Frigate: An Object-Oriented File System for Ordinary Users*

Ted H. Kim Gerald J. Popek†
Department of Computer Science
University of California, Los Angeles

Abstract

Vendors cannot provide all the operating system services that users demand. As a result, there has been a persistent desire to make operating systems more flexible and customizable. It is natural that object-oriented technology would come to bear on this area. However, many solutions have been disappointing when it comes to ease of use.

This paper describes the design and implementation of Frigate, an object-oriented file system. The goal of Frigate is to provide a modular, extensible framework. The framework allows new extensions to be "plugged-in" on the fly. Frigate's focus differs from most other file system designs in that it is targeted for use by *ordinary users* rather than by sophisticated operating system gurus. Thus, ease of use is a very important concern in the design. Frigate is fully implemented and supports a set of example file system extensions.

1 Introduction

Vendors cannot provide all the operating system services that users demand. On the one hand, it is economically infeasible, and on the other hand, many times they do not know what the users desire. Vendors rightly concentrate on providing a few general purpose services. Users can either lobby the vendor to include some desired service or implement it themselves. Adding services to the operating system is a difficult proposition. Traditionally, special privilege and access to source (and possibly a license agreement) is required. Programming the kernel can demand special care and knowledge. Debugging modified operating systems is difficult as well usually requiring specialized kernel address space tools as well as continual rebooting. Convincing someone else that your extension of the operating system is not too risky is not always easy. Distribution of modifications may encounter similar trust problems and be limited by license restrictions. Compatibility between independently written extensions and the ability to further modify them are also potential problems.

The long standing desire to make operating systems more flexible and customizable led to various architectural innovations, including loadable device drivers, streams [42], vnodes [28] and micro-kernels. All of these ideas were aimed, at least in part, at providing extensibility. It is natural that object-oriented technology, with its themes of modular extensibility, would eventually come to be applied to this problem.

Most attempts at applying object-oriented technology to operating systems are attempts to internally restructure the operating system into a more modular organization. The object-oriented model is not generally exported to the users. The intended audience of such solutions are expert operating system architects, who are conversant in the intricate internals of the operating system. Such tools are extremely hard for ordinary users to use.

Our particular problem domain is the file system. Within this domain, Frigate takes a different approach. The intended audience of Frigate is ordinary users. Frigate's object model is not just for internal use. It is fully exposed and usable by ordinary users. A programmer using Frigate needs to be familiar with object-oriented concepts and system-call programming but does not need to be an operating system guru. We believe we have constructed tools that can be easily and widely used. This enables a much larger audience to do powerful, modular extensions of the file system.

Our overall goal is to provide a way to add value to the file system easily. Towards this end, Frigate attempts to provide a modular, easy-to-use, persistent object framework that also allows incremental usage and is fully compatible and integrated with the current filing environment.

*This work was supported by the Advanced Research Projects Agency under contract DABT63-94-C-0080. The authors can be reached at the Department of Computer Science, UCLA, Los Angeles, CA 90095, or by email to {tek,popek}@cs.ucla.edu.

†Gerald Popek is also affiliated with Platinum *technology*.

2 Architecture

The Frigate system consists of five main components built on top of UNIX. At the lowest level, Frigate uses *typed* files. File System extensions are stored in an external *Repository*. When extensions are used, they are instantiated as server processes. Within the operating system, Frigate provides a *Dispatcher* module, which manages the servers and intercepts file system calls, passing them out to the servers. This module is built on the Stackable Layers framework. Finally, an object-oriented programming interface is provided by using Xerox PARC's Inter-Language Unification (ILU) system [25]. The runtime architecture of Frigate is illustrated in Figure 1.

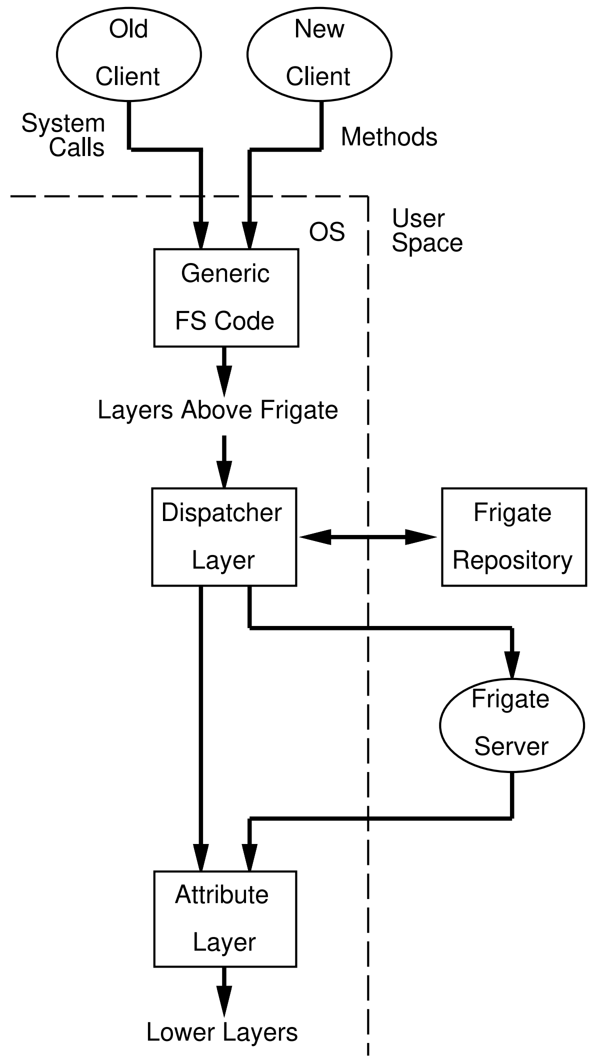

Figure 1: Frigate Architecture

2.1 Stackable Layers Infrastructure

Frigate marries two different worlds together; it connects a UNIX file system model with a CORBA [36] based object model. On the UNIX side, Frigate uses the Stackable Layers framework, which can be seen as a generalization of the Virtual File System (VFS) [28]. VFS is the basis for most modern UNIX file system implementations. In VFS, the file system portion of the operating system is divided into a generic portion and specific file system implementations (e.g., Unix File System (UFS), NFS, PC-FS). All files (directories, pipes, etc.) are represented in VFS by an abstract type called a vnode (virtual node). All calls into the specific implementations are operations on this abstract type. This allows other specific implementations to be added without any modification of the generic code. In practice, though, this is difficult as each specific implementation is still quite large and must be developed and debugged in the operating system address space.

