Distributed I/O using an Object-based Protocol

David R. Cheriton
Department of Computer Science
University of British Columbia
Vancouver, B. C., CANADA V6T 1W5

Technical Report 81-1
January 1981

Abstract

The design of a distributed I/O system is described. The system is distributed in being implemented by server processes, client processes and a message communication mechanism between them. Data transfer between processes is achieved using a "connectionless" object-based protocol. The concept of file is generalized to that of a view of an object or activity managed by a server. This allows many objects, including application-defined objects, to be viewed or accessed within the program I/O paradigm. Files are instantiated as file instance objects to allow access to the associated data. Conventional byte-stream program input/output facilities are supported by a subroutine library which makes the message-based implementation transparent to applications.

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

Distributed I/O using an Object-based Protocol

Keywords and Phrases: distributed system, byte-stream, interprocess communication, message-based, object-based protocol

CR Categories: 4.35, 3.80

1. Introduction

Message-based systems and byte-stream program input and output have both been used extensively. The design of an I/O system that implements byte-stream I/O in terms of messages is described with examples of its use. In justifying the design, we document the issues encountered and our solutions. Some of these issues arose due to the goals we chose and some arose due to our choice of execution model - particularly the semantics of the message primitives. It is therefore helpful to first describe the progression that lead to this work.

A research group at UBC have been experimenting with distributed system structuring principles and techniques in a research system known locally as Verex [5,15]. Verex, as a direct descendant of Thoth [4,6], provides many, small (inexpensive) processes communicating via messages plus a data transfer facility. It was necessary to invent protocols for transferring data between processes; simply sending a message was not adequate in several cases. The receiver may be requesting the data from the sender so some request conventions are required. It is also difficult, if not impossible, for the receiver to accept arbitrarily large amounts of data whether or not it was requested. Given that the data constituting one logical unit cannot be transmitted in one indivisible message, problems of consistency, packetizing, and the general communication protocol problems arise. There was sufficient uniformity in the data transfer requirements to warrant developing one common protocol for the system.

A common protocol was further motivated by the need to dynamically interconnect processes to form cooperating program activities similar to the UNIX **shell** [18] program constructions. In UNIX, a shell program may be constructed from component programs joined together by byte streams called pipes.

A somewhat orthogonal motivation came from the prevalent use of byte-stream I/O as a useful programming abstraction. We have considerable experience working with byte-stream I/O and a large collection of programs using it. The byte is also an efficient, portable data unit for both applications and hardware. Despite the availability of messages at the application level in our research system, device-independent byte-stream I/O has remained the primary logical interface between a program and its external environment. Our data transfer requirements were compatible with this program I/O model.

These considerations inspired the goal of developing a common data transfer protocol between processes that would also support a conventional program I/O model.

Several considerations further refined our goals. First, a prime goal of our experimental system design has been to explore designs that minimize the use of connections. A connection is a an association between two or more processes similar to links in DEMOS [1] or virtual circuits in X.25 [7]. Connections impose an overhead in both space and execution time. They also present conceptual and implementation problems in systems of autonomous nodes where, for example, failure and possible recovery at one end of a connection requires sophisticated exception handling.

Second, our message primitives are oriented to supporting the remote procedure call (RPC) model of communication. Each process has a single message port with which is associated a message queue, a message buffer and a global identifier. The operation of sending a message appears to the sending process as a remote procedure call; the sender is blocked until the message has been received and replied to by the receiving process, corresponding to the suspension of the caller during a procedure invocation. These semantics are simple and efficient to implement. They also allow the system to be structured using the familiar procedure call model of program decomposition. However, concurrency must be achieved by using multiple processes, not by multiple outstanding messages sent by the same process. This contrasts with most message systems in which non-blocking sending of messages is central to the design. Receivers of messages can exploit message queuing, non-serial receiving and replying to messages as well as the ability to forward a message to another process. Providing the full power of message-passing in the receiver's view allows processes acting as servers to provide sophisticated scheduling of activities in response to messages received. Message communication in this model is illustrated in **Figure 1.** The major primitives in our execution model are described in greater detail in **Appendix A.**

In line with these considerations, we have developed an object-based protocol that is connectionless because it consists of idempotent operations on objects called file instances. Server processes implement the file instance objects on which clients request operations via messages. The operations are defined with procedural semantics. Reclamation on failure is handled by loosely tying the existence of a file instance to the existence of its creator/owner. Each server implements a simple garbage collection scheme to reclaim file instances when it detects the owner of an instance no longer exists.

This object-based approach suggested detaching the semantics of the object - file instance - from its implementation. We have thus generalized the concept of file to that of a view of some data associated with an object or activity of interest. There is no fundamental association of storage with a file, although that is a possible attribute. In many cases we are dealing with dynamically created, often application-defined objects or activities. For example, our mail system provides access to the inter-user mail messages as file instances, allowing the mail system to use the I/O protocol and providing access to the mail facility from the I/O system.

Another design goal was to allow for economy and flexibility in server design. The disparate range of objects and activities that we wished to view as files and their corresponding servers varied sufficiently in properties that "lowest common denominator" semantics were inadequate for applications and prevented full exploitation of the servers. For example, it is necessary for uniformity that either all servers support random access to file instance data or that none does. Consequently, we identified a set of type attributes associated with a file instance that further determine the semantics of operations on this object. We found these type attributes necessary in implementing a subroutine library (or package) that provided server-independent (conventional) byte-stream I/O at the application level.

The next section defines and discusses object-based protocols in the general context; **Section 3** describes the I/O system, including our concept of file and file instance, the protocol and one program environment supported by this system. **Section 4** discusses the design rationale; **Section 5** gives examples of I/O servers we have implemented. We close with a discussion unresolved problems and plans for future research.

2. Object-based Protocols

An object-based protocol (OBP), is a protocol between communicating processes defined in terms of an object - implemented by one of the processes called the server - and operations on the object that the remaining processes, called clients, may request of the server. The client sends a request message to the server for a particular operation on the object and the server returns a reply message after performing the operation.

These protocols differ from other protocols primarily in being defined in terms of an abstract object. The state transitions of the protocol are the state transition of the object; the protocol operations are the operations allowed on the object; and the protection and access control is that associated with the object. Object-based protocols are also asymmetric; all operations are requested or initiated by the clients. The object is passive (consistent with the technical use of the term object [12]). That is, the server only communicates with a client in reply to an operation request. No signals [18], emergency messages [12] or other spontaneous communication are used by the server. Finally, every request message generates a reply from the server for which the client is expected to wait before issuing a further request.

Object-based protocols support two levels of communication: client to server and client to client (indirectly via the server and object). This is illustrated in **Figure 2.** We are only concerned here with the client-server communication; the second level is client-defined at a semantic level similar to two processes communicating via an intermediary file in a conventional system.

Object-based protocols are of interest for several reasons. The object model applied to protocols provides insight and design methodology similar to its application to programming languages such as CLU [14] and operating systems such as Cal-TSS [11] and Hydra [21]. For example, one is disciplined to identify a well-specified abstract object with implementation-independent operations. Conversely, object-based protocols arise naturally in implementing an object-based system that uses messages for inter-module communication instead of procedure calls.

Object-based protocols are also compatible with the RPC model of interprocess communication. Their asymmetry is similar to the asymmetry existing between the procedure caller and the activation of a procedure. The client acts as the caller and the server acts as the procedure invocation. These properties allow a client process to be written in the procedural programming model while (transparently) using messages to communicate with a server process. The message communication then allows servers and clients to be distributed over several machines or systems.

An object-based protocol is end-to-end (between the client and server) because all operations requested by a client are received by, and semantically meaningful to, the server. This contrasts with protocols in which interaction is defined in terms of an intermediate entity such as the DCE in X.25 [7].

Object-based protocols are good candidates for lightweight protocols. Lightweight protocols are protocols that can be implemented with low overhead in processing, storage and lower-level communication support. They are of interest for environments providing high-speed, low delay, low error rate communication as is characteristic of local-area networks and message-oriented operating system kernels. In these environments, longhaul (heavyweight) protocols such as X.25 [7] reduce throughput and increase cost by the unnecessary overhead they introduce on the communication link and network nodes. Several protocols have been developed with this view [2,8].

We have experimented with object-based protocols that, to minimize overhead, do not implement connections or virtual circuits between the client and server. Instead, as end-to-end protocols, they exploit end-level semantics and the "perfect" knowledge the server has of the state of the object used in the protocol. Several techniques to handle the synchronization, access control, sequencing, flow control, errors and failure notification usually handled by the virtual circuit are described below.

State is maintained as the state of the object. Because the object is implemented by the server, the integrity of the object is easily guaranteed. Because interaction is confined to the RPC model, synchronization is trivial. The server sends a positive acknowledgement to each successful operation. The client waits for the acknowledgement before updating its local record of the state, if it chooses to maintain such a state.

