November 1986

Report No. STAN-CS-86-1134

UIO: A Uniform I/O System Interface for Distributed Systems

by

David R. Cheriton

Department of Computer Science

Stanford University
Stanford, CA 94305

UIO: A Uniform I/O System Interface for Distributed Systems

David R. Cheriton
Computer Science Department
Stanford University

Abstract

A uniform I/O interface allows programs to be written relatively independent of specific I/O services and yet work with a wide variety of the I/O services available in a distributed environment. Ideally, the interface provides this uniform access without excessive complexity in the interface or loss of performance. However, a uniform interface does not arise from careful design of individual system interfaces alone; it requires explicit definition.

In this paper, we describe the UIO (uniform I/O) system interface that has been used for the past five years in the V distributed operating system, focusing on the key design issues. This interface provides several extensions beyond the I/O interface of UNIX, including support for record I/O, locking, atomic transactions and replication as well as attributes that indicate whether optional semantics and operations are available. We also describe our experience in using and implementing this interface with a variety of different I/O servicer plus the performance of both local and network I/O, We conclude that the UIO interface provides a uniform I/O system interface with significant functionality, wide applicability and no significant performance penalty.

This work was supported in part by the Defense Advanced Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431, by Digital Equipment Corporation, and by the National Science Foundation. The initial work was supported by the Natural Sciences and Engineering Research Council of Canada.

General Terms: Design, Performance, Experimentation, Standardization
Keywords and Phrases: distributed system, input/output, interface, byte stream, remote procedure call, interprocess communication, files.
CR Categories: D.4.4 Input/Output

1 Introduction

A *uniform I/O* interface specifies the syntax and semantics for a set of I/O operations that are provided in a sufficiently similar form across the I/O services of an operating system such that programs can be written to be relatively independent of the data sources and sinks to which their I/O operations are eventually bound. A familiar example of such an I/O interface is the UNIX I/O interface [29] [1] consisting of the system calls open, read, write, seek, close and ioctl. [2] Most programs written using this I/O interface can work with a variety of different I/O services including disk files, terminals and pipes.

Without a uniform I/O interface, each program would only work with the set of I/O services it had been explicitly programmed to handle. In the extreme, every program would have to be explicitly programmed to handle (for example) output to a terminal as well output to a file or else only work with one or the other. The result would be greater burden on programmers, greater program complexity and less flexible interconnection of programs with I/O services.

A well-specified and powerful uniform I/O interface is even more important in a distributed system because of the multiple different file systems, printers, network services and so on that can be interconnected and the wide range in function they can provide. Proper specification is essential so that existing services can be retrofit with a facade that implements the interface and so that new services can be designed to work with the existing applications. In this regard, the UNIX I/O interface is neither well-specified, nor well suited to a distributed environment based on remote procedure call (RPC) [3] or message-passing, nor does it provide support for locking, replication and atomic transactions.

A uniform interface does not arise from careful design of individual system interfaces alone. There needs be a common abstraction and interface for a class of services with the interface of each service in that class designed to be compatible with the common interface. For example, if a read operation is part of a common I/O interface then all I/O services should implement a read operation with the same syntax and the same genetic semantics [3].

In addition, the interface should specify common procedures for all reasonably general I/O functionality and distinguish between *compulsory functionality*— that which all I/O servers need to provide-and *optional functionality.* Optional functionality in a uniform interface specifies a common interface if the functionality is present, but does not require every service to implement it. For example, not all servers can support random access to data, yet this is an important functionality to provide in a uniform way to clients. In this vein, a uniform interface that simply specifies the "lowest common denominator" of the I/O services is inadequate, since that could not include random access.

Finally, the designer should recognize that some functionality is too specialized to include in the uniform interface and instead provide a standard escape mechanism for accessing this functionality. For example, setting a broadcast network interface into so-called "promiscuous mode" is highly specified to a certain class of devices and is not appropriate as part of the uniform interface. Nevertheless, access to the functionality is important. The appropriate choice of a common abstraction for I/O and partitioning of functionality across the compulsory, optional and "escape" portions of the interface is a significant design challenge.

Additional considerations are introduced by the use of RPC or equivalent as the transport-level communication mechanism.[4] First, there is no connection maintained between calls so the I/O interface must provide

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] In fact, Thompson [34] describes the UNIX kernel as primarily an "I/O multiplexer".

[3] The exact semantics of, for example, reading from terminal differ from reading from a file. However, at some level of abstraction, namely the genetic level for the class under consideration, the semantics are the same.

[4] For example, the V IPC facilities are equivalent to RPC for our purposes here.

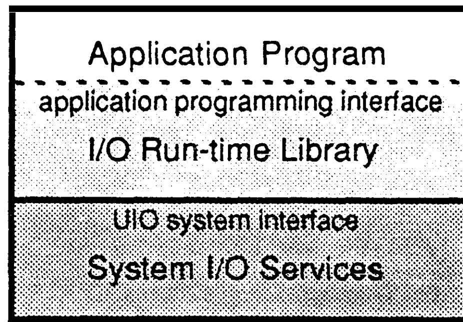

Figure 1: Application I/O versus System I/O Interface

connection support itself [5]. By connection, we refer to the state and association that span multiple calls. For example, a multi-window service must be able to associate multiple independent calls with the same window. Similarly, an I/O service requires criteria for reclaiming the state record and resources associated with a client, especially when the client terminates abnormally. Finally, there needs to be uniform criteria for access control to the I/O level connection state and resources.

With a virtual circuit-based communication mechanism, the server can maintain one virtual circuit per window and associate all data coming in on one circuit with the same window. Also, the server is normally informed when the circuit is cleared as a result of client failure so the service can reclaim the state at that point. Finally, the service may assume that any client able to use the virtual circuit is authorized on this I/O connection.

Although these considerations may appear to point out deficiencies in the RPC model of communication, we argue that I/O state maintenance, reclamation and protection can be done easily at the I/O interface level, allowing for greater optimization by the I/O service modules. In addition, we strongly support the view that the RPC model simplifies the transport service compared to using a virtual circuit model.

In this paper, we describe the uniform I/O (UIO) system interface used in the V distributed operating system [6,9] as an example of a uniform interface that addresses the above issues. The UIO interface defines a common abstraction for a diversity of services that fit into the general notion of I/O. We focus on the key design issues and on the rationale behind the design choices made in developing this interface. We also describe our experience in using and implementing this interface with a variety of different I/O services as well as the performance of both local and network I/O. We conclude that the UIO interface provides a uniform I/O system interface with significant functionality, wide applicability and no significant performance penalty. We also propose the UIO interface as a good design for the system I/O interface in other RPC-based distributed systems.

In the following discussion, it is important to distinguish the I/O system interface from the application I/O interface which is the abstraction seen by the application programmer. The I/O system interface is an operating system interface designed for maximum performance, reliability, security and flexibility to handle a variety of different application I/O interfaces. The application I/O interface is defined by the programming language, as in Ada or Pascal, or by a standard run-time library, as in C [22]. It is implemented on top of the system I/O interface as a set of standard procedures, the *run-time library*, linked directly with the application, as shown in Figure 1.

This distinction is significant in centralized systems because of the cost of protection boundary crossing associated with the system interface. For example, byte-stream I/O implemented by get-byte and put-byte system calls would be significantly slower than an application run-time implementation because of the system

[5] An RPC mechanism may use an underlying connection mechanism. However, it is not generally available to the RPC client or server level.

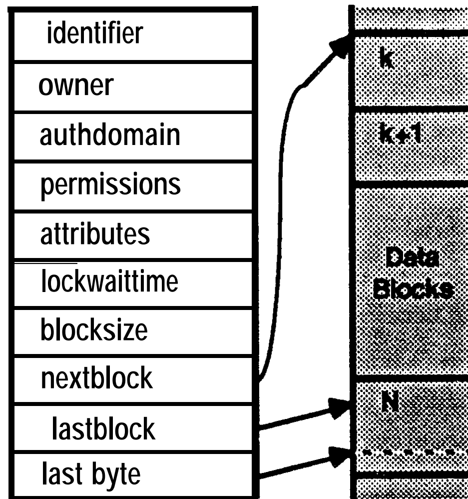

Figure 2: Client-visible Generic UIO State

call overhead. In modern distributed systems, this distinction is even more pronounced owning to several factors that favor putting as much function as possible in the application run-time routines. These factors include the following:

- The cost of a remote system call is higher since it entails network communication and all the associated costs. Thus, performance is improved by adding function to the application run-time procedures if it reduces the frequency of these remote calls.
- Server processing is a critical shared resource since each server machine that implements these system calls provides service to multiple nodes running applications, each with their own processor(s). Overall system performance is improved by adding to the application run-time procedures function that transfers processing load from the server processor to the client processors.
- The falling cost of semiconductor memory has reduced the role of the system module as a mechanism for run-time code sharing, allowing interface design to be based on reliability, security and communication/processing overhead considerations alone. In our experience, these criteria tend to shift function from the system service modules to the application run-time routines compared to the function placement in conventional operating systems.

Recognizing these factors, the UIO interface is designed to minimize load on I/O servers and the communication mechanism, relying on a significant layer of I/O run-time procedures to build a conventional application programming I/O interface on top of the UIO interface.

The remainder of the paper is organized as follows. The next section describes the key design aspects of the UIO interface. Section 3 describes our experiences with this interface and its associated protocol in V, including performance. Section 4 indicates the relation of this work to other work. We close with some conclusions. The UIO interface is summarized in the Appendix.

2 Key Design Aspects of the UIO Interface

The UIO interface is defined in terms of an abstract object called a *UIO*. The operations on UIO objects constitute the operations of the UIO interface; the UIO object itself represents the state of the interface, the *I/O state*. The client-visible generic UIO state is shown in Figure 2. By generic, we mean that this client-visible state is present in all UIOs. UIO objects for specific services may include client-visible state specific to the service. For example, a UIO object corresponding to a window may include its current screen coordinates as part of its client-visible state.

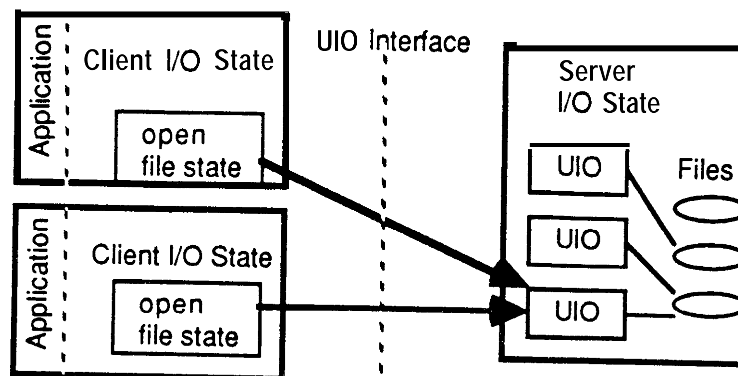

Figure 3: Partitioning of Client and Server I/O State

The UIO object represents the I/O state as some control state and a sequence of data blocks, as depicted in Figure 2. The control state includes information required to access the data. The sequence of data blocks represents the data that can be read and written through this interface. The control state portion defines: an *identifier* as a short name for the object, *owner*, *authorization domain* and ***permissions*** as criteria for protection, accounting and reclamation of the I/O state, and, *attributes* that indicate client-visible differences between UIOs. For example, some I/O services support random access whereas others must restrict reading and writing to sequential access. These differences arise from optional aspects of the UIO interface that different services elect to support, or are able to support. The *lockwaittime* indicates the maximum time to wait for a lock. The sequence of data blocks associated with a UIO is described in the control state by *nextblock*, the current lowest numbered block, *lastblock*, the current highest numbered block, *lastbytes*, the number of bytes in the last block. and *blocksire* bytes, the maximum size for blocks.