A further refinement of VFS is the UCLA Stackable Layers file system [23]. In this model, each specific implementation is made up of a stack of protocol layers in the kernel similar to System V streams [42]. Each layer is a much smaller entity than the large monolithic file system implementations of pure VFS. Generally, layers implement a specific increment of file system features. Operations not implemented by a particular layer are passed down to lower layers in the stack, via the "bypass" operation (a form of delegation). The layers use the vnode interface for inter-layer calls and can be independently developed, replaced and composed together. For example, a file system might be composed of a layer for naming/directory services and a layer for storage. Later, the storage layer could be replaced with one implementing a log-structured storage [43] without requiring any changes to the naming layer. Stackable Layers was used to implement the Ficus replicated file system [18, 19]. Layers need not be configured strictly linearly as stacks but can also be placed in tree configurations. As we will see, Frigate extensions are implemented in this model as layers.

2.2 Documents

Frigate uses typed file entities, which we call *documents*. A file entity is anything that can be named in the file system: file, directory, pipe, etc. Along with the standard UNIX file attributes, each document has associated class and version identifiers. The class is a name for a particular method interface and hence a set of services provided by the document. The version is used with the class to pick a particular implementation at *bind* time. To prevent name clashes, but at the same time, permit independent development, the actual identifiers are NCA UUIDs [52], which are globally unique but can be generated in a distributed fashion. This obviates the need for any central registry. In our current implementation, the storage for these attributes is provided for in a separate Stackable Layer [50].

Documents are objects in our system. However, they have some distinctive features. In our model there is a standard implicit UNIX file class, which all document classes inherit from directly or indirectly. Vnode operations appear in the Frigate model as methods. Document classes can declare that their associated versions have modified vnode method implementations. This redefinition of vnode methods is possible for all file operations, except for a few reserved for Frigate's infrastructure. Of course, document classes can add any other methods also required for their application usage. Because documents have an associated UNIX file entity, they can be named by using file pathnames and can also use file data storage.

2.3 Repository

The repository stores the interfaces and implementations for Frigate extensions. Each class has an interface description indexed by class identifier. The interface description consists of identifiers for vnode methods that are redefined by the document class. This information is used by the Dispatcher to determine if a vnode operation should be intercepted and passed to an extension server. The Dispatcher also sends vnode operations carrying ILU method invocations to the servers.

Implementations stored in the repository are the actual servers that implement objects. The implementations are indexed by class and version. One implementation for each class may be marked as *preferred*. Applications can specifically request the preferred implementation. A preferred implementation is usually the latest or most stable version. Stored with the implementation is load balancing information and configuration parameters for the server.

The repository also supports *alias* records. An alias is an alternate name for an implementation or another alias record. It defines a mapping from one implementation identifier to another identifier. As described later, this feature is used to aid in version management.

The repository is implemented by a repository server and database. Requests from both the Dispatcher and front-end user programs are handled by the server. The repository server handles various requests from the Dispatcher involving server management, interface query and binding of implementations. The process of binding involves mapping a class/version pair to a specific server implementation, taking into account preferred implementations and alias records.

Front-end user programs interact with the repository server mainly to add and remove class descriptions, server implementations and alias records. Security features only allow the class owner to manipulate the class description and associated implementations. Otherwise, all users are capable of adding and managing classes and implementations.

2.4 Dispatcher

The Dispatcher is a stackable layer that stacks in the operating system above the layers that provide file storage and Frigate attributes. This layer intercepts vnode operations for documents and directs the management of the servers that implement document objects. From the view of Stackable Layers, this is just another layer. The Dispatcher layer, however, does not have the same behavior for all files. Instead, the behavior of the Dispatcher varies according to the class and version of the document.

When the Dispatcher vnode for a newly referenced file is constructed, some special actions occur. First, it is determined whether the file is a document or not. A document has Frigate attributes; other file entities do not. If the file entity is not a document, the vnode is set up to simply bypass all operations to the layer beneath it. Thus, ordinary files behave as if the Dispatcher layer did not exist at all.

If a vnode belongs to a document, then the class attribute is used to find its interface by querying the repository server. (Information is cached, so subsequent queries may avoid RPC with the repository server.) The interface information from the repository is used to determine which vnode operations should be intercepted and sent to the server. All other operations bypass to the layer beneath.

Servers are lazily started; no attempt is made to start one until some method or vnode method is called. In this way, no unnecessary server starts are performed. When an operation that requires a server is invoked, the document is actually bound to a particular implementation. This late binding allows the most recent repository changes to be reflected in the binding, even if these changes occurred since the time of class interface lookup. When the actual implementation is determined, the list of active servers is checked for a matching server. If a matching implementation server has spare capacity, then the document is assigned to that server. If no server is available, then one is started.

After a document has been assigned to a server, methods and vnode methods are passed to the server using Stackable Layer transports. Methods pass through special vnode operations that carry ILU information to and from the server. A server may handle requests from several different users. For the duration of each request, the server's credentials are changed to that of the process that invoked the method. Thus, the server will have exactly the privileges of the client. Between requests, the server is given special reduced privileges. If we did not have such a mechanism, it would not be possible to have a single, possibly *untrusted*, server handle multiple users. Each user would have to have his own server to assure correct access credentials.

When a document is no longer referenced, the vnode for the document is destroyed. During this process, a message is sent to inform the server. When a server no longer has any documents assigned to it, the Dispatcher directs the repository server to terminate the server.

2.5 Servers

Stackable Layers provides some facilities to run user-level file system servers. Frigate servers are modified and enhanced versions of the Stackable Layer servers. Frigate servers are structured somewhat differently to allow modular construction from ILU files and to accommodate method dispatch at runtime. Since Frigate servers are user-level, server programmers do not have to worry about kernel programming restrictions. Frigate provides the means for running more than one implementation side by side for any particular class. Also, multiple servers can run for load balancing.

Documents are persistent objects, which can be in one of two states: *active* and *passive*. An active document has been assigned to a server and is ready to service requests (i.e., has a running implementation). A passive document has no server. Whenever a document is newly assigned to a server, methods are automatically invoked on the document to activate it. Typically, these methods initialize the document's implementation from its persistent file data. When the last reference to a document is removed, methods are automatically invoked to deactivate the object. Usually, these methods force any pending updates out to persistent storage.

The actual implementation of a method lies in its handler code. If necessary, the handler may call on services from layers lower down in the stack. Calling on lower layers leads back into the operating system rejoining with the in-kernel stack. This is accomplished by using a special Stackable Layers transport.

