An Architecture for Interprocess Communication in UNIX*

— DRAFT of June 22, 1981 —

William Joy and Robert Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

ABSTRACT

This proposal describes a set of extensions to UNIX integrating interprocess communication mechanisms (IPC) for use in an networked environment. The proposed extensions provide virtual circuits and datagrams, both of which admit simple and efficient implementations. To support multiplexing of communications in a single process both a synchronous facility similar to the ADA *select* statement and an asynchronous software-interrupt (signal) based facility are proposed.

The IPC facilities are integrated into the current UNIX name space by *portals*, entries in the file system that invoke server processes when accessed. Portals are used to build services accessible in the standard UNIX name space. We describe how the basic IPC facilities and portals may be used to provide services on a single machine and in a internetwork environment.

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

TABLE OF CONTENTS

1. Introduction

This is a proposal for enhancement of UNIX to provide inter-process communications facilities for a network environment.

An examination of the reasons for the widespread use and popularity of UNIX should provide a good point of departure for designing new system facilities. There is strong evidence that the clean and simple interfaces of UNIX are the reason for its success. The command invocation interface, the file system, and the *pipe* mechanism all have straightforward semantics, admit simple and efficient implementations, and are easy to use. They allow UNIX programmers to access programs, files and to build new applications by composing the actions of existing programs, often without writing any new programs. The functional style of program building and the existence of many single-function tools contributes greatly to UNIX's accessibility [Kernighan 81]. Related research in programming environments attempts to apply a similar tool-kit approach [Osterweil 81].

We are not attempting to design a monolithic IPC facility that will provide everything needed by UNIX users. We believe that this is neither desirable nor possible. Instead, we are interested in constructing IPC mechanisms that will be simple to use, admit efficient implementations, and allow higher level mechanisms for communications to be constructed. The UNIX kernel functions largely as an input/output multiplexor [Thompson 76], and we believe that this is also a proper role for a IPC kernel for UNIX.

We have several specific applications in mind for the IPC facilities. There are many programs running on a single UNIX system that need better IPC facilities and the handling of multiple i/o activities provided by the IPC design. The ability to access to both local and distant networks is also important. To the extent possible, we wish to mask the differences between the datagram and virtual circuit facilities of the various networks so that network portable programs can be written. Remote file access, remote login and communication between parts of distributed user applications need to be supported on all networks.

We also intend to use the IPC facilities to build both a distributed UNIX system of autonomous personal machines and a tightly coupled central computing facility providing a distributed file system and load-sharing. Tightly-coupled distributed UNIX systems have been or are being constructed by a several groups working with UNIX [Hwang 81a] [Popek 81] [Rowe 81], and the IPC design presented here can be used to construct such systems. Loose coupling of personal computers will be important in our environment where a large number of machines are under local control. Providing sharing of resources while retaining autonomy is a problem addressable within our design and one worthy of further study [Clark 80].

This paper has 7 sections. Section 2 outlines issues in distributed systems and develops a set of goals and non-goals for our IPC design. Section 3 describes the IPC primitives for internetwork communication in UNIX in great detail. Section 4 discusses the interaction of the IPC primitives with existing UNIX facilities to allow servers and other facilities to exist in the UNIX name space. Section 5 compares the IPC design presented in sections 3 and 4 to other work. Section 6 outlines how the IPC mechanisms can be implemented and reports on the efficiency of a prototype implementation. Section 7 summarizes the paper and notes areas for further investigation. An appendix describes the layered models of internetworks and internetwork systems that are a basis for this work and with which it can be compared.

2. IPC design issues

We see at least four distinct areas where UNIX IPC will be important:

* In the support of single machine applications involving simple data transfer between unrelated processes. Efficiency of the intra-machine IPC mechanisms is critical in this case.
* In accessing communications facilities of the network and internetwork architectures, for communication between processes on different processors. Efficient translation from the IPC facilities to the network facilities and the ability to communicate with non-UNIX systems are especially important here.
* In constructing services on a tightly coupled set of machines to build a network operating system [Kimbleton 76] [Watson 80a]. Naming, protection, synchronization, resource management and performance issues are important for such systems.
* In retaining the autonomy of individual machines while allowing coherent access to distributed resources. Some researchers believe that this is the major research issue in distributed computing [Svobodova 79] [Clark 80]. Providing uniform and coherent access to distributed objects while meeting local needs is a difficult issue.

In achieving these goals we wish to provide facilities that complement the current UNIX facilities in style, and appear integrated as a total system. The extent to which we have achieved this goal in our design can be judged only subjectively, but we have tried to be conscious of this goal and hope we have designed culturally compatible extensions to UNIX.

2.1. Layered approach

To approach the many issues raised by these requirements, distributed system and network design is based on layered systems and protocols. Important examples include the Arpanet protocol family based on the Internet Protocols [Cerf 74] [Lyons 80] [Postel 80b] [Postel 80c], the Pup Internetwork architecture [Boggs 79] as supported by the Pilot operating system [Redell 79], the ISO model of Open Systems Interconnection [ISO 79] [Zimmerman 80] which incorporates the X.25 protocol [Rybczynski 80] [Folts 80], and models for network operating systems such as that of [Watson 81a] given in [Lampson 81a]. Readers unfamiliar with these models should refer to Appendix A.

We first present considerations at the lower levels of the models and then proceed to higher level considerations. We begin by presenting our assumptions about the internetwork environment. Successive sections consider facilities desirable in the kernel interface to the internet, data type and representation conversion issues, and the provision of services easily accessible from UNIX programs.

2.1.1. Internetwork interface

A system that wishes to provide an efficient and reliable interface to several networks must abstract common facilities and features of those networks into a consistent model for network communication. We believe the following abstract network features are critical and non-controversial, and likely to be constructible from the concrete facilities available in most networks and internetworks:

- Datagram as well as virtual circuit access must be provided in the internetwork. These are a minimal set for the construction of other higher level protocols. Arguments that suggest that only circuit access should be provided to public networks are largely political [Pouzin 76].
- A universal internetwork address space should be available, and no a priori restrictions should be made on intercommunications in this space. The exact form of the internetwork address space is not critical to us, so long as entities that wish to communicate in the internetwork are able to obtain distinct addresses.
- All messages should contain origin and destination addresses.

We will assume these features in the design of our facilities.

It is anticipated that more than one form of address will be applicable to the internetwork environment. Generic resource addresses and location-independent names are particularly useful. We expect that internetwork resources will be accessed by resource identifiers and translated by network servers to addresses in a location independent way. These higher-level addresses are supported at a level above the bare internetwork addresses, perhaps as part of capabilities used to access the remote resources. See [McQuillan 78], [Shoch 78a] and [Saltzer 78] for more discussion of such issues; specific considerations for naming designs are given in [Abraham 80], [Powell 81], [Voydock 80] and [Watson 81b].

2.1.2. Kernel IPC services

It is crucial that the operating system kernel IPC services should correspond closely to the internetwork IPC facilities. There should be datagram and circuit facilities in the IPC kernel. To the extent technologically possible, both datagrams and circuits should be supported over the entire internetwork address space. Within the context of the local system, provisions should be made for using datagrams and virtual circuits without invoking the network transport layer to increase efficiency.

To support construction of servers that multiplex activities we believe that there must be a way for a server to *select* i/o channels in need of activity from a specified set of channels. Support for the style of selection provided in ADA [Ichbiah 79] seems natural here, and is a generalization of the *await* mechanism of [Haverty 78], providing a timeout, and specifying the interesting set of i/o channels at each call instead of having the active set as part of a more global, per-process state.

There will certainly exist some servers that need to service some i/o channels with low latency. To allow such servers to be written in a natural way without polling, we believe that an interrupt-driven style of i/o access should be provided, where software interrupts are presented to the server when events occur on specified channels.

2.1.3. Data representation and abstract types

The transmission of typed data in message communication is a critical feature that must be provided to application programs [White 76] [Sproull 78] [Liskov 79] [Feldman 79]. We recognize the value that would accrue if a useful standard external data representation were in common use, but believe that many applications that will need representation transformation are likely to find that pre-defined system transformations are inadequate and incomplete. Complex mechanisms are required to handle the transportation of structures for which internal type representations differ from the external representation in

which messages are communicated. The general case where self-referential structures are to be transmitted or relationships between objects in separate messages are to be preserved is deeply involved with the semantics desired by the application language [Herlihy 80] [Nelson 80].

We agree with [Voydock 80] that work in this area is in its infancy, and this suggests that such mechanisms should not yet be made a part of the base IPC design (as in [Rashid 80]), but layered in at a level outside the IPC kernel, as in the ISO model where they are part of the Presentation layer [ISO 79] [Zimmerman 80]. Under our proposal, the data type interpretation and transmission will be the function of the language libraries provided by the applications packages, and involve use of language-specific storage management facilities and global state normally unavailable to the kernel. This approach is conservative, flexible and acceptably efficient.

2.1.4. Applications support

To make IPC available to UNIX applications in a natural way, the IPC facilities must be made part of the UNIX name space, supported by the UNIX file system naming conventions. Such facilities are part of most distributed UNIX systems proposed or implemented [Chesson 79] [Rowe 81] [Ward 80]. We believe that the system should:

* Make it possible to provide the different types of facilities available in current UNIX files using server processes.
* Allow server processes to be created on request instead of than requiring them to exist before they are needed.
* Enable applications programs to rendezvous using the UNIX file system as a simple name server.

We propose to meet these needs by providing a mechanism for a server process to place hooks in the UNIX file system name space that provide access to server processes. The *portal* mechanism that allows these service processes to be accessed is described in section 4.

2.2. Protection

Providing protection for applications operating in a distributed environment is more difficult than in a single machine. In the context of a single machine or a set of machines under a single administrative authority the control of access can be dealt with within the UNIX kernels, protecting files and access to communications using the UNIX protection mechanism.

In a rich internetwork environment resource sharing will be desirable across administrative domains, and protection mechanisms cannot (and should not) be provided for totally in the kernel. Mechanisms for control of access in such environments involve the use of encryption [Kent 76] [Needham 78] [Popek 79]. Encrypting conventional capabilities can be used to control access [Chaum 78] [Needham 79]. A standard encrypted form including address, rights and authentication bits is believed to be valuable in implementing such an environment [Watson 80a].

Our basic IPC mechanisms provide UNIX protection within the context of a single machine but no a priori protection in the internetwork. We expect to experiment with several internetwork protection facilities within the framework of our IPC mechanisms.

2.3. Systems with similar goals

At the internetwork and system facilities level, our design choices closely agree with the facilities provided by the Pilot operating system operating in the Xerox Pup environment [Redell 79]. The applications facilities attempt to allow construction of facilities such as those provided by Cocanet [Rowe 81].

We desire universal addresses for communicating entities. Pup provides such addresses to Pilot. At the IPC facilities level, we believe that the system should provide both a datagram and a stream interface. Pup and higher level protocols (such as *NetworkStream*) do this in Pilot.

The proposed facilities for UNIX and the facilities of Pilot differ in the way in which input/output multiplexing takes place. Pilot provides for several Mesa processes within a single address space. Individual tasks in this address space can block in the process of sending and receiving messages and Pilot will continue with other active tasks. Mesa monitors provide synchronization for processes in a single Pilot process [Lampson 80].

In UNIX multiple independent processes are available for concurrent programming activities. Only a single logical thread of control is provided by the system for each process context. If multiple simultaneous input/output activities are to be permitted to a single process it is necessary to provide the user with non-blocking primitives or a mechanism for determining whether a given system call will block.