In a system such as the V distributed system using the UIO interface, each I/O service is abstracted as 'one or more UIO objects. For example, the V transmission control protocol (TCP) service [11] is abstracted as two UIO objects per connection, one for reading and one for writing. Allocation and initialization of I/O state is abstracted as the creation of one or more UIO objects using the *CreateUio* operation. Cleanup and reclamation of I/O state is handled by the deletion of UIO objects using the ***ReleaseUio* operation.** With respect to the TCP service again, CreateUio sets up a connection and ReleaseUio closes the connection. Reading and writing is represented as access to the UIO data blocks using the operations *ReadUio* and *Write* **Uio.** Operations are also provided for querying and modifying the UIO control state. The CreateUio operation effectively creates an abstraction of the underlying file-like object or I/O service in the form of a UIO object, setting up a translation from the UIO operations to specific operations on the real I/O service. For example, the ReadUio operation for a UIO corresponding to an Ethernet interface must be translated into the device-specific operations in order to read from the interface.

The UIO interface partitions I/O state between the client run-time library and the I/O server, as shown in Figure 3. The UIO object corresponds to the portion of the open file connection state maintained in the service module. The remainder of the I/O state is maintained in the client run-time library. In particular, the client run-time state includes open file descriptors that maintain current file position and perform the translation between byte-stream (if used by the client) and block array, as supported by the UIO interface. Maintaining this state in the client, rather than the server, reduces the space and communication costs to the server. For instance, there may be multiple open file descriptors corresponding to a single UIO. This separation of the "opening of files" at the client level from the creation of I/O state in the server also reduces the processing load on the servers.

The following subsections describe the rationale behind key aspects of the UIO interface design. The reader is referred to the Appendix for further details on the UIO interface.

2.1 Data Access Model

I/O functionality is centered around the reading and writing data between local memory and an external storage or communication service. Each I/O operation must specify the unit of data to read or write, how that unit is addressed or referenced, and how many units can be read or written at one time as well as the ordering constraints on data access, i.e. sequential access, random access or sequential with rewind, etc. Different I/O services exhibit considerable diversity with respect to these parameters. For example, disk files may be addressed by sector number whereas record files may be addressed by character string keys. Similarly, disk files may contain fixed-size blocks but user terminals deliver variable-length line records. A file server can easily support large read and write operations yet these are difficult to provide in a terminal, network connection or pipe. Finally, network connections are strictly sequential in nature, tape drives are sequential but allow rewind or reset, and disk files provide random access, A uniform I/O interface must specify a data access model that accommodates these differences yet provides a common abstraction.

In the UIO interface, the data access model is defined in terms of the sequence of data blocks associated with a UIO object. The unit of data access is a *block,* with each block addressed by a (32-bit) *block number.* The maximum size of block is given by the UIO state variable *blocksize.* In the common case of fixed-size blocks, all but the last block are the fixed maximum size. All UIO objects support sequential access. The ability to reset or perform random access is optional. A strictly sequential or *stream* UIO object only allows read operations to the nextblock and write operations to the lastblock+l. These variables are incremented as part of read and write operations in this case. Multi-block read and write operations are also optional. The following paragraphs justify this abstraction.

The use of *block* as the unit of addressing has several advantages. First, the block unit is used to model logical records such as those that arise in structured files, networks and interactive terminals. In the case of structured files, each record corresponds to a block. In the network setting, each network packet is **a** block. With an interactive terminal, each input line corresponds to a block. In each of these cases, a byte-stream abstraction is difficult to implement and obscures record boundaries that may have important semantics to the client [6].

For I/O services without client-meaningful logical records, blocks serve as the unit of transfer between client and server with the client otherwise ignoring the block boundaries. In these cases, the service module picks the size of block and how it corresponds to the underlying implementation to provide a fast and simple implementation. In this vein, most I/O services have some internal form of block as the unit of transfer, buffering or allocation. Examples include memory page, network packet, disk buffer and disk sector. If the block corresponds to this implementation block or multiples thereof, the server module is simplified and server overhead for client requests is reduced. In particular, all client read and write requests start on a block boundary so the server need not implement access to an intermediate point in its buffers. Optionally, the server can limit its service to single block read and write operations, thus making these operations even simpler to implement. The major disadvantage of our approach is that it requires the client to allocate buffer space at least as large as the server's block size. For example, an application using our UIO interface in the V distributed system would have to have at least a 1024-byte buffer to read a file whereas in UNIX, for example, it could read a byte at **a** time from the kernel. However, with the low cost of memory, we do not regard this as a significant disadvantage.

Multi-block reading and writing are defined as optional parts of the interface because, with some services such as file service, it improves performance and is easy to provide, whereas with other services, such as terminal input, they are difficult to provide and of limited performance benefit. If a client attempts to read or write more than the server can support, the server transfers up to the maximum it supports and returns an error indication plus the amount actually read or written. For example, the V file server software supports arbitrary size read and write operations, whereas the the V TCP implementation supports single block (packet) operations only.

Addressing data blocks using a 32-bit block number is simple, efficient and adequate for most I/O services.

[6] **Note that a byte-stream is a subcase of the block sequence model with the blocksize restricted to one byte.**

For example, with a disk file, the block number designates the ordinal number of the block within the file. For a terminal (or any stream) it acts effectively as a sequence number for input records. Moreover, the block number is adequate to handle record-structured files in which records are indexed (or addressed) by string keys using one of the following techniques. When the record file is accessed sequentially, the block number is adequate to enumerate the records, being interpreted as the ordinal number of the desired record. When the file is to be accessed by index key, the CreateUio operation can be used to create a *view* [16] of the file corresponding to the subset of records with the desired key by having the name used in the CreateUio operation specifying the file and key. For example, the name "/dbase/employees/name=smith" in a CreateUio operation specifies that the database manager create as the UIO a view containing all employee records with name key "smith". With this mapping of structured files onto UIO objects as views, an application uses block numbers to randomly access this subset of records as well as to read the view sequentially. This approach effectively transfers the complexity of key-based addressing to the naming facility. Alternatively, the client can access the index and the base set of records as separate UIO objects, with the record identifiers in the index corresponding to the block numbers in the second UIO. That is, for example, if the index gives records 17, 39 and 61 as corresponding to the key "smith", the client reads blocks 17, 39 and 61 from the UIO corresponding to the base set of records to get, these records. This latter approach has the advantage of requiring less mechanism and state in the server.

As further support for structured files, an I/O server may optionally support two operations, InsertUioBlocks and DeleteUioBlocks for inserting new blocks in a file and deleting existing blocks. The DeleteUioBlocks operation also provides the ability to truncate UIOs on arbitrary disk files. The insert and delete operations effectively renumber the resulting block sequence as necessary.

Several other aspects of the data access model merit discussion. First, the data blocks associated with a UIO are addressed as a *single* sequence of blocks because this is adequate for the common cases of random access files and a uni-directional streams, i.e. sequential reading or writing. The most common case not directly supported by the UIO interface is the bi-directional stream, such as a TCP connection [11]. A bi-directional stream can be handled as two uni-directional UIO-based streams. However, there needs to be a way to associate the second UIO with the same underlying bi-directional stream as the first. For example, if a UIO is created corresponding to the writing side of a TCP connection, the second UIO needs to be associated with the reading side of the same TCP connection. Similarly, if a UIO is created corresponding to a window in a multi-window system, the second UIO needs to be associated with the input stream from that window. To achieve these associations, the normal CreateUio operation is used to create the first UIO, corresponding to the stream in one direction. The CreateRelatedUio operation creates the second UIO, taking as argument the first UIO, so that this second UIO is associated with the same stream but in the other direction. The direction of the first stream is either reading or writing, depending on the specified access mode. An implementation may restrict the order and modes used in creating these related UIO objects. For instance, it may require that the writing stream be specified in the initial CreateUio call if the read stream is to be opened at all. In addition, an implementation may effectively have to create both streams in response to the first call (since this is required by some network protocols, for example) and simply create the UIO for the second stream in response to the second call. This scheme generalizes to the creation of multiple related UIO objects.

An alternative scheme for creating multiple related UIO object is to have the first CreateUio operation create the multiple UIOs with a fixed association between the identifier for the first UIO and those of the others. The client can then determine the identifiers of the other UIOs from the first and proceed to use all of them after a single CreateUio operation. For example, the V pipe server implementation creates two UIOs in response to a CreateUio request, the UIOs representing the two ends of a new pipe. [7] The parameters for the output UIO are returned in response to the CreateUio operation. The identifier for the reading end is one greater than that of the writing end so the client can easily determine its uio-id and use it immediately without additional create operations. This approach is best applied when all the related UIO objects are

[7] A single UIO is adequate in theory to represent a pipe since a pipe is a uni-directional stream. However, the V pipe server uses two UIOs to allow the writing end to be released independent of the reading end.

generally required whenever the first one is created, as is true with pipes.

The UIO block model does not explicitly support sparse files, that is files in which large holes of undefined data blocks may exist between the start of the file and the last block in the file. For this case, the nextblock and lastblock fields act as upper and lower bounds on the range of valid blocks. Reading an undefined block causes the ReadUio operation to return an error code indicating the block is undefined. This provision is adequate because most uses of sparse files impose an access structure that guides access to the defined portions of the file, avoiding the holes. It rarely makes sense to read a sparse file sequentially.

Finally, ReadUio and WriteUio specify the block number in each invocation eliminating the need for the server module to maintain a per-client current block number and implement a seek operation to modify this current block number when random access is implemented. (With the UIO interface, the seek operation is instead performed locally in the client run-time library, which also maintains the current block position.) Explicit specification of the block number also allows the ReadUio operation to be handled as *idempotent* because the block number is an absolute address. That is, rereading from block K should have the same effect as the original read operation unless there has been an intervening write. The disadvantage is that it is more difficult to share a file pointer between two different programs, than it is in UNIX for example.

Handling remote operations idempotently is attractive because it reduces the cost of providing for retransmission of the return packets. Without idempotency, the server (at the transport-level) needs to retain copies of all return packets for some timeout period in case they need to be retransmitted. With idempotency, the server can handle the retransmission by simply redoing the request to regenerate the response. The cost of retaining a copy of the return packets can be significant, especially for read operations returning large amounts of data. In particular, the time taken to copy this data is approximately the same as the time to transmit the data, at least on local networks. Thus, in the normal case, idempotency can reduce by half the time to send return packets. Moreover, the cost of redoing an idempotent operation is incurred infrequently because of the low error rates and low delay on local networks. In addition, with disk files, for instance, the file server frequently has the requested blocks in its buffer pool from the original request handling, so redoing the read operation can be handled as a new request at minimal cost, with zero cost imposed on the normal case (when there are no retransmissions.) Given that file reading is often 80 to 90 percent of file activity, the performance benefits of idempotency in this one case alone can justify its support in the design of both the I/O interface and the transport level.

