universal (AHHP and TCP) Connection States

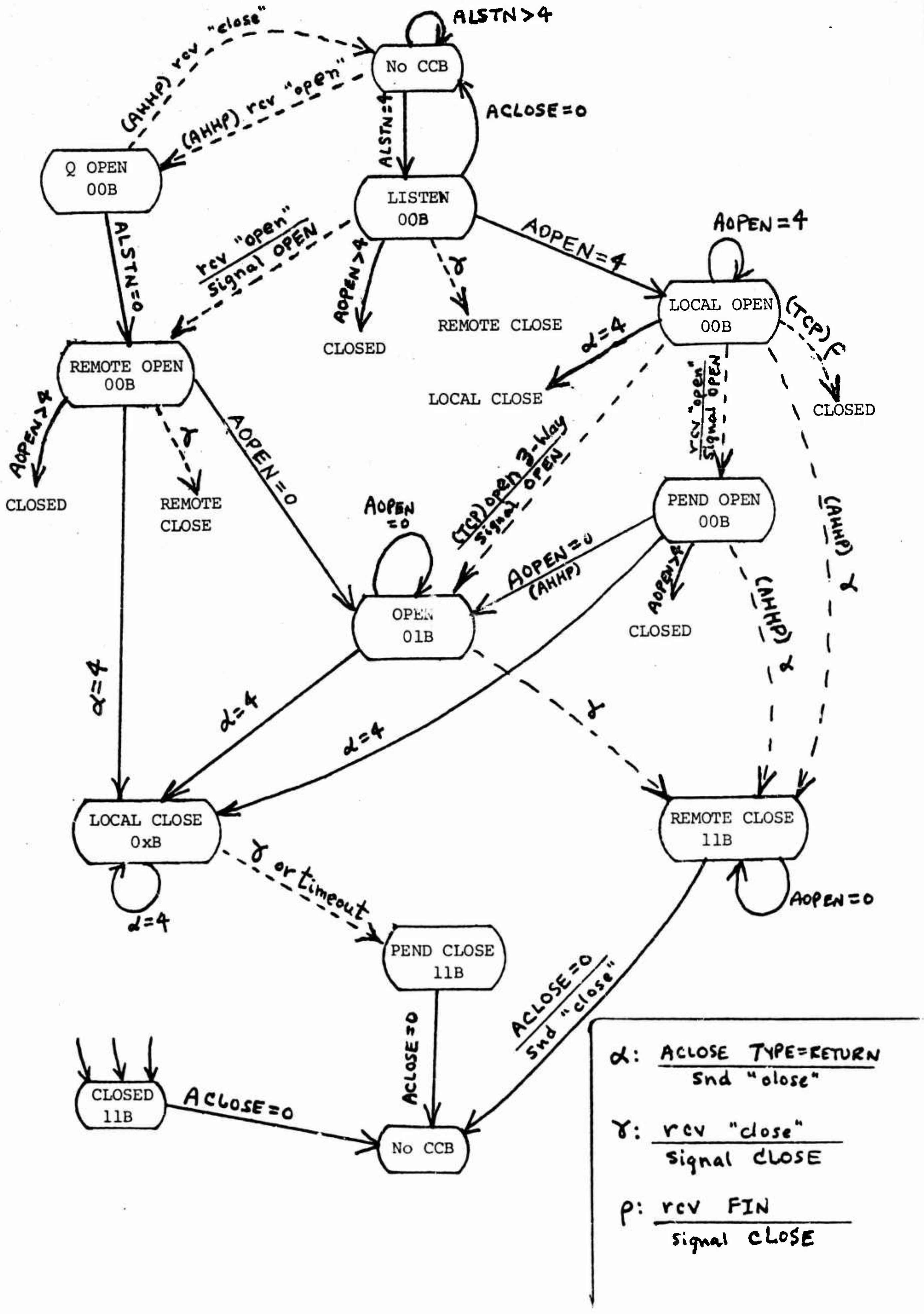
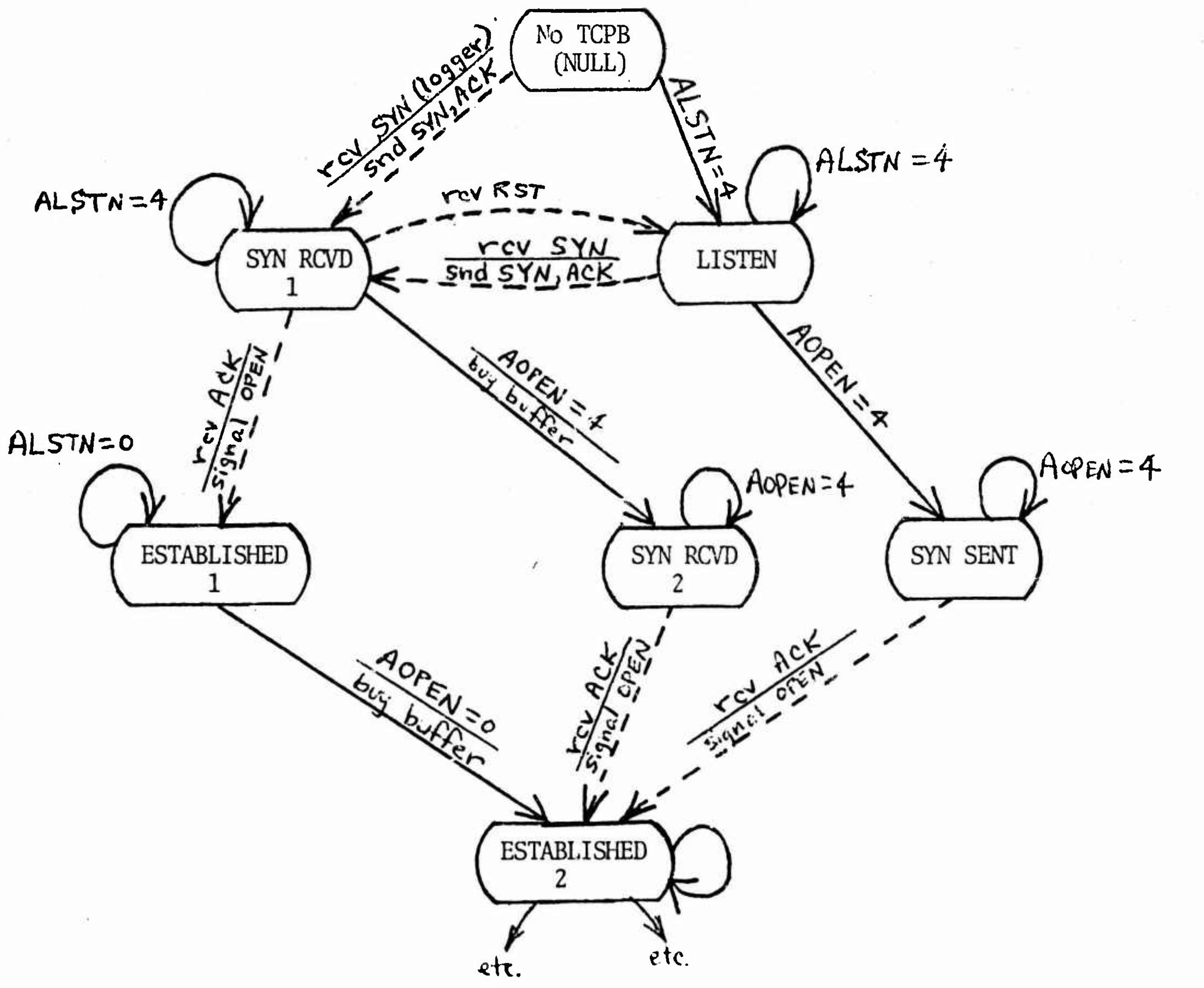