A fundamental mechanism proposed for the Pilot environment is the use of remote procedure calls to access remote resources. Such facilities have been implemented in RIG [Ball 76] [Lantz 80]. If similar facilities were to be provided in UNIX we would place much of the relevant code in the language libraries (at the Presentation level of the ISO model), since much of this processing is concerned with transmission and reception of language-specific information.

At the application level, the Pilot system provides Mesa interfaces, while the UNIX system provides inter-process interfaces based on the UNIX i/o system. Closer comparisons of our proposed facilities can be made to the applications facilities of Cocanet [Rowe 81], Datakit [Chesson 79], Locus [Popek 81] or Trix [Ward 80].

Cocanet, in particular, provides both remote file access and IPC facilities, providing users with datagram, unicast (circuit) and multicast messages. The datagram facilities are accessed using names in the directory "/dgram"; similar directories exist for unicast messages (virtual circuits) and multicast messages (which send data reliably to multiple recipients). These directories allow access to name servers providing special protocols for message sending. Similar special naming directories provide access to remote file systems.

We allow such name extensions to the system to be constructed by applications programs using the *portal* facility described in section 4. Special portals supported by kernel or user processes can be used to provide IPC facilities similar to those provided by Cocanet.

2.4. Goals and non-goals for the design

We can now state a set of goals and a set of non-goals to guide our IPC design. Goals are to:

+ Provide simple and efficient inter-process communication on a single processor. The IPC primitives should be implementable on different machines in a small communications oriented module in the kernel.

+ Provide IPC primitives that will interact well with facilities available in current networks and on other systems.
+ Provide facilities for extending the single-machine UNIX programming environment to include services written using IPC facilities. In particular, it should be possible for existing naive programs to make use of new server-provided facilities without being aware that servers are involved.

We are specifically not attempting to provide a single approach for:

- The naming and accessing of services in a internetwork environment.
- The control over information access and protection of communication in the internetwork environment.
- The transmission of structured information between processes.

We believe that there are a several interesting approaches to these latter problems that merit further investigation. These approaches can be investigated within the context of our IPC facilities.

3. IPC primitives

We describe the internetwork facilities: addressing, sockets, datagrams and circuits, and the facilities for multiplexing i/o.

3.1. Addressing

We assume that the transport layer of the system provides us with an internetwork wide address space. Each message to be sent on the internetwork includes source and destination addresses. In the sequel we refer to a type *in_addr* which represents a universal internetwork address.

The internal form of a network address is not important to most applications. A few addresses will be widely known, but most will be obtained from servers. It is, of course, critical that the addresses be translatable to the forms required by the various networks. Networks based on X.25, PUP and the IP protocols use different internal formats for the addresses. It is necessary that the single representation used by UNIX be mappable to and from these different formats.

For definiteness the reader may assume that an address is in an internetwork format:

```
typedef struct in_addr {
        int     ipaddr;             /* internet address */
        int     moreprecise;        /* sub-addressing at destination */
} in_addr;
```

3.2. Sockets

Our implementation makes use of a *socket* concept for both the datagram and circuit abstractions. Sockets are the destination of all internetwork communication. If a socket is not active when communications is attempted to it the information may be discarded, or a server may be created and presented with the open socket.

The types of sockets available are represented by the type *in_proto*:

```
enum in_proto { SOCK_DG, SOCK_CALL, SOCK_VC };
```

Each socket has some amount of buffering associated with it; the different socket types buffer different objects. SOCK_DG datagram sockets have a fifo queue for datagrams. SOCK_CALL call director sockets have a queue for incoming and outgoing calls. SOCK_VC virtual circuit sockets have a queue for incoming data and logically reference a matching SOCK_VC socket where transmitted data is stored.

Active sockets in a process are referenced by small integer descriptors drawn from the same pool as UNIX file descriptors. Processes that wish to multiplex their communications activities often wish to choose from among a set of active sockets the ones that are ready for service. To specify such a set the type *in_sockets* is introduced. A variable of type *in_sockets* takes on values which are bit masks representing sets. On the VAX the *in_sockets* type could be defined by:

```
#define     NBBY        8           /* number of bits per byte */
#define     IN_SOCKETS  20

typedef struct in_sockets {
        char    bits[(INSOCKETS+NBBY-1)/NBBY];
} in_sockets;
```

Here IN_SOCKETS is the count of sockets representable in type *in_sockets*, and is currently limited by the limit on per-process descriptors to 20. We expect to increase this limit before the first release of the IPC facilities and have constructed the IPC facilities to allow the limit to be easily changed from program to program or system to system.

The C library will provide a set of standard operations on the socket-set represented in an object of type *in_sockets*. The *zerosocket* routine removes all elements from the set of sockets in an argument socket-set. The operation *setsocket* adds a socket index to the socket-set. The *getsocket* routine picks a socket from the argument socket-set and returns it as a small integer, also performing a *clrsocket* operation to remove this socket index from the socket-set. The implementation of these operations is trivial and they will not be defined here.

3.3. Datagram facilities

A datagram is a short piece of data sent to a specific address. The system does not guarantee delivery of datagrams. Datagrams may have a limit on maximum length imposed by the medium over which they are communicated. Particular networks may attempt to return datagrams in a network-specific format when datagram delivery fails.

3.3.1. Creating datagram sockets

To send a datagram both a source and a destination address are required, so that the recipient of a datagram can receive the address of the sender. This corresponds to the format of datagrams supplied by the common networks. To create a datagram socket one issues a *socket* call:

```
int s;              /* socket descriptor */
in_addr addr;       /* assigned address */
in_addr pref;       /* preferred address */

s = socket(SOCK_DG, &addr, &pref);
```

Socket calls return a UNIX "file descriptor" in *s*, and the internetwork address of the socket in *addr*. A preferred address for a *socket* call may be specified by *pref*, to be used in setting up well-known sockets. If the *pref* argument is 0, then the system chooses an address for the socket. If *pref* is specified, but unavailable (e.g. in use, not permissible on this machine, or forbidden for use by the current user) then an error is returned.

As with other UNIX system calls, an error may be returned by *socket* indicating that there are no more sockets available, or that the preferred socket is not available. The C interface provides a return value of −1 in these cases, and the external variable *errno* will be set to a characteristic error. The interface to other languages may generate exceptions in these cases.

3.3.2. Sending/receiving datagrams

To send a datagram from a socket a *send* primitive is provided. Thus:

```
int s;
in_addr dest;
char *msg; int len;

... initialize values of s, dest, ident ...
send(s, &dest, msg, len);
```

sends *len* bytes starting at *msg* to the socket specified by *dest* from the socket associated with *s*. This datagram could be received by a process with access to the *dest* socket if the process had code like:

```
int d;
in_addr source;
char msg[MAXMSG]; int len;

... initialize socket d with addr dest as above ...
len = receive(d, &source, msg, MAXMSG);
```

When a message arrives it will be placed in the buffer at *msg* with the message length returned in *len*. If the length of the datagram is greater than MAXMSG then this length will be returned but the part that would not fit in the *msg* buffer will be discarded. Thus each *receive* call returns a single datagram, and the *source* address from whence it came.

3.3.3. Properties of datagrams

Datagrams are not guaranteed to be delivered, nor are they capable of containing an arbitrary amount of information. While we expect that some networks will provide fragmentation facilities for datagrams that are too large to be directly transported in the network, there may be a maximum size on the datagrams imposed by limited buffering for reassembly or other considerations. This would be expected, for instance, if only intra-network fragmentation was provided and a long datagram had to cross a network for which it was too large. Since the fragments of the datagram have to be reassembled at the exit gateway we cannot be sure that there would be adequate space at that gateway to reconstruct an arbitrarily long datagram [Shoch 78b].

The X.25 datagram protocol allows for only 128 data bytes in a datagram [Folts 80]. The Xerox Pup specifications and the IP datagram specification suggest that messages with more than about 512 bytes of data be sent only if it is known that the message can be delivered without problems in gateways. This seems like a reasonable suggestion to make. Although we impose no a priori restrictions on the length of a datagram, we intend that a length of 512 bytes be supported as much as possible. The constraints of the X.25 datagram protocol seem to be the most difficult issue here. If UNIX systems are communicating on top of X.25 then a packet reassembly protocol could easily be used, but the case where datagrams are being sent to a system other than UNIX that doesn't include reassembly facilities is harder to address.

3.3.4. Discarding datagram sockets

Datagram socket descriptors are drawn from the same pool as file descriptors, and are duplicated as the normal file descriptors are in the UNIX *fork* system call.

A datagram socket *s* may eliminated with a

```
close(s);
```

call; the socket is deactivated at the last close. Sockets not associated with "portals" or "associations" will have their buffered data discarded when they are closed. Portals and associations are discussed in section 4.

3.4. Circuit facilities

To use a virtual circuit one first obtains a SOCK_CALL call director socket that is associated with a specific network address. Calls may be placed from and answered at this socket. Each call placed or answered yields a distinct new SOCK_VC virtual circuit socket that allows for the reliable, flow-controlled transmission of arbitrary amounts of data to and from the party at the other end of the circuit. Circuits allow specially marked *urgent* information to be sent and give out-of-band notification of the presence of urgent data. They also allow record boundaries to be marked in the stream.

3.4.1. Creating circuit sockets

So that incoming and outgoing calls may be queued, a process must have access to a call director socket to place or receive a call. A SOCK_CALL socket is created with a *socket* call:

```
int s;
in_addr addr, pref;

s = socket(SOCK_CALL, &addr, &pref);
```

The returned *s* is a "file" descriptor for a socket for establishing virtual circuits, by calling and receiving calls. When calls are placed or answered additional descriptors are obtained for the SOCK_VC virtual circuit sockets corresponding to the calls.

3.4.2. Answering calls

A call is received by doing:

```
int t;
in_addr caller;

t = answer(s, &caller);
```

This returns a descriptor for the new SOCK_VC socket for the conversation with *caller*. Several *answer* calls may be done on a single call director socket; each yields a SOCK_CALL virtual circuit socket representing a single conversation.

3.4.3. Placing calls

To place a call establishing a circuit one must first have access to a SOCK_CALL call director socket at some address. Assuming the SOCK_CALL socket exists as *s* created as in 3.4.1 above, then a call could be placed by:

```
int t;
in_addr callee;

... initialize callee ...
t = call(s, &callee);
```

After placing a call, a new descriptor is obtained corresponding to the new

SOCK_VC virtual circuit socket. If the call fails then a value of −1 is returned for *t* and the external variable *errno* is set to a characteristic error, as is usual for a C program. When the conversation with *callee* is complete, the virtual circuit socket *t* can be closed.

3.4.4. Input and output on circuits

We now describe the features of i/o on SOCK_VC virtual circuit sockets. The facilities to be supported are derived from those available in common virtual circuit protocols, and are similar to the interface described in [Gurwitz 81].

3.4.4.1. Reading and writing in stream mode

Processes can send and receive data on a circuit with the normal UNIX *read* and *write* calls. Conversations are flow controlled by the underlying mechanisms; if the sender writes data faster than the receiver can accept it, the sender will block. If the receiver reads data when none is available, it will block pending receipt of more data.

In the default stream mode, a read returns as soon as data is available and the system does not preserve any boundaries within the information stream.

3.4.4.2. Record mode

Circuits support a record mode, where each piece of data written on the circuit is considered a single record, and reads return complete records. This allows records to be read and written conveniently. The call

```
recordmode(s, 1);
```

sets a virtual circuit socket to be in record mode. A newly created virtual circuit socket is not in record mode. Record mode may be disabled by doing

```
recordmode(s, 0);
```

If you read only part of a record while in record mode because the buffer supplied to *read* or the read buffering of the socket is insufficiently large to contain the entire record, then the remainder of the record made available on successive reads. The call

```
isbetween(s);
```

returns 1 if the specified stream is at a boundary between records, or 0 if it is not.