Note that even stream UIOs can be made idempotent to the degree required here with relatively little overhead, recognizing that only the last read ever needs to be retransmitted. To provide for retransmission of the last block read, the server module saves this block in a buffer. A read of block K+1 acts as an acknowledgement of block K, allowing the server to overwrite the buffer containing block K.

In general, we have optimized the UIO data access model for the common cases to achieve efficiency and simplicity, identified techniques for handling less common cases, eliminated the need for per-client state in UIO objects, and allowed for use of efficient idempotent read operations, as described above. [8]

2.2 Representation of I/O State

A uniform I/O interface needs to support *I/O* state, state that is associated with *access* to the file or device, as opposed to state associated with the file or device itself. For example, I/O state arises with conventional disk-based files as file and record locks. Similarly, the temporary versions of a file created prior to atomic update or abort are I/O state. A writer updates a temporary version of the file that logically only becomes part of the underlying file after the commit point. Similarly, a reader may see a snapshot of the file that

[8] **Idempotency, while contributing to system efficiency, has disadvantages in the reliability and error reporting in the system. For example, if a server does not implement idempotency properly, it may only show up as a timing-dependent problem under particular system load, not during the testing of the server. Also, with an idempotent ReadUio operation on a stream, the error of reading repeatedly from the same block is not reported, even though it would cause a failure in a non-idempotent implementation. However, we see idempotency as an essentid technique for performance, especially when our general I/O design must stand up to the performance comparisons with more specialized file access protocol such as used in Locus [35].**

remains unchanged despite updates made to the file subsequent to the reader opening the file. Examples of I/O state in other I/O services include:

- buffering and synchronization associated with pipes [29].
- screen coordinates, border, size and contents associated with a window or virtual terminal in a multi-window system.
- protocol state associated with a network connection.
- status and contents associated with a printer spool file, all of which are created as a result of "opening" the printer.

In each of these examples, the state is created when the entity is opened and typically discarded (in some service-specific way) when it is closed. For example, the spool file is logically appended to physical printer on close whereas the network connection state is just discarded (after a timeout period). Similarly, all the state associated with a pipe is created when the pipe is opened and discarded when it is closed since a, pipe only exists when it is open.

In the UIO interface, the I/O state maintained by the server is represented by the UIO object. The CreateUio operation provides for creation and initialization of this state. For example, a CreateUio operation specifying a TCP connection allocates a descriptor and initializes the protocol state required to setup a TCP connection as well as the generic UIO state. In general, the ReleaseUio operation allows release and reclamation of this state, as well as service-specific actions associated with the UIO. For example, ReleaseUio on a disk file may cause any associated modified buffers in the server to be written to disk.

The use of the UIO object for I/O state has several advantages. First, it is general enough to handle all the different I/O services we have considered. Second, it defines a uniform client-visible view of the I/O state plus provides a basis for handling service-specific state. That is, each server can store additional state in its UIO records and clients can access that state using the UioControl procedure. Finally, the UIO interface separates the I/O state representation from the communication mechanism. This is essential for systems using RPC as the communication abstraction because RPCs do not support state between calls (at least from a user's standpoint). Although a virtual circuit-based transport level might directly support open file connections with one virtual circuit per open file, this approach is expensive at the transport level in the number of connections required and does not eliminate the need for state at the I/O level for every open file. Multiplexing multiple open file connections on a single transport-level connection introduces complexity that eliminates many of the benefits of virtual circuits without eliminating the virtual circuit overhead. For instance, multiplexing on top of virtual circuits means that separate flow control is required for the multiplexed streams and that the virtual circuit level flow control is useless and perhaps even a liability.

Despite the simplification offered by stateless interfaces such as used in WFS [31], state is necessary for the generality required of a widely applicable uniform I/O interface. For example, it does not appear possible to handle multi-window systems, pipes, network connection and numerous other I/O services in a stateless interface. The WFS design, while simple and elegant, only handled simple disk files and had no provision of other devices and I/O services.

2.3 I/O State Management

The provision of state in the I/O interface introduces the need to manage this state, specifically: protect the state from unauthorized access, account for the costs of maintaining it, and reclaim the resources associated with the state at some point. The I/O state needs to be protected since any process can attempt to invoke an operation on any UIO simply by specifying the UIO's identifier. In contrast, a virtual circuit or connection model can treat the connection as a *communication capability* as done in DEMOS [2] or Accent [27] to control unauthorized access. Similarly, the server module must have some criteria for determining that it can discard the I/O state because the RPC model provides no transport-level notification that a client has terminated, unlike that present in most virtual circuit-based communication systems.

To deal with these issues, each UIO has an associated *UIO owner* process or task, *authorization domain* and access **permissions** recorded as part of the UIO state. The initial owner of a UIO is the process that requested its creation. The initial authorization domain is that associated with the initial owner. Initial permissions are those required by the **CreateUio** access mode, namely read if the access mode is READ else read and write.

On each UIO operation, the server checks that the requesting process is in the authorization domain associated with the UIO and that the operation is consistent with the permissions, (or that the process is associated with the authorization domain of the UIO owner). If not, the operation is terminated with the error code NO-PERMISSION. The **SetUioPermissions** operation is used to modify the initial authorization domain and permissions as required. Only processes in the same authorization domain as the owner of the UIO or the owner of the underlying file can change the permissions. Moreover, they are restricted from giving greater access to the data than provided during the original UIO creation or provided by the underlying file. Using SetUioPermissions, the UIO owner can give a process access to **a** particular UIO without providing the process or its associated authorization domain with access to the underlying file. For example, a command interpreter can pass open files to a program run in a restricted authorization domain that would not be able to open the files itself. In general, a process can only get greater access to a file than authorized by the file protection by being passed a UIO with such access by **a** process that is so authorized.

The UIO interface bases accounting and reclamation on the UIO owner. The cost of maintaining the UIO state, if accounted for, is charged to the account associated with the **UIO** owner. The UIO state is automatically reclaimed by the server if the owner process is determined to be invalid.

The **SetUioOwner** operation is used to transfer ownership to another process as well as optionally change the authorization domain and permissions. A common case in which this occurs is command execution, in which the command interpreter opens the input and output files and then passes these open files to the executed program. Using the UIO interface, the ownership of the **UIOs** associated with these open files is transferred to the executed program. This ensures that these open files are automatically reclaimed (or closed) if the program terminates abnormally without explicitly closing these files, and that the use of the open files is charged to the same account **as** that used for the program execution. It also gives the program access to the UIO even though it may not be authorized to open the associated file itself.

This design has several advantages. First, it is simple and efficient for the server to implement, since it need only record three fields for each UIO, the owner, the authorization domain and permissions. The code to check permissions against the authorization domain and permissions field is simple, as is the code to check the validity of the owner. [9] It is also quite efficient, given that the client is normally the UIO owner. Second, it provides a means to control access to the UIO as desired yet allows access to be effectively passed to a process without providing access to the underlying file. Third, it does not require communication with the server to establish access except when the UIO is created or its ownership or access control is changed. For example, a command interpreter can pass access to a UIO to a newly created program without contacting the I/O server simply by passing information on the UIO object to the new program (assuming the new program is executed with access to the UIO object). Finally, it does not rely on the communication mechanism for protection or notification of abnormal termination. As noted earlier, the remote procedure call abstraction does not generally provide either, thus simplifying the transport level.

2.4 Compulsory and Optional I/O Functionality

To provide a uniform interface without restricting the interface to the lowest common denominator service, the UIO interface defines certain functionality as optional. For example, not all servers implement **InsertUioBlocks** and **DeleteUioBlocks**. [10] The UIO interface effectively defines an interface that is a **superset** of the

[9] Since the validity check requires network communication plus retransmission and timeout if the owner node has crashed, several V servers do the owner validity checking in a separate helper process. This is relatively easy to do in V but might be a greater complication in a system without lightweight processes.

[10] Alternatively, these operations can be viewed as implemented uniformly across all servers since each operation is defined to either perform a particular action or else return an error code indicating the reason. One possible error code is OPERA-

functionality naturally available from any of the I/O services we have considered.

The basic compulsory functionality of the UIO interface includes implementation of CreateUio, QueryUio, ReleaseUio, ReadUio, WriteUio, SetUioOwner, SetUioPermissions and UioControl. With these procedures, a service must support creation of at least one UIO object that can be either read or written sequentially and released. Thus, the base level functionality is a readable or writeable stream. For such UIO objects. the run-time library knows to write to the next block number each time it writes out a block, such as when flushing terminal output. For randomly accessible UIO objects, the current block is filled until it reaches the block size, independent of the flush operations executed by the application. [11] Without this attribute being explicitly communicated, the run-time library would have to discover that the UIO was a stream by having a write operation fail due to attempted non-sequential access.

As a final part of the compulsory functionality, the service must return standard error codes in response to requests for functionality it does not support or problems with requests for functionality it does support. For example, MODE,NOTSUPPORTED must be returned in response to a CreateUio operation that specifies a mode that is not supported by this module. This standardization allows error returns to be interpreted by the I/O run-time in some cases, particularly in dealing with optional aspects of the UIO interface. For example, a client that requests to read more data than the maximum amount supported by the server on a multi-block read is guaranteed to receive the error return BAD-BYTE-COUNT on return. Thus, the run-time routines can detect this case easily and adjust the amount read or write accordingly, and not interpret this error return 'as more serious than it is.

Beyond this level of functionality, the generic optional functionality available in each UIO object is specified by its UIO attributes. The rest of this section describes some of the UIO attributes and their use. Most of these are primarily motivated by the needs of I/O run-time library in implementing conventional byte-stream I/O.

RESETTABLE : The stream can be reread or rewritten starting from the beginning, by simply starting at block 0. Otherwise, access must be sequential.

RANDOM-ACCESS : The data blocks can be read and written (depending on the access mode) in any order. RANDOMACCESS subsumes RESETTABLE and strictly sequential access.

STORAGE : Reading a block returns the last data written to that block. That is, it has *storage* semantics. In particular, a block that is reread returns the same data as the previous read if there were no intervening writes. A UIO with the RANDOMACCESS attribute may allow random access reading yet not provide storage semantics. For example, the memory of a frame grabber attached to a video camera may allow random access reading yet the contents of the frame grabber may change on each new frame, independent of UIO activity.

FIXED-LENGTH : The length of the UIO is fixed to that specified by the lastblock and lastbyte fields. This is true of a disk file accessed for reading as well as a physical device such as the disk itself independent of access mode. It is not necessarily true for stream I/O where the end may not be known a priori, and it is not true for disk files open for writing, given that they may be extended by writing past the last block. This attribute allows the run-time library to determine if it has reached the end -of file without an additional read operation past the end of file in common cases like reading disk files 'while still allowing on-going reading of unbounded streams such as terminal input. For a UIO object without the FIXED-LENGTH attribute, the lastblock and lastbyte fields indicate the current end of the available data blocks. For example, with a UIO object representing the input portion of network connection, the last block indicates the last block queued to be read from that connection and thus how many blocks can be read without blocking for additional data. The semantics are similar for the

---

TION_NOT_SUPPORTED.