Figure 10 -- Effective TCP States for ULPP



4.3.13. Incoming Logger Function

A remote host can create a new server session using TCP by simply opening a connection to the appropriate "well-known port" (<u>WKP</u>). This invokes a mechanism commonly known as the <u>incoming logger</u>. TCPMOD behaves as if there were always an idle server ULPP listening for a connection on each WKP. In fact, a server ULPP is not created until the initial connection request actually arrives.

The incoming logger function for TCP is initiated by a <SYN> message from the remote user host. This message will specify the ports (U, WKP), where U is the remote (user) port and WKP is the local server port. This message is received by the IPP, which builds a new association (hence, ICB) for it, and passes the <SYN> message and the ICB address to TCPIN. The ICB points to a TCPB which is cleared to zero. In particular, the TCPB specifies the "Null" state (zero value), indicating to TCPIN that this is an incoming logger request. This will cause TCPIN to take the following actions:

(1) Build a parameter list and call INTERNET START to create a new session in the internet environment.

(2) If the START fails (e.g., because of a bad contact socket), send a "believable" <RST> segment and discard the <SYN>.

(3) Else, call the ALSTN A-Service to initialize the TCPB in the "Listen" state.

(4) Process the <SYN> segment in "Listen" state, advancing the state to "Syn Received" and sending a matching <SYN>.

Since TCPIN executes under INPTASK, INTERNET START does also; hence the ULPP which is forked will be inferior to INPTASK. INTERNET START sets up an ACE, which is chained from the P3CB pointed to by the IPB. It also assigns a session number and stores the proper ICV parameters in the primary PTAUSER fields.

Calling ALSTN at this time simplifies the code because it maintains the consistency of the appearance that the process was passively waiting all the time. It also allows the segment tracing mechanism, if enabled, to trace the <SYN> segment and the session creation.

4.3.14. Tracing TCP Transactions

The A-Service ATRACE [WolBr79] will build a trace buffer containing variable-length entries. The trace buffer is controlled by a pseudo-CCB called a "TRB" (Trace Block), using standard NCP circular buffer pointers. A TRB address is called a "trace handle".

To aid present and future TCPMOD debugging, provisions have been built into TCPMOD to associate a trace buffer with each connection. The TCPB includes a field for a trace handle for this buffer. If tracing is enabled, trace entries will be built by TCSEND, TCSEGOUT, TCLSTN, and TCPIN.

TCP tracing is enabled by a TCPB flag bit (TCPFlTRC). This bit is copied from a corresponding ICB flag, which is initialized from the IPB. Thus, the IPB controls the default for tracing. However, a systems programmer can turn on the trace bit in a particular TCPB at any time.

Freeing a TCP trace buffer has presented some difficult system design problems. There are two issues:

(1) There is an inherent race condition between closing and deleting a TCPB, and deleting its corresponding TRB. The problem arises in AEXIT, that will call ACLOSE for both the TCPB and the TRB, in the order in which they appear on the all-CCB chain. If the TRB is closed first, the trace handle in the TCPB may point to free storage. Note that AEXIT does not know about trace buffers as a resource; even if it did, the offset of the trace handle in the TCPB is assumed to be specific to the higher-level protocol, so AEXIT couldn't find it.

(2) Normally, we want a trace buffer to disappear when its connection is closed; otherwise, memory would quickly fill with "dead" trace buffers. However, during debugging we will sometimes want a trace buffer to be saved after the TCPB is deleted.