If only the writer is in record mode, then reads will never return data across record boundaries. If only the reader is in record mode then data will normally be aggregated to requested lengths before being presented to the reader.

A record may be created from data presented in multiple *write* calls by turning record mode off, writing data as required, and turning record mode on just before the last write in the record.

3.4.4.3. Urgent data

Circuits support a notion of urgent data. A circuit can be set into urgent mode by doing

```
urgentmode(s, 1);
```

or disabled by specifying a second argument 0. Data transmitted while in urgent mode is marked, and causes the recipient of the data to process it specially. By

default, urgent data arriving on a circuit causes generation of a signal SIGURG. This signal may be ignored if urgent data is to be processed synchronously.

The set of channels with urgent data may be determined by doing

```
in_sockets whichareurg;

... initialize whichareurg to interesting sockets ...
geturgent(IN_SOCKETS, &whichareurg);
```

This selects out of the sockets in the bit-mask *whichareurg* those with pending urgent data; all other bits are cleared.

While a socket has pending urgent data the

```
moreurgent(s);
```

call will return true. When the next byte to be read is part of urgent data the predicate

```
isurgent(s);
```

will return true.

The normal way of processing urgent data is to read out records from the input until the *moreurgent* flag drops. Then the last piece of urgent data will remain in the input buffer.

A single *read* call never returns both urgent and non-urgent data; it therefore suffices to check *isurgent* before each call to *read* to determine the type of the data to be read.

3.4.5. Failure of circuits

If a permanent failure occurs in a circuit the circuit will be marked invalid. A process that attempts to read from or write to a failed circuit will be given an error indication and then sent a signal indicating a broken connection if further reads or writes are attempted. When processing circuits asynchronously a notification is sent immediately when a circuit fails; see section 3.5.3.

3.4.6. Circuits simulating pipes

A circuit can be used to simulate a pipe directly as the semantics are upward compatible; the reverse direction of the circuit will not be used, and can be disconnected to prevent accidental use. If the circuit fails, the signal sent on the next access to the circuit performs the same function as the SIGPIPE signal for pipes.

We will discuss the use of circuits to simulate other UNIX objects such as files and file systems in section 4.

3.4.7. Closing sockets and circuits

The call

```
disconnect(t);
```

reports to the other party in a call that the call is no longer needed by sending an end-of-file on the connection. The call will continue while the other party is sending, and more data can be received on *t*, but no more data may be sent.

When all copies of the descriptor *t* created in *fork* or by *dup* have been destroyed, the circuit will be shut down after allowing the write buffers to drain.

Calls pending when a call director socket closes normally cause a new server to be created to service it if the socket has a server via a *portal* or a

association; otherwise the pending calls are aborted (see section 4).

3.5. Multiplexing communication

In writing communications oriented programs it is often desirable to process information arriving from more than one source. The proposed IPC facilities provide three mechanisms for use in handling communication with more than one party: a synchronous facility based on the *select* statement, a facility for preventing i/o operations from blocking, and an asynchronous facility based on software interrupts.

The facilities proposed in this section are generally useful for UNIX and we expect they will be gradually made available for more system services and devices.

3.5.1. Synchronous processing

To support synchronous processing of information from more than one source we provide a *select* call, of the form:

```
int nsockets, nready;
in_sockets reads, writes;

nready = select(nsockets, &reads, &writes, timeout);
```

The *select* call is provided with a structure describing sockets that are interesting; *reads* for sockets where readability is interesting and *writes* for virtual circuit sockets where writability is interesting. The system examines each specified socket to see if there is an input or output operation possible on it, and returns in *reads* and *writes* a list of all such sockets. *Nsockets* gives the count of sockets representable by type *in_sockets* so that the size of the second and third arguments to *select* need not be fixed in the system, but may vary from program to program.

Either *reads* or *writes* may be specified as 0 to denote that no sockets are interesting to read or write. If no socket comes ready within *timeout* milliseconds, the *select* returns, indicating that no sockets are ready. *Timeout* may be 0 for immediate return or −1 to not return prematurely.

The name *select* is chosen from the name of the statement in the ADA language [Ichbiah 79] whose semantics are similar. The *select* statement is also similar to the *await* mechanism described in [Haverty 78], with the exception being the way that the interesting sockets are described and returned. With *await* the system keeps a list of interesting file descriptors internally, instead of having it specified at each call, and the return value is an array of integers instead of than a bit mask. *Await* does not provide the timeout facility. Library routines to simulate *await* could easily be implemented using the facilities of *select*.

An important point in the semantics of *select* is that it imposes no bias. The mechanism for selecting among sockets that can be processed is left to the user.

3.5.2. Non-blocking operations

To support servers and other processes that wish to not block in doing communications processing, a call to set a *socket* or other UNIX file descriptor into a non-blocking mode is provided:

```
nonblocking(s, 1);
```

After setting a socket non-blocking, operations that would block because of insufficient buffering on output or lack of available data on input will return a new error ENBLOCK. This is normally returned to a caller in C as a −1 return from a system call, with the global variable *errno* set to ENBLOCK.

The operation can be retried later, as *select* will report the socket ready when it becomes unconstipated.

A *call* placed on a non-blocking call director socket will immediately return a SOCK_VC virtual circuit socket descriptor, even though the call is not complete. The returned file descriptor will *select* as ready for writing when the call completes or fails to connect. At that point a *status* operation can be done on the circuit socket to determine the status of the *call*. A *timeout* may be used with the select to limit the length of time spent waiting for a call to complete.

3.5.3. Asynchronous processing

Certain applications may require that they be notified immediately whenever input/output is possible. If such asynchronous operations are required, this can be enabled by doing:

```
asynchronous(s, 1);
```

Then when input is available or output becomes possible after a blockage the process that is doing *asynchronous* processing on the socket is notified with a SIGIO signal. A *select* with a *timeout* of 0 can be used to identify the subset of the asynchronous sockets that need service.

Asynchronous can also be used in addition to *nonblocking* when placing and receiving calls. The sequence:

```
in_addr addr, dest;
int s, c;

... initialize dest in some manner ...
s = socket(SOCK_CALL, &addr, 0);
nonblocking(s, 1); asynchronous(s, 1);
c = call(s, &dest);
```

places a call on the socket *s* and immediately returns a descriptor *c* because the socket *s* is marked non-blocking. Because *s* is marked asynchronous, a SIGIO is posted when the call to *dest* succeeds or fails and the call socket *c* will appear in a *select* as ready for writing. A *status* call, described below, can be used to determine whether the call succeeded or failed.

A similar technique can be used with *answer;* if a *call* were placed to socket *s* in the example above then a SIGIO would also be generated, and the socket *s* would show as being readable, the data being the incoming call. A *answer* could be used establish connection.

SOCK_VC virtual circuit sockets marked asynchronous cause SIGIO to be sent immediately when the circuit fails.

Because of the specialized nature of *asynchronous* i/o and to avoid difficult semantic and implementation difficulties only one process may mark a socket asynchronous at a time.

3.6. Socket status

A *status* operation can be used to get the status of the socket:

```
in_status state;

status(s, &state);
```

in the following structure:

```
typedef struct in_status {
        in_proto protocol;          /* SOCK_DG, SOCK_CALL or SOCK_VC */
        in_addr source;             /* socket address */
        in_addr dest;               /* destination address, for circuits */
        in_state state;             /* state of the connection */
        struct in_water srcwm;      /* watermarks for sending */
        struct in_water rcvwm;      /* watermarks for receiving */
} in_status;
```

The *protocol* field tells the protocol the socket supports; the currently defined protocols are SOCK_DG for datagram protocols, SOCK_CALL for call director sockets where *call* and *answer* are possible, and SOCK_VC for the virtual circuit sockets resulting from *call* and *answer*. The field *addr* is the address of this socket. The field *dest* is used only for SOCK_VC sockets, where sockets obtained by *call* or *answer* report the socket they are connected to here.

The field *state* shows the state of a call in a SOCK_VC, and has the values:

IN_CALLING	Call is pending
IN_CALLFAILED	Call failed
IN_OPEN	Call has succeeded and circuit is open
IN_CLOSING	Call is closing
IN_CLOSED	Call has closed
IN_BROKEN	Call broke due to some failure

The watermark fields specify the amount of transmit and receive buffering in this socket. Each has the following structure:

```
typedef struct in_water {
        int     lowat;
        int     hiwat;
        int     timeout;
} in_water;
```

The *hiwat* watermark reflects the total amount of buffering available. The *lowat* and *timeout* are used in non-blocking input/output. On output, a non-blocking sender will receive an error when the high water mark is reached and the data is not transmissible within *timeout* milliseconds. The sender will be notified when the amount of output pending drops to the *lowat* watermark.

A receiver will be notified if *lowat* data accumulates, or if any data has accumulated and *timeout* time has elapsed.

The *lowat* and *hiwat* are in bytes, and the *timeout* is measured in milliseconds. Reasonable defaults for the various fields are set by the system. The watermarks may be set by the user by

```
in_water rdwm, wtwm;

watermarks(s, &rdwm, &wtwm);
```

where either the second or third argument may be specified as 0 to specify that

the read or write watermarks are not to be changed.

3.7. Ioctl, control and option setting

The interpretation of options for data transmissions such as priority and security classifications varies from network to network and tends to be interpreted in ways that are hard to generalize to different networks. This is akin to device control, where different devices will allow different operations. Instead of specifying all possible options with each message to be sent, which would involve complicated processing for each message, we will use per-socket state to localize most of the option setting to the socket setup phase.

UNIX currently provides an *ioctl* operation to deal with device specific control operations, and we wish to use a similar mechanism for socket option specification. The *ioctl* mechanism suffers from a lack of specification of the lengths of the control information being exchanged. We intend to define a new operation that has *ioctl's* semantics but with full parameter specification. This *control* operation will have the form

```
int f;
char *request;
char *idata; int ilen;
char *odata; int olen;
int reslen;

reslen = control(f, request, idata, ilen, odata, olen);
```

Here *f* is a UNIX file descriptor, *request* is a null-terminated string specifying the request, *idata* is a string containing input for the request of length *ilen*, and *odata* provides a place for storing the corresponding result value of maximum length *olen*. The returned *reslen* is the length of the result, which may be shorter than *olen*.

Specially interpreted in this context are 0 values for *idata* and *odata*, indicating that no input parameters are given or output results are expected respectively. To allow for the easy use of null-terminated strings in *idata*, a *ilen* of −1 will be interpreted as indicating that *idata* is a null-terminated string.

Within this framework we can define *control* operations on sockets to set options. For example:

```
control(f, "precedence", "high", −1, 0, 0);
```

could set the precedence of the circuit *f* to be high and

```
char security[32]; int slen;

slen = control(f, "security", 0, 0, security, sizeof (security));
```

might return the current security of *f* as a character string to *security*.

The *watermarks* primitive of the previous section might be implemented by:

```
watermarks(s, rdwm, wtwm)
        int s;
        in_water *rdwm, *wtwm;
{

        if (rdwm)
                control(s, "readwm", (char *)rdwm, sizeof(*rdwm), 0, 0);
        if (wtwm)
                control(s, "writewm", (char *)wtwm, sizeof(*wtwm), 0, 0);
}
```

We intend to study the appropriate standard set of *control* operations for sockets and provide suggestions for such a set at a later date.