2.6 Programming Interface

The Inter-Language Unification (ILU) system [25] from Xerox PARC is used by Frigate to provide an object-oriented programming framework. ILU provides a CORBA [36] based object model. In ILU, type definitions and interface definitions for objects (including multiple inheritance relationships) are written in CORBA IDL (or ILU's own ISL). These definitions are compiled into target language mappings, providing client calling stubs and server method skeletons. The target language mappings are compiled along with application code to construct the actual ILU client and server programs. At runtime, ILU clients and servers communicate by using transports selected from ILU's library.

Frigate integrates document objects and vnode methods with ILU. Currently, Frigate implements support for documents in both IDL and ISL and provides support for the ANSI C target language. Frigate applications are programmed just like other ILU applications, except that servers must be assigned a version number and registered with the repository to be used. Compiling interface definitions automatically produce the required calling stubs and skeletons for any vnode methods. Server method handlers may be programmed using the Stackable Layers vnode model or the regular UNIX system call model.

Frigate document classes appear in the ILU world as ILU object types. They are distinguished from other ILU objects by inheriting object types from the implicitly defined UNIX file interface. This avoids any syntactic change to ISL or IDL. Only documents may have vnode methods. The semantics of inherited vnode methods, though, are different from those of other inherited methods. Even if a particular vnode method is not inherited, a document will still try to delegate that method invocation by using the Stackable Layers bypass mechanism. Vnode methods are, by design, inherited only. The signatures of vnode methods are externally fixed and so cannot be redefined inconsistently with the rest of the framework.

At runtime, Frigate documents are located by using file system paths. The file system namespace acts as the published object registry. Through the use of Frigate utility routines, the file system name of a document is resolved internally into an object handle and then into a local proxy object. Thereafter, messages for ILU methods or vnode methods can be sent to the document object by using the calling stubs defined in the language mapping. Method invocations made on proxy objects are automatically converted into calls to the Frigate document servers. The calls are transported to Frigate servers via the Frigate transport, which packages the Stackable Layers infrastructure as an ILU transport.

It should be noted that vnode methods can be invoked both through ILU and through ordinary UNIX system calls. UNIX system calls are serviced by the VFS generic file system code, which in turn makes vnode operation calls. These calls are selectively intercepted by the Dispatcher just as direct vnode method calls would be. This means that Frigate document servers can even enhance the behavior of programs that have no knowledge of the Frigate object model by enhancing standard UNIX file operations. Thus, almost all existing UNIX programs can transparently use Frigate.

2.7 Versions

The mechanisms provided by Frigate allow for multiple models for version management. In all cases, the repository accommodates installation of classes and implementations without any disruption of system operation. Currently active documents will continue to execute with their bound implementations; any new use will immediately reflect any repository updates. Multiple implementations for the same class can run side by side. New implementations can be installed and tested in isolation before general use by proper version management.

Documents marked with preferred versions always use the implementation so marked. This feature is used to support an "eager" model of version management. In this model, the newest accepted version is designated as preferred. In this way, documents can automatically use the most recent version. New versions, though, should not be designated as preferred until tested sufficiently.

A more selective model uses aliases to effect the update. After acceptance of a new version, the old version is replaced by an alias that maps to the new version. Any documents with the old version identifier automatically use the new version. There is no need to hunt down existing documents and update their version identifiers. As documents are used, they *may* have their version identifiers rolled forward. However, it is not necessary for *all* old versions to be replaced by aliases. Situations where there are incompatible changes in data formats or limited trust placed in a new version can be handled. This is a "lazy" model where no change is made until an old version is explicitly mapped to the new version. It is also possible to have chains of aliases that converge to allow future coalescing of versions.

In many object-oriented systems, subclasses are often defined to provide a specialized implementation even when there is no interface specialization. For example, an image object class may have subclasses for GIF and TIFF formats. Since implementation in Frigate is completely divorced from class definition, the version system can allow specialized implementations *without* subclassing. No duplication of interface is needed. Separate specialized implementations can be installed under the same class. Thus, two different versions do not necessarily represent different stages along a single line of development. They may, in fact, be parallel separate lines of implementation. In Frigate, there may be GIF and TIFF implementations for the same image class.

The version system can, in effect, provide a form of polymorphism. The advantage of doing this is that simple repository commands can give the user complete control over the binding process. By using aliases, the evolution along any particular line of development can still be transparently managed. If future implementations merge lines of development, aliases chains can be made to converge as well.

3 Examples

Frigate has a variety of possible uses. Here we outline some example possible applications of Frigate to illustrate the flexibility of the framework. We have actually implemented a transparent encrypted file system, image files, intentional files and a simple access control system.

Frigate can be used to provide file system features such as transparent compression, encryption and migration. For the most part, these are services that continue to use the ordinary system call interface. In Frigate, such services are provided transparently and automatically by redefining vnode methods. The new definitions for vnode operation enhance the behavior of standard system calls. In this way, no change to old programs is necessary to take advantage of enhanced behavior.

Another example of this type is the *intentional* file. Instead of explicitly storing the contents of a file, the contents are instantiated on the fly. Traditionally, UNIX stores information on users, hosts and networks in a series of files. Sun's NIS service modifies the standard library calls to access network servers instead of the files. Frigate can provide the same sort of service through intentional files. The file contents are generated on demand based on information from the network servers. The advantage here is that even programs that have the old library code can benefit. Dynamic information, such as network loads, logs or realtime data can similarly be provided by an intentional file interface. Intentional files can also be used as a form of compression for large but easily regenerated datasets.

Other applications might use a mode of operation that is not strictly transparent. In such cases, the interface is extended to provide additional functions. An example is an enhanced file access control system. In such a system, there are additional methods to manage access control lists, provide different levels of information to users, read logs, etc. However, the majority of the file system access (e.g., read/write) still goes through the ordinary system call interface.

Frigate also provides an object-oriented model of access. Programs using this model can take advantage of polymorphism. A single document class for image files might have several implementations, each of which services a different image format. A single program could be written to manipulate any of the image files via the document interface. The program would not have to be changed, if available formats changed or if a format was added after the program was written. This is because the implementations (servers) are completely separated from the interface (class definitions) and from the client program binary. Despite the generic nature of the program, format specific code is executed for each particular file for efficient access.