The ability to save a trace buffer is provided by an ICB bit that specifies "Test Mode". In Test Mode, a TRB will be owned by the permanent internet ptask INPTASK rather than by the ULPP ptask; as a result, the TRB will not be deleted when the ULPP exits. At present, there is no way to limit the number of old trace buffers built up in Test Mode; to delete them, it is necessary to issue the operator command that closes the IPP.

The first problem was solved by requiring that the trace buffer contain a TCPB pointer, whose offset in the TRB is assumed to be standard in the internet environment. Then the compatibility A-service routine ARPICLSE was designed to handle closing of a TRB specially; if there is a pointer to a TCPB, it closes the TCPB <u>first</u>.

4.4. AREAS FOR FUTURE WORK

There are three TCPMOD design issues to be addressed:

* Compatibility Interface Design

  As discussed earlier, we need a better conceptual model to assign functions to the ARPIxxxx and the TCxxxx routines of the compatibility interface.

* Transaction-oriented Interface

  The compatibility interface suppresses the datagram-like features of TCP, in favor of connections. A new transaction-oriented ULPP interface should be designed and implemented for TCP.

* Positive Notification of Send Complete

  We have mentioned some complexities in the current TCPMOD implementation that are required because the outgoing gateway returns no positive signal when it has sent a packet to the IMP. Impending changes in the IMP I/O driver code of the NCP will allow a positive signal to be returned, and this in turn could be used to simplify TCPMOD.

Beyond these issues, further TCP development will be concerned with testing and tuning the flow control and buffering strategies.

For example, the current formulas used to calculate retransmission timeout should be verified experimentally, by doing throughput tests with a variety of (known) distributions of round-trip delay and packet loss.

Handling internet traffic with large delays will require more reassembly buffers than are now provided, and may demand larger segments. It may be necessary for a particular TCP connection to choose its segment size dynamically. Similarly, liberal receive-window strategies should be tried in high-delay, high-bandwidth situations.

5. INTERNET TEST ENVIRONMENT

Development of the IP/TCP implementation required modifications and extensions to the existing NCP code. Errors in these changes, or in the INTMOD and TCPMOD modules themselves, could severely impact the running NCP. Furthermore, the debugging facilities within the NCP are largely static, while the general-purpose time-sharing TSO has a powerful interactive debugger. We therefore decided to create an internet test environment within TSO.

This TSO test environment included several new pieces of software:

* "Raw Packet" Interface to the NCP.

* NCP environment simulator.

* Gateway simulator.

* TSO test driver.

We will briefly describe each of these in turn.

5.1. Raw Packet Interface

As we discussed previously, a process within the IBM system obtains access to the ARPANET by opening an Exchange window to the NCP using the appropriate "well known tag", and then sending and receiving data through this window. The process normally employs a canonical internal user-level protocol, which is translated into the actual ARPANET user-level protocol by a ULPP within the NCP [Bra77].

For developing and testing new protocol modules, it is useful to allow a process to send and receive ARPANET messages at the "raw packet" level. Such a raw packet interface was implemented [Bra79A] to allow the internet test environment under TSO to use the ARPANET. However, the interface has already found other uses.

The raw packet interface is basically a new ULPP, named ARAWPKT. The process opens a window with the tag "ARAWPKT" and sends ARAWPKT an ANMOC parameter list that defines an NMC input intercept filter. The result is to create a full-duplex internal packet communication path to the process.

The process sends a packet through the Exchange window in the form of a WRE followed by the data that the WRE references. ARAWPKT makes minimal modifications to this WRE and calls ACPX QUEOUT to send it to the outgoing gateway.

A packet of data received through the Exchange window is prefixed by the 8-byte buffer header that the NMC intercept attaches to a message. This header specifies the lengths of the packet and the leader.

ARAWPKT also has an internal "loop-back" mode, in which each output packet is reflected into the receive circular buffer without traversing the hardware path to the IMP and back.

5.2. NCP Environment Simulator

To test the IP/TCP modules under TSO, is was necessary to construct a sufficiently-complete software environment for their execution. The first requirement was a subset of ICT that could be executed as a user program under TSO. An existing ICT simulator was adapted and extended for this purpose.

The next requirement was a LOGGER ptask; this was created as a subset of the real LOGGER. The test LOGGER performs the functions:

* Fork two fixed ptasks: NCP and INTERNET.

* Act as an outgoing logger by issuing pending Exchange opens for two tags: INPOLOG and ARAWPKT.

Thus, LOGGER forks INTERNET, which forks INPTASK. INPTASK will call ANMOC in the ARPANET gateway to create its input buffer.

Finally, a sufficient subset of the resident NCP module ARPAMOD was assembled and linkage edited together. This included the A-service transfer vector and all the AHHP modules which are shared by the internet environment, as well as the Telnet access method modules. Note that these modules are being assembled from exactly the same source programs that is used (or will be used, after testing) in the production NCP.

5.3. Gateway Simlator

The gateway simulator is contained in a module named INTEST. It uses the raw packet interface to extend the real gateway into the TSO test environment. INTEST includes the entry points:

* ARPANMOC

  This code simulates the NMC intercept routine, by opening an Exchange window to ARAWPKT in the NCP and passing across the parameter list; ARAWPKT then passes it to the real ANMOC within the NCP. It returns to its caller the address of an assembled-in pseudo-CCB.

* QUEOUT

  This code simulates the QUEOUT routine of the NCP. It is invoked by the ACPX QUEOUT macro to enqueue a WRE on the NOW queue and awaken the NCP ptask to send it to ARAWPKT.