[11] The flush operations referred to here are client I/O run-time flush routines that simply write any modified data in the local I/O buffer to the associated UIO object. Flushing as part of the UIO interface is provided as part of the ReleaseUio operation, as described later.

read end of a pipe. When writing a **non-FIXED_LENGTH** UIO object, a data block written past the end of the last block is appended, making it the new last block. (This updated information can be read from the server at any time using the QueryUio operation.)

MULTI-BLOCK : The server supports multi-block reading and writing. The maximum amount supported is not specified and is allowed to change between requests.

VARIABLE-BLOCK : Blocks shorter than the full block size can be returned in response to read operations other than due to end-of-file or other exception conditions. Also, writing blocks less than the UIO blocksize is handled properly.

INSERTDELETE : The **InsertUioBlocks** and **DeleteUioBlocks** operations are supported.

LOCKED, **INTENT_LOCKED** : The UIO has a file-level lock or intention lock, respectively. A UIO that is *intent-locked* supports block-level locking, as described in Section 2.6.

SNAPSHOT : The UIO is a consistent snapshot of the data associated with this UIO at the time of the UIO was created and will not change except by operations on this UIO.

RECOVERABLE : The UIO allows changes to be aborted as well as atomically committed and retained independent of failure, as required for atomic update. This is only applicable if the UIO is WRITE-ABLE.

REPLICATED : The underlying file for this UIO is replicated across multiple servers. Section 2.8 describes how this is handled.

INTERACTIVE : The UIO represents a text line-oriented input stream on which a break or attention handler can be defined (by **SetBreakProcess**). It has the connotation of supplying interactively (human) generated input. This allows programs to tailor their behavior for interaction and for batch operation.

Other attributes such as READABLE, WRITEABLE and APPEND-ONLY have obvious semantics. Clearly, a large number of the combinations are not meaningful and therefore not used.

Some of the UIO attributes are known to the client creating the UIO by the nature of its request. For example, if a UIO is created in READ mode, the UIO is READABLE unless an error is returned. These attributes are included in those specified by the server so the UIO attributes returned by a QueryUio operation are complete. Thus, a client, having been passed a UIO identifier can determine the same information as available to the original creator by querying the server (using QueryUio).

2.5 Specification of the Desired I/O Service

For the I/O interface to be truly uniform, there needs to be a uniform way of specifying the desired I/O service despite the choice of a wide diversity of different types of I/O services with many different parameters and options. For instance, with a network service, there are various different protocols, types of service and destination endpoints, With a printer, there may be different printers, types of forms, priority and so on. With a multi-window system, there may be different window sizes, data representations and formats. In contrast, most applications simply take a string-name for a file, open the file and proceed to use the file. fully ignorant of specialized options.

In the UIO interface, a new UIO object is specified by the parameters to the **CreateUio** operation. namely: a character-string *name* and *context* in which to interpret the name. [12] All specialized options and parameters for a new UIO object are handled in one of three ways. First, the string name (and context identifier) can be used to refer to a directory entry which specifies the details of the UIO, including these

[12] **The access mode is used only to specify variations of read or write access, not further aspects of the new UIO object. Similarly, the transaction identifier is only used to associate the new UIO object with a transaction.**

options. For example, the directory entry "/printers/mailinglabels" may specify all the information required to direct output to a printer with the options for correctly printing mailing labels. This approach is used in UNIX. For example, the indication of whether an open file representing access to a tape drive should use 800, 1600 or 6250 BPI is specified in the file name used in opening the file. The second way is for the name to be an encoding of the required information. For example, "/net/tcp/36.8.0.8" describes a protocol and a host address to use for creating a network connection corresponding to the requested UIO. Finally, the UioControl operation can be used to set or modify various service-specific parameters controlling the UIO object after it has been created. This is only applicable to cases in which it is feasible to set these parameters and options after the UIO object has been created and the application knows to set these parameters.

Conventional disk files can be regarded as an example of the first technique. The directory specifies the file properties and contents [13] which are then associated with the UIO. For example, "/bin/emacs" names a directory entry that describes the data blocks to be associated with a UIO created using this name. It also implicitly specifies the access procedures to be associated with the UIO operations, namely the standard ones incorporated into the file system. In general, the directory system can be regarded as a "dictionary" that defines the attributes associated with a particular name. The CreateUio operation then creates a UIO object matching the dictionary definition.

The use of a name parameter for UIO specification allows the CreateUio operation to be uniform across all objects and forces the non-uniformity associated with different specific types of objects to be handled in the name definition facility. [14] Since the name definition facility is used far less widely and less frequently than the I/O access mechanisms, this simplifies the design of client applications and does not detract from performance.

This design assumes that either the I/O servers implement the portion of the name space corresponding to the objects they implement or else there is a private interface between the directory system and the I/O server, not visible to the client. That is, either "/printers/mailinglabels" is interpreted by the printer spooling when used in a CreateUio operation or else the directory system must pass all the information associated with this name to the printer spooler. The former approach seems preferred because it eliminates the need for these private interfaces, allows servers to specialize their directory information, distributes the naming interpretation overhead, and requires less network overhead compared to when the directory system and I/O server are not co-resident. This is especially true when the name references are largely local, which tend to occur in systems providing a *current working* directory mechanism. However, it does mean that every server needs to manage a portion of the (global) name directory. This decentralized approach to naming is used in the V distributed operating system [7].

2.6 Locking

Lock support is included as part of the UIO interface, instead of as a separate service, for several reasons. First, the I/O server is the most efficient point to locate lock management because it may be the only common point of communication between two clients that need to synchronize. Second, locking and unlocking can be handled as side-effects of I/O operations, minimizing the communication overhead for locking. For example, a record can be locked as a side-effect of the read operation. Third, incorporating locking as part of the I/O interface allows one to define interactions between locked and unlocked access to a file. For example, an exclusive lock on a file may inhibit all other updates to the file data, whether or not these other accesses request locks. Finally, locking and recoverability are tightly coupled to I/O atomic transactions are being implemented. Since recovery management is most efficiently located as part of the I/O server [15], locking seems best located there as well.

[13] This information is stored either directly in the directory entry or indirectly by a pointer in the directory entry to a separately stored file descriptor record, as in the UNIX file system [34].

[14] This is similar to the UIO handling of string-keyed record files, as described in Section 2.1.

[15] This comment applies to recovery using logging or shadow pages. We do not consider the use of network logging of communication [26].

	File Lock Mode				
Request	***IShare***	Share	***IExcl***	***Excl***	***Share-IExcl***
Share	OK	OK	NO	*NO*	*NO*
IShare	OK	OK	OK	*NO*	OK
Excl	NO	NO	NO	*NO*	NO
IExcl	OK	NO	OK	*NO*	NO
Share-IExcl	OK	NO	NO	*NO*	NO
Block-Share	OK	NO	OK	*NO*	NO
Block-Excl	NO	NO	OK	*NO*	OK

Table 1: Lock Compatibility Table

The UIO interface supports both file and block-level locking. The CreateUio operation includes two flags, LOCK and INTENTIONLOCK, that can be set in the mode parameter. When the LOCK flag alone is set, the file is locked on open according to the access mode.

READ : Share lock. Other CreateUio operations are allowed only in READ mode, locked or otherwise.

CREATE, MODIFY : Exclusive lock. No other CreateUio operations are allowed other than in READ non-locked mode. (Unlocked read access is used here to examine updates in progress and previous versions of the file.)

APPEND : Share lock on existing file data and exclusive lock on appended data. Thus, the existing data, but not the newly appended data, can be read by other clients and no other writing is allowed on the file.

If the server does not support file-level locking, it returns an error message in response to the CreateUio.

A file-level lock restricts access to the same file according to the standard share/exclusive lock compatibility. The notion of what is the "same file" is not defined in the protocol. However, intuitively, file-level locking in CreateUio locks the data associated with the UIO to achieve the expected semantics of shared and exclusive locks. For example, if the UIO represents a view of some portion of the database, then that portion of the database is locked even though it may span multiple physical files and relations. Beyond this basic semantics, I/O servers are free to implement *type-specific* locking [30].

The INTENTION-LOCK flag indicates the intention to do block-level locking of the file. If the server does not support block-level locking, it returns the appropriate error code. With the INTENTION-COCK flag set, the file is intention *locked* for sharing if the access mode is READ; otherwise an exclusive intention lock is acquired. The intention locks on the file prevent the acquisition of a file-level lock that would interfere with the modes of block-level locks that the client intends to acquire on this file and forces all clients that are allowed to access the file to contend at the block level for locks. If both the LOCK and INTENTION-LOCK flags are set, the file is locked with the shared lock as well as the intention lock [16]. Table 1 shows the compatibility between file-level intention locks on the file and requests for both file-level and block-level locks, These compatibility rules are similar to that used in System-R [16]. In this table, IShare, IExcl and Share-IExcl are the intention-shared, intention-exclusive and shared intention-exclusive modes respectively. Intention locks are only used in READ and MODIFY modes. Use of the INTENTION-LOCK with the CREATE or APPEND modes is an error.

Locks only provide synchronization between two different UIOs. Locks on the same UIO are always compatible. Duplicate lock requests on the same UIO are ignored without error indication.

The ReadUio operation on an intention-locked UIO acquires a share lock on each block it reads. Similarly, WriteUio acquires an exclusive lock on each block it writes, with no writing taking place if it fails to acquire

[16] **The two flags are only used in MODIFY mode.**

the lock(s). Locking that does not fit this behavior must be performed explicitly using the LockUio operation to set locks on specific blocks of the UIO data blocks. In particular, LockUio is used to convert a share lock to an exclusive lock if that is needed before attempting the write.

All locks are released when the UIO is released, minimizing communication overhead for the common case of two-phase locking [12]. In addition, the *mode* parameter in ReleaseUio allows one to release the block-level locks associated with UIO without releasing the UIO itself. UnlockUio is also provided to unlock specific blocks. However, it should be used with care since it unlocks a block independent of the number of times the block was locked. In general, it is safest to use ReleaseUio for all unlocking nd to unlock only at the point the client is completely finished with the UIO, thereby avoiding cascading aborts, premature unlocking and the communication overhead of selective unlocking.

When a client requests a create, read, or write operation that is incompatible with a lock held by another process, the operation is suspended for a maximum of the *lockwaittime* milliseconds, returning the error code LOCKED if the incompatible locks are not released within this time period. The UIO instance variable lockwaittime is set using the operation *Set UioLockWaitTime*. Using timeouts when waiting for locks avoids the need for a global deadlock detection mechanism across all UIO servers. Placing timeouts on waiting for locks rather than on the locks themselves (so they timeout after some period of inactivity) avoids requiring the client to periodically interact with the server to prevent lock timeout. Note that, if the client crashes or is deleted, the server detects the owner of the UIO to be invalid and the locks it holds are released when the UIO is released.

The point of this section is to show how locking can be incorporated into a uniform I/O abstraction, not to present a new form of locking. We claim that this locking abstraction satisfies three major requirements, namely: (1) it minimizes communication overhead for locking, especially in the common two-phase locking paradigm, (2) it is easily implementable by the server module, and (3) it provides an adequate basis for client synchronization, including atomic transactions. The next section shows how the UIO interface provides for atomic transactions.