3.8. Examples

We now give examples using the described facilities. The first is a time server program that creates an internetwork datagram socket to which a message can be sent causing a message with the time to be returned. It could be used by a small computer on a network to obtain the time of day from a central server.

```
#include <inet.h>
#include <types.h>
#include <wellknown.h>          /* defines WWV_ADDR and others */

/* tsaddr is the well-known-address of the time server */
struct    in_addr  tsaddr = WWV_ADDR;

main()
{
     char buf[1]; int len;
     in_addr addr;
     int s;
     char *ctime(), timestr;
     time_t t;

     s = socket(SOCK_DG, 0, &tsaddr);
     if (s < 0) { printf("can't get socket\n"); exit(1); }
     for (;;) {
          /*
           * We receive a datagram and discard its contents,
           * to get the address of the sender.  A more sophisticated
           * time server might handle several requests based
           * on the contents of the received datagram.
           */
          receive(s, &addr, buf, sizeof (buf));
          time(&t);                         /* get binary time */
          timestr = ctime(&t);              /* convert to string form */
          send(s, &addr, timestr, strlen(timestr));
     }
}
```

Here the *socket* call associates this process with the time server socket whose address is specified, returning −1 if there is something wrong with *ts_addr* (i.e. not providable on this machine) or if the socket is already in use (e.g. by

another instance of the time server). If the socket is openable the server loops reading a packet from the socket for the sole purpose of obtaining the address it came from and sending back the time without further ado.

The second example is that of a telnet server creating server processes (login commands) each time someone connects to the telnet socket:

```
#include <inet.h>
#include <signal.h>
#include <wellknown.h>

struct in_addr teladdr = TELNET_ADDR;

main()
{
    void reaper();
    int s = socket(SOCK_CALL, 0, &teladdr);

    if (s < 0) { printf("can't get socket\n"); exit(1); }
    sigset(SIGCHLD, reaper);
    for (;;) {
        int t = answer(s, 0);
        if (fork() == 0) {
            dup2(t, 0); dup2(0, 1); dup2(0, 2);
            close(s); close(t);
            execl("/etc/tellogin", 0);
            exit(1);
        }
        close(t);
    }
}

#include <wait.h>
/* reaper() allows all children which have died to exit */
void reaper() { while (wait3(0, WNOHANG, 0) >= 0) continue; }
```

Here the basic server *answers* to the telnet socket it created. Each time a connection is made to the virtual circuit socket a new instance of a special login server /etc/tellogin is created. When a login is complete, the child exits and the *reaper* routine is called with a signal; it collects the terminated children.

The previous program makes use of an asynchronous facility for handling process termination. A reasonable extension to UNIX would be to provide a record on a special circuit when child processes terminate. This program could then be written using *select* to service the two circuits synchronously.

Assume that a call *waitsocket* yields a socket on which messages of type *child_status* are placed when child processes terminate. The above example could be written as follows:

```
#include <inet.h>
#include <signal.h>
#include <wellknown.h>

#define  FOREVER     -1

struct in_addr teladdr = TELNET_ADDR;
in_sockets sandp, choose;

main()
{
     int s = socket(SOCK_CALL, 0, &teladdr);
     int p = waitsocket();
     int t;

     if (s < 0 || p < 0) { printf("can't get socket\n"); exit(1); }
     setsocket(&sandp, s); setsocket(&sandp, p);
     for (;;) {
          choose = sandp;
          select(IN_SOCKETS, &choose, 0, FOREVER);
          while ((i = getsocket(&choose)) >= 0) {
               if (i == p) {
                    child_status chstatus;
                    read(p, &chstatus, sizeof (chstatus));
                    continue;
               }
               t = answer(s, 0);
               if (fork() == 0) {
                    dup2(t, 0); dup2(0, 1); dup2(0, 2);
                    close(s); close(t);
                    execl("/etc/tellogin", 0);
                    exit(1);
               }
               close(t);
          }
     }
}
```

4. Application support

We now consider system facilities for provision of services to applications. UNIX application programs normally interact by using names in the UNIX file system name space to gain access to resources. The traditional UNIX file system is a hierarchical structure containing files, directories and special devices. We extend this space by allowing a new type of object, a *portal*, to be placed in the file system name space. A portal is a gateway to a server process.

Different types of portals exist corresponding to the different object types in the UNIX file system: files, devices and directories. An additional portal type provides access to symmetric virtual circuits. Portals allow rendezvous between processes to be controlled by the UNIX protection mechanisms and allow servers to build remote file systems and other resources representable in the traditional UNIX name space.

Server processes for portals are created if the portal is referenced and the server does not exist. We also describe here an *association* mechanism that allows for automatic creation of servers for the internetwork name space.

4.1. Creating servers in UNIX name space: portals

The mechanism whereby services may be created in the UNIX file system name space involves creating a bridge between the file system name space and an IPC socket called a *portal*. Portals are client/server links and as such are asymmetric. The client accessing the portal may well be unaware that the object referenced is not a traditional UNIX object; in all but the most trivial cases, the server of the portal is interpreting a protocol and is cognizant of the existence of the portal.

4.1.1. Creating portals

A portal is created by the call

```
enum { PORTAL_CALL, PORTAL_FILE, PORTAL_DEV, PORTAL_DIR; } kind;
char *name;
int mode;
char *server;
int s;

s = portal(kind, name, mode, server);
```

where *name* is the pathname for the portal, *mode* is the UNIX protection mode for *name*, and *server* is a specification for the server to be invoked when the portal is accessed. The *kind* specifies the type of portal, and thereby specifies the protocol generated by the kernel for operations by client processes on it. The *s* returned is a descriptor for a SOCK_CALL call director socket to which the kernel will place calls when *opens* are done on *name*.

The socket types are implemented by the kernel by translating system calls applied to the file descriptors returned from *opens* on a *portal* into data on the SOCK_VC sockets the server receives when it answers *calls*.

A PORTAL_CALL portal acts like a virtual circuit socket, and simply passes reads and writes onto the underlying SOCK_VC socket.

A PORTAL_FILE translates reads and writes on the underlying SOCK_VC resulting from an *open* into a record-oriented request packet to the server. The kernel expects an appropriate reply to complete the operation for the client. Operations *fstat* and *lseek* are also possible on descriptors obtained by clients by opening a PORTAL_FILE.

A PORTAL_DEV is like a PORTAL_FILE, but also allows *control* operations, the generalization of *ioctl* that was described in section 3.7. A PORTAL_DEV thus can be used to simulate a general UNIX device, such as a terminal.

A PORTAL_DIR can be used to simulate a UNIX directory, as calls such as *open*, *unlink* and *creat* are translated into appropriate protocol. A result of such a call is often another connection to a service process to provide a file interface via the PORTAL_FILE or PORTAL_DEV protocol.

We describe these protocols in more detail below.

UNIX protection modes are used to control access to the sockets associated with a portal. The call director socket for a portal is not accessible using internetwork addresses. It is therefore accessible only using a reference through the file system name space.

4.1.2. Server establishment

The service process need not exist when a portal is first referenced. If it does not, a socket is created and associated with the in-core information about the file system entry for the portal. The *server* string is taken as a path name of the server program and that server is created in the environment of the process referencing it, receiving as descriptor 0 the socket associated with the portal, inheriting the current directory and user-id of the accessing process. The server process may be set-user-id to allow it to run in a different protection domain. The server process created has as parent the process that created it but is marked to not notify the parent when it finishes execution, since the accessing process is not aware of its presence.

The portal process may service more than one request on the descriptor or exit at any time. Processes accessing a portal may wait for the server to service them much as callers wait for an *answer* to occur on a virtual circuit.

When a portal is created the *portal* call returns a descriptor for the portal. Portals thus are created *live*. If the pointer to the *server* in a portal call is 0, this portal is accessible only while it is live; the portal will be closed if the server dies. A process may thus establish a portal that it will serve and bypass the server creation mechanism.

4.2. UNIX service protocols

As mentioned above, the UNIX system defines a protocol for each portal type that the server should implement. Such protocols have been studied for a many systems; some recent examples are [Donnelley 80] [Lantz 80] [Rowe 81] [Ward 80]. Complete specification of the protocols to be supported by UNIX will not be given here. We will instead outline some simple protocols to show the flavor of such facilities.

4.2.1. PORTAL_CALL protocol

The protocol for a PORTAL_CALL is trivial: each *open* system call on the portal causes a *call* to occur to a virtual circuit socket corresponding to the portal. The server does an *answer* to establish a circuit in the normal way. The resulting descriptors may be used by both the client and the server as a normal, symmetric virtual circuit.

4.2.2. PORTAL_FILE protocol

The PORTAL_FILE protocol extends the PORTAL_CALL protocol by imposing a request/response exchange on each *read*, *write*, *lseek* or *fstat* operation on the portal. Like all portal protocols except PORTAL_CALL, this protocol is asymmetric: the portal process acts as a server and the process that opened the portal acts as a client.

A portal operating in request/response mode is placed in record mode, and each record corresponds to a request or a piece of data for a request.

A *read* request on a PORTAL_FILE generates a record containing the word "read" (as an ASCII string) and the length of data to be read in bytes. So that the information is portable, the data length is given as a printable ASCII string. Thus a *read* of 10 bytes would result in the string

read\n10\n

being presented as a record to the server process.

The server process would answer such a request with either

error\nerror-message\n

or

data\n10\n*the-data*

These responses are interpreted by the kernel to complete the call from the user.

Write, *fstat* and *lseek* define request/response pairs in a similar way.

If an error occurred in accessing the PORTAL_FILE requiring cancellation of a pending operation by a client, the kernel would resynchronize the connection by sending a urgent message to the server. The server would respond with an urgent message of its own, completing the resynchronization.

When the client closes the file the server receives an end-of-file on the circuit, in the normal way, allowing it to clean up.

4.2.3. PORTAL_DEV protocol

The PORTAL_DEV protocol extends the PORTAL_FILE protocol by allowing the operation *control* to be applied by the client. If a *control* is applied to a PORTAL_FILE by a client it is simply refused by the kernel.

4.2.4. PORTAL_DIR protocol

To build a name space that is accessible directly in the UNIX file system we can build a directory server, handling calls such as *open*, *creat*, *chdir* and *unlink* with pathnames that have as prefix the name of a portal, but may include an additional suffix that is interpreted by the portal server. A typical use of a directory portal would be to build a remote file system.

Protocols for such directory servers have been constructed in Cocanet [Rowe 81] and in Trix [Ward 80]. They are also present in link-based systems such as Demos [Baskett 77]. A directory server protocol specifies a translation of normal file system operations into messages on connections to the directory server. An access to a name in a directory server space would create a connection to the directory server and pass as parameters the identity of the user, identification of the request, the path name suffix, and parameters of the operation. The translation of the operation into a record sent on a dynamically created circuit occurs in the UNIX kernel.

The directory server responds to such a request in one of several ways, as specified by the options of the protocol. It could provide the information requested by the operation directly and thereby complete the operation. It could implement the connection implied by a file *open* by reusing the connection used to present the request, perhaps using a child process to provide the required services. It could provide the caller with an internetwork address and authentication for the caller to use in continuing the operation by interacting with another server.

The exact options to be used in the standard UNIX directory server protocol will be the result of experimenting with these and other possibilities and is beyond the scope of this paper. There are a many techniques adequate for use in the protocol; the choice of a standard protocol will be made because of convenience, efficiency and generality.

To fully support the notion of a virtual file system or other resource name space we require a slight generalization of the UNIX process state, such that the notion of a current directory is generalized from being a reference to a device or file in the file system to being a communications link. This allows the current directory to be in server supplied name space.

4.3. Creating servers in internetwork address space: associations

Recall that *portals* are not accessible using the internetwork addressing mechanisms, so that UNIX protection applies to them. It is thus necessary to provide a separate facility to allow servers to be dynamically created as a result of internetwork address space references.