* NCP

  This code executes as a ptask under TSO to simulate the action of the fixed ptasks NCP and IMPIO of the real NCP. That is, it performs the actual data transfers across the Exchange window to ARAWPKT.

  It is awakened by QUEOUT when there is output to send, or by Exchange when input arrives. For output, NCP assembles the WRE and data into a single packet, modifies the WRE slightly, and sends it through the Exchange window; then it dequeues the WRE from NOW and marks it "Complete".

  When data is received over the Exchange window, NCP moves it into a circular buffer under control of the pseudo-CCB. Then NCP signals the INPUT semaphore of the ptask that called ARPANMOC (INPTASK).

* ARPAHIO

  This routine, which is invoked by ACPX HALTIO, purges WRE's enqueued on the local NOW queue.

This set of routines effectively extends the gateway into the TSO job, so the internet routines can access the ARPANET gateway as if they were in the NCP.

5.4. PL/I DRIVER

For testing IP/TCP, we wanted to be able to invoke its services in a controlled manner, and to create nicely-formatted diagnostic listings. We wrote an interactive TCP driver using PL/I plus a set of small assembly-language subroutines that interface to the rest of the test environment.

The PL/I driver accepts the commands listed below. The driver prompts interactively for the parameters which are listed in parentheses after each command.

* OUTLOG (<outlog parm string>)

This command invokes the outgoing logger function. Specifically, it opens an Exchange window to invoke INPOLOG, and passes <outlog parm string> to it. INPOLOG, INTMOD, and TCPMOD operate as they would in the real NCP, creating a new user session as a ULPP ptask.

Successful completion prints out the session number.

* OPEN (<session number>)

  This command causes the ULPP ptask with the specified session number to issue an ATOPN.

* SEND (<session number>, <length>, <data string>)

  This command causes the ULPP ptask with the specified session number to issue an ATPUT for the specified data.

* RECV (<session number>)

  This command causes the ULPP ptask with the specified session number to issue an ATGET call, and prints the resulting character string on the terminal.

* CLOSE (<session number>)

  This command issues an ATCLOSE call.

* DUMP

  This command prints out the contents of the trace buffers associated with all TCP connections.

* ARB(<ICB address>, <TCP header and data>)

  This command sends an arbitrary TCP segment on a specified association.

The IPP in the TSO test environment is configured with logical host number 1, so it can open connections to the production IPP (logical host 0) within the NCP.

6. CONCLUSIONS

This report has described an implementation of the internet protocols IP and TCP for an IBM 360/370 computer. This implementation is currently able to communicate with the other internet hosts supporting these protocols and Telnet. The test of "communication" is basically the ability to log into the remote system using the Telnet protocol. The OAC TCP is available on the ARPANET 24 hours a day, and we believe that it could be used for production access to TSO, for example.

Our initial goal, a system-call interface for ULPP's which is compatible between TCP and AHHP, was largely realized. The major differences that remain are due to real differences in the two protocols. As noted earlier, the majority of ULPP's are insulated entirely from these differences because they use the Telnet access methods.

We believe that the current NCP, including IP/TCP, could be installed on any IBM system running OS/MVT. During the next year, the NCP will be converted to the virtual memory operating system, MVS. The IP/TCP implementation contributes no operating system dependency to this conversion. On the other hand, the existence of the new internet protocol implementation gives additional weight to the requirement that the existing NCP be converted with minimal changes.

There are a number of tasks for the future development and support of the OAC implementation of the internet protocols. We will list some of them here.

(1) Maintenance

Little stress-testing has been performed, and we anticipate that the IBM implementation still contains obscure bugs at this time. Reliability tests using a traffic generator and Plummer's "Flakey Gateway" [Plum78] would be useful in finding these bugs.

(2) Status and Test

The current test and monitoring facilities are still inadequate for long-term maintenance of the NCP using IP/TCP. For example, NCP code is needed for dumping trace buffers, manipulating the IP/TCP parameters, and displaying the status of TCP connections. In addition, better means for operator monitoring and control are needed (for AHHP as well as TCP).

(3) Performance

Although the IP/TCP code gathers some rudimentary statistics, there is no provision for recording or observing them. In addition, we need to create simple measurement tools, including a traffic generator, an echo server, and a discard server.

(4) Additional Features

Earlier sections described a number of areas that may require extensions or improvements. In addition, we expect that the protocols themselves will continue to evolve, particularly in the areas of routing, type of service, and optimizing the algorithms for flow control and retransmission. This evolution will inevitably require changes in the code described here.

(5) Convert FTP and MSG

There are a number of design decisions in the current implementation whose correctness can only be established (or contradicted) when other higher-level protocols than TCP are implemented, and when user-level protocols other than Telnet are converted to TCP. Serious candidates include MSG, the transaction-oriented interprocess communication protocol for the National Software Works, and File Transfer Protocol. It is unclear whether MSG should be interfaced at the IP level or the TCP level.

Finally, we are anxious to acknowledge the major contribution to this effort made by Denis de la Roca, who helped code a number of the IPP and TCPMOD routines. He was patient in the face of unforgivable bugs as well as numerous shifts in design as the protocols evolved. Lou Rivas was also an immense help in getting the code to actually function within the NCP environment.

7. REFERENCES

BBN1822

BBN. "Specification for the Interconnection of a Host and an IMP", Report 1822, Bolt Beranek and Newman, Cambridge, Massachusetts, revised January 1976.

Bra76

R. Braden. "The National Software Works", Technical Report TR9, Office of Academic Computing, UCLA, December 1976.

Bra77

Braden, R. "A Server Host System on the ARPANET", Fifth Data Communications Symposium, Snowbird, Utah, September 1977.