2.7 Atomic Transaction Support

The atomic transaction interface is an important abstraction of I/O access when an update must be indivisible with respect to failures and other **accesses.** However, an atomic transaction protocol is logically separate from the I/O interface since atomic transactions can include non-I/O operations and atomic transaction operations operate on transactions and only indirectly on UIO objects. Thus, the UIO interface specifies a means of including UIO operations in an atomic transaction (providing the server supports the atomic transaction interface), but not the atomic transaction protocol itself,

In the UIO interface, a new UIO is created as part of a transaction by specifying a non-zero transaction identifier as part of the CreateUio operation. All subsequent operations on this UIO are considered as part of the specified transaction. An I/O server only accepts a CreateUio operation with a non-zero transaction identifier if it supports the system atomic transaction protocol. Otherwise the operation returns an error message. (The atomic transaction protocol in V consists of the operations CreateTransaction, PrepareTo-Commit, CommitTransaction and AbortTransaction as well as various query and management operations.) This approach allows UIO objects to be grouped into transactions with no additional operations. It also allows servers to do a simple check at creation time to see whether or not the UIO object needs to be included in a transaction.

Normally, a UIO that is part of a transaction is either locked or intention-locked. Without locking specified, the server may presume that the client is handling concurrency control separately and need only provide recoverability. A lock request on a UIO associated with one transaction may conflict with a lock held as part of another UIO even though it is part of the same transaction. This prevents a transaction from unknowingly performing conflicting updates.

The CreateUioVersion operation is provided to support nested subtransactions. It creates a UIO object corresponding to the current state of another UIO object. When the newly created UIO object is committed,

its changes are incorporated into the UIO object from which it was derived. If aborted, the changes to the new UIO object do not affect the original one. Using CreateUioVersion, a subtransaction can abort without undoing changes performed by its parent transaction.

2.8 Replication

Files are replicated for fault-tolerance and improved performance in a distributed system. A uniform I/O interface should provide access to replicated files (and other services) in a uniform way so that replicated objects are easy to access and the benefits of replication are available.

The UIO interface uses the read-any/write-all approach with the modification that, "all" is defined as the set of available *sites* [15]. All sites maintain a common notion of the set of available sites.

When a UIO object corresponding to a replicated object is created, it has the UIO attribute REPLICATED. A UIO object created for READ access is handled by one of the available sites picked by the name mapping mechanism, ideally the closest or least loaded server. Subsequent operations on the UIO proceed the same as for the non-replicated case. If this site fails, the I/O run-time reopens the file at another site. Thus, the reading a replicated file using the UIO interface takes advantage of the replication of the file for both performance and fault-tolerance.

When a UIO object is created for MODIFY, CREATE and APPEND access, the server randomly selected by the naming mechanism returns a uio-id representing the set of available copies plus a list of the available servers. In V, for example, the uio-id is represented as the identifier for the process group [9] of available servers plus a group-wide unique local UIO identifier. Each update operation is multicast to the group of servers, with the client collecting responses from each server to acknowledge the operation, and thus ensuring that the multicast was reliable. (In the absence of multicast, the list of available servers is used to make a series of identical calls, one to each server, to simulate multicast.) Exclusive lock operations are handled similarly. Read, shared lock and query operations can be directed to an individual server.

The I/O run-time routines have mechanisms for handling replicated access, making it transparent to applications that a file is replicated. In particular, the run-time routines handle the reliable multicast write to the available sites with no change to the client interface.

To simplify failure modes, the update of a replicated file is best done as part of an atomic transaction given that a lock conflict, server failure or communication problem at any point can endanger the consistency of the replicated copies. In general, there needs to be a means to return all replicas to a consistent state if a problem arises.

This design has the advantage (associated with read-any approach) of fast and simple read access. In addition, assuming reasonable multicast support at the RPC or IPC level, this design allows efficient update since the transmission of a write is done in one IPC operation, assuming packets are not lost. Moreover, the handling of replication is transparent to the application level. Recovery from failures in the case of reading is transparent to the application except for the delay in reopening the file. Failure during writing is just another reason for transaction abort.

2.9 Mapped I/O

Mapped I/O normally refers to the binding of disk files into a virtual address space such that a reference to that region of the virtual space is effectively a reference to the corresponding portion of the file. Mapped I/O in this form allows the application to access data in files the same as data in memory. It also takes advantage of the read-on-demand and write-if-modified facility of a demand-paged virtual memory mechanism, which is important for random but "sparse" modifications to large files. That is, on first reference to a page in such a bound region, the page fault handler copies the appropriate file data into a page frame and maps the page frame into this portion of the virtual address space.

Mapped I/O can be provided in conjunction with the UIO interface by a binding mechanism that binds the data blocks of a UIO to virtual addresses. The V mechanism for this binding is implemented by

BindRegion(pid, addr, len, uio, startblock, permissions)

which binds the specified UIO starting at *startblock* to the process address space, starting at *addr* for *len* bytes. Any UIO can be so bound provided that it has the READABLE, WRITEABLE and STORAGE attributes and not the VARIABLE-BLOCK attribute and that it supports reading and writing of data of the size of the virtual memory page size. Thus, the UIO interface supports mapped I/O by providing an object to bind and attributes with that object to indicate whether it can support mapped I/O, using to the criteria above.

Another form of mapped I/O arises as a performance optimization such as that used in Accent [13]. Data requested in a conventional read operation is mapped into the address space instead of copying it providing it satisfies alignment restrictions. (Writing is handled in a similar way.) The UIO interface does not provide any explicit support for this optimization, relying on the run-time library to page-align buffers when possible and the I/O services and virtual memory mechanism to take advantage of this optimization when feasible.

2.10 Handling Specialized Operations and Options

Many I/O facilities require service-specific operations as well as specialized options that, for example, modify the behavior of read and write operations. For example, a multi-window system requires a service-specific operation for changing the mapping of a window. Similarly, the TCP specification [11] calls for a "push" flag and an "urgent" flag to be passed on each write operation. Also, a terminal read operation could use flags indicating whether the input is be to echoed and line-edited.

To handle these specialized operations and options, the UIO interface defines the UioControl operation. The exact parameters and their interpretation is specific to each I/O server although there is some standardization on the format, such as the initial parameter being an operation code. For options that act on a read or write operation, it is necessary to set them using a UioControl operation before issuing the read or write operation (and potentially reset them afterwards). Although this requires additional operations over including the options as part of the read or write operations, several techniques minimize the overhead. First, the default setting of the options can be selected to the most commonly used values so the use of these operations is minimized. Second, options that are typically set or cleared for a single operation can be automatically reset after the next operation, thereby eliminating the need for an explicit reset operation. For instance, in getting a password, echoing is normally turned off for just the next read operation. Thus, the server can automatically reset the echo option after a read operation. (There can also be a way to turn off echo without autoreset.) With these techniques, we have yet to encounter a case in which the UioControl operations constitute a significant overhead.

An alternative approach would be to provide a set of option flags as part of the ReadUio and WriteUio operations. This approach has several disadvantages. First, it is not feasible to provide sufficient flags and option fields to support all possible options so some modules would require the above solution anyway. Second, if a given option field had a different interpretation depending on the UIO server module that handled it, the I/O interface would no longer be uniform. For instance, there is no sensible meaning to a "urgent flag" on a tape drive yet there is the temptation to overload option flags in order to handle the many different options that arise. In general, option flags on the main UIO operations seem like a poor idea. The UIO interface has survived for over 5 years without reading and writing options despite numerous challenges along the way.

To summarize this section, we have presented the key design issues in the UIO interface and the rationale behind our design decisions. In doing so, we have tried to cover the issues that would arise with the design of any such a uniform I/O interface, particularly in an RPC-based distributed system. The combination of this section plus the Appendix should provide the reader with a fairly complete description of the UIO interface. We now go on to describe our experience with using and implementing the UIO interface.

3 The UIO Interface in the V Distributed Operating System

The UIO interface is an abstraction of I/O services. The real test of any abstraction is how useful it is and how implementable it is. The UIO interface has been used in the V distributed operating system for access to all I/O-like services. In this section, we describe our experience with its use and implementation over the past five years. We also draw on experience with an earlier version of the interface [4] that was used in a version of Thoth [5] starting in 1980.

The V distributed system is based on the V kernel [6], a communication-oriented kernel that supports an efficient request-response form of message passing tuned for remote procedure call behavior. The rest of the system is structured as a set of server modules running at the process level plus application run-time libraries that provides a standard application interface to system services. In particular, V has a standard client I/O run-time library which provides the C "stdio" interface as an application I/O abstraction on top of the UIO interface. The UIO interface is mapped onto V messages according to the (so-called) V I/O protocol. [17]

The following subsections describe the I/O server modules we have implemented, the client I/O run-time library and the performance of these components.

3.1 I/O Services using the UIO Interface

In V, all services that are sufficiently close to the I/O paradigm use the UIO interface. Examples of servers implementing the UIO interface currently in use in the V-System include:

Storage : A standard UNIX-like hierarchical file server [8] based on the Thoth file system [5], with some extensions for atomic update. We are currently extending this server to support the complete atomic transaction facility provided in the UIO interface.

. **Pipe** : Implements UNIX-like pipes, i.e. a temporary file on which reading and writing by separate processes are synchronized. The two UIO objects representing the two ends of the pipe have UIO attributes VARIABLEBLOCK and READABLE or WRITEABLE. Zwaenepoel [36] gives a more extensive report on this server and its performance.

Internet : Implements TCP connections as well as UDP and IP services. Access to one end of a TCP connection is provided as two UIO objects representing the read and write portions of the connection.

Display : Provides multiple overlapping windows and views on a single bitmap display. Each output window is a single UIO. **An** input UIO can be optionally associated with a window. It uses the kernel device server and mapped I/O to access the frame buffer. Further details on this server are described elsewhere [23].

Device : Implements UIO access to the serial line, frame buffer, network interface (6 different types of Ethernet interfaces), disks, tape and mouse. Servers such as the display server, storage and internet server use this device interface to implement their higher-level abstractions.

Printer : A spooled printing server, which uses the file service of the storage server to temporarily store spooled data and the device server to access the raw printer.

UNIX service : A UNIX user-level server program that implements the UIO interface to UNIX files as well as the V inter-kernel protocol, allowing file (and program execution) services of a UNIX system to be access from the network using the V protocols.

"The V I/O protocol refers to the UIO interface described in this paper in terms of procedures plus the mapping of these procedures onto messages. This mapping should be considered as part of a general presentation-level remote procedure call protocol, in the ideal, automated by a remote procedure call stub generator. Thus, the V I/O protocol is not of direct interest beyond the UIO interface described in this paper.

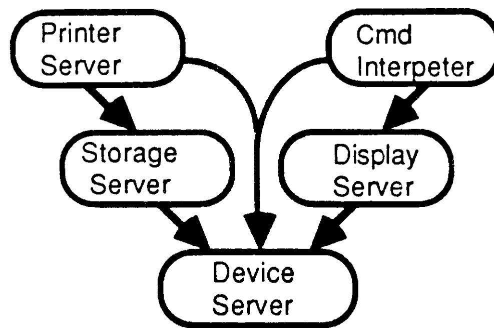

Figure 4: UIO Server Dependencies

In addition, a variety of servers use the UIO interface to provide access to a logical directory of information about their state or the objects they are managing. Examples include the display server and its directory of current windows, the program manager and its directory of executing programs, and Internet server with its directory of connections.

