University of California
Los Angeles

Frigate: An Object-Oriented File System

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Ted Hoon Kim

1998

The dissertation of Ted Hoon Kim is approved.

Richard R. Muntz

D. Stott Parker

Kirby A. Baker

Wesley W. Chu, Committee Co-chair

Gerald J. Popek, Committee Co-chair

University of California, Los Angeles

1998

To Him who is able to do immeasurably more than all we ask or imagine

Table of Contents

List of Figures

List of Tables

Acknowledgments

This research would not have been possible with out the expert guidance of my advisor, Dr. Gerald Popek. I would like to also thank Dr. Wesley Chu, Dr. D. Stott Parker, Dr. Richard Muntz and Dr. Kirby Baker for serving on my dissertation committee. I am also grateful for all the support provided by Dr. Peter Reiher and the members of the Ficus research group, especially Dr. John Heidemann. Generous financial support was provided by ARPA research contract DABT63-94-C-0080. I would also like to thank Monique, Janice and Alta for arranging schedules under difficult circumstances.

Of the many who have encouraged me, I would like to especially thank Tom Page, Caroline, Verra Morgan, my mother Young, my brother Yul and his family, the members of my Men's Fellowship Group (Steve, Erik and Peter) and the prayer warriors at Venice.

Vita

1962	Born, Boston, Massachusetts.
1981-82	Computer Lab Assistant, MIT EECS Department.
1982-84	Systems Programmer, MIT Laboratory for Computer Science.
1984	B.S., Computer Science and Engineering, MIT.
1984-85	Teaching Assistant, UCLA Computer Science Department.
1985-87	Research Assistant, UCLA Computer Science Department.
1988-92	System Administrator, UCLA Computer Science Department.
1992-97	Research Assistant, UCLA Computer Science Department.
1997	M.S., Computer Science, UCLA.

Publications

Ted Kim. "XWIP Reference Manual, Version 0.4." Technical Report CSD-880079. UCLA Computer Science Department, October 1988.

Ted H. Kim and Gerald J. Popek. "Frigate: An Object-Oriented File System

for Ordinary Users." In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, pp. 115–129. USENIX Association, June 1997.

Abstract of the Dissertation

Frigate: An Object-Oriented File System

by

Ted Hoon Kim

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1998

Professor Gerald J. Popek, Co-chair

Professor Wesley W. Chu, Co-chair

It is impossible to design and implement file systems for all possible uses. Only a few general purpose services can typically be included in an operating system. However, it is highly desirable to be able to modify or extend file systems within an operating system.

Modifying the operating system for any purpose has traditionally been a difficult, if not impossible, task. Micro-kernel architectures, Streams, vnodes, loadable device drivers, and various "file system in a server" schemes have all attempted to add some measure of extensibility. We propose a new file system extensibility approach based on Frigate, an object-oriented file system.

Frigate provides a modular, extensible file system framework that allows new extensions to be "plugged-in" on the fly. Frigate is based on user-level servers but with some new twists. The interfaces are cast into an object-oriented framework to provide the advantages of polymorphism and inheritance. Complete integration with other file systems is provided by building on the Stackable Layers architecture. In addition, the design allows for compatible and incremental use.

The goal of Frigate is to lower the barriers to file system extensibility. Unlike most other systems, Frigate is targeted for use by *ordinary users* rather than by sophisticated operating system gurus. This dissertation describes the design, implementation, use, and performance of Frigate.

CHAPTER 1

Introduction

It is impossible to design and implement file systems for all possible uses. The designers of operating systems have rightly sought to provide implementations of general purpose services that are widely usable. But at the same time, they have also conceded the ability to support some specialized applications. Many applications that could benefit by specialized file services are then left with unenviable alternatives that lead to awkward implementations. In a few cases, through special vendor arrangements, there has been special support for certain applications such as databases. However, only large companies are capable of successfully lobbying vendors for special features. The ability to support any one of a myriad of applications is also limited by the simple fact that only the users, themselves, know what is required.

This leads one to an alternative strategy. That strategy is to modify the file system ourselves. Indeed, the aim of our research is to provide the means to easily extend the file system. Unfortunately, in most systems available today, the barriers to extending the file system are insurmountable. This can result in new functionality being placed where it does not belong simply because it is easier (or only possible) to do it elsewhere.

Some systems, mostly research systems, have sought to make this task easier by providing specific features for extensibility. Instead of attempting to anticipate all user needs, an interface is provided to allow parts of the operating system to be modified or extended. We also take this route, though our approach is somewhat different from that taken by the other systems. Before going further, we first outline the nature of the challenge before us.

1.1 Problem Definition

The fundamental problem we wish to address is how to easily *add value to the file system.*

The "file system" can be many things to different people. Traditionally, it is thought of as a persistent storage system based primarily in disk media. It is usually considered distinct from backup or archival storage systems, except where transparently integrated to appear under the same interface as disk files. By the same token, some other services such as so-called "RAM disk" or address space mapping are also often integrated under a file interface, even though they do not store persistent data.

File systems usually have a namespace designed to be human readable. Within the operating system, the file system is usually considered to occupy a position above the device drivers but below the system call interface. For our purposes, we would like to use a broad definition, where the file system is not so much defined by media or even persistent storage, but that it offers services through the file namespace and interface. While we don't define the file system by its position in the system architecture, we will see that there are advantages to placing functionality in the operating system.