Bra79A

Braden, R. "Gateway Interfaces within the ARPANET NCP", Technical Report TR17, Office of Academic Computing, UCLA, October 1979.

Bra79B

Braden, R. "Interface Specifications for Programming a Higher-Level Host-Host Protocol using Internet Protocol", Technical Report TR19, Office of Academic Computing, UCLA, December 1979.

BraFe72

Braden, R. and Feigin, S. "Programmer's Guide to the Exchange", Technical Report TR5, Office of Academic Computing, UCLA, March 1972.

BraTCP

Braden, R. "Program Logic Manual for TCP", Office of Academic Computing, UCLA, in preparation.

BraIP

Braden, R. "Program Logic Manual for Internet Protocol", Office of Academic Computing, UCLA, in preparation.

CerKa74

Cerf, V. and Kahn, R. "A Protocol for Packet Network Intercommunication", IEEE Transactions on Communication, vol. C-20, 5, May 1974.

FeinPos

Feinler, E. and Postel, J. eds. "ARPANET Protocol Handbook", NIC 7104, published for the Defense Communications Agency by SRI International, Menlo Park, California, revised January 1978.

McKen72

McKenzie, A. "Host-Host Protocol for the ARPANET", NIC 8246, January 1972. Revised and published in [FeinPos].

McKen73

McKenzie, A. "Telnet Protocol", RFC 562, NIC 18638, August 1973. Revised and published in [FeinPos].

Plum78

Plummer, W. "Flakeway in Operation", ARPANET message to TCP and Internet Protocol groups, September 1978.

Pos71

Postel, J. "Official Initial Connection Protocol", NIC 7101, June 1971. Published in [FeinPos].

PosINS

Postel, J. "Internet Name Server", IEN-116, August 1979.

PosIP

Postel, J. "Internet Protocol", IEN-111, August 1979.

PosTCP

Postel, J. "Transmission Control Protocol", IEN-112, August 1979.

RivLB77

Rivas, R., Ludlam, H, and Braden, R. "An Implementation of the MSG Interprocess Communication Protocol", Report TR12, Office of Academic Computing, UCLA, May 1977.

RivWo77

Rivas, R. and Worth, D. "Server FTP Program Logic", Systems Document Q049, Office of Academic Computing, UCLA, February 1977.

Tol77

Tolomei, V. "Server FTP Program Logic", Systems Document Q049, Office of Academic Computing, UCLA, February 1977.

WolBr79

Wolfe, S. and Braden, R. "Programming User Level Protocol Processes for the ARPANET NCP", Technical Report TR18, Office of Academic Computing, UCLA, November 1979. Revision of OAC document Q039A.

Wolfe74

S. Wolfe, "ICT Monitor Services and Macros", System document Q037, Office of Academic Computing, UCLA, revised September 1974.

8. APPENDIX A -- OUTGOING LOGGER PARAMETER SYNTAX

The internet protocol program (IPP) includes a mechanism to initiate the outgoing logger function. When a local process opens an Exchange window to LOGGER, LOGGER forks a transient INPOLOG ptask, and passes the Exchange window to it.

INPOLOG issues an Exchange to get from the local process a character string that defines the internet host, higher-level protocol, the contact port, and possibly the ARPANET gateway to be used. This Appendix defines the syntax and semantics of this Outgoing Logger Parameter string.

We use an extended BNF, with square brackets [ ] surrounding optional items.The terminal symbols are:

<hlp name> ::= <name>

<AHHP name> ::= <name>

<internet host name> ::= <name>

> an arbitrary string of letters (upper and lower case are equivalent), digits, and the break characters "-" and "_"; the first character must be a letter.

<dec number> ::= <string of digits>

> a decimal number, i.e., a string of digits (0-9).

<octal number> ::= #O<string of of octal digits>

> an octal number, i.e., only digits 0-7.

<hex number> ::= #H<string of hex digits>

> a hexadecimal number, i.e., a string of digits 0-9, A-F.

Delimiters are < > ( ) : ,

None of these terminal symbols may contain imbedded blanks, but blanks are allowed freely between terminal symbols.

We can now present the syntax.

```
  <Outlog Parameter String> ::=

      <AHHP string> [ , <socket> ] |

      [ <hlp name> ] : <internet string> [,<port> ]

          This syntax provides a compatibility  interface  to
          the  outgoing logger; either the old AHHP syntax or
          the new internet syntax is acceptable. An  internet
          address  string  must  have  start  with  a  colon
          (optionally   preceded   by   the   name   of   the
          higher-level protocol).

      <AHHP string> ::=

          <ARPANET host address>

      <ARPANET host address> ::=

          <AHHP name>

              This is a standard ARPANET  host  name,  as  it
              appears  in  the AHHP host tables.  It may be a
              full name, or a "nic-name", and is  limited  to
              12 characters.

          | <dec number> / <dec number>

              This  is  a  24-bit ARPANET host number, in the
              standard form: <host #>/<IMP #>.

           | <hex number> |<octal number>

              A    hexadecimal    or    octal    number    is
              right-justified in 24 bits.

           | <dec number> [ / ]

              This  form  (with  an  optional trailing slash)
              defines the old-form 8-bit ARPANET host number,
              <host  #>*64+<IMP  #>.  It will be converted to
              24 bits.

      <port> ::= <dec number>

      <socket> ::= <dec number>

              <port>  must  be  less  than 2**16 and <socket>
              must be less than 2**32.
```

<internet string> ::=

<internet host string> [(<gateway spec>)]

In most cases, the <internet host string> will imply a gateway to reach the specified host. However, in any case the gateway can be specified explicitly.