Many services implement additional operations beyond the UIO operations. In fact, the main focus of the service may lie outside the I/O paradigm. For example, the primary role of V program manager [33] (also referred to as the *team server)* is to initiate and manage programs in execution. It implements the UIO interface as a means for client, to read information about the currently executing programs.

A UIO server may also be a client to other UIO servers, allowing a hierarchical structuring of UIO implementations. For example, the printer server implements printer UIO objects using UIO objects of *provided by the storage server that in turn implements its UIO objects in terms of the disk UIO objects provided by the kernel device server. 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.

Most V services are implemented as one or more dedicated processes executing a service module, and responding to client request messages. However, implementing these servers using RPC framework should not significantly affect the implementation of the UIO interface or the performance, assuming efficient stub routines.

In our experience, the UIO interface has worked out well for this diversity of I/O services. The interface is simple enough that it does not entail a significant amount of extra code compared to using a specialized interface for each service. Moreover, the existence of multiple servers for the UIO interface provides both a model and a source of software for the programmer when implementing a new server, minimizing the programming cost of each new server. [18]

In addition, we have implemented a command interpreter that provides much of the functionality of the UNIX shell including I/O redirection and pipelining. With its sophisticated manipulation of I/O, a command interpreter is a good test of any I/O interface. In our experience to date, the UIO interface is adequate in this area.

Finally, our preliminary experience with the UIO interface for mapped I/O in conjunction with a virtual memory mapping mechanism is also favorable. We regard this form of I/O as an important facility for efficient access to databases and so-called *persistent memory* [32].

In general, the UIO interface appears more than adequate to cover a wide range of I/O services, certainly the ones we have encountered to date. With the addition of each new server, some design time is required to map the services into the UIO interface. Inevitably, there is the temptation to add new operations or options.

[18] With an adequate stub generator, the sharing of code for parameter handling and invocation would be less relevant. However, the shared code for handling the UIO interface would remain.

However, the interface, particularly the ReadUio and WriteUio operations has remained fairly stable despite these pressures. Moreover, we have developed a better understanding of how to accommodate diversity of services in the interface with experience. Finally, with the established functionality of the client I/O library and the growing number of applications that use this library, the benefits of conforming to the UIO interface are substantial.

3.2 Client I/O Run-time Library

The client I/O run-time library supports the C stdio interface [22] plus various extensions for block-oriented I/O. It has also been used to support Pascal I/O and could easily be modified to support other similar I/O interfaces, including that defined for Ada. In general, the library provides a conventional byte-stream and block I/O interface with only minor semantic and functional differences from conventional program environments. Thus, applications are independent of the underlying UIO interface.

The basic I/O library for byte stream and block I/O is 997 lines of C code, of which 309 lines is header information, that is message formats and manifest declarations. This compiles to 1696 bytes on the VAX. An additional 1082 lines of code implements the C stdio interface, compiling to 4920 bytes. Most of the latter code is common to that required in a more conventional system such as UNIX. For example, over 3000 bytes of this code is for the ***printf*** and ***scanf*** routines, which are built on the byte-stream mechanism and thus independent of the use of the UIO interface. The library routines are machine-independent, currently running on both the VAX and Motorola 68000 architectures.

Each local file structure requires 48 bytes of storage for local state information plus ***blocksize*** bytes for the local buffer if used in byte-oriented mode. The number of open files an application may use is limited only by the amount of memory it has for these descriptors, although an individual server may limit the number of UIO objects it supports at any time.

In general, the space cost of supporting conventional I/O abstractions on top of the system-level UIO interface is modest. In the next section, the performance cost is also shown to be reasonable.

3.3 Performance of the V I/O System

The performance of V I/O can be decomposed into three components: the performance of the client I/O run-time library, the performance of the communication facilities between clients and the I/O service modules, and the performance of the I/O service modules themselves. The performance of the V communication facilities has been studied and reported elsewhere [9,10]. We also list relevant IPC times with our measurements given below. In our measurements, we determine the cost of byte-stream and block-level I/O to a server incorporating the cost of the UIO interface, but no service-specific overheads such as disk overhead. All measurements were made on Microvax II's running the V distributed system and include the costs for byte stream and block-level reading to a local and a remote server. A local server executes on the same machine as the client. A remote server executes on a separate machine, connected to the client machine by 10 Mb Ethernet.

The client I/O run-time supports two abstractions: the byte stream and the block-level. The basic cost of the byte stream over block-level I/O is the cost of reading a byte at a time from a local buffer and checking if the buffer is empty or writing a byte at a time to a local buffer, checking if the buffer is full. When the local buffer is empty or full respectively, block level I/O is used to fill or empty it as appropriate.

Table 2 gives the performance for a client reading from the V storage server using the standard V I/O library. The V storager server [8] is a disk-based file server that supports a UNIX-like file system. The get-byte column lists the basic processing time for calling the V version of the UNIX getc to return 1024 bytes, not including any filling or flushing of the local buffer. (This measurement works out to 160.5 kilobytes per second or 6.08 microseconds per byte.) The next two columns give the elapsed time per kilobyte to read using byte streams and blocks from the V storage server. Because of the way that the experiment was conducted, all the data being read is resident in the file server buffer pool so our measurements do not include disk access overhead. Otherwise, the time is indicative of the cost of file access in V. We chose to

Server Location	get-byte	Disk (bytes)	Disk (blocks)	IPC
Local	6.23	9.91	3.47	1.79
Remote	6.23	14.63	8.18	6.34

Table 2: Reading: Elapsed Time per Kilobyte on Microvax II (in milliseconds)

Server Location	Byte Stream	Block I/O
Local	19.1	48.4
Remote	14.1	22.11

Table 3: UIO Overhead as a Percentage of Elapsed Time

factor out disk access costs because, with a large buffer pool (8 megabytes on our current file server), disk access is infrequent in general and, with use of increasingly large buffer caches, the effect of disk overhead can be expected to decrease even further [25]. Also, it is difficult to interpret measurements incorporating disk overhead, given the variations on seek times, rotational latency and disk transfer rate. The IPC column gives the **elapsed** time for the basic IPC primitives used as part of the I/O to the file system for each row in the table.

The cost of getting 1024 bytes from a file a byte at a time, 9.91 milliseconds locally and 14.63 milliseconds remotely, is composed of 6.23 milliseconds for 1024 calls to getc plus 3.47 milliseconds and 8.18 milliseconds respectively for getting the file block into the local buffer from the file server, leaving about 0.21 milliseconds (both locally and remotely) for the I/O routines that map between getc and the block I/O routines. Similarly, the block I/O read operation includes the basic IPC cost (1.79 milliseconds locally and 6.34 milliseconds remotely) leaving roughly 2 milliseconds (1.68 milliseconds locally and 1.84 milliseconds remotely) for local I/O routine overhead and server processing time.

These measurements indicate that the cost of block I/O reading is dominated by the cost of the IPC operation to transfer the data. We have shown elsewhere [6,10] that the performance of the V IPC operations is dominated by the hardware time for data transfer, particularly for non-trivial amounts of data. Moreover, with increasing block size, the client and server processing times remain almost identical but the IPC times grow (more or less) linearly with the size of block transferred. In particular, the cost of a ReadUio operation to a remote storage server increases by 2.8 milliseconds for each additional 1 kilobyte increase in the number of bytes to be read and 2.52 milliseconds of this cost is due to the IPC overhead. Thus, the UIO interface has a modest overhead with 1 kilobyte blocks and can be reduced further by using larger block sizes.

For byte-stream I/O, the I/O library and UIO interface overhead is dominated by the cost of get-byte operations. Moreover, we argue that the cost of a *getc* is close to the minimum achievable by conventional techniques because it is about 6 microseconds per byte on a Microvax II, a 2 MIPS machine, and it includes little more than a test for empty buffer, an indexed byte reference (assigning the returned byte to a register), and an increment to the buffer pointer. Moreover, any techniques that would reduce this cost in other get-byte implementations should also be applicable here. [19] Thus, excluding the get-byte operation time, the overhead for the UIO client library and server overhead can be calculated as

overhead = elapsed-time - get-byte-time (if any) - IPC-time

As indicated in Table 3 this overhead is less than 25 percent of total time for all but local block I/O (and this percentage decreases with increasing block size). Since all schemes would have a comparable cost for get-byte, no I/O scheme can have significantly less overhead, assuming similar costs for the basic IPC or RPC communication with the server module.

[19] *getc* **is executed as a (portable) macro for these measurements the same as in our standard T/O implementation.**

Server Location	put-byte	Disk (bytes)	Disk (blocks)	IPC
Local	6.70	12.57	5.66	1.79
Remote	6.70	17.90	10.99	6.34

Table 4: Writing: Elapsed Time per Kilobyte on Microvax II (in milliseconds)

These measurements do not fully measure the UIO processing overhead when the server is remote since both the client and server processor are then executing in parallel. However, the fact that the difference between local and remote I/O times is almost entirely accounted for by the difference in IPC times and the server takes the same action for both local and remote requests indicates that this "invisible" overhead is not significant when reading.

Table 4 gives the comparable measurements for writing. The basic put-byte time is approximately the same as for get-byte. The higher block I/O cost, and therefore byte-stream cost, is due in part to an extra overhead in the storage server for writing. Currently, it copies each block it receives as part of a write operation in a multi-block buffer in the disk buffer pool.

In summary, the UIO interface provides efficient I/O interface relative to the costs of communication and byte-level operations. Moreover, we see considerable potential to further improve the performance, particularly in the server implementations. Work on a faster version of the V storage server is in progress.

4 Related Work

There appears to be little other work that addresses the key focus of our work, namely the design of a uniform I/O interface.

In the operating system area, the need for uniform interfaces has been recognized but not studied to any degree. The UNIX operating system interface [29] provides a familiar example of a uniform input and output interface yet it has a number of deficiencies. First, the semantics of the I/O operations are incompletely specified relative to that required by programs. For example, there is no guarantee that writing data to a file and then reading the data back returns the same data. In fact, there is no guarantee that rereading the same part of a file returns the same data, although this is the case with "normal" disk-based files. Programs that use files as temporary storage need to be able to check that the I/O service to which they have bound provides storage semantics to guarantee their correct behavior. The UIO interface clarifies the semantics of I/O by allowing a program to determine that a UIO being used as a storage container, such as for a temporary file, does in fact have storage semantics. Second, the UNIX interface does not support logical records in a uniform way because it treats some cases, such as terminal input and raw network input in special ways and treating others as transparent byte streams. Several years ago, Ritchie [28] observed the incompatibility that can arise in UNIX between terminal and file I/O because of the lack of logical record support, and suggested an extension similar to that provided in the UIO interface. The UIO interface allows one to preserve logical record boundaries in a uniform way using the data block abstraction. Finally, the UIO block abstraction is easier and more efficient to support in a distributed system than the byte stream abstraction defined by the UNIX "read" and "write" system calls, especially if the local kernel does not contain the file system. With the UNIX interface, the remote file server has to support arbitrary byte stream read and write operations; it cannot treat read operations as idempotent; and it must maintain a "file pointer" for each open file. Even in the local case, block I/O is more efficient, especially for applications like database management systems. Overall, the UIO interface benefits considerably from the UNIX design yet makes several key improvements.