The UNIX file interface is limited to addressing file entities. Frigate's object-oriented model can address other sorts of objects. Messages may be sent directly to parts of a compound document represented by objects. This allows compound documents similar to those in OLE [35] or OpenDoc [14] to be constructed in Frigate. In this case, a document object is a container for other "content" objects, which contain text, images, audio, spreadsheet cells, graphs, etc.

Frigate's facilities can be used to provide *links*. Links are potentially interfile, persistent references to content objects that can be stored in documents. (ILU object handles are not persistent.) With a link tracking service, updates of a source can automatically be reflected in any document containing link references. Frigate can also be used to provide more general models of interfile consistency. Changes in one document can be automatically propagated, appended, merged, etc. into other documents.

UNIX file operations are low level. Frigate can provide much higher level file system abstractions. Documents might have operations to print, archive, install and move (with all other associated objects). Operations can be tailored to be more meaningful to the application. Documents are capable of any action that can be accomplished by a program, because documents are implemented by server programs. For example, a document might automatically enforce integrity constraints or warn other users of important changes by email.

4 Extensibility

Our overall goal of making the file system easier to extend has several aspects. We provide for extensions that are not resident in the operating system address space. This frees us from the classic problems of operating system development including: privileged access, source access, specialized operating system knowledge, programming and debugging restrictions, rebooting, potentially disastrous effects of bugs and redistribution restrictions.

Frigate is designed to scale. Frigate class and version identifiers are assigned in a distributed fashion. Extensions are not limited by having to all fit into a single address or overlay space. Practically speaking, Frigate is limited by runtime resources (i.e., number of simultaneous running servers).

Frigate provides an object-oriented interface to the file system. This provides a single coherent client interface paradigm, rather than a hodgepodge of ad hoc interfaces. Variants are cleanly described through interface inheritance. The object interface also provides the ability to use polymorphism with file operations. Frigate's late binding mechanism allows implementation to be changed right up until a server is required. Since the actual implementation is not part of the client program, no recoding, recompilation or relinking is ever necessary to make full use of polymorphism.

The class provider is given a structured environment that merges vnode and ILU method programming into a common framework. Frigate remains compatible with Stackable Layers but with a better programming interface. The familiar file descriptor programming model is also available to program methods.

True encapsulation that cannot be bypassed, accidently or purposefully, is provided around file entities. Redefined vnode operations can enhance the behavior of old programs without any change to the program.

Frigate provides a mechanism for flexible version management. When new versions are installed, there is no need to reboot or restart the system. Multiple implementations can run side by side with no interference. New runtime instances are automatically assigned the updated versions. There is no need to hunt down every document instance to update it.

Except for initial installation of the framework, all facilities can be used with full capabilities by unprivileged users. Anyone can write, install, debug and use extensions to the file system.

5 Safety and Security

Frigate extensions are placed in server processes outside of the operating system address space. The process boundary around each server acts as a firewall. Buggy or malicious behavior is confined to the server process. Extensions communicate with the operating system through Stackable Layers transport layer interfaces and via system calls. Extensions do not have direct access to the operating system address space. Thus, the operating system is protected from the server just as well as it is protected from any other user process.

During the servicing of a method, the server holds the same access rights as the calling process. In between method calls, the access rights are changed to that of a special unprivileged Frigate user. The rights granted during method service are identical to that given to a program executed by a user. The difference is that access rights are being granted on a lower level of granularity and access rights change over the life of the server. It is not possible to use the varying credentials to accumulate access rights. Even if access is granted previously, the next access may be barred on the basis of the submitted credentials. Overall, malicious extension code cannot do anything not possible through ordinary malicious program execution.

However, Frigate does offer more danger as a Trojan horse. Precisely because Frigate can operate transparently, it is somewhat easier to unknowingly run an untrusted Frigate extension than to execute an untrusted program. This is especially true if a user defines a malicious vnode method. Frigate attempts to mitigate the problem in two ways. First, privileged (root) clients do not pass their rights to Frigate servers. Thus, the only actions that a Frigate server can perform for a root user are those that any non-privileged program could. Root users can always disable Frigate, if necessary. In addition, it is always possible for users to examine an accessible file entity for Frigate attributes. (Frigate does not allow the relevant vnode operations to be redefined.) In this way, a user can safely determine if a suspicious file entity is an untrusted Frigate document by examining its attributes.

The Frigate repository is shielded from unauthorized modification by file protections. Requests typically issued by the operating system are only accepted from protected ports. Only class owners can modify a document class and its implementations.

Frigate offers complete encapsulation of documents. Since Frigate is a part of the operating system and below the system call interface, it cannot be bypassed accidently or maliciously. Frigate allows any specified operation to be intercepted, except those used by Frigate itself. This ensures that security or data integrity mechanisms will not be bypassed. Malicious users cannot defeat encapsulation by altering the attributes on documents. Only the owner or the root user can change the attributes on a document.

6 Compatibility

Frigate remains compatible with the vnode programming model. As previously described, some Frigate extensions provide their new functionality exclusively through vnode methods invoked indirectly through the UNIX system call interface. As a result, this type of extension provides complete backward compatibility for existing programs. Old programs work "as is" and require no changes at all, not even recompilation or relinking, to take advantage of the new functionality. This preserves the current software investment. In many cases, this means that application programmers do not need to learn new paradigms to take advantage of Frigate capabilities.

Since Frigate is packaged as a Stackable Layer, Frigate also offers an inherent form of forward compatibility. The other layers of the stack may be updated or reconfigured independently of Frigate. This means that a Frigate system can take advantage of new features in other layers without any change to Frigate itself. For example, an enhanced storage substrate might be provided by substituting a new storage layer for the standard UFS layer.

Frigate also offers *incremental* use. While Frigate offers a new paradigm, one is not forced into an "all or nothing" decision to use it. Frigate can coexist with standard UNIX. A user need only use Frigate as much as is desired. There is no need to conform all activity to the new object-oriented environment. Porting of existing facilities is not necessary because the current environment continues to exist on a Frigate system. Frigate documents are distinguished by class and version attributes. Frigate documents may be stored in any file system that can support those attributes. Otherwise, there is no restriction on storing ordinary files and Frigate documents in the same volume.

In some cases, it may desirable to port old applications to the object-oriented Frigate paradigm. For example, the benefits of being able to write generic clients, which take advantage of polymorphism, may justify the porting cost. In other cases, applications may wish to use Frigate's version management system. The incremental nature of Frigate aids application migration. Migration may be accomplished by "wrapping". Old files are converted to documents by wrapping class and version attributes around them. The addition of the Frigate attributes brings them under Frigate management. Old programs are "wrapped" as methods by having method handlers simply execute the programs directly. In this way, an old application is quickly enabled to run under Frigate. Further changes may, of course, require more extensive restructuring.