The call

```
in_addr addr;
in_proto kind;
char *server;

associate(&addr, kind, server);
```

specifies that a server of type *kind* is to be provided for internetwork address *addr*; the address must be on the current machine. A reference to the address *addr* causes the specified *server* to be created and given access to the newly created socket of type *kind*, either SOCK_DG or SOCK_CALL. The created process will be run with user-id and group-id of the user who supplied the association, from the root directory of the file system, and with the system initialization process as parent. The power to create associations may be limited administratively on a particular machine. It is likely that certain internetwork addresses will be reserved to privileged user-id's, and that normal users would not be allowed to specify these addresses for associations.

An association may be removed by a

```
disassociate(&addr);
```

For a general discussion of the kinds of protocols and facilities that are appropriate for the network address space see [Abraham 80] [Donnelley 79] [Lantz 80] and [Sproull 78].

4.4. Examples

We first show a mail server utility that looks up forwarding addresses:

```
main()
{
        int p;
        char *lookup();

        unlink("forwarding");
        p = portal(PORTAL_CALL, "forwarding", 0666, 0);
        for (;;) {
                int s, len;
                char name[128]; char *addr;

                s = answer(p, 0);
                recordmode(s, 1);
                len = read(s, name, sizeof (name));
                addr = lookup(name);
                write(s, addr, strlen(addr));
                close(s);
        }
}
```

The server creates a portal named *forwarding* of virtual circuit type. If you want to look up a forwarding address you can do:

```
FILE *f = fopen("forwarding", "rw");
recordmode(fileno(f), 1);
fprintf(f, "jones\n");
fgets(f, buf);
```

We could also write a server to be created automatically instead of manually. We would create the portal using a call:

```
portal(PORTAL_CALL, "/etc/forwarding", 0666, "/etc/forwarder");
```

Then when the file /etc/forwarding is first referenced, a /etc/forwarder will be created to service it. This portal would normally be created by a shell command:

```
$ portal call /etc/forwarding /etc/forwarder
```

The server would be created with descriptor 0 referring to the portal, and would be written:

```
main()
{
        char *lookup();

        for (;;) {
                int s, len;
                char name[128]; char *addr;

                s = answer(0, 0);
                recordmode(s, 1);
                len = read(s, name, sizeof (name));
                addr = lookup(name);
                write(s, addr, strlen(addr));
                close(s);
```

```
        }
    }
```

A server could be created in internetwork space by using a *socket* instead of a portal, or automatically created on reference in internetwork address space using a *association*. Assume that an internetwork registry exists on the local network and we wish to create a service program that will be known to the registry. The following program creates an association for the server and registers it with the registry. This program could be invoked as

```
$ register servicename program
```

to register *servicename* to access *program*. We assume that the registry operates by accepting a *call* from the program followed by three records on the connection: the operation type as the first record, consisting of the word *register* for registration requests. For registrations the second record is the name to be registered, and the third record is the internetwork address.

Note: in this example we use *printf* to print error messages; in a production program we would use the C library routine *perror* that looks up an error message, and can yield more precise system characterizations of the error. We use *printf* here since the error messages in the source can help understand the program while calls to *perror* would all have the form

```
perror(x);
```

where *x* would be *s* or *t*. This is not enlightening to the code reader.

```
#include <inet.h>
#include <wellknown.h>

in_addr  registry = REGISTRY_ADDR;      /* well-known */
in_addr  serviceaddr;
char     response[128];

/*
 * register servicename program
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int s, t;
    char *servicename, *program;

    if (argc != 3) {
        printf("usage: register servicename program\n");
        exit(1);
    }
    servicename = argv[1];
    program = argv[2];
    /*
     * Get a socket to call the registry with.
     * Since both this and the socket to be registered
     * are assumed to be call director sockets we simplify
     * the program by just registering the socket we are talking on.
     */
    s = socket(SOCK_CALL, &serviceaddr, 0);
```

```
        if (s < 0) { printf("no sockets available\n"); exit(1); }
        t = call(s, &registry);
        if (t < 0) { printf("registry doesn't answer\n"); exit(1); }
        if (associate(&serviceaddr, program) < 0) {
            printf("can't associate service\n");
            exit(1);
        }
        recordmode(t, 1);
        write(t, "register", 8);
        write(t, servicename, strlen(servicename));
        write(t, &serviceaddr, sizeof (serviceaddr));
        disconnect(t);          /* so it knows we think we are done */
        if (read(t, &response, sizeof (response)) < 0) {
            printf("no response from registry\n");
            exit(1);
        }
        if (strcmp(t, "ok") != 0) {
            printf("error registering: %s\n", response);
            disassociate(&serviceaddr);
            exit(1);
        }
    }
```

We note in passing that the placement of the service name in the registry and the placement of the association of the name in the local association table would ideally be done as a single distributed atomic operation.

5. Comparisons

In this section we contrast the proposed facilities with current facilities in UNIX, examine other protocols and primitives available for IPC, look at facilities of tightly coupled networked systems, and examine the facilities of other systems that allow access to networks and network IPC facilities.

5.1. Current UNIX facilities

The version 7 UNIX systems include four facilities that may be used for interprocess communication: *pipes*, the *mpx* multiplexed file facility, *signals*, and the debugging facility *ptrace*.

5.1.1. Pipes

The *pipe* facility provides a reliable one-directional byte stream, and is much like a half of a virtual circuit, except that no facilities other than raw data transmission are possible and no record boundaries are available. Pipes are a strong facility for program composition [Kernighan 81], but as they are permitted only between processes that have common ancestry, they provide only a weak form of interprocess communication.

The virtual circuits in our proposal can be used to supplant pipes. The default semantics of a virtual circuit allows them to be used as inter-machine pipes. The error reporting and closing of a virtual circuit is upward compatible with that for a pipe. Note that PORTAL_CALL portals can be used to build generalized "named" pipes.

5.1.2. Multiplexed files

The *mpx* multiplexed file facility provides for creation of trees of UNIX file descriptors by attaching one or more file descriptors to a multiplexed file descriptor. Reads from the multiplexed file select from data available on any of the files in the multiplexed tree, returning the source channel address and the data. Opens on the multiplexed file provide a control record on the multiplexed file. A response to this connects or refuses the connection. Terminals may be incorporated into a multiplex tree and control operations on the terminal are turned into special packets allowing the multiplexing process to simulate the control operations and provide responses.

Our facility provides an analogue of the tree creation by allowing a server to *select* the set of serviceable file descriptors. The server can choose a descriptor to service first. This is more general than the multiplexor scheme of picking the descriptor to be serviced in a way that the user cannot control. The *select* facility also provides a timeout not available in the multiplexor scheme.

Connections to circuit servers in our facility would normally be answered by an *answer* call creating a new circuit. This is in contrast to the multiplexor scheme where a *attach* or *detach* follows receipt of a record on the multiplexed file.

Virtual circuits in our scheme are symmetric while multiplexor connections are asymmetric and thus resemble *portals* of type PORTAL_CALL. The generation of control records on a terminal device so that higher level emulation may be done is a useful feature of the multiplexor, provided in our scheme by using a PORTAL_DEV.

5.1.3. Signals

Signals in UNIX provide a weak interprocess communication channel. A process may send a signal to another, causing it to execute a parameterless signal handler. In the Berkeley VAX versions of UNIX signal handling has been extended so that signals may be handled reliably, processing them as software interrupts much in the style of hardware interrupts.

A different implementation of UNIX-like signals has been designed for the Series/1 distributed system [Sincoskie 80]. SODS/OS is a message based distributed system whose aims are location transparency and distributed control in a decentralized system. The objects of the system are processes and exchanges, where an exchange is a fifo queue of messages. Access to exchanges in SODS is controlled by capabilities maintained by the kernel.

SODS is interesting in that it implements several mechanisms that appear distinct in UNIX as different message classes. Thus signals and exceptions appear a special messages as do messages that interrupt a process when they arrive at its exchange. This view of these conditions is necessary to a simple model of the system, and is a good goal for evolution of UNIX mechanisms.

5.1.4. Ptrace

The process trace facility provides a weak form of interprocess communications used for debugging. The facility is unpleasant to use and the bandwidth available to the debugger is low. Casting this facility as a interprocess communications protocol would clean up this messy facility, make debugging much more efficient, and allow debugging of distributed programs, permitting the process being debugged to be on a different machine.

Development of a protocol for debugging in UNIX seems a simple and worthwhile exercise; such a protocol can easily be layered on top of the virtual circuit facility.

5.2. Other communications primitives

There exist other protocols and styles of IPC than the ones we are proposing. In this section we present some of these alternatives and comment briefly on each.

5.2.1. Reliably delivered messages

Many distributed systems services are based on the exchange of single messages to accomplish a transaction, where little state is kept at the end points. A large amount of effort and several messages must be sent to set up a connection for a single message exchange [Belsnes 76]. This suggests that either a pair of datagrams or a special protocol should be used to accomplish message exchange.

The Livermore Network Operating System includes a acknowledged message-exchange protocol as the basis for its network IPC facilities [Donnelley 79] [Fletcher 80] [Watson 80a] [Watson 80b]. This facility is built using a timer-based protocol that depends on the limited lifetime of packets in the internetwork [Sloan 79].

We do not hold with this style of acknowledgments in a system. We believe that the fundamental responsibility for consistency of system action belongs at higher levels of the system, and that a distributed transaction based system should be built from a simpler model of unacknowledged datagram facilities. Low-level acknowledgments can be used to increase performance of transport protocols, rather than being used to impart semantics to message delivery.

If a network supports the lifetime counters needed to allow timer based protocols and an implementation of these protocols is available, the facilities could be easily be added using a new socket type.

We note that protocols such as that of [Finn 79], that provide reliable fail-safe message delivery in a network capable of "resynchronization" are of theoretical interest, but feel that wide-ranging synchronization will be impractical in a large internetwork, and thus that this protocol and similar protocols will be of use only in limited contexts. Mechanisms based on them were thus not considered for inclusion in the IPC facilities.

5.2.2. X.25 fast select message exchange

The X.25 fast select facility [Folts 80] provides a message exchange, whereby a message is sent and a response or refusal returned. The initial message is piggybacked on a circuit open request and the response is piggybacked on the refusal to connect. This refusal must be acknowledged, completing a three-way handshake. The fast select facility can be used for short query-response situations.

Because of the timing constraints imposed on the response (it must occur quickly) and the limitation to 128 bytes in the messages exchanged, we do not see fast select as being nearly as useful as datagrams or circuits, and did not include fast select primitives in our IPC facilities.

5.2.3. Broadcasting and multicasting

Where local hardware admits or networks support broadcast capabilities we expect that they will be made available at a well-known address in the internetwork address space. Reliable transmission of the same message to a specified list of sites, referred to as *multiaddressing* or *multicasting* is useful in data base work [Rowe 81] and appropriate protocols can cut the traffic significantly. If protocols to do *multicasting* become more available, a system interface to multicasting facilities can be provided based on the virtual circuit facilities.

The use of multi-addressing impacts the naming and addressing considerations in the internet, especially the way in which location independence is achieved [McQuillan 78]. Our scheme is sufficiently flexible to allow various approaches to be tried.

5.2.4. Remote procedure calls

Some interprocess communications mechanisms use message exchanges to build remote procedure calls [Feldman 79] [Nelson 80] because the interprocess communication can be made to have semantics like the more familiar single processor case, although there are several difficulties in making remote procedure calls transparent [Lampson 81b]. Other researchers consider that hiding the semantics of remote invocation is not a good idea, and that servers for a distributed environment should instead be written to deal with the semantics of communication [Svobodova 79]. As an example, the occasional duplication of messages can be remedied by making the servers aware of this and constructing them to be idempotent [Liskov 79].