The Alpine file system [24] provides similar file and block-level locking and atomic transaction functionality to that specified as part of the UIO interface. There are minor differences, such as the Alpine interface does'not release locks when a file is closed. However, a key difference is that the Alpine interface is designed specifically for a file server whereas the UIO interface is intended to be far more general.

This work has been strongly influenced by the connectionless protocols and idempotency in the WFS design [31]. In fact, an earlier version of our design [4] attempted to realize a uniform I/O interface with no state. However, experience with I/O services such as terminal and network connections showed the infeasibility of this approach.

Overall, the need for well-designed interfaces for services has been recognized by the operating system area [21] but techniques for achieving uniformity across these interfaces have not significantly explored.

The work on abstract data types and classes has some relevance to our work. In particular, to first approximation, the UIO interface can be regarded as specifying an abstract data type. However, some established work in the area [18] places a very restricted definition on the notion of abstract data type which precludes viewing UIO specification as an abstract data type. For example, the Guttag-Horning notion of *sufficient* completeness of an abstract data type definition prohibits the semantic differences that can arise between different UIO objects. Moreover, it is not apparent how to relax these definitions without significant loss of results and admission of pathological cases.

Similarly, the Smalltalk [20,14] notion of *class*, *subclass*, *abstract* class and *metaclass* appear similar to the UIO interface, which effectively defines a set of types with the commonality defined by the UIO operations. However, 3malltalk classes stress economy of representation, not semantics. That is, a class is implemented as a subclass of another to avoid repeating the specification of instance variables and methods. It is free to override the semantics of messages handled by its superclass with arbitrary semantics of its own, at least for its own objects. In contrast, the UIO interface defines a class of types based on common generic semantics with no consideration as to whence the representation of each individual type or I/O service originated. In general, this distinction between *economy of representation* versus *conformity of semantics* separates the work on type classes and type hierarchies [1] from the requirements of uniform interfaces.

5 Concluding Remarks

The UIO interface has been successfully used in a distributed system of considerable scale and diversity to couple a range of applications and I/O services. In this regard, it demonstrates the feasibility of a uniform I/O interface that reduces the complexity and increases the flexibility and function of a system. It also takes a significant step beyond the functionality and semantic integrity of the C stdio interface and the UNIX operating system interface.

The UIO interface and its design rationale are interesting on several points. First, the block abstraction for data access is a departure from the more commonly espoused byte stream model. Second, the explicit incorporation of state in the interface contrasts with the stateless approach advocated by some designers. Third, the provision of ownership and protection information in the I/O state eliminates the need for the communication system to support these functions, allowing the use of a pure RPC communication abstraction. Fourth, the distinction between compulsory, optional and service-specific operations in the interface allows full access to functionality without excessive burden on restricted services and unnecessary restriction on service functionality, and without excessive complexity for clients. In general, the UIO interface implicitly identifies a number of generic classes of I/O behavior and function, imposing a taxonomy on the apparent diversity of I/O-like services. Fifth, the provision of attributes to describe UIO objects with respect to the optional functionality and standard error returns simplifies the client software in dealing with the optional aspects of the interface and also allows client behavior to be more efficient. Sixth, the use of the name directory for handling options and specialized parameters avoids complicating the basic I/O interface. Finally. we have pointed out some novel mappings of the record file access onto UIO data block model, showing how structured file access can fit the UIO model. In general, the design focuses on efficient support of the "common case" while identifying techniques that accommodate the other cases without loss of function.

Our work to date on the UIO interface leaves a number of research issues for future exploration. First. there is the need to explore detailed aspects of the interface with a wider set of services and applications Our experience with the interface in V is significant but far from complete. For example, the locking and atomic transaction interface has only been implemented as part of a student project and not integrated

to date into the V software we use and distribute. Similarly, the replication portion of the design needs further implementation experience. Moreover, several of the servers do not follow the protocol exactly as described here, in some cases because of improper implementation; in others because the UIO interface has changed slightly since their original implementation. Although the create, release, read and write aspects of the interface have been quite stable for five years, there have been additions and modifications along the way. While our experience gives us confidence in the basic UIO design, further refinements in the exact parameters and their semantics are expected. In general, there seems no prospect of "proof of optimality" or its converse for this type of work. Confidence in this design can only come from further experimentation and actual use.

There also needs to be more exploration of other approaches to the design of uniform interfaces-I/O as well as others. Quantitative trade-offs within the design need to be better understood. To a large degree, the UIO interface simply represents the author's current "best guess", supported by some experience and (it is hoped) convincing arguments.

Finally, the specification of a uniform interface raises a number of semantic and notational issues. For example, different UIO objects may be sufficiently different that it would be impossible to regard them as the same data type using conventional data type theory and semantics. This work also does not allow instances of a type to have operations or semantics beyond that specified for the type, as arises with UIO instances. A semantic theory and notation is required for generic *abstract data types* in order to formally describe the semantics of uniform interfaces like the UIO interface. The author has taken some tentative steps in this direction, closely modeled after the work of Guttag and Horning [19,17]. However, developing a formal replacement for the informal semantic descriptions used in this paper remains a significant challenge. A more formal and complete specification is essential in reducing ambiguity and ensuring compliance, just as with conventional standard protocols and interfaces. In fact, with the growing use of remote procedure calls in distributed systems, the specification of interfaces, as opposed to protocols, becomes the focus of distributed systems design at all but the transport and RPC presentation levels.

In general, we view that uniform interface design is a difficult yet critical aspect of large-scale system design. Further research is required to understand how to design such interfaces and what issues are involved. Concurrently, more examples of such interfaces are required along with credible experience with the implementation and use of these interfaces in a diversity of real system contexts. We put the UIO interface forward as but one candidate for such an interface and one step in this research direction.

6 Acknowledgements

Steve Deering, John Demco and Brent Hilpert contributed to the early development of these ideas. I am indebted to the members of the Stanford Distributed Systems Group for the implementation of the UIO interface in the V distributed system and the further refinement of the interface that resulted, particularly Tim Mann, Ross Finlayson, Karen Lam and Joe Pallas. Finally, I am grateful to Michael Stumm and the referees for helpful comments that greatly improved the content and the presentation.

References

[1] A. Albano.
Type hierarchies and semantic data models.
In *Proc. SIGPLAN '83 Symp. on Programming Language Issues in Software Systems,* pages 178-186, ACM SIGPLAN, New York, 1983.
Also available in SIGPLAN Notices, Vol. 18, No. 6, 1983.

[2] F. Baskett, J.H. Howard, and J.T. Montague.
Task communication in DEMOS.

In *Proceedings of the 6th Symposium on Operating System Principles,* pages 23-31, ACM, November 1977.

[3] A. Birrell and B. Nelson.
Implementing remote procedure calls.
ACM Trans. on Computer Systems, 2(1), February 1984.

[4] D. R. Cheriton.
A loosely-coupled I/O system for a distributed environment.
In A. West and P. Janson, editors, *Local Networks for Computer Communication,* North-Holland, Amsterdam, August 1980.
IFIP WG 6.4 International Workshop on Local-Area Networks, Zurich Switzerland.

[5] D.R. Cheriton.
The Thoth System: Multi-process Structuring and Portability.
American Elsevier, 1982.

[6] D.R. Cheriton.
The V kernel: a software base for distributed systems.
IEEE Software, 1(2):19–42, April 1984.

[7] D.R. Cheriton and T. Mann.
A Decentralized Naming Facility.
Technical Report STAN-CS-86-1098, Computer Science Department, Stanford University, April 1986.
Also available as CSL-TR-86-298.

[8] D.R. Cheriton and P. Roy.
Performance of the V storage server: a preliminary report,
In *ACM Computer* Science *Conference,* March 1985.

[9] D.R. Cheriton and W. Zwaenepoel.
Distributed process groups in the V kernel.
ACM Trans. on Computer Systems, 3(2), May 1985.

[10] D.R. Cheriton and W. Zwaenepoel.
The distributed V kernel and its performance on diskless workstations.
In *Proceedings of the 9th Symposium on Operating Systems Principles,* ACM, 1983.

[11] DARPA.
DOD Standard Transmission Control Protocol.
Technical Report IEN-129, Defense Advanced Research Projects Agency, January 1980.

[12] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notions of consistency and predicate locks in a database system.
Commun. ACM, 19(11):624–633, November 1976.

[13] R. Fitzgerald and R.F. Rashid.
The integration of virtual memory management and interprocess communication in Accent.
ACM Trans. on Computer Sys., 4(2):147–177, May 1986.

[14] A. Goldberg and D. Robson.
SmallTalk-80: the language and its *implementation.*
Addison-Wesley, 1983.

[15] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries.
A recovery algorithm for a distributed database system.
In *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database* Sys., March 1983.

[16] J. Gray.
Notes on database operating systems.

In R. Bayer, R.M. Graham, and G. Seegmueller, editors, *Opemting Systems: An Advanced Course,* chapter 3, pages 393-481, Springer-Verlag, New York, 1979.

[17] J. **Guttag** and J.J. Horning.
Formal specification as a design tool.
In *Seventh ACM Symp. on Principles* ***of*** *Programming Languages,* January 28-30 1980.
Las Vegas, also Xerox PARC Tech. report CSL-80-l.

[18] J.V. **Guttag** and J.J Horning.
The algebraic specification of abstract data types.
Actu Informutica, **10(3):27–52,** 1978.

[19] J.V. **Guttag,** J.J. Horning, and J.M Wing.
The Larch family of specification languages.
IEEE Software, **2(5):24–36,** September 1985.

[20] D. **Ingalls.**
The Smalltalk- programming system.
In *5th Annual Symp. on Principles* ***of*** *Programming Languages&,* pages 9-16, ACM **SIGACT/SIGPLAN,** 1978.

[21] A.K. Jones.
The object model: a conceptual tool for structuring software.
In R. Bayer, R.M. Graham, and G. Seegmueller, editors, *Opemting Systems: An Advanced Course,* chapter 2, pages-7-16, Springer-Verlag, New York, 1979.

[22] B. W. Kernighan and D. M. Ritchie.
The C Programming Language.
Prentice-Hull ***Software*** Series, Prentice-Hall, New Jersey, 1978.

[23] K.A. Lantz, D.R. Cheriton, and W.I. Nowicki.
Third Generution Graphics ***for*** *Distributed Systems.*
Technical Report STAN-CS-82-958, Computer Science, Stanford University, 1982.

[24] K.N. Kolling M.R. Brown and E.A. Taft.
The Alpine file system.
ACM ***Trans.*** *on Computer Sys.,* **3(4):261–293,** 1985.

[25] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson.
A trace-driven analysis of UNIX 4.2 **BSD** file system.
In *10th Symp. on Opemting System* ***Principles,*** ACM **SIGOPS,** 1985.