Frigate is also compatible with standard languages: OMG IDL and ANSI C. All inheritance support is confined to the ILU interface generator. Thus, no extensions to C are necessary and the investment in C compilers and programming environment can be preserved.

	Layer Configuration			
	UFS		Dispatcher Attribute UFS	
Phase	mean	stddev	mean	stddev
mkdir	4.3	2.28	4.8	1.04
cp	11.6	2.01	16.9	1.93
find	10.1	1.50	11.8	1.84
read	15.4	0.97	16.2	0.96
make	103.0	1.85	105.6	1.77
user	83.0	0.30	83.3	0.28
sys	37.7	0.35	39.9	0.36
elapsed	149.8	3.12	161.1	2.39

All runtimes in seconds.

Phase	Percentage
mkdir	112 %
cp	146 %
find	117 %
read	105 %
make	103 %
user	100 %
sys	106 %
elapsed	108 %

Frigate time as percent of UFS time.

Table 1: Non-Use Times

7 Performance

We measured Frigate's performance in two respects. First, we measured the cost of merely having, but not using, Frigate's framework. We also compared the performance of Frigate to alternative solutions.

All of our performance measurements were carried out under a SunOS 4.1.1 kernel on SPARC IPC (25 MHz, 15.8 MIPs) machines with 12 MB of memory and a SCSI 207 MB (3.5", 3600 RPM) disk. The operating system was modified to use Stackable Layers. All test results are derived from 30 runs.

7.1 Framework Overhead

We would like Frigate to have minimal performance impact when we are not using its facilities. To measure this impact, we ran the Modified Andrew Benchmark [24, 37] in a file volume with and without the Frigate service.

No direct use was made of Frigate in the benchmark. The configuration without Frigate was simply the standard UFS, packaged as a Stackable Layer. The configuration with Frigate included a Stackable Layer to implement extended (class and version) attributes and the actual Frigate Dispatcher layer. The results for each configuration and each phase of the benchmark are shown in Table 1. Frigate times are also shown as a percentage of UFS runtime.

Overall, the overhead of having Frigate, but not using it, amounts to an 8% elapsed time penalty. Additional measurements were made with just the extended attribute layer to see how much of the overhead came from that support layer. About seven of the eight percent of overall overhead comes from the extended attribute support layer. For general use, Frigate's overhead would have to be reduced. The obvious strategy would be to improve the performance of the attribute service.

7.2 Comparison to Library

Our second performance study compared Frigate to a user-level library. In many cases, the unprivileged user has few alternatives but to implement his extension as some type of user-level library. Our example extension provided file encryption services. The encryption algorithm is an enhanced version of the one used in the German World War II Enigma

File Size (KB)	Library		Frigate	
	mean	stddev	mean	stddev
8	0.3	0.05	1.3	0.44
80	2.8	0.23	4.7	1.56
800	26.4	0.18	32.5	0.55
8000	295.6	5.80	343.2	6.58

All runtimes in seconds.

File Size (KB)	Percentage
8	433 %
80	168 %
800	123 %
8000	116 %

Frigate time as percent of Library time.

Table 2: Single Client Times

File Size (bytes)	Library		Frigate	
	mean	stddev	mean	stddev
1	0.2	0.05	1.0	0.23
8 K	5.5	0.22	1.7	0.26
80 K	52.4	0.97	8.4	0.51
800 K	556.3	109.77	74.1	0.20

All runtimes in seconds.

File Size (bytes)	Percentage
1	500 %
8 K	31 %
80 K	16 %
800 K	13 %

Frigate time as percent of Library time.

Table 3: Shared File Performance

encryption machine [16]. While not a strong encryption method, this example did provide a predictable processing overhead on each read and write.

The library implementation of our encryption extension simply followed any read of the encrypted file by a function call to decrypt the incoming block of data. Writes were preceded by a call to encrypt the outgoing data. File operations were performed on a UFS file system without any additional stackable layers.

The Frigate implementation redefined the read/write vnode method to provide transparent encryption and decryption. A new ILU method was added to provide the session key to the extension. Because Frigate is positioned below the system call interface, file data could be buffered and shared between clients in the clear. To prevent unauthorized parties from gaining access to clear text data, other vnode methods were also redefined with added security features.

Our first experiment used a single client, which provided a mix of file I/O operations and compute processing. Each block in the file was read, modified, and written back into another location in the file. The elapsed time of the library and Frigate implementations is shown in Table 2. Frigate times as a percentage of library runtimes are also shown.

Overall, we see for small cases (e.g. 8 KB file size) that the library performs much better. The reason for this result is the server startup time for Frigate. As file size (and hence overall I/O) increases, the server startup time is more effectively amortized over the entire run. In both library and Frigate cases, the same amount of processing is done by the driving client application and in encryption operations. The difference in performance comes from the lower throughput of Frigate. The additional overheads in Frigate include the two additional stackable layers, the limited speed of our transport, and the need to context switch to the server process to handle requests. For general use, further performance tuning of Frigate is probably necessary. In this case, the largest payoffs in improving performance are in improving our IPC throughput and server startup time.

Our second experiment involved two clients sharing access to an encrypted file. One process read, modified, and wrote each block of the file. Before proceeding to the next block, the second client also read, modified, and wrote the same block. The entire file was processed 10 times in this fashion.

The elapsed runtimes for the library and Frigate implementations are shown in Table 3. The runtimes for Frigate as a percentage of the library runtimes are also shown. The file size, which is directly tied to the total amount of I/O, was varied in our experiment. (The 8000K case was omitted because the library solution had an excessive runtime.)

The server startup cost again makes the library implementation faster in the smallest cases. Since so little data is actually transferred in the one byte file case, it essentially only measures the overhead of the framework. However, in the larger cases, Frigate does significantly better. The ability to share text in the clear reduces runtime substantially. In the largest case shown, Frigate runs in less than one-seventh the time of the library implementation. The savings is mostly due to avoiding redundant encryption and decryption operations. Frigate need only perform these operations between its buffers and the disk. The library implementation must perform such operations on each I/O. This fact more than compensates for any I/O throughput reduction suffered by Frigate. In this case, Frigate's architecture provides a unique advantage not otherwise available.