We consider remote procedure calls to be largely a linguistic facility, as some of the semantics of remote procedure calls cannot be hidden easily. Because of the linguistic nature of remote procedure call, the involved semantic issues, and the lack of a single clear implementation for this facility, we defer the implementation to the applications at the language level. Remote procedure calls can be built as a protocol on either the datagram or virtual circuit facility.

5.3. Other styles of IPC

We next consider different IPC styles, including the link-based style of Demos [Baskett 77], the use of capabilities in IPC [Fletcher 80] [Chaum 78], and the port-based IPC of [Rashid 80].

5.3.1. Link-based IPC

Several systems have been constructed based on the link model used in Demos [Baskett 77] [Solomon 79]. Links provide a good basis for IPC in distributed systems, but do not generalize easily for use across protection domains. For instance, in a single protection domain the system can guarantee properties of links such as "use once" and "do not duplicate", but when multiple domains are involved a single kernel cannot completely control the distribution of links, and a different scheme must be used.

Link-based IPC is an interesting paradigm for control of communication. We expect that link-based schemes can be constructed on top of our IPC by applications that do some authentication on communications and pass addresses with authenticating information as the link tokens. This is really a special case of distributed capabilities. The difficulties implicit in providing reliable messages without the overhead of explicit connections must also be dealt with [Belsnes 76]. Techniques used for implementation of remote procedure calls may prove useful [Nelson 80] [Lampson 81b].

5.3.2. Authentication in multiple protection domains

The difficulties of supplying authenticated access to objects in multiple protection domains can be overcome by using encrypted capabilities [Chaum 78] [Needham 79]. The Livermore NOS design [Watson 80a] supports a standard form of a network capability that includes a network address, resource identifier and authorization information [Fletcher 80]. Such a convention, when combined with the IPC mechanisms described in section 3, can allow convenient authentication of access to distributed resources.

We expect that standard capability forms will be developed for use by servers in an internetwork environment. Servers will allow access to their services only when presented with the authenticatable capabilities. Such services cannot be assumed to be provided by the basic IPC mechanism because authentication is both a distributed and layered problem. See [Nelson 79] for a discussion of the role of encryption in layered systems, and [Kent 76], [Needham 78] and [Popek 79] for general discussions of cryptographic protection in systems.

5.3.3. CMU ports

A UNIX IPC mechanism based on ports and typed messages has been implemented at CMU [Rashid 80]. Ports are the destination of messages and may be moved from machine to machine. The port originating a message is received with the message but it is not possible to use the origin to determine the location where the message came from. Capabilities are used to access ports, and may be passed in messages. The IPC facility also has the capacity to collect structured information to be passed from scattered location in the users address space, and to then return the structured information, although the information must be presented linearly in the target address space.

The port naming scheme is high level, since the names cannot be used to determine port location. Port names are not related to network addresses, but are manipulated and protected internally by the IPC. This makes them difficult to use in a heterogeneous environment, containing systems not running CMU

IPC.

A different approach is to build naming and protection as a layer visibly above the internetwork addressing. Our approach would be to build the names for IPC objects from more primitive addresses, making explicit the role of service identifiers and the problems of authentication in an internetwork. See [Abraham 80] [Redell 79] and [Watson 80a] for system designs with similar approaches.

The facilities for passing typed messages are asymmetric: the kernel can gather the information from but cannot scatter it back to the target address space. We believe this reflects improper layering, and feel that abstract data type transmission is the proper function of the presentation layer of the system, not the IPC kernel. In a system like UNIX where the kernel is primarily an i/o multiplexor [Thompson 76] we believe such functionality should be in the language libraries.

We expect that mechanisms similar to those of [Herlihy 80] are the ones required for a clean and efficient implementation of typed message passing. Similar facilities will be needed in Mesa run time support to support remote procedure calls [Nelson 80]. Until such facilities are well understood they should be isolated in a layer where they can be easily changed as the software technology advances.

5.4. Tightly coupled systems

There are a several tightly coupled network based operating systems relevant to the discussion of the IPC. In general, our IPC architecture views tightly coupled systems as a single machine. The problems of protection and autonomy are expected to be addressed for the tightly-coupled machines acting as a single entity.

5.4.1. Cocanet

Cocanet [Rowe 81] is a distributed UNIX system for a local network environment. It provides remote file and command execution as well as three kinds of interprocess communication: datagrams, unicasting (circuits) and multicasting for database applications. The facilities are constructed to tightly couple UNIX systems into a single computing resource.

As we noted in section 2.3 and in the section on Broadcasting and Multicasting above, the interprocess communication facilities of Cocanet are available in our scheme or easily provided. The naming implicit in interprocess communication and remote file access can be constructed using appropriate *portals* and protocols. It will be interesting to compare the complexity and efficiency of these two approaches to providing distributed UNIX services.

Cocanet does not provide any services for access to resources in the internetwork outside its domain, or for more general communications.

5.4.2. Locus

Locus [Popek 81] also provides distributed system services, and is much more tightly knit than Cocanet. While Cocanet services are provided by server processes, the sharing and multiprocessor nature of Locus is provided by low-level interchanges between the system kernels. Locus is ambitious, providing transaction based file system operations, replication, and attempts to continue to operate in the presence of system partitioning failures.

While the Locus facilities can be provided in our scheme much in the style of WFS [Swinehart 79] or DFS [Sturgis 80], we expect that differences in

complexity, style and performance characteristics of the two approaches may be non-trivial, since the facilities to be provided in Locus are complex.

In relating our IPC proposal to Locus we would consider that the single-machine case for our IPC, where the *portal* facility would be used, would encompass the set of machines accessible using the facilities of Locus. Outside this set of machines the Locus facilities would be unavailable, but similar facilities could be provided using the facilities of the IPC and directory portals, perhaps by interacting with the facilities of a non-UNIX DFS.

5.4.3. Trix

The TRIX system design [Ward 80] provides a view of an operating system where all objects can be represented by processes. An asymmetric view of communication is taken, where the relationship between processes is always that of a client and a server. Interprocess communication occurs on bi-directional streams. The TRIX design includes protocols implementing the standard system object types much as we define the protocols for the different types of portals. It uses descriptors passed through the IPC facility in a link-passing style to control access to the communications facilities.

Trix has several difficult issues to deal with because it treats all system objects as processes. In particular, consistency of the system across a crash, reachability of objects in the file system and garbage collection of unreachable objects, buffering of messages, the limited lifetimes of processes and management of removable storage volumes all seem troublesome.

These issues are not prominent in the UNIX IPC design because we are not designing an entire system based on the *portal* model. We believe, however, that the *portal* facility and the provision for automatic server creation will provide facilities nearly as flexible as those proposed for Trix, without the attendant problems.

5.4.4. Livermore NOS

The Livermore Network Operating System [Donnelley 79] [Fletcher 80] [Watson 80a] provides homogeneous network services on a heterogeneous set of machines building from a layered system model. The model used in developing this system was also used in our design, and hence our system facilities have corresponding facilities in the LLL NOS. LLL NOS uses timer-based message exchange protocols as were previously described, and is thus based on different primitives than we are assuming for our internetwork.

As we have already mentioned, LLL NOS has a universal address space for communication and also develops a standard form for a capability with authentication. Levels of kernel services provide interpretation of these capabilities.

We expect that levels of services will be provided by UNIX systems much like those of LLL NOS. Standard forms for authenticated capabilities may be desirable in the UNIX environment, corresponding to those of LLL NOS, and conventions for these forms can be easily experimented with.

5.4.5. Accent

Accent is a network operating system being constructed at CMU absed on the CMU ports described in section 5.2.3 [Rashid 81]. The IPC facility for Accent has been implemented in VAX UNIX [Rashid 80], while the full Accent implementation is supported by PERQ microcode.

In Accent the kernel provides naming, typed message transmission and authentication mechanisms. All system facilities are provided by the kernel and

server processes in a message passing style. Accent is thus a network system in the style of LLL NOS [Watson 80a], with most of the layers pre-defined by the system.

In contrast to Accent, our service naming, type conversion and internetwork authentication facilities are layered outside the basic IPC mechanisms and outside the UNIX kernel. This allows different approaches to providing these facilities to be tried. Our IPC proposal is thus more in the style of Pilot [Redell 79].

We believe that both approaches to IPC are valuable, and that several different systems with high-level facilities similar to those of Accent can be tried based on our IPC primitives.

5.5. Loosely couplable systems

In this section we examine systems and subsystems that provide IPC facilities in a less tightly coupled (usually more layered) fashion.

5.5.1. BBN UNIX TCP/IP

The BBN TCP/IP implementation for UNIX [Gurwitz 81] provides access to the facilities of the Transmission Control Protocol and Internet Protocols developed by the Internetwork Working Group [Postel 80b] [Postel 80c] for use in the ARPANET and the military AUTODIN II network. The facilities provided by the Transmission Control Protocol were the model for the virtual circuit facilities described in section 3, and therefore a clean and simple interface between the TCP and the IPC should be possible. The modest requirements of the datagram interface should be easy to provide using the IP Internet Protocol.

Using the IPC it should be possible to build and access services in the ARPANET and connected networks that speak the Internet protocol in a uniform way.

5.5.2. Pilot

The Pilot system [Redell 79] is Xerox's operating system for a personal computer in the Ethernet. As previously discussed, the communications architecture of the IPC is closely related to the structure of Pilot. In particular, we intend that the internal structure of UNIX systems that are to act as internetwork gateways will be similar to that of Pilot.

We will not repeat here the more detailed comparison with Pilot; see section 2.3.

5.5.3. Purdue ECN UNIX

A local network of computers has been constructed at Purdue University in the Electrical Engineering department based on a store-and-forward facility using the service computers as the message processors [Hwang 81a] [Hwang 81b]. The transport protocols used in this network are simple and efficient, and the throughput in the network is high. The basic primitives for accessing the network include a primitive to obtain a free network file from a pool, a primitive to wait for a connection to a specified local socket, and a primitive to connect to a specified foreign socket. These correspond in a natural way to the primitives of our proposed IPC facility.

The high-level facilities of the ECN include remote command execution, remote login, and a load-sharing command execution facility. Provision of similar facilities in an internetwork of systems using our IPC design should thus be straightforward.

6. Implementation

We now describe the way in which the various proposed facilities are (or are to be) implemented, and then give some indication of the performance of a prototype implementation compared with CMU's UNIX IPC.

6.1. Sockets

Each socket consists of a structure containing the basic parameters (related to the *status* of the socket) and some information linking it into the internetwork address space so that it can be located as necessary. In addition, there is an internal data structure used to store data queued in the socket; this data structure consists of a circular buffer in virtual memory. Data arriving to the socket is stored in the circular buffer wrapping end-around as needed. Record boundaries are given by control information included in the circular buffer.

A single socket structure and a single data buffer suffice to build a datagram socket and also build UNIX pipes. A virtual circuit socket from which one can do *call* and *answer* operations can be instantiated by a single socket structure with some additional information about waiting calls. Two sockets, one for input and one for output, each with an associated buffer build a local virtual circuit.

These structures are the basis for a efficient implementation of the IPC primitives.

6.2. Datagrams

Sending a datagram to a socket involves allocating the space needed in the socket for a header describing the message (sender and length) and copying the data into the associated buffer space. The operation is simple and fast, and receiving a datagram is equally simple.

6.3. Virtual Circuits

Sending data to a virtual circuit in the single machine case is similar to sending data to a datagram socket, except that when record mode is not in effect no record boundaries are created. The urgent data operation is supported by having urgent markers in the socket data structure much as they are kept by the TCP finite state machine [Postel 80c] [Gurwitz 81].

6.4. Select statement

The select statement begins by examining each of the sockets in the argument bit masks. If none of these sockets is ready, then the kernel marks each socket to show that it is being watched. When such a socket is ready for an operation to take place the waiting process is awakened, and continues by returning a mask of sockets that are ready to be operated on.