Access to the object is controlled entirely by the server according to protection information it associates with the object. The permission may be determined from an access control list associated with the object or according to a capability offered by the client. Any client is allowed to access the object if it has the requisite permission. No "open" or "connect" operations are required. Thus, protection is not required or assumed provided by the communication subsystem.

The naming of objects used in the protocol is end-to-end; there is no name mapping or conversion in the protocol. The server provides a name or identifier that is used by clients to identify the object. Although the same name is used by all clients, this does not necessarily imply a global naming scheme for objects. Object identifiers may be unique and meaningful only relative to the object server.

There is no notification on client failure or communication failure. Servers reclaim resources by garbage collection using criteria for reclamation having end-level semantics. We have avoided providing general exception notifications due to the overhead and complexity in the semantics, implementation and use.

Reliability is achieved by checksumming and idempotency. Standard redundancy techniques such as checksums can be used to detect corrupted messages, which are then discarded. Requests are retransmitted when the client does not receive a correct reply within a time-out period. This can cause the server to receive duplicate messages which are handled by the idempotency of the protocol. That is, the same operation issued multiple times has the same result as the operation being issued once so there is no need to filter out duplicate messages. The communication subsystem is not required to provide perfect communication, only best efforts data transport [2].

The remainder of the paper describes a distributed I/O system implemented in terms of an object-based protocol. This is presented both as an example of using OBP's as well as a novel realization of an I/O system based on message communication. d

3. I/O System Description

The I/O system provides a uniform means of data transfer between processes allowing simple interconnection of files, programs, services and resources. The I/O system is defined in terms of files and file instance objects.

3.1 Files and File Instances

A file is a collection of data viewed as a sequence of variable-size records or blocks. This data is defined as the data associated with an object, activity or resource in the system. For example, an object in a storage system can be viewed as a file yielding the conventional file system model. Less conventionally, data describing the current state of a system can be viewed as a file even though the data may not be physically stored as such and may be rapidly changing. The same data may be viewed as different files, differing in some property of the view such as the ordering of the records. For example, the data stored on disk can be viewed in the block units and order defined by the physical sectors of the device, as the logical records and order defined by a logical disk file in which these sectors are contained; and as a text file in which the blocks are defined by the line delimiters. This is illustrated in **Figure 3**. Thus, the way data is viewed is separated from its underlying representation.

Files are conceptual entities. In order to access data with a particular file view, the file must be instantiated. A file instance is an object that represents a version or snapshot (or "instantiation") of a file. For example, a file instance of a storage system object represents a snapshot of the data associated with the object, similar to the versions defined by Reed [17]. A file instance of an X.25 logical channel is a (connected) virtual circuit. A file instance of data describing current system activity is a (consistent) snapshot of the data at the time the instance was created.

We have identified four logical usage modes for file instances as follows:

READ	data associated with the file instance is read but not changed.
CREATE	A new set of data is to be created by the client and associated with the file instance, discarding any previously associated data.
APPEND	Data is appended to the current sequence of data blocks. Data previously associated with the file remains unchanged.

MODIFY	Existing data is to be modified and possibly appended to.

The usage mode is specified when a file instance is created, allowing economy of implementation and immediate detection of incorrect usage. Usage modes could otherwise be ignored except for the side-effect that CREATE has of discarding the previous data sequence. That is, given a special operation for discarding the file instance data, the usage modes could be omitted from the design and usage could be deduced from the operations performed on the file instance.

A file instance is further defined in terms of the operations on it as an object.

CREATE_INSTANCE(file specification)

QUERY_INSTANCE(instance)

RELEASE_INSTANCE(instance, mode)

READ_INSTANCE(instance, block_number, buffer, bytes)

WRITE_INSTANCE(instance, block_number, buffer, bytes)

SET_INSTANCE_OWNER(instance, new_owner)

Table 1. File Instance Operations

* CREATE_INSTANCE creates a file instance according to the file specification. For a disk-based storage system, the file specification identifies an object stored on disk. For a virtual terminal [13], the file specification may describe an area of the terminal screen or a logical input source. When a file instance is created, an instance identifier is returned along with information describing maximum block size, last block number (written), bytes in last block, type and next block number (to read). The file instance is initially owned by the client that created it.

* QUERY_INSTANCE returns the same information as returned by the CREATE_INSTANCE. This allows a client that is passed the file instance identifier to get sufficient information about the instance to use it.

* RELEASE_INSTANCE invalidates the instance identifier, releases resources dedicated to the instance and performs a file-dependent function with the file instance data depending on the release mode. For example, with a printer spool file instance, the data is printed providing the mode is zero, otherwise the data is discarded. With a updated transaction file, the file instance atomically replaces the "real" file if the release mode is zero. File instances may be released when the creator of the instance no longer exists or after a prolonged period of inactivity on the instance in the case of, for example, instances representing connections to public data networks.

* READ_INSTANCE transfers the specified number of bytes to the client's buffer from the file instance starting at the specified block.

* WRITE_INSTANCE transfers the specified number of bytes from the client's buffer to the file instance starting at the specified block.

* SET_INSTANCE_OWNER sets the owner of a file instance to the specified new owner. This is used for transferring a file instance representing a user login device such as a incoming network call from the login handler process to the user's command interpreter program. It is also used by the command interpreter to transfer ownership of the standard input and output file instances of a program it is executing to that program.

Although these operations are defined for all file instances, it is not possible to provide the same semantics for them on all file instances without some restrictions. For example, a communication line is logically a stream of blocks; random access to not meaningful to support. Consequently, the operations that may be performed on the file instance as well as the semantics of these operations are indicated by the file instance <u>type</u>. The type is specified as a combination of the following attributes.

READABLE	READ_INSTANCE operations are allowed.
WRITEABLE	WRITE_INSTANCE operations are allowed.
APPEND_ONLY	WRITE_INSTANCE operations are only effective to bytes in the file instance beyond the current last byte.
STREAM	All reading and writing is strictly sequential. Each READ_INSTANCE operation must specify the same block number as that specified as the next block number returned by the CREATE_INSTANCE and QUERY_INSTANCE operations. This next block to read is incremented after each READ_INSTANCE operation. Similarly, each WRITE_INSTANCE operation must specify a block number one greater than the last block number, which is incremented after every write operation. A file instance without the STREAM attribute must have its associated data stored to allow non-sequential access.
FIXED_LENGTH	The associated sequence of data blocks is fixed in length. The length is specified by the last block and last byte returned from a create or query instance operations. Otherwise the file instance grows to accomodate the data written or the length of the file instance is not known as in the case of terminal input.
VARIABLE_BLOCK	Blocks shorter than the full block size may be returned in response to read operations other than

	due to end-of-file or other exception conditions. For example, input frames from a communication line may differ in length under normal conditions. With a file instance that is VARIABLE_BLOCK and WRITEABLE, blocks that are written with less than a full block size number of bytes return exactly the amount written when read subsequently.
MULTI_BLOCK	Read and write operations are allowed that specify a number of bytes larger than the block size.
INTERACTIVE	The file instance is a text line-oriented input stream on which a prompt can be specified and user-generated breaks can be received. It also has the connotation of supplying interactively (human) generated input.

Not all of the possible combinations of attributes yield a useful file type. The file instance type is dependent on the server, file specification and the usage mode. For example, the storage system provides file instances with type attributes READABLE, FIXED_LENGTH and MULTI_BLOCK in response to a CREATE_INSTANCE operation specifying READ usage mode. File instances of X.25 virtual circuits have type attributes READABLE, WRITEABLE, VARIABLE_BLOCK and STREAM when created with CREATE usage mode (the only mode supported). This example illustrates that a single file instance can support both an input and an output stream.

File instance types reflect differences between file instances that are apparent in the semantics of the file instance operations. They do not cover higher-level semantics such the meaning of writing to a particular file instance. One of the objectives of our on-going research is to investigate the limitations of our current type attributes and possible extension or modifications to these attributes.

Our concept of file differs from the conventional one in that a file is purely a conceptual object, not a "real" object. File instances are the real objects. For instance, the type information is associated with a file instance, not the file. File instances differ from what are conventionally called "open files" in several ways. There is no connection between the user of a file instance and the file instance that persists between operations on the file instance. Consequently, there is no user-dependent state such as read/write pointer maintained in the file instance; only the state of the instance is maintained. Also, no concurrency control is provided between users of file instances. Finally, the implicit creation associated with some open files such as spool files, dial-up lines and X.25 circuits is made explicit in file instance creation.

3.2 I/O Servers

An I/O server is a process that implements file instances for client processes. An I/O server provides remote file access to data to which it has access. For example, the storage server provides file instances of the objects it is storing. A file instance created by the storage server for CREATE usage has type attributes READABLE, WRITEABLE and MULTI_BLOCK. It atomically updates the underlying storage object when the file instance is released. The printer server implements file instances similar to "open printer spool" files. Only CREATE usage mode is supported, with type attributes READABLE, WRITEABLE and MULTI_BLOCK. It makes use of the storage server to store the data for printing.