<gateway spec> ::=

<ARPANET host address>

A full gateway specification requires not only the ARPANET host addresss, but also the link number and the service level (standard vs. uncontrolled). There is currently no syntax for explicitly setting the last two.

<internet host string> ::=

<internet host name>

This is an internet host name appearing in INAMTBL. It implies the full internet host address (8-bit network number and 24-bit <internet host number>),the default higher-level protocol, and the full gateway specification. The higher-level protocol and gateway host address can be explicitly overidden.

| [<Network=ARPA>] <ARPANET host address>

[/ <logical host>]

This entry implies the full internet address and the ARPANET gateway address; however, the link number and service level for the gateway are not implied, so the defaults will be used.

The logical host number can be specified. Note the forms:

a/b => a * 2**16 + b (24-bit host number).

e//h => convert 8-bit host number 'e' to 24 bits and add h * 2**8 (logical host).

a/b/h =>(a * 2**16 + b)+ h * 2**8.

```
        [| <Network~=ARPA> ] <internet host number>

             If  the  network is specified by number or name
             and exists in INAMTBL, it will imply  the  full
             gateway  specification  (gateway  host address,
             link number, and service type).

             No  default  higher-level  protocol is implied.

    <Network=ARPA> ::=
    <Network~=ARPA> ::=

        <'<'> <network name> <'>'>

        | <'<'> <dec number> <'>'>

             The network name is enclosed in <  >  brackets,
             and    may    be  specified  either  by  name  or
             numerically.

   <internet host number> ::=

        <dec number> | <hex number> |<octal number>

             This   defines  a  full  24-bit  internet  host
             number.
```

9. APPENDIX B -- NCP A-SERVICES

This appendix lists all the A-services for both the AHHP and the internet environments, giving the function and the name of the module which implements each. The ACPX services used internally by the host-host routines are also included. Finally, we list the resident modules included in ARPAMOD that are not A-services; these are the fixed ptask modules and the tables.

When different modules are invoked by the AHHP and internet transfer vectors, then the AHHP module name is followed by the internet module name. When a functions is performed by an entry point within another module, the entry point name is given in square brackets following the name of the containing load module.

9.1. Commutator Support Services

* PATTACH

  Function: ptask initialization following PATTACH call: PLOAD inferior module, and propagate A-service transfer vector from superior ptask.

  Module: ARPAATCH

* PDETACH

  Function: complete PDETACH (null routine).

  Module: ARPADTCH

* PEXIT

  Function: free ARPANET-dependent resources when ptask exits.

  Module: ARPAEXIT (Note 1)

* "A-SPIE"

* "A-STAE"

  Function: Link to user abend (SPIE/STAE) exit.

  Module: ARPADBUG [ARPASPIE, ARPASTAE]

9.2. Environment Creation and Control Services

* ACEBUY

Function: Create a session by buying and initializing an ACE.

Module: ARPALOG [AACEBUY] (Note 1)

* ACESELL

  Function: Delete a session by unchaining and deleting an ACE.

  Module: ARPAEXIT [AACESELL] (Note 1)

* ABUF

  Function: Create, delete a receive circular buffer. Called internally by ARPAOPEN and TCOPEN.

  Module: ARPABUF

* AGHCT

  Function: Find or create an AHHP Host Control Task for a given host.

  Module: ARPAGHCT (Note 2)

* ATRACE

  Function: Create a variable-length entry in a circular trace buffer.

  Module: ARPTRACE

9.3. ARPANET Gateway Services

* ACPX QUEOUT

  Function: Enqueue a message for the outgoing gateway.

  Module: NCP [QUEOUT]

* ANMOC

  Function: Create or destroy an NMC intercept filter.

  Module: ARPANMOC

* ACPX HALTIO

  Function: Purge the outgoing gateway queues of all WRE's for a given CCB/ICB.

Module: ARPAPRGE [ARPAHIO]

* (no macro)

  Function: Used by ARPANET gateway for sending host-IMP messages.

  Module: IMPIO [DOQP]

* AHLUP

  Function: Map ARPANET host number to and from host id.

  Module: ARPAHLUP

9.4. Connection Services

* ALSTN

  Function: "Listen", i.e., passive open of new connection.

  Module: ARPALSTN, ARPILSTN

* AOPEN

  Function: Active open of a connection.

  Module: ARPAOPEN, ARPIOPEN

* ASEND

  Function: Send data over ARPANET connection.

  Module: ARPASEND, ARPISEND

* APURGE

  Function: "Purge" all active ARPANET I/O on a connection.

  Module: ARPAPRGE, ARPIPRGE

* ARECV

  Function: Receive data from an ARPANET connection.

  Module: ARPARECV

* ARLSE

Function: "Release" received data from circular buffer. Called internally by ARECV MOVE.

Module: ARPARLSE, ARPIRLSE

* AINT

  Function: Send a host-host interrupt (or for TCP, make the data sent so far "urgent").

  Module: ARPAINT, ARPIINT

* AALLC

  Function: For AHHP, send deferred allocation command.

  Module: ARPAALLC (Note 2)

9.5. AHHP Protocol Modules

* (no macro)

  Function: Send host-host command on control link.

  Module: ARPACMND

* (no macro)

  Function: Segment and send AHHP message(s).

  Module: ARPALGO

* (no macro)

  Function: Map host and link into CCB address.

  Module: ARPAFCCB

* (no macro)

  Function: Map ARPANET host number to and from host id.

  Module: ARPAHLUP

9.6. Telnet Access Method

* ATOPEN

  Function: Open a Telnet connection.