8 Related Work

Frigate is comparable to a number of different systems. In general, we classify these systems according to their structure and functionality. A key question is where the extension is added to the overall system. File service extensions can be added to client processes, system libraries or the operating system. If extensions are implemented in servers, then some sort of "hook" is inserted into one of these locations to intercept the relevant calls.

Frigate uses servers. The intercept mechanism for Frigate is the Dispatcher layer, which intercepts selected vnode operations inside the operating system. Since all file accesses pass through the layer, Frigate can ensure that an extension has complete control over file operations. Frigate's user-level servers allow easy development. The process boundary around servers also protects the operating system from buggy extensions. The drawback is that performance can suffer due to the need for IPC between the Dispatcher and server. Frigate's object-oriented environment is provided for the client and server by using a CORBA based solution. The intercept mechanism in between is essentially ignorant of objects other than vnodes.

8.1 Languages & Libraries

Programs written in object-oriented languages can use objects with support for persistent storage. Typically, such objects can provide a high-level method interface appropriate to the application instead of the primitive system call interface. Support for persistence may be a builtin language feature. An elementary example is found in the Eiffel programming language [34]. The "storable" class provides methods to read and write an internal language representation of an object to a file. Other classes gain persistence by merely inheriting the storable class.

When languages have support for persistent objects, they can offer a natural, enhanced filing interface that is fully integrated with the rest of their language environment. On the other hand, their rich world is not available outside of their own language environments. They are not integrated with the operating system and cannot provide any encapsulation of persistent objects outside of the language environment. These systems and others that choose to remain above the system call interface can be free of the problems inherent to developing in the operating system address space. However, by not being able to go below the fixed system call interface, their flexibility and ability to enforce policy is limited. For example, it is difficult above the system call interface to even ensure unambiguous file identity. As we saw in the performance tests, it can also be a performance liability.

Most CORBA [36] based object frameworks are designed to integrate with target language environments. Thus, for the most part, they share the characteristics of language-based solutions. Object frameworks with "applet" and "serverlet" features offer a novel form of modularity. However, when actually executing, they too share the characteristics of language-based solutions. Some other object framework cases, such as OLE 2 [35] and ActiveX [8], are less clear, since it is difficult to determine how much has been subsumed by the operating system.

Another related approach uses libraries. Commonly used system libraries are modified to enhance the behavior of file operations. Often these libraries are shared and dynamically loaded. This allows existing compiled programs to use the modified library without recompilation. Of course, statically linked programs will not be able to take advantage of redefined, enhanced operations and will break encapsulation. Examples of library systems are the 3-D File System [29], COLA [30] and IFS [12]. These particular systems provide additional (albeit not object-oriented) functionality by intercepting library calls to the standard UNIX system calls. Like language-based solutions, library systems reside above the system call interface and thus inherit the associated tradeoffs.

8.2 Servers

Another form of operating system extension is the server. The server usually executes at user-level outside of the operating system (or at least outside of privileged execution modes). The intercept mechanism may be above or below the system call interface. Intercepted calls must be passed to the extension by some form of IPC. This may cause some loss in performance.

The Object-Oriented File System [47] intercepts UNIX system calls at the library level and passes them to servers using pipes or UNIX domain sockets. The modified library must be explicitly linked with clients and thus will only work with new applications. Despite its name, it does not offer an object-oriented programming interface. There are no classes, methods, inheritance, etc. in this system.

The Distributed Object-Based System (DOBS) [11] provides automatically started servers that respond to object methods. Communication between client and server process is through SunRPC. This system is similar in flavor to the ILU part of Frigate. DOBS defines its own simple interface language to describe methods. It does not appear to have the ability to manipulate any non-file objects, nor is there any inheritance. Also, it does not have any integration with the UNIX file operations. No existing UNIX file operations can be redefined; one can only define new operations. As a result, the system cannot be used to affect the behavior of existing programs and encapsulation is not enforced.

Some operating systems offer special debugging mechanisms to intercept system calls. As long as the intercept can be properly set up, complete encapsulation is possible. Interposition Agents [26] use this strategy. An object-oriented toolkit is provided as a framework for implementing new services. However, the toolkit is not exposed to clients. Instead, each agent is provided with a symmetric system call interface allowing the layering of agents. Thus, clients continue to see the same unextended system call interface. In this system, agent(s) are attached to specific clients and are apparently not shared.

The File Monitor Interface [49] provides a user-level server facility that responds to vnode calls intercepted inside the operating system. The intercept mechanism is similar to the Frigate Dispatcher and can selectively intercept calls based on special file attributes. However, there is no provision for an object-oriented programming model.

Similar facilities were also built for other operating systems. Pseudo-File-Systems [51] were developed for Sprite [38]. Userfs [15] provides this service for LINUX. Stackable Layers also provides this functionality on vnode operations through transport layers. These three systems use the mount mechanism to provide a volume granularity redirection to servers. They also require servers to be started prior to use.

Watchdogs [4] and Extensible Streams [41] go somewhat further in that servers are started automatically, if none is running. Both also use type identifiers similar to Frigate's class attributes. This allows specifying servers on a file level granularity.

Micro-kernels take the server approach and apply it to reorganize the entire operating system. A modest number of basic abstractions are implemented in a small ("micro") kernel. The bulk of the operating system is moved into separate servers. Examples include Mach [1] and Chorus [45]. Exokernel [13] and Cache Kernel [9] attempt to push this organization as far as possible. In a micro-kernel, the file system is usually implemented by one or more servers. In the process of reorganization, many micro-kernels and their file systems have also acquired an object-oriented flavor. An example is Mach 3.0 [17, 46]. While this object orientation is of great use in the *internal* development and specialization of the operating system, it is not generally exported to users. Users continue to see the same unextended system call interface.

8.3 Extensible Operating Systems

A number of approaches to increase extensibility of the traditional file system have been tried. Early attempts to add extensibility interfaces to the operating system included streams [42] and dynamically loaded device drivers. The latter is included in systems such as SunOS and Chorus [2]. Other attempts to restructure the file system include VFS [28] and UCLA Stackable Layers [22, 23], which we have already described. An alternative model of stacking is described in [44]. The Spring operating system [27, 40] also offers stacking with additional object-oriented features.

Recently some new operating systems extend their functionality by downloading "safe" modules into the operating system. The modules are made safe by restricting address references [48] or by using safe languages. An example of the latter is the SPIN operating system [5], which uses Modula-3 [7] as its extension language. With the use of the techniques mentioned above relative safety can be assured. In some cases, though, there is limited space to add extension code and thus large extensions are not possible.