[26] M.L. Powell and D.L. Presotto.
Publishing: a reliable broadcast communication mechanism.
In ***Proc.*** *9th ACM Symposium on Operating Systems Principles,* ACM, 1983.

[27] R. **Rashid** and G. Robertson.
Accent: a communication oriented network operating system kernel.
In *Proceedings* ***of*** *the 8th Symposium on Operating Systems Principles,* pages 64-75, ACM, December 1981.

[28] D. Ritchie.
The UNIX timesharing system: a retrospective.
The Bell System Technical ***Journal,*** **57(6),** July-August 1978.

[29] D. M. Ritchie and K. Thompson.
The UNIX timesharing system.
● *ommunicutions* ***of*** *the ACM,* **17(7):365–375,** July 1974.

[30] P.M. Schwartz and A.Z. Spector.
Synchronizing shared abstract data types.
ACM Trans. on Computer Sys., 2(3):246–250, 1984.

[31] D. Swinehart, G. McDaniel, and D. Boggs.
WFS: a simple file system for a distributed environment.
In *Proc. 7th Symp. Operating Systems Principles, 1979.*

[32] S. Thatte.
Persistent memory: storage architecture for object-oriented databases.
In *Proc. Int. Workshop on Object-Oriented Database Systems,* ACM, New York, September 1986.
Pacific Grove, California.

[33] M.M. Theimer, K.A. Lantz, and D.R. Cheriton.
Preemptable remote execution facilities in the V-system.
In *10th Symp. on Opemting System Principles,* ACM SIGOPS, 1985.

[34] K. Thompson.
UNIX implementation.
The Bell System Technical Journal, 57(6), July-August 1978.

[35] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel.
The LOCUS distributed operating system.
In *Proceedings of the Ninth Symposium on Operating Systems Principles,* pages 49-70, ACM, October 1983.

[36] W. Zwaenepoel.
Implementation and performance of pipes in the V-System.
IEEE Trans. on Computers, C-34(12):1174–1178, December 1985.

A UIO Interface

The UIO interface is specified as a set of procedures including, for each procedure, the syntax of the call, the data types of the parameters and return values, and the generic semantics of the procedure. For brevity, the following description omits some of the details. However, this description together with the main body of the paper should communicate the key points of the design.

(uio-desc,error) := CreateUio(name, context, mode, transact-id) - creates a UIO object according to the *nume* in the specified *context* and in the specified *mode,* associated with the transaction identified by *transact-id,* if non-zero. It returns **a** uio-desc of the new UIO, or else returns a reason for failure. The mode indicates the way the object is be used, namely:

READ : Data associated with the object are to be read but not changed.

CREATE : A new sequence of data blocks is to be created by the client and associated with the object, discarding any previously associated data.

APPEND : Data are to be appended to the current sequence of data blocks. Data previously associated with the object remains unchanged.

MODIFY : Existing data are to be modified and possibly appended to.

These modes are further modified by LOCK and INTENTION-LOCK flags. The LOCK flag alone causes the file to be locked in shared or exclusive mode, depending on the access mode. The INTENTION-FLAG causes the lock to be an intention lock, providing for block-level locking. The combination of the LOCK and INTENTION-LOCK flags used with MODIFY mode provides a shared file lock plus an intention-exclusive lock on the file. The server returns the error LOCKED if the file is already locked

in a mode incompatible with that requested. The server returns the error **MODE_NOT_SUPPORTED** if a mode of access is requested that the server does not support, e.g. locking. Similarly, if transact-id is non-zero and the server does not support the atomic transaction protocol, it returns an error message. The uio-desc record consists of the following fields:

uio-id - the system-wide unique UIO identifier, normally structured as I/O server identifier and local identifier.

owner - the process owner of the UIO, initially the creating client process.

authdomain - authorization domain for access to the UIO.

uio-perms - access permissions on the UIO. The possible permissions are read and write for those in the authorization domain specified by authdomain.

uio-attributes - bit flag attributes describing the UIO, including **READABLE, WRITEABLE, AP**-PEND-ONLY, RESETTABLE, **RANDOM_ACCESS, STORAGE, VARIABLE_BLOCK**, MULTI-BLOCK, **FIXED_LENGTH**, INSERTDELETE, SNAPSHOT, RECOVERABLE, LOCKED, INTENT-LOCKED, REPLICATED, and INTERACTIVE.

lockwaittime - maximum time an operation can suspend waiting for **a** lock to be available before returning an error indication.

blocksize - maximum data block size that can be read or written with this UIO object.

nextblock - block number of the first block in the UIO array of data blocks.

lastblock - block number of the last block in the UIO array of data blocks.

lastbytes - number of valid byes in the last block.

uio-servers - list of the servers that implement this object. This is only returned when the file is REPLICATED and accessed for update.

(uio-desc,error) := CreateRelatedUio(uio-id, mode) - creates a UIO object that is the related UIO object to the specified UIO, typically corresponding to the other direction on a bi-directional stream.

(uio-desc **,error) := CreateUioVersion(uio-id, mode, transact-id, ptransact-id**) - creates a UIO object that is a version of the specified UIO object in the specified access mode as part of the specified transaction, with parent transaction specified by ***ptransact-id***.

(uio-desc, error) := QueryUio(uio-id) - returns an uio-desc record describing the specified UIO, as describe above.

error := ReleaseUio(uio-id, releasemode) - completes the updates performed using this UIO object if any, releases any locks associated with this UIO object and invalidates the object identifier, reclaiming the resources associated with this UIO. Service-specific actions may be involved with completing the updates and reclaiming resources. For example, on a file server, buffered data may be flushed to disk and possibly to a log. **A** mode of 0 specifies the default normal action. The release mode consists of three bit flags that control the three actions of release, as follows:

ABORT_CHANGES : Do not complete, but undo if possible the changes reflected in this UIO.

HOLD-LOCKS : Do not release the locks held by this UIO object. This is only legal if the following flag is not set.

RETAIN_UIO : Do not invalidate this UIO object or release any resources associated with it.

Using combinations of these flags, **ReleaseUio** can be used for creating savepoints, just releasing the locks associated with a UIO object and aborting changes made using this UIO, if RECOVERABLE.

(count, error) := ReadUio(uio-id, blkno, buffer, count) - transfers *count* bytes to the client's buffer from the UIO object starting at the specified block and returns the number of bytes read and the error reason for reading fewer bytes than requested if appropriate. If the UIO object has the STORAGE attribute, the data bytes read match those last successfully written to that block, providing the operation succeeds and a successful write operation has occurred to this block of this object during its lifetime. The operation returns the END-OF-FILE code if the object is FIXED_LENGTH (see below) and the block number is larger than the lastblock field or it is equal and the byte count is greater than lastbyte. A share lock is acquired on the data blocks read if the UIO is intention locked.

(count, error) := WriteUio(uio-id, blkno, buffer, count) - transfers *count* bytes from the client's buffer to the UIO object starting at the specified block and returns the number of bytes written and the error reason for writing fewer bytes than requested, if appropriate. If the UIO object is FIXED_LENGTH, the data written must fit within the existing data sequence. After a successful write operation, the lastblock and lastbyte values are updated as necessary to include the data written. An exclusive lock is acquired on the data blocks written if the UIO is intention locked.

(count, error) := InsertUioBlocks(uio-id, blkno, buffer, blkcount) - creates new blocks numbered *blkno* through to *blkno + blkcount* - 1, renumbering the blocks currently assigned this range of number to consecutively follow the new data blocks. The server returns an error code if the UIO does not have the INSERTDELETE attribute.

(count, error) := DeleteUioBlocks(uio-id, blkno, blkcount) - deletes the specified range of blocks from the UIO, renumbering the remaining blocks so they form a consecutive sequence. The server returns an error code if the UIO does not have the INSERTDELETE attribute. However, the server for a UIO with the STORAGE attribute and not the FIXED-LENGTH attribute is expected to support deletion from the end of the block sequence at minimum, i.e. truncation.

. **(oldowner,error) := SetUioOwner(uio-id, newowner, authdomain, perms)** sets the owner of a UIO to the specified new owner, authorization domain and permissions. This can only be executed by a process in the same authorization domain as the UIO owner.

(error, oldauthdomain, oldperms) : = SetUioPermissions(uio-id, authdomain, newperms j - sets the access control on the UIO to the specified new authdomain and permissions and returns the previous authdomain and permissions. This can only be executed by a process in the same authorization domain as the UIO owner.

(error, oldbreakprocess) := SetUioBreakProcess(uio-id, newbreakprocess) - set the new break process for this INTERACTIVE UIO object **and** return the previous break process. The *break process* for a UIO object receives a signal **or** exception when a user-generated break or attention occurs on this UIO.

(oldtime, error) := SetUioLockWaitTime(uio-id, newtime) sets the lock wait time of a UIO to ***newtime***, returning the previous value of this parameter. This can only be executed by a process in the same authorization domain as the UIO owner.

error := LockUio(uio-id, lockmode, blkno, blkcount) - lock the specified range of data blocks in the specified UIO in the given lockmode, SHARED or EXCLUSIVE. An LOCKED error code is returned if the request is in conflict with an existing lock after waiting for *lockwaittime* seconds.

error := UnlockUio(uio-id, lockmode, blkno, blkcount) - unlock the specified number of data blocks in the specified UIO to the given lockmode, starting at blkno.

(error, ret urnrec) := UioControl(uio-id, parameters) - perform a query or modification to the UIO specific to this type of UIO according to the service-specific parameters, returning an error code and the return record.

Except for **QueryUio, SetUioOwner, SetUioPermissions, LockUio** and **UnlockUio,** these operations may also have specific semantics supplied by the service-specific module implementing the object. Consider as an example the implementation of a printer spool file. When this specific type is used, besides creating a UIO object, the CreateUio operation creates a disk "spool" file into which the data can be stored temporarily and associates this with the new UIO object. Logically (and perhaps in practice), a **WriteUio** operation on this object simply invokes the corresponding operation on the disk file to append the new data to the disk file. A **ReleaseUio** operation with a 0 release mode queues the disk file to be printed by a printer daemon. With an ABORT-CHANGES mode, it aborts the printing by discarding the disk file. In general, the specific semantics are only constrained to be consistent with the generic UIO semantics.

An important aspect of this uniform I/O interface is uniformity in the error return codes, since they must be interpreted by the I/O run-time, and not just passed back to the user in **a** print statement. In the interests of brevity, we only list a sample of the error returns we have used in V.

OPERATION_NOT_SUPPORTED - the request operation is not supported by this service module.

MODE_NOT_SUPPORTED - mode in the CreateUio operation is not supported by this server module.

UNDEFINED-BLOCK - attempt to read a block that has never been written, such as in reading a so-called sparse file or reading beyond the end of file.

BAD-BYTE-COUNT - attempt to read or write a larger amount than **that** supported by the server. The server should read or write the maximum it does support, as indicated in the returned bytecount.

BAD_BLOCK_NUMBER - attempt to read or write non-sequentially on **a** stream UIO.

LOCKED - The file or block **is already** locked in a way that is inconsistent with the request lock.

END-OF-FILE - the read operation received the end of the file, i.e. read the last byte in the last block. This return code can also be generated on **a** terminal input stream.