  Module: ATOPN (Note 1)

* ATCLOSE

  Function: Close a Telnet connection. Also used internally by AACESELL to free TCCB's.

  Module: ATCLS (Note 1)

* ATPUT

  Function: Send data on a Telnet connection.

  Module: ATPUT (Note 1)

* ATGET

  Function: Receive data from a Telnet connection.

  Module: ATGET (Note 1)

9.6.1. V-Cons

The following address constants appear on either the A-service or the ACPX transfer vector:

* (no macro)

  Function: Address of list of outgoing logger (Exchange window) control areas.

  Address: V(PROTLIST)

* INTERNET P3CB

  Function: Address of IPP control area, P3CB.

  Address: V(INTP3CB)

* (no macro)

  Function: Address of internet transfer vector; entry point of transient module INTMOD.

  Address: V(INTNETRV) (Note 3)

* ACPX LFLAG

  Function: Address of logger control flags.

  Address: V(LFLAG)

* (ACPX macro)

Function: Address of transfer vector for internal ACP interfaces.

Address: V(ACPXTRV)

9.6.2. Internal ACP Interfaces

The following routines are used internally by the ACP, and are not expected to be directly called by ULPP's; therefore, they are not true A-services.

* (no macro)

  Function: Used internally by AHHF to obtain a CCB and add it to environment chains. This routine does not appear on any transfer vector.

  Module: ARPAMCCB

* ACPX SOCKET#

  Function: Used internally to allocate a new session number.

  Module: ARPASOCK

* ACPX INSRCCB

  Function: Insert a CCB (or internet equivalent) into control block chains to create normal environment for ULPP.

  Module: ARPALSTN [INSRCCB]

* ACPX REMVCCB

  Function: Remove a CCB (or internet equivalent) from control block chains.

  Module: ARPACLSE [REMVCCB]

* ACPX ULSTART [ULSTART]

  Function: Create a new session by buying an ACE, issuing PATTACH to create the primary ULPP, and setting the ICV.

  Module: ARPALOG [ULSTART]

* ACPX OLOGERR [OLOGERR]

Function: Standard interface to ARPACOMS transient module, to report outgoing logger error to user process.

Module: INPTASK [OLOGERR]

Notes:

Note 1: the same module is used in both internet and AHHP environments, but acts slightly differently in each environment.

Note 2: appears only on the AHHP transfer vector.

Note 3: appears only on the internet transfer vector.

Finally, we list the resident modules which are not A-services or ACPX services. These are:

* ARPAMOD -- the A-service and ACPX transfer vectors, for all environments.

* ARPALOG -- LOGGER and HCT fixed ptask code.

* IMPIO -- IMPIO fixed ptask code.

* ARPANCP -- NCP fixed ptask code.

* INPTASK -- Internet protocol program fixed ptask code.

* INPTASK[INTERNET] -- Internet control ptask code.

In addition, ARPAMOD includes the following resident tables:

* HOSTS -- ARPANET Host tables

* ARPAICP -- Incoming and Outgoing Logger tables

* ARPAICP[PROTLIST] -- Outgoing Logger chain

* ARPAMSG -- WTO text table

* IPBLIST -- Internet Protocol Block ("IPB") list

* INTP3CB -- Internet control area ("P3CB")

10. APPENDIX C -- CONNECTIONS

This section contains some details of the semantics of connections. This information is important to the programmer of a ULPP or for the implementation of a new higher-level protocol.

10.1. Opening / Closing a Connection

For compatibility, ULPP's in the AHHP and TCP environments use a "universal model" for the apparent states of a connection. This section describes that model in terms of the system call sequence for the ULPP, and also notes any specific exceptions for AHHP or TCP. Figure 9 shows the universal state diagram.

To create a connection, the ULPP must first issue ALSTN. The possible results of this call are:

10.1.1. ALSTN Return Code > 4:

Fatal error, no CCB was created.

10.1.2. ALSTN Return Code = 4:

CCB was created and its address is returned in R1. The connection is passively awaiting a remote open request. The local process may:

(1) Call AOPEN to actively open ("initiate") the connection.

The possible results are:

(a) AOPEN Return Code > 4 and CCBLOG = 11B (closed).

Fatal error in AOPEN. Call ACLOSE (which should delete CCB and return 0).

(b) AOPEN Return Code = 4:

Open is pending, awaiting completion of handshake. After OPEN semaphore is signalled (and CCBLOG=01B), repeat this step.

However, if CLOSE semaphore is signalled (and CCBLOG is set to 11B), call ACLOSE to delete the CCB. Note on TCP: the second AOPEN call is unnecessary, if the first call specified a circular buffer size.

(c) AOPEN Return Code = 0 and CCBLOG = 11B (closed)

Connection never opened, or opened and then closed immediately. Call ACLOSE (which should delete CCB and return 0).

Note on TCP: the "Reset" bit may be on (TCPFLRST) to indicate that the connection was refused by the remote host. In any case, ACLOSE should be called.

(d) AOPEN Return Code = 0, CCBLOG=01B (open).

Connection is open. The OPEN semaphore will have been signalled, as well.

(2) Call ACLOSE to retract open request.

Normally, ACLOSE will delete CCB and return 0.

Note on TCP: for logging connection, ACLOSE TYPE=RETURN may return 4 (pending); in this case, issue PWAIT CLOSE and then call ACLOSE again.

10.1.3. ALSTN Return Code = 0:

An open request was received from the remote host already.

The ULPP should immediately either:

(1) Call AOPEN to complete open.