Any process can act as an I/O server if it provides the file instance operations in response to request messages from clients. This allows applications to introduce new I/O servers to implement new types of files as well as filter access to system-implemented files. Servers may (and typically do) provide operations beyond the file instance operations. In general, a server provides the file instance operations to support the clients viewing its objects or activities as files. This may be secondary to the real function of the server.

Acting as an I/O server does not preclude a process from also being a client to other servers although the graph of client/server dependency must be acyclic to avoid deadlock. The dependency between some of our current I/O servers is illustrated in **Figure 4**. An arrow from one server to another indicates that the first server uses a service provided by the second server.

The choice of how each server implements its file instances, the protection imposed and the additional operations provided are decisions local to the design of the server. However, the I/O servers we have implemented conform to the following general structure (given in a C-like language[9]).

```
IO_server()

        \ Base function executed by an I/O server.
  {

     state = Initialize();

     repeat
       {
         id = Receive( message );

         select( OPERATION[message] )
           {
             case CREATE_INSTANCE:
                     reply = Create_instance( state, message, id );
             case RELEASE_INSTANCE:
                     reply = Release_instance( state, message, id );
             case READ_INSTANCE:
                     reply = Read_instance( state, message, id );
             case WRITE_INSTANCE:
                     reply = Write_instance( state, message, id );
             case SET_INSTANCE_OWNER:
                     reply = Set_instance_owner( state, message, id );

             ( other requests handled)

             default: reply = ILLEGAL_REQUEST;
           }
         if( reply == NO_REPLY ) next;
         REPLY_CODE[message] = reply;
         Reply( message, id );
       }
  }
```

General Form of I/O Servers

3.3 I/O Protocol

The I/O protocol is an object-based protocol in which the objects are file instances. Most of the properties of this protocol were covered in the general discussion of object-based protocols in **Section 2.** The protocol consists of a request message format for each file instance operation plus the format of the reply message received in response to each of these request messages. The semantics of these request and reply messages follow from the semantics of file instances and file instance operations described previously. The request and reply messages for reading and writing instances are described in greater detail below.

The format for a read or write instance request message is

request ::=

operation code	- the operation requested
file instance id	- server-generated instance identifier
block number	- starting block to read or write
bytes	- number of bytes to read or write

buffer	- location of data buffer

The format for the reply message is

reply ::=

reply code	- indicating success or reason for failure
count	- number of data bytes transferred

The file instance identifier is that specified by the server when the instance is created. The file instance is uniquely and globally identified by specifying the server implementing the file instance and the file instance identifier.

The block number is used as a sequence number for file instances with type attribute STREAM and as a location specifier for non-stream file instances to make read and write operations idempotent. For non-stream file instances, a read or write request message that is retransmitted because the reply message was lost results in the read or write being performed again (with no side-effects). For stream files, the block number allows the server to recognize that the read or write request is out of sequence. To perserve idempotency, the server must reply with the same reply message as the client would receive if the retransmission had not occurred as well as avoid side-effects on the file instance such as, for example, duplicate data blocks appearing on a user terminal. A stream I/O server thus replies without error indication (and without performing a read or write) to a read request that specifies a block number one less than the next block to read from the stream and to write requests that specify a block number equal to the last block written. Effectively, a read or write request for block number N serves as an acknowledgement to the reply to the read or write request for block N-1. This does not detect erroneously generated read or write request messages that look like retransmissions. However, this is an unlikely client error and only reduces the support for client error signalling. It does not impinge on the integrity of file instances or the server. It is one of the features of the I/O system design that each server can be designed independently to trade-off the cost of distinguishing a retransmitted request message from an erroneous message with the level of client error signalling it chooses to support. One of the objectives of our on-going research is to explore this trade-off.

A similar problem with idempotency arises in releasing a file instance. The reply to a request to release a non-existent file instance cannot signal an error in case this is a retransmission of an earlier request that successfully released the file instance. Further, the retransmission of a request to create a file instance results in the creation of a second file instance. In the worst case, the first file instance is released by the server garbage collector when the creating process is determined to no longer exist. However, for servers such as a tape server that only support one file instance at a time, the server assumes that the request is a retransmission if it is received from the same process that owns the existing file instance and specifies the same file.

Requests and replies are sent by short control messages. Data being read or written is normally transferred by the separate data transfer facility. If the data is short enough to fit in the control message starting at the buffer field, it is instead appended to the request (in case of writing) and the reply (in the case of reading). This may always be the case in systems providing large messages. Thus, the design can take advantage of larger messages than our current 8-word messages. This also provides efficient data transfer for small amounts of data such as the byte-level interaction with a terminal used by a screen editor. It is especially desirable to avoid using the data transfer facility when the data to be transferred is smaller in size than its description, namely the buffer pointer. Using Verex on a 16-bit machine, 8 bytes of data fit in a control message.

The server may defer satisfying the request until the data is available (for example, from a terminal or a pipe) by not replying immediately, and continue to handle other requests. It may also reply indicating the process should retry the request if it is unable to service the request immediately.

The simplicity of this protocol has allowed I/O servers to be implemented by undergraduate students with no previous experience with communications.

3.4 Symbolic File and Server Naming

Symbolic naming of both I/O servers and files is required to provide a user-friendly environment and server-independent file identification. We follow a scheme common to several distributed systems in using a name server to perform name mapping. The name server also handles requests to change the symbolic mapping by addition, deletion, aliasing, and modification of symbolic names.

The name server accepts a symbolic name and maps it to a server and a file specification, returning this information to the client. For certain requests such as CREATE_INSTANCE, the name server forwards the request to the appropriate I/O server with the symbolic name replaced by the file specification associated with the symbolic name. In the latter case, the client receives the I/O server's reply to the "create instance" request as illustrated in **Figure 5.** The technique of forwarding after name mapping is an extension of the name mapping services for efficiency. The client can instead issue a separate CREATE_INSTANCE request using the server and file specification returned to it by the name server.

The file specification provided by the name server for the create instance request may contain part of the symbolic name to allow the recipient I/O server to complete the name mapping. For example, a remote storage server may map part of the symbolic name in the context of the remote system it represents. However, we have minimized the use of symbolic names by the servers for simplicity and efficiency.

The name server is not part of the I/O system but a service used by the I/O environment library to allow symbolic naming of files and servers.

3.5 The I/O Environment Library

The program I/O environment is the interface to the I/O system provided for application programs. It serves to insulate applications from the unusual nature of the underlying structure of the I/O system. We have to date implemented one I/O environment (subroutine) library. It is described below to illustrate how conventional input and output can be implemented using the object-based I/O protocol described in the previous section.

The programming environment implemented is similar in flavor to the C standard I/O library [9,10]. It supports device-independent byte-stream I/O as well as block-oriented I/O. To access a file, the file is first opened using a symbolic name to identify the file and a second argument to specify the desired mode of access: READ, CREATE, APPEND or MODIFY. A process can then get a byte at a time from the file or put a byte at a time to the file to access the file in a byte-oriented mode. It can also read or write a sequence of bytes to or from the file in the block-oriented mode. Closing the file releases the resources dedicated to the open file and ensures that all changes to the file have taken place. An (**Open_file**) operation allows a file instance to be opened directly by specifying its server and instance identifier. The program I/O environment is described in greater detail in the **Appendix B.** Note that "open files" only exist as objects implemented by the environment library.

The internal design and implementation of the library is fairly apparent given the I/O model and the underlying server and communication support. The following discussion attempts to focus on points of interest.

Open issues a CREATE_INSTANCE request to the name server to map to the server and file specification to produce an instance of the file for simulating an open file connection. This use of the name server is illustrated in **Figure 5.** Part of the mode specified to **Open** indicates whether the file is to be used in the byte or block mode. For the byte mode, the local file structure created by **Open** includes a buffer equal in length to the block size of the file.

Close generates a RELEASE_INSTANCE request if the instance being closed was created by the **Open** that allocated the local file structure. (This is indicated by a flag stored in the local file structure.) Other functions either do not cause interaction with server processes or do so by calling **Read** or **Write,** which generate READ_INSTANCE and WRITE_INSTANCE requests respectively.

Get and **Put** are implemented by getting and putting a byte at a time into the local buffer until it is empty or full respectively. When the file is being read, the buffer contains up to a block of data read from the associated file instance. When the file is being written, the buffer contains up to a block of data to be written to the file instance. To handle switching between reading and writing, **Get** and **Put** use separate indices into the local file buffer. When reading, the read index maintains the byte position within the buffer. The write index is set to a value greater than the block size so as to cause **Put** to call a write "fix-up" function when next called. The write fix-up function performs the necessary actions to continue putting bytes into the local buffer, setting the write index to the current byte position in the buffer. This is the same fix-up function called when the buffer is full after a sequence of **Put's.** Similarly, the read index used by **Get** causes it to call the read fix-up function if the local buffer is empty or if the file was being written.