Unfortunately, any facility requiring development and debugging in the kernel address space needs special tools and privilege. Also, these tools were designed for expert specialization of the operating system. They are not meant for casual users. However, these approaches potentially perform better than servers, because there is no need to cross process boundaries to reach the extension.

8.4 Object-Oriented Operating Systems

Another approach is to provide operating systems that are built on an object-oriented paradigm from the ground up. The services they offer, including the file system, are object-oriented. Extensibility is provided by using subclasses and polymorphism. This is to be distinguished from merely writing a system in an object-oriented language. See [21] for further discussion on this point. Of course, the two are not mutually exclusive; many such systems are actually written in an object-oriented language or use one for its client interface.

Choices [6, 32, 33] implements an object-oriented operating system written in C++ with some extensions. Operating system abstractions, including files, are provided as objects in a class hierarchy. Extensions can be added by subclassing existing classes. However, the extension framework is not targeted for use by the ordinary user. Rather, it is oriented toward controlled specialization of the operating system by privileged expert users. Choices does try to address the problems of operating system debugging by providing a user-level emulator.

Clouds [10, 39] offers a complete object-oriented operating system. All services in Clouds are offered as objects accessed through capabilities. The radical persistent object model makes no distinction between memory objects and file system. Essentially, all objects are located in a large distributed virtual memory. Objects in Clouds are relatively large grain with operating system enforced encapsulation. The COOL [20] system provides a similar model built on top of the Chorus micro-kernel.

True object-oriented operating systems can provide powerful extensible environments. Their main drawback is their incompatibility with the rest of the world. The current investment in software is lost. Any desired feature must be ported and possibly rewritten to fit with the new paradigm. One must make an "all or nothing" switch to the new system. Only a few systems such as Solaris MC [3] and Frigate explicitly attempt to integrate compatible use with a new object-oriented operating system interface.

9 Future Work

Frigate presents a number of directions for future work. These concepts are enhancements that can be incrementally added to the current, fully operational system.

Frigate is currently implemented on the SunOS 4.1.1 infrastructure of Stackable Layers. As Stackable Layers is ported to other platforms, Frigate can also be ported with only modest effort. On its current platform, the Frigate client environment could also be expanded to include other ILU languages such as C++ and Modula-3 [7].

Some improvements could also be made to the server structure. The most fundamental would be to allow object types to be dynamically loadable. This would allow implementations to be decomposed into a shared server executable and an application specific set of dynamically loadable implementations. Servers would tailor themselves while running and load implementations only according to demand. Currently, a compound document server must have implementations of all possible component classes linked into the executable. In a dynamic architecture, compound documents would load only what was demanded by the access pattern of their constituent components. New classes and versions would not require relinking a server executable. In effect, we would be extending late binding to server binary construction as well.

Another possible improvement is to allow servers to be distributed on other hosts. This would allow additional server configuration flexibility and better load balancing. To allow this possibility, a new distributed alternative to the Stackable Layers user-to-kernel transport must be provided along with some enhancements to the binding process.

A number of performance improvements are possible as well. Probably, the largest payoffs in performance are in improving the separate attribute service, the communication throughput with the server and the server startup time. For servers on the same host, considerably better performance is probably possible. Current kernel-to-server transports are based on implementations of NFS. Much more efficient interprocess communication could be used. Improvements would probably use strategies similar to those used in micro-kernel designs (see [31]).

Server startup is an expensive process involving new process creation and process image overlay. When in the critical path of execution, this can result in a substantial delay. One strategy to improve things would be to cache servers. Servers would not be killed until a timeout period had expired. Further uses of the same implementation would not require a new server to start. Another complementary strategy would start up a number of dynamically loadable servers ahead of time. Unassigned servers may even be preloaded with some implementations. As servers are needed, they are assigned from the pool of servers. In this way, the server startup time is avoided, though, the time cost for dynamic loading must still be paid.

10 Conclusion

Operating systems and the file systems they contain have been designed as general purpose instruments. Specialized file system support for applications could provide great benefits. Unfortunately, it is very difficult to extend operating systems and file systems. What is needed is a file system infrastructure with extensibility designed in. While some extensibility features have appeared, they do not go far enough in providing open access for extensibility.

Frigate attempts to provide such open access to all users, not just to those with system privilege or specialized training. The philosophy driving Frigate is that the user or ordinary programmer is best equipped to add value to his applications, not some distant expert. Therefore, the challenge is to expose the power of underlying mechanisms in a safe and easy to use way.

Towards this end, Frigate combines a user-level server framework, which is fully integrated with the operating system, with an object-oriented interface. The user-level structure provides adequate performance and, at the same time, frees developers from the constraints of development in the operating system address space. It allows the freedom to provide an implementation that does not have to mirror the structure of a corresponding implementation inside the operating system address space. It is *expected* in this environment that untrusted extensions will be run. We use the server process boundary as a firewall, protecting the operating system and other extensions.

The object-oriented framework provides a coherent programming model for extension that provides the benefits of inheritance, polymorphism and encapsulation. Users do not have to learn a new programming paradigm for each extension. Inheritance allows the leveraging of previous work. Polymorphism allows generic programs to be written that need not change as new versions or types appear. Frigate also provides full encapsulation that is enforced and cannot be bypassed.

Frigate provides compatibility. By being able to redefine vnode operations, Frigate can extend the behavior of existing programs. Vnode operations are smoothly integrated into the object-oriented programming model. Frigate uses standard languages and compilers. The implementation of object-oriented mechanisms are carefully isolated so that current compilers need not be disturbed. The value of the current software investment is preserved. By using Stackable Layers technology, Frigate can also take advantage of new work packaged as layers.

Frigate allows incremental use. Frigate objects can be freely mixed in the same file system with ordinary files. Frigate code can cooperate with other programs using the standard system call interface. Frigate does not force one to chose between paradigms. Frigate can be used as much or as little as desired. It is not an "all or nothing" approach.

Frigate also provides a structure for controlled change without any disruption of operation. Not only are interfaces named but versions are as well. Particular versions can be named and relevant distinctions can be maintained. Yet at the same time, versions can be transparently upgraded. The installation and upgrade of versions never causes a need to reboot, reinitialize or interrupt operation of the system.

Frigate scales well. Interfaces and versions can be developed independently of each other without fear of name clashes. The namespace for interfaces and versions is assigned in a distributed fashion without the need for a central registry. Practically speaking, Frigate is only limited by runtime resources for server processes and not by arbitrary design limits.