By using counters it is possible to avoid any work looking at the sockets waited on when the process reawakens (such as clearing out their state), and to avoid subtle wakeup-waiting problems. It is possible to arrange to examine each socket exactly once for each call to *select*.

Timeouts on *select* can be implemented using the internal UNIX *timeout* mechanism.

6.5. Asynchronous and non-blocking i/o

Asynchronous notification of change in state on a socket is provided by storing in each socket a pointer to the process that is interested in asynchronous notification. This is possible because we allow only one such process for each socket. When an event causing notification occurs, we send a signal to this process.

Sockets flagged for non-blocking i/o return an error indication when a write socket is full or a read socket empty, allowing the caller to continue.

6.6. Portals

A portal is implemented by a UNIX file of a special type. When a portal is referenced, the system looks to see if a corresponding server is active. This is easy, because if a server is active then the file will have an entry in the in-core file information referencing the socket accessed by the server.

If there is an active server, then the new reference to the portal is presented to it in the portal-protocol specified way. If there is no active server, then one is created and then presented the new reference.

Translation of system calls and the interpretation of the server protocols in the kernel is straightforward.

6.7. Associations

Associations between internetwork addresses and processes can be implemented in several different ways. A simple implementation is to associate a directory with such servers (say "/assoc"), and search it for files with specific names when sockets are addressed that are not in use. This has the disadvantage that the file system operations may be time consuming.

A plausible alternative is to have the files in that directory represent the servers, but to have a kernel table rebuilt from that directory at boot.

Server creation for associations is straightforward.

6.8. Network interfacing

A goal of the IPC architecture was to easily interface the available networks. We believe that interfaces to the ARPANET, networks based on X.25, and to local area networks running either the TCP/IP or PUP protocols will be straightforward. It should be possible to build the circuit and datagram primitives on these networks in a network-transparent way, by translating the UNIX IPC virtual circuits and datagrams into the local network protocol equivalents.

We have designed the IPC to be able to interface to non-UNIX applications running on the internetwork. The transformation of the UNIX primitives into those of the various internetworks is straightforward, with a few minor exceptions (such as the limited length of datagrams in X.25 networks.)

6.9. Performance of a prototype implementation

The following sections compare the performance of different IPC facilities. We first describe the measurement method and then present some measurements.

6.9.1. Measurement method

We wish to measure the overhead in sending messages using different IPC facilities on a single machine, on a local network and over a long-haul network. Currently available for measurement are traditional UNIX pipes, a single-machine implementation of CMUs IPC [Rashid 80], and a prototype implementation of the IPC described here, which supports pipes and datagram services on a single machine.* We are thus limited to single machine measurements.

System overheads from CPU rescheduling and system entry and exit affect all the IPC mechanisms. In these first measurements we are attempting to factor out the effect of these overheads. We hope later to measure the degree to which the different mechanisms can avoid these overheads.

To avoid introducing rescheduling time, we timed the mechanisms sending messages through the IPC back to the originating process. Thus the basic test loop consists of repeated message exchanges, represented in the pipe case by writes and reads.

The program:

```
main(argc, argv)
char **argv;
{
        char buf[2048];
        int i;
        int len = 64;
        int illegalfd = -1;
        for (i = 10000; --i > 0; ) {
                write(illegalfd, buf, len);
                read(illegalfd, buf, len);
        }
}
```

was run to determine the overhead implicit in the test program. It determined that there was 440 usec of time spent in each *for* loop iteration, not attributable to any internal mechanisms. With this knowledge, the following program:

```
main(argc, argv)
char **argv;
{
        int len = atoi(argv[1]);
        int pv[2];
        char buf[2048];
        int i;
        pipe(pv);
        for (i = 10000; --i > 0; ) {
                write(pv[1], buf, len);
                read(pv[0], buf, len);
        }
}
```

was run to gather mechanism timings. By subtracting 440 usec per loop from the time accumulated by this second program, we can determine the amount of time spent in a message send-receive pair.

* The UNIX *mpx* mechanism is not measured here, because of problems that do not permit sending of messages longer than 200 bytes. The times for *mpx* are roughly equivalent to those for traditional UNIX pipes.

6.9.2. Measurements

For the measurements presented here we used a prototype implementation of our IPC proposal, traditional UNIX pipes as implemented in the latest release of the Berkeley VAX system, and numbers reported to us by Rashid for the CMU IPC described in [Rashid 80].

Our IPC mechanism implements pipes as restricted virtual circuits, and the numbers reported for our IPC use these pipes. As the datagram, circuit and pipe facilities are nearly identical in implementation on a single machine, the prototype numbers will not differ much from those that would be measured for datagrams or circuits. For the CMU IPC we have derived numbers equivalent to our test program from the data supplied by Rashid; we hope to run a program similar to that above to obtain directly measured numbers soon.

The first measurement is of the numbers of messages that can be passed per second under the three mechanisms in the current system:

messages per second to self			
length	UCB proto	CMU IPC	old pipes
4	1234	400	588
256	1052	303	526
512	833	285	454
1024	526	222	357

From these measurements we can determine the internal overhead in the IPC mechanisms. We first assume that reads and writes take the same amount of time. We then take the time spent by the benchmark program, subtract the time spent by the program that computed the overhead per iteration, and divide by twice the iteration count to determine the overhead for a single read or write:

usec per read or write			
length	UCB proto	CMU IPC	old pipes
4	185	1030	630
256	255	1430	730
512	380	1530	880
1024	730	2030	1180

Finally, we can compute the rate at which the internal IPC mechanisms can be executed, measured in exchanges per second. This is a limiting rate and cannot be exceeded under any single-machine scenario. These numbers could be approached by providing system primitives to send and receive vectors of messages in a single system call:

max. possible exchanges per second			
length	UCB proto	CMU IPC	old pipes
4	2702	485	793
256	1960	303	684
512	1315	285	568
1024	684	222	423

In the current incarnation, the prototype UCB IPC mechanism is significantly faster than old pipes, which are faster than the CMU IPC

mechanism.

Source and object code size comparisons for the different mechanisms are given in the following table. We give two numbers for the proposed IPC: the size of the current prototype implementation and an estimate of the size of the final single-machine implementation when virtual circuit and portal support is added. To obtain the size estimate we have assumed that that addition of circuit and portal support will result in code 2 to 3 times as large as the prototype implementation.

IPC facility	source lines	object bytes
old pipes	230	600
UCB proto	689	2100
mpx	1600	6032
UCB IPC (est)	1700	5250
CMU	4107	12840

We have included the size of the current UNIX *mpx* facility in our figures. The IPC facilities proposed here will provide functionality equivalent to both *mpx* and *pipes* so that it will be possible to replace these facilities with the new IPC. We thus believe that a system with our new IPC will not be substantially larger than the current system, as the older support for these facilities can be removed.

We hope to make further tests of these mechanisms soon and report numbers for both inter-machine implementations using TCP/IP protocols and a full single-machine implementation soon.

7. Conclusions

We believe that the IPC design specified here meets the needs of a wide variety of UNIX uses and the goals we outlined for it in section 2. Specifically:

- The IPC primitives are fast in the single machine case, supporting multiprocess applications that require rapid and low overhead message passing.
- The design matches closely the facilities of available networks and will be usable with a internetwork containing heterogeneous network protocols.
- The *portal* facility makes it easy to make internetwork resources available to processes in the UNIX name space. The *association* facility makes it easy to make local resources available to remote processes.
- The autonomy of local nodes is supported by the restriction of access to *portals* to machines on the local node. External access through *associations* and internetwork addresses are subject to higher level authentication procedures in the communicating server processes.

We believe that our IPC will be applicable to applications areas we have not foreseen. Its applicability is enhanced by extensibility: new socket types can be defined for specific applications with special semantics appropriate to those applications; new portal types and protocols can be defined to extend the kinds of resources available in the UNIX name space; conventions on addressing in the internetwork can make special network resources such as broadcast or multicast hardware available.

We believe that the IPC will be simple to build and efficient enough for the most demanding applications. It should be possible to build IPC support for C programs on other UNIX and non-UNIX processors, even ones with limited address space.

We hope that other IPC facilities will be built for UNIX, both as layers atop our IPC and extending or providing replacements for layers that we have defined. In particular, at the IPC kernel layer, connectionless message-response protocols and multicast protocols deserve further investigation. Standard protocols for external data representation merit and are receiving much attention. To foster experimentation with these and other areas we have made a conscious effort to keep the proposed extensions simple, providing bases for facilities that seem worthy of further investigation instead of than building in a single, unsatisfying attempt at an answer. Several interesting experiments with additional facilities and protocols can be tried simply, and we are planning such experiments in the near future.

Acknowledgments

We would like to thank the members of the IPC working group at Berkeley for their help in discussions that led to the present proposal. Ken Birman and Michael Powell presented proposals to this working group that help us in the formative phases. Raphael Alonso, Eric Cooper, Doug Terry and David Ungar provided constructive criticism. Rob Gurwitz of BBN helped by carefully describing their approach to interfacing a TCP/IP implementation to UNIX, and also provided comments on early drafts of this paper. Rick Rashid of CMU provided benchmark programs and measurements used in preparing section 6.9.

References

[Abraham 80] Abraham, S.M. and Y.K. Dalal. Techniques for Decentralized Management of Distributed Systems. *Proc. Spring COMPCON 1980.* San Francisco, Ca. 430-437.

[Ball 76] Ball, E., J. Feldman, W. Low, R. Rashid and P. Rovner. RIG, Rochesters intelligent gateway: system overview. *IEEE Trans. on Software Engineering* SE-2,4 (Dec. 1976) 321-328.

[Baskett 77] Baskett, F., J.H. Howard and J.T. Montague. Task Communication in DEMOS. *Proc Sixth ACM Symposium on Operating Systems Principles*. Nov. 1977, 23-31.

[Belsnes 76] Belsnes, D. Single-Message Communication. *IEEE Transactions on Communication* COM-24(2), Feb. 1976, 190-194.

[Boggs 79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe. *Pup: An Internetwork Architecture*. Report CSL-79-10, XEROX Palo Alto Research Center, July 1979.

[Cerf 74] Cerf, V. and R. Kahn. A Protocol for Packet Network Interconnection. *IEEE Transactions on Communication*. COM-22(5), May 1974.

[Chaum 78] Chaum, D.L. and R.S. Fabry. Implementing Capability-Based Protection Using Encryption. University of California, Berkeley, Electronics Research Laboratory, Memorandum UCB/ERL M78, July 1978.

[Chesson 79] Chesson, G.L. Datakit Software Architecture. *Conference Record, International Conference on Communications*. June, 1979, 20.2.1-20.2.5.

[Clark 80] Clark, D.D. and L. Svobodova. Design of Distributed Systems Supporting Local Autonomy. *Proc. Spring COMPCON 1980.* San Francisco, Ca. 438-444.

[Donnelley 79] Donnelley, J. Components of a Network Operating System. *Computer Networks*. Vol. 3, 1979, 389-399.

[Feldman 79] Feldman, J.F. High Level Programming for Distributed Computing. *Comm. ACM* 22(6), June 1979, 353-368.

[Finn 79] Finn, S.G. Resynch Procedures and a Fail-Safe Network Protocol. *IEEE Transactions on Communications* COM-27(6), 840-845.

[Fletcher 80] Fletcher, J.G. and R.W. Watson. Service Support in a Network Operating System. *Proc. Spring COMPCON 1980.* 415-424.

[Folts 80] Folts, H.C. X.25 Transaction-Oriented Features – Datagram and Fast Select. *IEEE Transactions on Communications* COM-28(4), 1980, 496-500.