The separate read and write indices allow a program to intermix calls to **Get** and **Put** while inexpensively maintaining a byte-level window on the file. For example, the execution of **Get** involves one simple test, namely on the read index to check whether a call to the read fix-up function is required. Because the local file structure records whether it is reading or writing, blocks need only be written if changed and only read if the data is required or for maintaining consistency of a block that is partially modified. **Seek** also exploits the knowledge of the current reading/writing mode to minimize interaction with the I/O server.

This implementation contrasts with the Thoth [6] byte-stream implementation in which a block is always written out before reading another block when a file is open for reading and writing, even though the block may not have been changed. It also differs from the UNIX I/O library [9,10] in which simultaneous reading, writing and seeking at the byte-level on a file using a local buffer is not supported.

Exceptions are handled at two levels. At the byte-level, both **Get** and **Put** return a special value **EOF** (Exception On File) which is generated by the "fix-up" function in response to an exceptional condition. The file can then be queried for a code indicating the nature of the exception which is usually the reply code from the last read or write request. Possible exception codes include: end-of-file, end-of-medium, device error as well as less expected exceptions such as non-existent file instance, permission denied, and server unavailable.

The exception-detecting scheme allows a single test to be performed to check for exceptions while getting or putting bytes to or from a file yet provides a means of determining the nature of the exception once it is detected. Moreover, simple programs can treat **EOF** as simply "end-of-file". Thus, the increased complexity of exceptions possible due to lack of real open file connections is handled with no significant increase in code.

Each local file structure currently requires 16 words of storage plus block-size bytes for a buffer if used in the byte-oriented mode. In our current implementation on a Texas Instruments 990 machine, the program library is approximately 4 kilobytes of code when loaded, including various standard input and output formatting routines. (The size is dependent on the subroutines referenced.) The cost of executing **Get** and **Put** is 13 instructions (when the fix-up function is not called) including the function call and return sequence.

This library provides a device-independent I/O interface with only minor semantic and functional differences from conventional program environments. Thus, existing programs and programming styles can be used without concern for the underlying distributed design of the I/O system. We have found it easy to convert our suite of software and document preparation programs to use this subroutine library and the I/O system even though they were originally written for and used with a conventional I/O system.

4. Design Rationale

The design was governed by the following design goals.

* It must support the conventional program I/O model of byte-streams while allowing extensibility to application-defined and implemented files.

* It must be distributed in that: resources requirements are distributed among the client and server processes; processes can exercise local autonomy in file instance implementation; and failures tend to be local to the failed components.

* It must allow efficient, inexpensive I/O servers as well as minimize the load on the communication subnetwork.

The subroutine library does provide a conventional I/O model implemented in terms of the I/O protocol and the message primitives. The I/O system is extensible because any process can act as an I/O server if it implements the protocol. The system is distributed in that clients and servers only interact via message-passing. Each server exercises autonomous control over the file instances it implements. Finally, efficiency is achieved by allowing flexibility of server implementation, server-specified file instance typing, and block-oriented reading and writing. Also, the program environment library strives for minimal server interaction. The main design issues addressed were the use of an idempotent object-based protocol, file instances, and block-oriented reading and writing. These issues are discussed below.

An object-based protocol was used to provide a simple, light-weight protocol. The protocol is efficent for the server because there is no connection to support, no data streaming and no spontanteous communication with a client. There is also no client notification scheme to handle. In our implementation experience to data, servers can easily reclaim resources using a garbage collector which is invoked in response to need for resources.

The protocol is simple and efficient for the client because the subroutine library can easily provide a simple procedural interface; the use of messages, file instances and servers is transparent to the application.

The protocol is also undemanding of the communication subsystem. We assume a fast, low-error rate but not necessarily reliable datagram service as is available in many local-area networks. The data streaming provided by long-haul protocols and other byte-stream protocols [2,8] is assumed not necessary due to the speed of transmission and not desirable due to the added complexity. The idempotency of the protocol means that duplicate messages need not be filtered out. Corrupted messages are discarded causing retransmission (and possibly duplicate messages).

File instances were invented to be the objects on which the protocol was based. This was done instead of using the "real" underlying objects because we recognized that these "real" objects were in many cases created as part of being accessed. This is modelled explicitly by the concept of file instance. As examples, a version of a file, an X.25 virtual circuit, a printer spool file, the current login information all represent file objects that are created as part of accessing them. The explicit support of this creation removes the need for separate "connect" or "setup" operations. For example, the call specification for a virtual circuit is specified in creating the file instance. Similarly, creating a tape file instance may require mounting a tape and creating a printer spool file may require specifying output formatting parameters.

File instances support accessing data in a consistent version because a file instance can represent a snapshot of the underlying object or activity, which may have changed since the file instance was created. Similarly, a file instance can represent a new version of an object that atomically updates the object when it is released, thus providing support for databases and transaction processing [19].

File instances also support the generalization of files to views of data that can be instantiated with a particular description. This generalizes the mechanism available in many systems in which the views of data are both implicit and fixed.

An earlier design [3] in which the objects of the protocol were files suffered problems in supporting the temporary, created objects described above.

File instances can also have shorter identifiers than the underlying objects, reducing the overhead when reading and writing. This is especially true when the file specification is a description of an object to create. Because the identification is generated (autonomously) by each server and is meaningful only relative to that server, an application can define its own files and use the I/O system for data transfer between its different modules.

The separation between the creation of a file instance and "opening" the file also recognizes that the instance may need to exist longer than that implied by the file opening and closing. This is solved in other systems by the server maintaining a count of the number of open file connections to this file and releasing it when all the connections are closed. Related to this, the process creating the instance may not need an open file connection to the instance. It may be creating it solely for use by another process, as is the case with the UNIX **shell** [18].

Block-oriented reading and writing has several motivations. The block is intended to reflect an efficient unit of communication for the server which may in turn be dependent on the underlying hardware. Because the block size is a parameter, no loss of portability need ensue from the use of a hardware-dictated block size. The constraint that all read's and write's must start on block boundaries reduces the hardware and software support required for interactions. For instance, if the block size matches the unit used by the device or communication line the server manages, it need not provide buffering to match different sized packets or blocks. Also, the block interaction facilitates the server providing atomic operations such as indivisible write's [19,20].

An application of block-mode access is a database system using B-trees for storing relations on disk. By using the block as the storage unit for nodes of the B-trees, the database could be guaranteed efficient access and indivisible update to nodes. However, the use of block numbers internal to data structures is contingent upon the block size remaining invariant. For example, the data could not be copied directly to another device with a different block size.

In the environment library, although the **Read** and **Write** operations do not allow starting at an arbitrary location in a file as do those of UNIX [10,18], the latter are easily implemented with **Get** and **Put.** Also, many uses of **Read** and **Write** in UNIX recognize the efficiency of the underlying physical block size and implicitly implement block-oriented operations.

The ultimate design rationale lies in the utility of the design. We have modified our research system to use this design. Currently, all the standard files and devices are available and many standard programs use the system for all their I/O activity. The next section describes some of the I/O servers with which we are experimenting.

5. Example I/O Servers

Four examples are given of I/O servers in our current system. All of these have been implemented and used, although the implementations in some cases are incomplete.

5.1 Storage Server

The storage server provides storage of data, using disks, tapes or whatever storage devices are available. It supports the ability to view the data as files in the conventional sense. The storage server provides a file instance as a snapshot of the data when the file instance is used in READ mode. When a file instance is used for CREATE or APPEND, the data written to the file instance atomically updates the underlying data of the storage container when the file instance is released with a successful release mode.

The storage server primarily provides services for storing data. It provides access to that data using the I/O protocol but it need not store data as files in the conventional sense.

This examples illustrates how the service, namely storage, is detached from the view of the stored data, i.e. as files. Our current work is engaged in expanding the devices used by the storage server to other devices than disk, implementing atomic update, and exploring novel file specifications that go beyond the conventional disk file model.

5.2 Session Server

The session server functions primarily to record the users currently logged in, handle the log-in sequence and control access to the passwords and user information. It is useful to have a program that prints the users currently signed on, requesting this information from the session server. It is difficult for the program to accept all the information at one time because there may be an unbounded amount of data returned. Conversely, if the program requests the information in packets, the information may be, in total, inconsistent.

Using the I/O protocol, this information is read as though contained in a file. In response to a create instance request, the session server creates a file instance that contains the information about users signed on at that time. This file instance can then be read by the program using the standard I/O routines with the guarantee that the information is consistent. The file instance is released according to the standard I/O protocol.

This example illustrates the use of the I/O system for providing a consistent data picture of an on-going activity when the data involved is changing.