Other systems have only addressed a few of these aspects. In short, we believe Frigate uniquely opens the file system to the world.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer Conference Proceedings 1986*, pages 93–112. USENIX Association, June 1986.

[2] F. Armand. Give a Process to Your Drivers! In *Proceedings of the EurOpen Autumn 1991 Conference*, Sept. 1991.

[3] J. M. Bernabeu-Auban, V. Matena, and Y. A. Khalidi. Extending a Traditional OS Using Object-Oriented Techniques. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems*, pages 53–63. USENIX Association, June 1996.

[4] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *USENIX Winter Conference 1988 Proceedings*, pages 267–275. USENIX Association, Feb. 1988.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Backer, C. Chambers, and S. Eggers. Extensbility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284. ACM, Dec. 1995.

[6] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-

Oriented System in C++. *Communications of the ACM*, 36(9):117–126, Sept. 1993.

[7] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report (Revised). Research Report 52, DEC Systems Research Center, Nov. 1989.

[8] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.

[9] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[10] P. Dasgupta, R. J. LeBlank Jr., and W. F. Appelbe. The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work. In *The 8th International Conference on Distributed Computing Systems Proceedings*, pages 2–9. IEEE Computer Society Press, June 1988.

[11] P. Dewan and E. Vasilik. Supporting Objects in a Conventional Operating System. In *Proceedings of the Winter 1989 USENIX Conference*, pages 273–285. USENIX Association, Jan. 1989.

[12] P. R. Eggert and D. S. Parker. File Systems in User Space. In *Proceedings of the Winter 1993 USENIX Conference*, pages 229–240. USENIX Association, Jan. 1993.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266. ACM, Dec. 1995.

[14] J. Feiler and A. Meadow. *Essential OpenDoc*. Addison-Wesley, New York, NY, 1996.

[15] J. Fitzhardinge. *Userfs – Filesystems Implemented as User Processes*. Softway Pty. Ltd., Aug. 1995.

[16] S. Garfinkel and G. Spafford. *Practical UNIX Security*. O'Reilly & Associates, Sebastopol, CA, 1991.

[17] P. Guedes and D. P. Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pages 114–117. IEEE Computer Society Press, Oct. 1991.

[18] R. G. Guy. Ficus: A Very Large Scale Reliable Distributed File System. Technical Report CSD-910018, UCLA Computer Science Department, June 1991.

[19] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71. USENIX Association, June 1990.

[20] S. Habert and L. Mosseri. COOL: Kernel Support for Object-Oriented Environments. In *OOPSLA/ECOOP '90 Proceedings*, pages 269–277. ACM, Oct. 1990.

[21] G. Hamilton, Y. A. Khalidi, and M. N. Nelson. Why Object Oriented Operating Systems are Boring. In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pages 118–119. IEEE Computer Society Press, Oct. 1991.

[22] J. S. Heidemann. Stackable Design of File Systems. Technical Report CSD-950032, UCLA Computer Science Department, Sept. 1995.

[23] J. S. Heidemann and G. J. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.

[24] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[25] B. Janssen, D. Severson, and M. Spreitzer. *ILU 1.8 Reference Manual*. XEROX Palo Alto Research Center, Mar. 1995.

[26] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93. ACM, Dec. 1993.

[27] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14. ACM, Dec. 1993.

[28] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer Conference Proceedings 1986*, pages 238–247. USENIX Association, June 1986.

[29] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the Summer 1989 USENIX Conference*, pages 147–156. USENIX Association, June 1989.

[30] E. Krell and B. Krishnamurthy. COLA: Customized Overlaying. In *Proceedings of the Winter 1992 USENIX Conference*, pages 3–7. USENIX Association, Jan. 1992.

[31] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188. ACM, Dec. 1993.

[32] P. Madany, R. Campbell, V. Russo, and D. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In *ECOOP '89 Proceedings*. Cambridge University Press, July 1989.

[33] P. W. Madany, D. E. Leyens, V. F. Russo, and R. H. Campbell. A C++ Class Hierarchy for Building

UNIX-Like File Systems. In *USENIX C++ Conference 1988 Proceedings*, pages 65–79. USENIX Association, Oct. 1988.

[34] B. Meyer. *Object Oriented Software Construction.* Prentice-Hall, New York, NY, 1988.

[35] Microsoft Corporation. *Object Linking & Embedding Version 2.0 Design Specificaton*, Apr. 1993.

[36] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report 96.08.04 (Revision 2.0), Object Management Group, July 1996.

[37] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256. USENIX Association, June 1990.

[38] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[39] D. V. Pitts and P. Dasgupta. Object Memory and Storage Management in the Clouds Kernel. In *The 8th International Conference on Distributed Computing Systems Proceedings*, pages 10–17. IEEE Computer Society Press, June 1988.

[40] S. Radia, P. Madany, and M. L. Powell. Persistence in the Spring System. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 12–23. IEEE Computer Society Press, Dec. 1993.

[41] J. Rees, P. H. Levine, N. Mishkin, and P. J. Leach. An Extensible I/O System. In *USENIX Summer Conference Proceedings 1986*, pages 114–125. USENIX Association, June 1986.

[42] D. M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, Oct. 1984.

[43] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[44] D. S. H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the Summer 1990 USENIX Conference*, pages 107–117. USENIX Association, June 1990.

[45] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report CS/TR-90-25, Chorus systèmes, Apr. 1990.

[46] J. M. Stevenson and D. P. Julin. Mach-US: UNIX on Generic OS Object Servers. In *Proceedings of the 1995 USENIX Technical Conference*, pages 119–130. USENIX Association, Jan. 1995.

[47] S. Summit. Filesystem Daemons as a Unifying Mechanism for Network Information Access. In *Proceedings of the Winter 1994 USENIX Conference*, pages 63–77. USENIX Association, Jan. 1994.

[48] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, Dec. 1993.

[49] N. Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the Winter 1993 USENIX Conference*, pages 219–228. USENIX Association, Jan. 1993.

[50] J. Weidner. An General Purpose Extended File Attribute Service as a File System Layer. Technical report, UCLA Computer Science Department, 1997. To appear.

[51] B. B. Welch and J. K. Ousterhout. Pseudo-File-Systems. Technical Report CSD-89-499, University of California, BerkeleyComputer Science Division, Oct. 1989.

[52] L. Zahn, T. H. Dineen, P. J. Leach, E. A. Martin, N. W. Mishkin, J. N. Pato, and G. L. Wyant. *Network Computing Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1990.