The possible results are exactly the same as those shown earlier for AOPEN, except here Return Code = 4 is impossible.

Note on TCP: this call is necessary to build a circular receive buffer.

(2) Call ACLOSE to "refuse" the connection.

Note: Note on TCP: "refusal" is not actually possible, as the connection is already open; hence, ACLOSE will simply close the connection immediately.

If the connection is now open, the ULPP can call ASEND to send data.

When the remote host sends a close request, the CLOSE semaphore is signalled and CCBLOG is set to 11B. The ULPP should continue to take data from the circular buffer (and call ARLSE) until the buffer is empty.

Note on TCP: ASEND may be called even if the CLOSE semaphore has been signalled and CCBLOG is 11B, until the ULPP calls ACLOSE or the "Reset" bit it turned on.

To close the connection, call ACLOSE. If the connection can be closed immediately, ACLOSE will delete the CCB and return 0. However, the non-blocking ACLOSE call (TYPE=RETURN) may result in return code 4 (pending); in this case, the ULPP should wait until the CLOSE semaphore is signalled and then repeat ACLOSE.

Note on TCP: there is an ACLOSE TYPE=ABORT call, that sends a <RST> and always returns 0.

10.2. AHHP Connection States

It will sometimes be useful to know the mapping of AHHP connection states into the universal state diagram seen by a ULPP. In particular, the bits in CCBLOG will have the values shown by the following table:

STATE	BITS IN CCBLOG	CCBLOG IN HEX
Listen	(none)	00
Local Open	FLLRF	04
Remote Open	FLRRF	08
Pend Open	FLLRF+FLRRF	0C
Open	FLLRF+FLRRF+FLOPN	4C
Local Close	FLLCL+(optionally) FLLRF+FLRRF+FLOPN)	4D
Remote Close	FLRCL+FLCLS	C2
Pend Close	FLLCL+FLRCL+FLCLS	C3

The first two bits of CCBLOG form a 3-valued state indicator used by the ULPPs. In particular, FLOPN is the "open" value 01B, and FLCLS is the "closing/closed" value 11B for these two bits. The other flags represent single bits.

10.3. CCB Contents

For compatibility, the following fields have the same offset in a CCB and in a hlpB. A ULPP which depends upon any other fields cannot be compatible with both the AHHP and internet environment.

* Flags (CCBFLG/TCPFLAGS)

  The flag bit CCBFlNHH will be off in all CCB's, and the corresponding bit will be on in all hlpB's.

* Open/Close State Bits (CCBLOG)

  These two bits must be tested by the ULPP to determine the state of the connection (as seen by the ULPP); see below.

* PTA Address (CCBPTA)

  This is the address of the PTA under which ALSTN was called, and which therefore owns the connection.

* Control CCB Address (CCBCTRL/TCPCTRL)

  For AHHP, this is the address of the appropriate "control CCB"; for TCP it is the address of the P3CB (pseudo control CCB).

* Local Socket Number (CCBLSCK)

  This is a 32-bit number used to label the CCB/hlpB; the high-order 16 bits must be the session number.

* CCBBUFB, -E, -R, -U, -L

  These five fullwords contain pointers and values controlling the circular buffer for receiving data.

  CCBBUFB= Address of beginning of buffer.

  CCBBUFE= Address of first byte beyond end of buffer.

  CCBBUFL= Length of buffer in bytes, i.e., CCBBUFE - CCBBUFB.

  CCBBUFR= Bit address of first user byte in buffer, or zero if there is none.

  CCBBUFU= Bit address of first bit beyond user data in buffer.

Note: "beyond" is meant in a circular sense: if the user data ends exactly with the last bit in the buffer, then BUFU will point to the first bit in the buffer (i.e., BUFU= 8*BUFB in this case). Because the data may wrap around to the beginning of the buffer, BUFU may be less than or equal to BUFR. The ambiguity between a full circular buffer and an empty one is resolved by making BUFR zero for an empty buffer but equal to BUFU for a full buffer.

* All-Connection Chain Word (CCBCCB)

  This word is used to as a link in a chain of all CCB's and hlpB's. This chain is used by AEXIT to close any open connection for a ptask which is exiting.

* ACE Address (CCBACE)

  This is the address of the ACE for the session under which this connection was opened.

* ACE Chain Word (CCBCHA)

  This word is used for the ACE chain of all CCB's for this session.

10.4. Pseudo-CCB

It is sometimes convenient to create pseudo-CCB's, blocks which are treated in the environment like CCB's but are not associated with real ARPANET connections. This allows the environmental control A-services to be used for these control blocks. In particular:

* ABUF may be used to obtain a circular buffer and set up the buffer pointers in the pseudo-CCB.

* ARECV will obtain data from this buffer.

* A pseudo-CCB is chained into the all-CCB chain.

* ACLOSE will delete a pseudo-CCB, and also free a circular buffer, if any, associated with it.

* AEXIT will call ACLOSE for a pseudo-CCB if the owning ptask exits without itself deleting the pseudo-CCB.

Thus, the pseudo-CCB can be used to ensure that the control block and circular buffer will be freed if the ptask abends. For this reason, NMC intercept filters and trace buffers are controlled by pseudo-CCB's, for example.

In order to be acceptable to the environmental A-services, a pseudo-CCB must satisfy some special constraints on the CCB fields listed above.

(1) Flag bits: CCBF1CTL, CCBF2NHH are off.

(2) Flag bit CCBF2BUF may be on to cause PCORE FREE to be issued for circular buffer.

(3) CCBLOG bits must be X'80'.

A CCB with this configuration will simply be unchained and freed by the AHHP ACLOSE module (ARPACLSE).