5.3 Mail Server

The mail server provides inter-user communication in the form of user-level mail messages. A mail server is dynamically created whenever activity is initiated on a user's mailbox (providing a mail server for that user is not already in existence). The mail server supports the I/O protocol for transferring new messages to the user's mailbox as well as reading messages in the mailbox. By also providing symbolic naming of user mailboxes, inter-user communication is easily coupled with the I/O system, reducing the need to use special "mail" programs. For example, the error output of a concurrently executing compilation can be connected by the user directly to his mailbox, causing him to be informed when the compilation completes as well as conveniently delivering any error messages resulting from the compilation. Similarly, general-purpose text editors and text formatters can be used easily in the preparation of messages.

The mail server uses the services of the storage server to store inter-user message data and management information. This illustrates how a server can provide a service that is basically a refinement and an extension of conventional files, yet still make the services available using the I/O system. It also illustrates how an application can provide a new type of file, namely mail message files in this case.

5.4 A Unix-like Command Interpreter

The I/O system raises some problems for applications that have used tightly-coupled I/O connections in other systems. An example of this is the UNIX **shell** [18] (command interpreter) which makes extensive use of byte-stream open file connections and shared file state information supported by the UNIX kernel. The shell supports a standard input, output and error output and the piping of a program's output into another program's input. It also supports redirection of input and output and so-called "here" documents, in which programs executed from a command file also take their standard input from the command file starting immediately after the point they are invoked.

To implement these features, we exploit two properties of the I/O system and the execution environment. First, the I/O system allows any process to act as an I/O server. Second, the command interpreter can consist of several processes in the same space (a team as in Thoth [4,6]), some of which can act as I/O servers for programs the command interpreter is controlling. The need for multiple processes arises because of the multiple concurrent reading, writing, program termination and exceptions that can occur.

Each of the standard input, output, and error output is specified to a program by server and file instance. A program can then access any one of these file instances by calling **Open_file** with these parameters. This specification is provided by the command interpreter.

The command interpreter determines the program input, output and error file from the symbolic names specified on the command line. In the case of pipes, it requests an instance of a pipe from the pipe server. All input, output and error output that is not redirected goes through the command interpreter. The standard input defaults to a file instance provided by the command interpreter process that handles command interpreter input. Similarly, the standard output and error output default to that of the command interpreter. The process interaction for a simple command pipeline is illustrated in **Figure 6**. In this example, the text editor is suspended waiting for input from the interpreter while the interpreter oversees the operation of the command pipeline. Stored text is being filtered through the text formatter and overstrike filter to the printer. Each arrow points from a client using the I/O protocol to an I/O server with which it is communicating.

The command interpreter input process reads a line and makes it available to the currently attached program if the line is not escaped to be a command line. Otherwise it interprets and executes the input line as a command. This makes the facility of escaping from a subsystem context to execute a command available during the execution of any program as well as providing the command interpreter context for the execution of the command line. It also makes features like attaching and detaching to concurrently running programs possible.

This example illustrates how a large application can be written as many separate programs that interact via the I/O system using application-defined files and file instances. The command interpreter is strictly an application program to the system; any user may write and select a command interpreter of his choice. The command interpreter program and its suite of command programs implement a command language. Because of the size of the language and the desire for extensibility, most individual commands are implemented as separate programs. Interaction between the managing interpreter program and the command programs takes place through the I/O system using files implemented by the interpreter program.

This approach to user interface design and program interconnection is equally applicable to virtual terminal models similar to RIG [12,13].

6. Discussion

We have presented the design of a distributed I/O system that uses an object-based protocol to provide uniform data transfer between a diverse set of files, programs, resources and services. A program library has been implemented that provides a conventional byte-oriented I/O abstraction and a block-oriented abstraction. The conventional I/O abstraction was considered important to retain because of the large number of programs that use this model and the programming experience we have with it.

Central to the design is the I/O protocol which defines interaction between clients and I/O servers. The protocol is implemented in terms of processes and message primitives so it is insulated from the underlying network architecture and the protocol is insulated from the correspondence between processes and processors. Because the protocol is implemented for clients by a subroutine library, it is detached from the program environment it supports and can support many different program environments.

The important aspect of the semantics of the message primitives used here is that they are simple transaction-oriented operations. A client process is only connected with a server process from the time that it sends to the server until the server replies to the client process. Beyond this, numerous different sets of primitives would support the design described here. For example, the I/O protocol could be implemented as a Level 2 protocol in the PUP internetwork architecture [2].

To date, to support the I/O system we have implemented terminal servers, an X.25 server, a printer server, a pipe server, a mail server, storage server and a command interpreter providing a subset of the UNIX **shell** facilities. The latter stages of preparing this report used a text editor and text formatter that use the I/O system. Our experience so far has been very flavourable. The major disadvantage has been a slight increase in the size of programs due to I/O functions being included as subroutines rather than being available as system calls, as was the previous scheme. Future projects include a local-area network implementation, new server designs, new models of user interfaces using the I/O system, and naming issues, particularly reverse name mapping, i.e. file instance to symbolic name.

Our design philosophy is in sympathy with a trend in local-area network communication away from sophisticated layered protocols and tightly coupled state-synchronized connections. These protocols seem best used for long-haul communications in which errors, propagation delay and complex network topology are important design considerations. The low delay and error rates of many local-area networks and message kernels suggest that simpler connectionless end-to-end protocols may be sufficient.

Acknowledgements

Discussions with Steve Deering and John Demco have contributed to the ideas and form of this paper. Their work and that of Brent Hilpert and Tony Kusalik on the implementation of the design are also recognized. The influence of UNIX on the program I/O model and the problems considered has been substantial.

References

1. F. Baskett, J. H. Howard and J. T. Montague, Task Communication in DEMOS. Proc. 6th Symp. Operating Systems Principles, Operating Systems Review, Vol. 11, No. 5, November 1977, 23-31.

2. D. R. Boggs, J. F. Shoch, E. A. Taft and R. M. Metcalfe, Pup: An Internetwork Architecture IEEE Trans. on Communications vol. COMM-28, pp. 612-624, April 1980.

3. D. R. Cheriton, A Loosely-Coupled I/O System for a Distributed Environment. IFIP WG 6.4 International Workshop on Local-Area Networks, Zurich, Switzerland, August 1980.

4. D. R. Cheriton, Multi-process structuring and the Thoth operating system. UBC Computer Science Technical Report 79-5, University of British Columbia March 1979. (based on the author's Ph. D. thesis, University of Waterloo 1978).

5. D. R. Cheriton, Designing an Operating System to be Verifiable. UBC Computer Science Technical Report 79-9, October 1979.

6. D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, Thoth, a portable real-time operating system. Comm. A.C.M. 22, 2 (Feb. 1979) 105-115.

7. CCITT Rapporteur on X.25 - Level 3, Draft revised recommendation X.25, CCITT COM VII no. 439, as amended, Feb. 1980.

8. M. A. Johnson, Ring byte stream protocol specification. Computer Laboratory, Cambridge, April 1980.

9. B. W. Kernighan and D. M. Ritchie, The C Programming Language. Prentice-Hall Software Series, Prentice-Hall, New Jersey 1978.

10. B. W. Kernighan and D. M. Ritchie, UNIX Progamming - Second Edition (Appendix), UNIX Version 7 documentation, Bell Laboratories, November 1978.

11. B. Lampson and H. Sturgis, Reflections on an Operating System Design. Comm. A.C.M. 19, 5 (May 1976), 251-265.

12. K. Lantz, Uniform Interfaces for Distributed Systems. Computer Science Technical Report TR63, University of Rochester, May 1980.

13. K. Lantz and R. Rashid, Virtual Terminal Management in a Multiple Process Environment. Proc. 7th Symp. Operating Systems Principles, December 1979, ACM order no. 534790.

14. B. Liskov, A. Synder, R. Atkinson and C. Schaffert, Abstraction Mechanisms in CLU. Comm. A.C.M. 20, 8 (August 1977), 564-576.

15. T. W. Lockhart, The Design of A Verifiable Operating System Kernel. UBC Computer Science Technical Report 79-15, November 1979.

16. D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules. Comm. A.C.M. 15, 12 (December 1972), 1053-1058.

17. D. P. Reed, Naming and Synchronization in a Decentralized Computer System. Ph.D. thesis, MIT/LCS/TR-205, September 1978.

18. D. M. Ritchie and K. Thompson, The UNIX timesharing system. Comm. A. C. M. 17, 7 (July 1974), 365-375.

19. H. Sturgis, J. Mitchell and J. Israel, Issues in the Design and Use of a Distributed File System. Operating Systems Review Vol. 14, No. 3 (July 1980), 55-69.

20. D. Swinehart, G. McDaniel and D. Boggs, WFS: A Simple File System for a Distributed Environment. Proc. 7th Symp. Operating Systems Principles, December 1979, ACM order no. 534790.

21. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Piersen and F. Pollack, HYDRA: The Kernel of a Multiprocessor Operating System. Comm. A.C.M. 17, 6 (June 1974), 337-345.

Appendix A: Program Environment Primitives

The following are the major functions used in our research system, Verex, for implementing the I/O system.

id = Create_process(priority)

creates a new process in an initial state awaiting a reply message from its creator and returns a unique port identifier which is used to specify the process subsequently in communication. There is one port per process so the terms process identifier and port identifier are interchangeable. Operations are provided for initializing the state of the process.

Destroy_process(id)

halts and removes the process associated with **id** rendering the identifier invalid. The kernel unblocks all other processes that are blocked on a nonexistent port (or process) with a time-out mechanism.

Send(id, messsage)

sends the message to the specified process port and blocks the sender until the buffer has been received and replied to.

id = Receive(message)

transfers the oldest unreceived message into the specified message buffer. The sender is now awaiting a reply. The invoking process is blocked until a message is received if necessary.

Reply(message, id)

replies to the specified process and unblocks this process providing it is awaiting reply from the invoking process.

Forward(message, id1, id2)

forwards the message to the process associated with **id2** and makes the process specified by **id1** appear as though it originally sent to **id2** (providing the first process was awaiting reply from the forwarding process).

Transfer_from(id, remote_array, n, local_array)

transfers **n** bytes from the remote array in the space of the process to the local array in the space of the invoking process. The process must be awaiting reply from the invoking process.

Transfer_to(id, remote_array, n, local_array)

is similar except the data is transferred to the remote array. It is also possible to transfer data directly between two processes awaiting reply from the invoking process.

The system also contains process state query and modification functions plus less-used varieties of message communication primitives.

Appendix B: I/O Program Environment

The following describes the major functions provided in our program I/O environment.

file = Open(filename, mode)

opens the file specified by the symbolic name **filename** with an access mode which is one of READ, CREATE, APPEND or MODIFY. The symbolic naming is similar to UNIX in syntax.

file = Open_file(server,instance,mode)

opens a file specified by the server and instance identifier. The identifier **file** is used subsequently to identify the file structure.

Close(file)

completes any outstanding updates to the file and releases the local resources and data structures associated with **file.**

The I/O model is further divided into a byte-oriented and a block-oriented set of operations.

Byte-Oriented Abstraction

In the byte-oriented abstraction, a file is a sequence of bytes numbered 0, 1, ... n-1 where n is the number of bytes contained in the file. Associated with an open file is a buffer and a current byte position.

byte = Get(file)

assigns to **byte** the value in the current byte location of the local file structure specified by **file** and increments the current byte location by one.

byte = Put(file, byte)

sets the byte value at the current byte location to **byte**, increments the current byte location by one, and returns the value **byte**.

It is an error to issue a **Put** on a file only opened for READ. Similarly, it is an error to issue a **Get** on a file that is not READABLE.

Flush(file)

flushes changes recorded in the local buffer to the "real" file.

Seek(file, offset, origin)

changes the current byte location of the file to that specified by **origin** and **offset**, where the origin may be the beginning of the file, the end of the file, or the current byte location. This allows random access to the file.

n = Byte_location(file)

returns the current byte location of the file.

Exceptions occurring when executing **Get** or **Put** are signalled by these functions returning a special value **EOF** (Exception On File) which corresponds to the conventional end-of-file indication. In the case of **EOF** being returned,

code = File_exception(file)

returns a code that indicates the nature of the exception, which can be end-of-file, end-of-medium, remote (device) error as well as less expected exceptions such as non-existent file, permission denied, and file unavailable. Standard procedures are defined for printing error messages and invoking program termination depending on which exception code is returned.

Exceptions occurring during the execution of other operations are indicated by an exception code returned by the operation (which was not shown for simplicity). The **Open** operation takes an optional parameter that is set to indicate the exception when it fails; a null file value is returned. If the optional parameter is omitted, **Open** performs the default exception action of printing a message and terminating the program.

Block-Oriented Abstraction

In the block-oriented abstraction, a file is a sequence of blocks of some maximum size, called the block size of the file.

block_size = Block_size(file)

returns the block size in bytes of the local file. Associated with the local file structure is a current block location.

block_location = Block_location(file)

All operations are on blocks.

count = Read(file, buffer, n)

reads a maximum of n bytes into the buffer from file and returns the number of bytes actually read. The bytes are read from the beginning of the block at the current block location of the file. The number of bytes read may be less than the number requested for several reasons including encountering the end-of-file, encountering the end-of-line (in the case of terminal input), as well as particular error conditions.

count = Write(file, buffer, n)

writes the first n bytes from **buffer** to the file starting at the beginning of the current block and returns the number of bytes actually written. If either **Read** or **Write** return a count less than that specified in the call, the procedure **File_exception** returns the exception code indicating the reason. It is an error to read or write with a byte count larger than the block size if the file is not MULTI_BLOCK.

Seek_block(file, block_offset, origin)

changes the current block location to that specified by **block_offset and origin.** Note that neither **Read** nor **Write** change the current block_location.