[Gurwitz 81] Gurwitz, R.F. Vax-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January, 1981.

[Haverty 78] Haverty, J., J. Davidson and R. Rettberg. A Standard for UNIX Interprocess Communication. BBN Report #3949, Oct. 1978.

[Herlihy 80] Herlihy, M.P. Transmitting Abstract Values in Messages. Laboratory for Computer Science, Massachusetts Institute of Technology. LCS TR-234, 1980.

[Hwang 81a] Hwang, K., B.W. Wah, and F.A. Briggs. Engineering Computer Network (ECN): A Hardwired Network of UNIX Computer Systems. *Proc. AFIPS NCC 1981.* Vol. 50, AFIPS Press, Chicago, Ill.

[Hwang 81b] Hwang, K., B.W. Wah, F.A. Briggs, G.H. Goble, W.R. Simmons and C.L. Coates. UNIX Networking and Load Balancing of Multi-Minicomputers for Distributed Processing. Manuscript, Electrical Engineering Department, Purdue University, W. Lafayette, In. April 1981

[Ichbiah 79] Ichbiah, J.D., J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, B.A. Wichmann. Rationale for the Design of the Ada Programming Language. *Sigplan Notices* 14(6), June 1979, Part B.

[ISO 79] International Organization for Standardization. Reference Model of Open Systems Interconnection, ISO/TC97/SC16 N227. Aug. 1979.

[Kent 76] Kent, S.T. Encryption-Based Protection Mechanisms for Interactive User-Computer Communication. Laboratory for Computer Science, Massachusetts Institute of Technology. MIT/LCS/TR-162, May 1976.

[Kernighan 81] Kernighan, B.W. and J.R. Mashey The Unix Programming Environment. *Computer* 14(4), 1981, 12-24.

[Kimbleton 76] Kimbleton, S.R. and R.L. Mandell. A Perspective on network operating systems. *Proc. AFIPS NCC, 1976.* 551-559.

[Lampson 80] Lampson, B.W. and D.D. Reddell. Experience with processes and monitors in Mesa. Comm. ACM. 23(2), Feb. 1980, 105-117.

[Lampson 81a] Lampson, B.W., editor. *Distributed Systems Architecture and Implementation: An Advanced Course.* Springer-Verlag, New York. To appear, 1981.

[Lampson 81b] Lampson, B.W. Remote Procedure Calls. In [Lampson 81a].

[Lantz 80] Lantz, K. Uniform Interfaces For Distributed Systems. TR63, Department of Computer Science, University of Rochester. Rochester, NY. May, 1980.

[Liskov 79] Liskov, B. Primitives for Distributed Computing. *Proc. Seventh Symposium on Operating Systems Principles.* Dec. 1979, Pacific Grove, Ca. 33-42.

[Lyons 80] Lyons, R.E. A Total AUTODIN System Architecture. *IEEE Transactions on Communications.* COM-28(9), 1980, 1467-1481.

[McQuillan 78] McQuillan, J.M. Enhanced Message Addressing Capabilities for Computer Networks. Proc. IEEE 66(11), 1517-1526.

[Needham 78] Needham, R.M. and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Comm. ACM.* 21(12), Dec. 1978, 993-998.

[Needham 79] Needham, R. Adding Capability Access to Conventional File Servers. *Operating Systems Review* 13(1), Jan. 1979, 3-4.

[Nelson 79] Nelson, J. Implementations of Encryption in an "Open Systems" Architecture. *Proceedings of the Computer Networking Symposium, 1979.* 198-205.

[Nelson 80] Nelson, B. Remote Procedure Call: A Thesis Proposal. Department of Computer Science, Carnegie-Mellon University. April, 1980.

[Osterweil 81] Osterweil, L. Software Environment Research: Directions for the Next Five Years. *Computer* 14(4), 1981, 35-43.

[Popek 79] Popek, G.J. and C.S. Kline. Encryption and Secure Computer Networks. *Computing Surveys* 11(4), Dec. 1979. 331-356.

[Popek 81] Popek. G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. Draft paper dated January 22, 1981. Computer Science Department, UCLA.

[Postel 80a] Postel, J. Internetwork Protocol Approaches. *IEEE Transactions on Communications* COM-28(4), 1980, 604-611.

[Postel 80b] Postel, J. DoD Standard Internet Protocol, RFC 760, IEN 128. USC Information Sciences Institute, Jan. 1980.

[Postel 80c] Postel, J. DoD Standard Transmission Control Protocol, RFC 761, IEN 129. USC Information Sciences Institute, Jan. 1980.

[Pouzin 76] Pouzin, L. Virtual circuits vs. datagrams – Technical and political problems. *Proc. AFIPS NCC 1976.* 483-494.

[Powell 81] Powell, M. Laissez Faire Naming for Distributed Systems. Computer Science, U.C. Berkeley. Submitted for Publication.

[Rashid 80] Rashid, R.F. An Inter-Process Communication Facility for UNIX. Department of Computer Science, Carnegie-Mellon University. Feb. 1980.

[Rashid 81] Rashid, R.F. and G.G. Robertson. Accent: A communications oriented network operating system kernel. Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Redell 79] Redell, D.D., Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, S.C. Purcell. Pilot: An operating system for a personal computer. *Comm. ACM* 23(2), Feb. 1980, 81-92.

[Rowe 81] Rowe, L.A. and K.P. Birman. A Local Network based on the UNIX Operating System. To appear: *IEEE Transactions on Software Engineering* 1981.

[Rybczynski 80] Rybczynski, A. X.25 Interface and End-to-End Virtual Circuit Service Characteristics. *IEEE Transactions on Communications* COM-28(4), 1980, 500-510.

[Saltzer 78] Saltzer, J. Naming and Binding of Objects. In *Operating Systems: An Advanced Course*. Springer-Verlag, New York, 1978.

[Shoch 78a] Shoch, J. Inter-Network Naming, Addressing and Routing. *Proc. Fall COMPCON 1978.* 72-79.

[Shoch 78b] Shoch, J. Inter-Network Fragmentation and the TCP. *Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks* Aug, 1978, 161-168.

[Sincoskie 80] Sincoskie, W.D. and D.J. Farber. SODS/OS: A Distributed Operating System for the IBM SERIES/1. *Operating Systems Review* 14(3), July, 1980, 46-54.

[Sloan 79] Sloan, L.J. Limiting the Lifetime of Packets in Computer Networks. *Computer Networks* Vol. 3, 435-445.

[Solomon 79] Solomon, M.H., and R.A. Finkel. The Roscoe Distributed Operating System. *Proc. Seventh Symposium on Operating Systems Principles* Dec. 1979, 108-114.

[Sproull 78] Sproull, R.F. High-Level Protocols. *Proc. IEEE* 66(11), Nov. 1978, 1371-1386.

[Sturgis 80] Sturgis, H., J. Mitchell and J. Israel Issues in the Design and Use of a Distributed File System. *Operating Systems Review* 14(3), July 1980, 55-69.

[Svobodova 79] Svobodova, L., B. Liskov and D. Clark. Distributed Computer Systems: Structure and Semantics. Laboratory for Computer Science, Massachusetts Institute of Technology. MIT/LCS/TR-215, March, 1979.

[Swinehart 79] Swinehart, D., McDaniel, G., Boggs, D. WFS: A Simple Shared File System for a Distributed Environment. *Operating Systems Review* 13(5), Dec. 1979, 9-17.

[Thompson 78] Thompson, K. UNIX Time-Sharing System: UNIX Implementation. *Bell Sys. Tech. J.* 57(6), 1931-1946.

[Voydock 80] Voydock, V.L. Features of Network Interprocess Communication Protocols. BBN Report 4489. September, 1980.

[Ward 80] Ward, S.A. Trix: A Network-Oriented Operating System. *Proc. Spring COMPCON 1980.* 344-349.

[Watson 80a] Watson, R.W. and J.G. Fletcher. An Architecture for Support of Network Operating System Services. *Computer Networks* Vol 4, 33-49.

[Watson 80b] Watson, R.W. Network Architecture Design for Back-End Storage Networks. *IEEE Computer* Feb. 1980, 32-48.

[Watson 81a] Watson, R.W. Distributed system architecture model. In *Distributed Systems: Architecture and Implementation: An Advanced Course.* B. W. Lampson, ed. Springer-Verlag, New York. To appear, 1981.

[Watson 81b] Watson, R.W. Identifiers (naming) in distributed systems. In *Distributed Systems: Architecture and Implementation: An Advanced Course.* B. W. Lampson, ed. Springer-Verlag, New York. To appear, 1981.

[White 76] White, J.E. A high-level framework for network-based resource sharing. *Proc. AFIPS NCC 1976.* 561-570.

[Zimmerman 80] Zimmerman, H. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications* COM-28(4), 1980, 425-432.

Appendix A: Layered models

This appendix gives short summaries of the various layered models mentioned in section 2.1.

A.1. Arpanet protocols: the IP family

IP is a datagram protocol for use in interconnected networks [Cerf 74] [Postel 80b] [Postel 80c]. It interfaces to local networks with a local network protocol, and the local networks interconnect through gateways, which are hosts on two or more networks. On top of this internetwork datagram model the Transmission Control Protocol (TCP) provides a logical connection service. At higher levels are function oriented protocols such as the File Transfer Protocol, FTP. These levels are summarized in the following table:

Level	Protocols
Application	FTP, TELNET ...
Connection	TCP, RTP, ...
Internetwork	IP
Local network	LNPs

The internetwork protocols are used in the Arpanet, in the military Autodin II [Lyons 80], and in commercial local area networks and network front ends.

A.2. Xerox Pup Architecture

The Pup Internetwork Architecture [Boggs 79] provides a common packet format for communication and at least 5 levels of protocols. The levels are:

4 Applications
3 Data structure/process interaction
2 IPC primitives and protocols
1 Internetwork datagrams
0 Packet transport mechanisms

Pup builds on a level of packet transporters, such as the Ethernet or packet radio at level 0. At level 1 is a internetwork datagram facility, described by a packet format, a hierarchical addressing scheme and an internetwork routing algorithm. This level in the scheme has only one implementation and unifies all possible implementation at the other levels. Level 2 abstracts IPC mechanisms and protocols. Level 3 adds structure to the data moved at level 2 and also specifies interaction between processes; this level consists of function-oriented protocols. Above level 3 are individual application protocols, and further levels may be defined by the applications themselves.

A.3. ISO Open Systems Model

The ISO Open Systems Interconnection Model [ISO 79] [Zimmerman 79] consists of seven layers. The layers are numbered from 1 to 7, with the hardware facilities at level 1 the applications facilities at level 7. The levels are:

7 Application
6 Presentation
5 Session
4 Transport
3 Network
2 Link
1 Physical

The application layer provides the facilities for the information processing activities such as file and facilities access. The presentation layer supports the applications by providing information transformations such as those required to account for heterogeneous machines. The session layer supports the dialogue between processes by providing facilities for information exchange. The transport layer provides end-to-end control by providing a network independent interface to transport services users. The network layer provides the functions for intra-network operations such as addressing and routing. The link layer provides for the reliable interchange of data between equipment connected at the physical layer. The physical layer specifies the physical, electrical, functional and procedural characteristics to connect, maintain and disconnect the physical circuits between equipment.

A4. Network Operating System Model

A layered model for a network operating system has been constructed by [Watson 81a]. The model consists of four basic layers, building on a concept of system objects:

1 Hardware/firmware
2 System kernel
3 System services
4 Applications

The hardware/firmware layer provided components such as processors memories and terminals. The kernel layer provides for device multiplexing, basic protection and security mechanisms and interprocess communication. The system services layer provides resource allocation and multiplexing of processes, memory and other primitives. The final layer is implemented by and for applications.