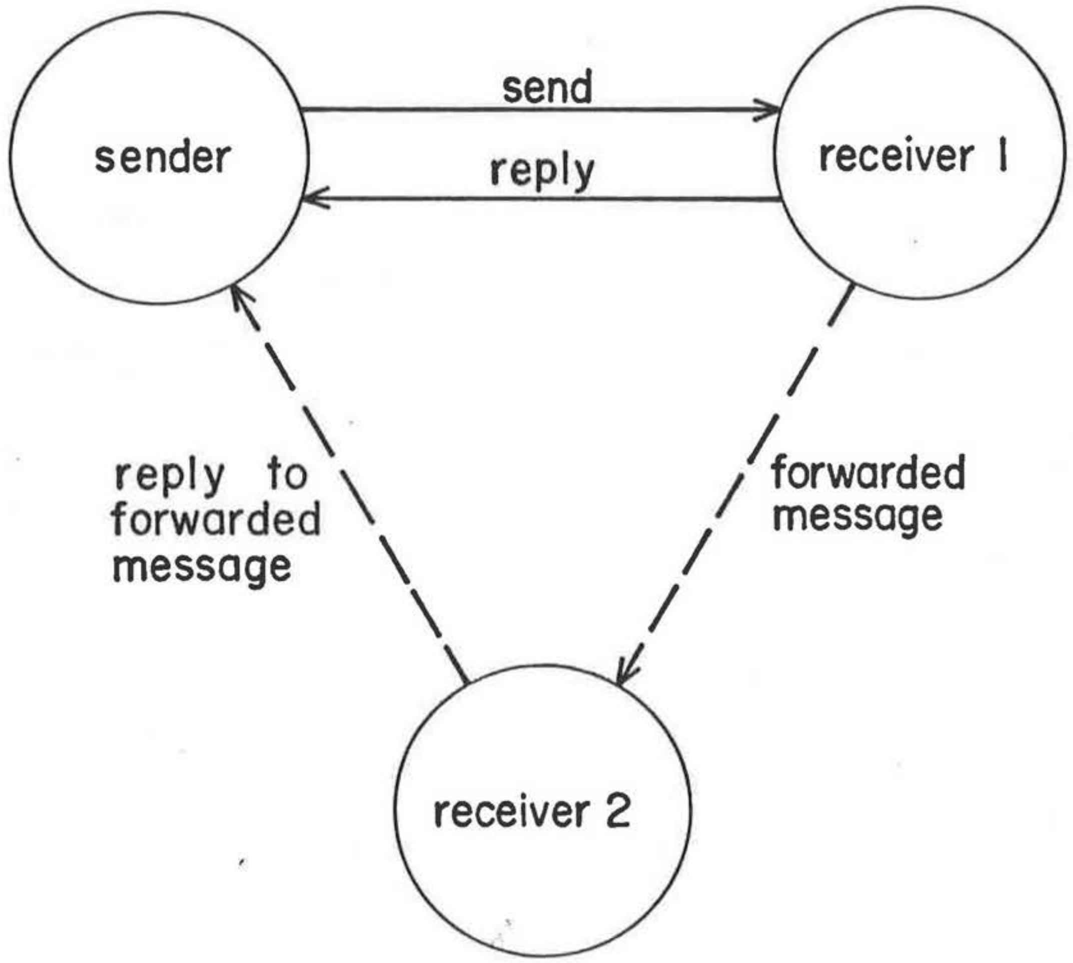

FIG. I Message Communication

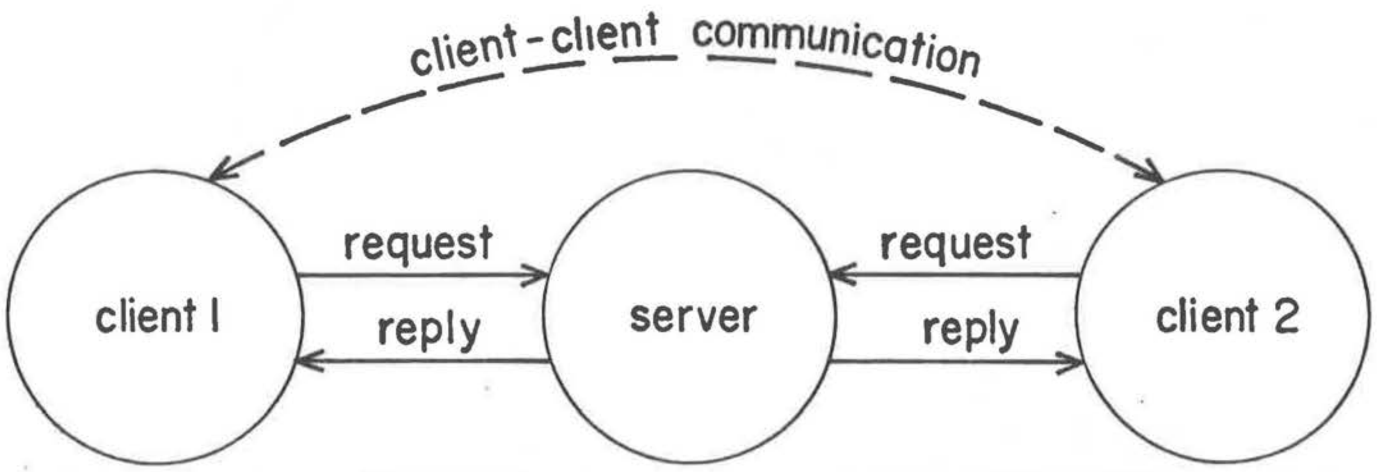

FIG. 2 Two Levels of Object-based Communication

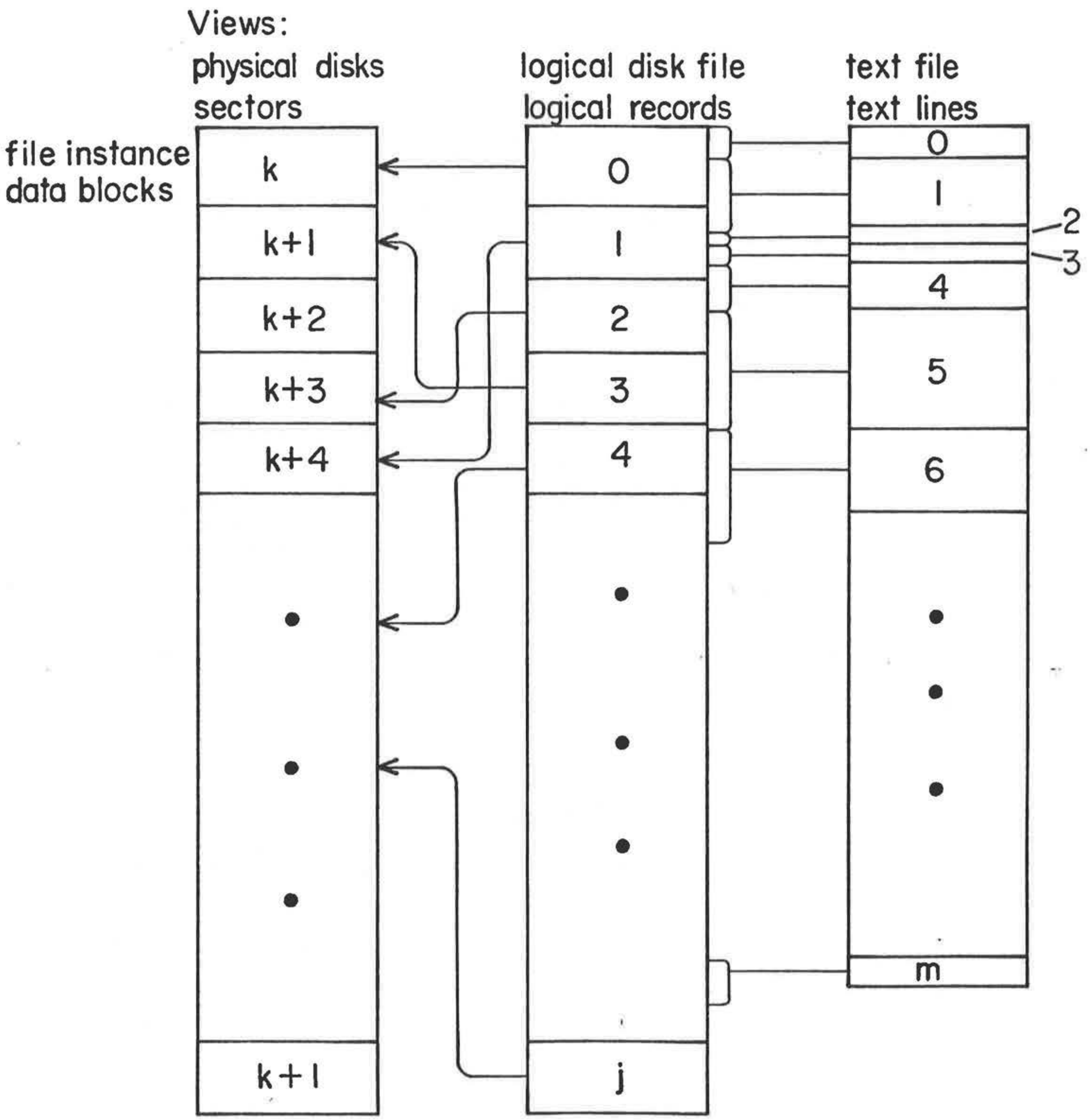

FIG. 3 Different Views of the Same Data.

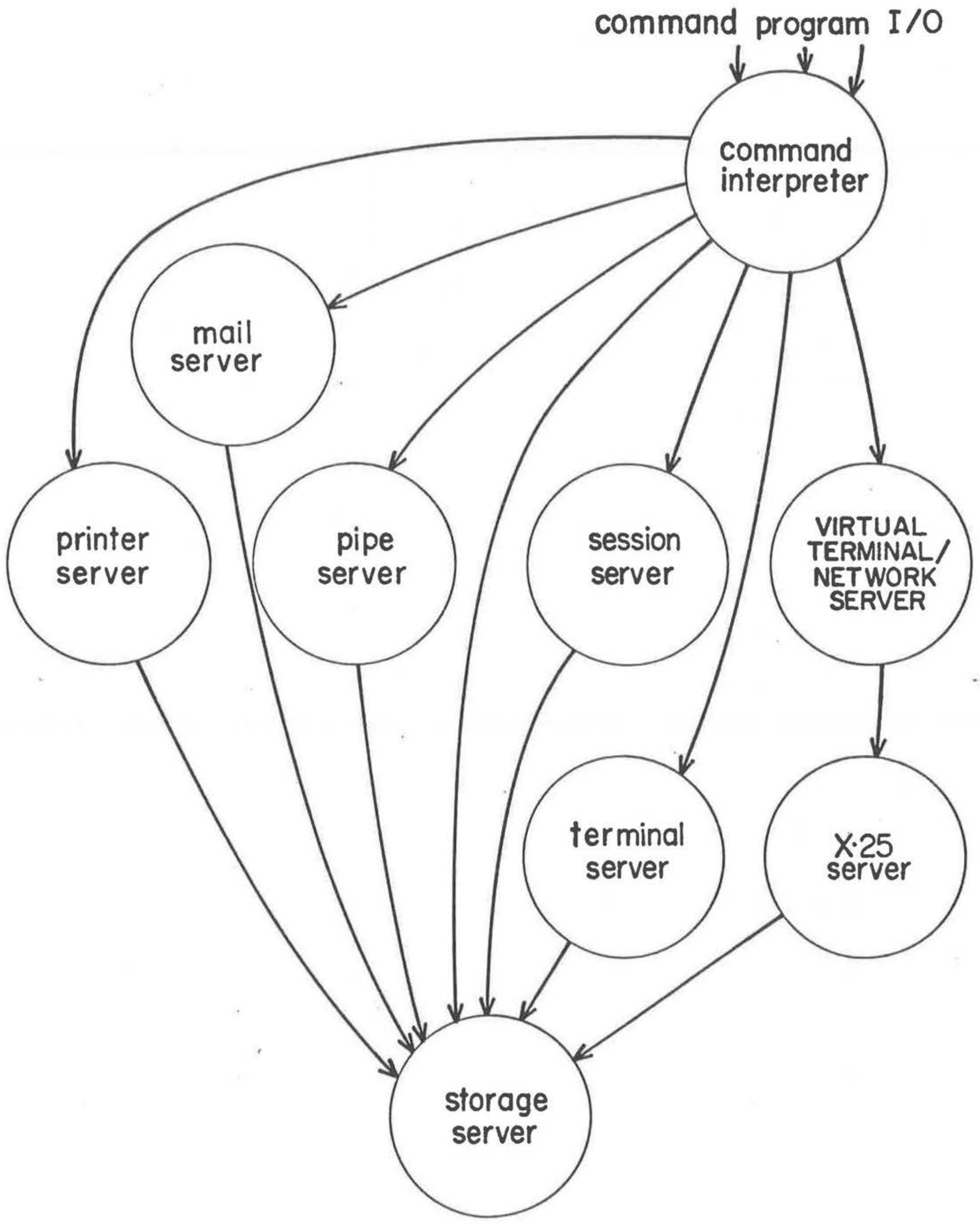

FIG. 4 I/O Server Dependencies

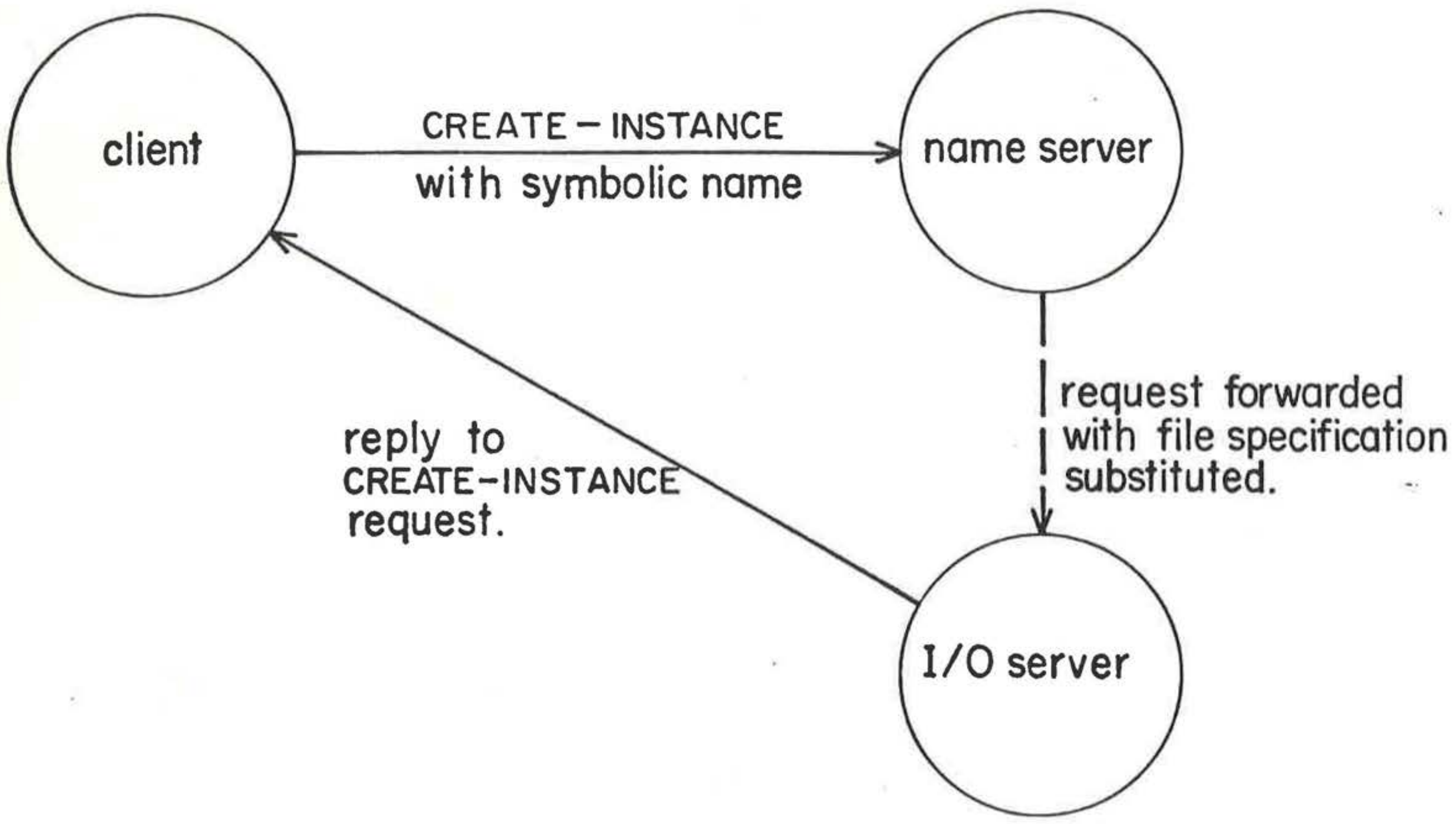
client
CREATE – INSTANCE with symbolic name
name server
request forwarded with file specification substituted.
reply to CREATE-INSTANCE request.
I/O server

FIG. 5 Symbolic Name Mapping by the Name Server.

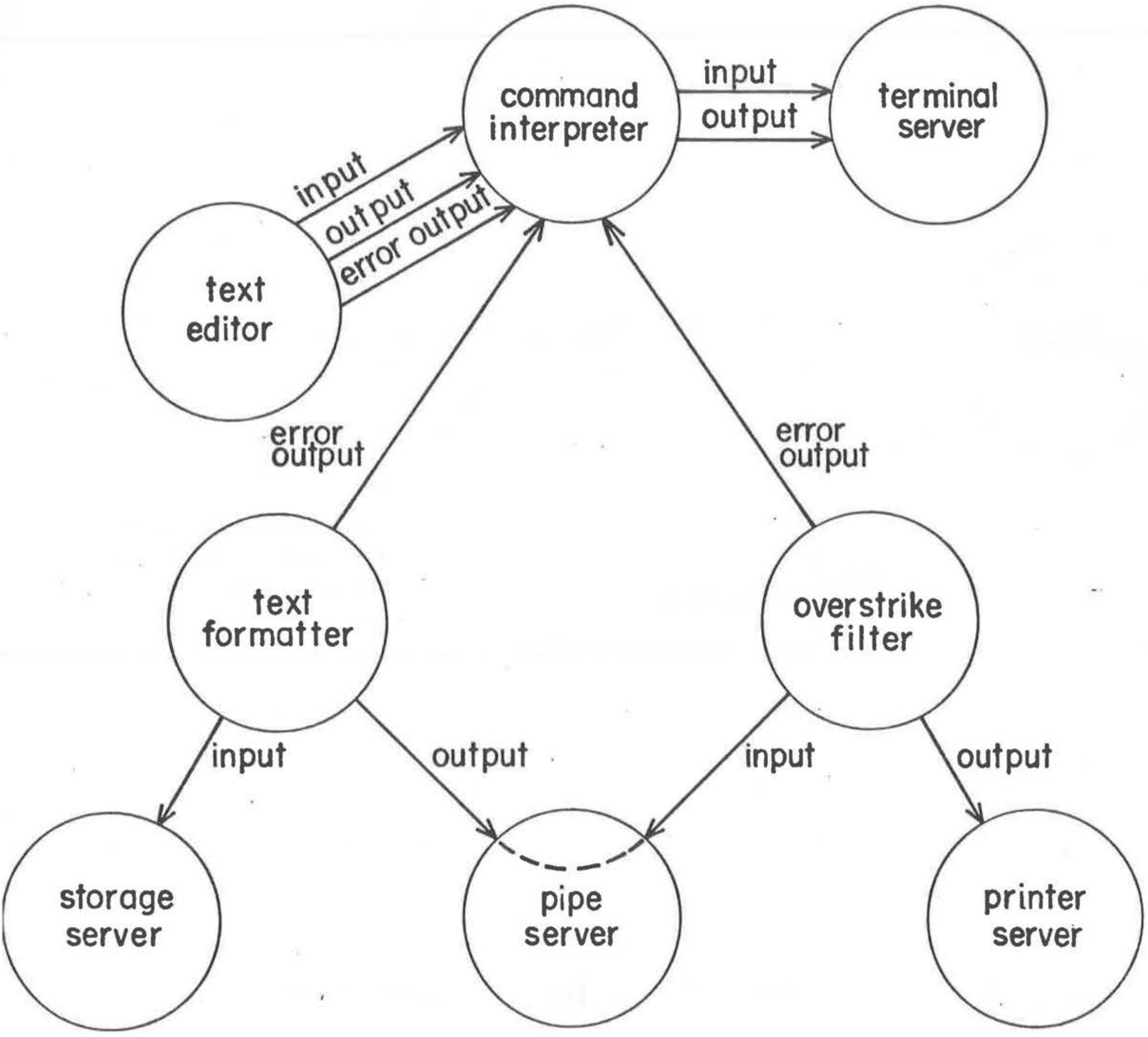

FIG. 6 Process Interaction for a Simple Command Pipeline.