Our desire is to "add value" to the file system. When we speak of adding value, we mean modifying or extending the file system to include any sort of useful functionality germane to our application. Traditionally, file systems have something to do with fixed data storage, though perhaps with special policies or characteristics (e.g., replicated, encrypted, compressed, batched, cached, or traced behavior). But with our broad definition, it can also include such things as generation of data on the fly, sending messages to file system objects, automatic maintenance of inter-file consistency, etc.

Finally, we wish to add value "easily". This characteristic is more subjective and only has a relative qualitative meaning. However, that is not to say that it is entirely intangible. In this area, our goal is to lower the barriers to extending the file system to the point where an ordinary unprivileged programmer can write, install and use an extension. We describe the barriers to extension in more detail below.

1.2 Barriers to Extensibility

Modifying the operating system for any purpose has traditionally been a difficult, if not impossible, task. The problem is that operating systems have generally not been designed with extensibility in mind. In recent years, some systems have made efforts in this direction especially to deal with heterogeneous hardware. Examples include features such as loadable device drivers, streams [Rit84], vnodes [Kle86] and micro-kernel architectures. There have also been various "file system in a server" schemes (e.g. [RLM86, WO89]). Nevertheless, on the whole, modification is still a daunting task.

The operating system, by its very nature, is a rather special program. It lies at the foundation of the system, mediating user access to system resources. As such, it is heavily optimized and makes use of privileged modes of operation. Commercial considerations have often made it a proprietary product or an instrument of market dominance.

Since the operating system is a privileged program, security and safety demand that access or modification requires special privilege or access rights. The problem is especially apparent when the system is shared between several users of different privilege levels. In short, only trusted users are allowed to change the operating system. Less privileged users cannot modify it.

Assuming we can modify the operating system, we also need programming tools compatible with the operating system environment. In the few systems that have any sort of extension interfaces, say for device drivers, only a small addition to the normal programming environment is necessary. But for many other systems, this implies having a substantial part of the operating system source (plus a special non-disclosure contract) and perhaps a special compiler. This can be costly and difficult to obtain because of the usual commercial considerations.

Programming for this environment can be quite challenging. Even if well structured, the operating system is a large program and may require lengthy study to understand the internals. Also, there are special programming restrictions that arise from the operating system's role in the system. The code is shared, reentrant and possibly multithreaded. Depending on the exact synchronization strategy taken, locking or similar mechanisms are required. The code must deal with possibility of blocking and deadlocks. Memory allocation is more tightly circumscribed. Programming in this environment requires more effort than in the user environment.

Debugging an operating system modification also presents special challenges. In order to debug user processes, the operating system provides the necessary support for underlying facilities, such as breakpoints. Debuggers combine this support with source code browsing. Often, support for debugging the operating system, itself, however, is nonexistent or very primitive. The interface to such debugging might be a low level ROM monitor. Full source debugging often requires a second system running a specially enabled debugger program. In this environment, a compile-debug cycle may be rather lengthy due to the time required to build such a large program as the operating system and the additional time to reboot. Debugging in this environment requires more effort than in a user environment.

After writing an extension, we must install it. For a monolithic operating system, this means rebooting the system. If there is a dynamically loadable interface, installation may not be as disruptive. But even when dynamic loading is available, unloading or replacement is usually not as easy, or is impossible, and requires rebooting.

If there is a bug, the consequences can be disastrous. Depending on the system's fault tolerance features, a bug may cause the system to run inefficiently, crash or worse destroy valuable data. Also, security may be compromised accidently or maliciously.

Even after an extension appears debugged, a modification may not be acceptable to other users of the system. Because the operating system may potentially affect everything on a system, untrusted modifications are risky. Often, errant code may potentially interact with large parts of the operating system, because it is not isolated by protection mechanisms from the rest of the operating system. Formal verification is not likely to be practical. The code can, of course, be reviewed and audited, but this can be a labor intensive task. Often, trust can only be built by a long period of experience, unless special mechanisms exist to limit risk to acceptable levels.

After an extension is debugged, and trusted, there is still further need to cope with installation issues. Inevitably, there will be some opportunity or incentive to improve the implementation. An upgraded version may be faster, smaller, more elegant, etc. The same problems associated with debugging, installation and trust can then reoccur.

Many of these issues arise again when trying to distribute enhancements. Often there are restrictions on redistributing source or even binary changes. Sometimes, source changes are only usable by sites possessing operating system source. Binary changes may not be compatible with other local changes, configurations and patches. Trust issues become magnified when exporting modifications.

1.3 Ad Hoc Extensions

Even if we are able to write extensions, there are further problems of software engineering. If later upgrades add new features, then there is a question about how to distinguish versions. Programs depending on newer features will naturally require the correct version to operate. How can such programs be assured of getting the correct version? Simply getting an error during program execution indicating that a vital feature is missing is not good enough. There must be a way to distinguish versions short of attempting to invoke new features.

Perhaps, some older programs will still require an old version. This can arise, for example, if the application source is not available for recompilation or if conversion of files to a new format has not taken place yet. Most systems do not have any provision to run both old and new versions side by side without problems.

When we need to address a slightly different need, we would like to adapt our current work. Naturally, when creating a variant of an existing package, our desire is to reuse existing interface and implementations. Unfortunately, most file systems have little support for adapting and leveraging existing work.

If two different authors write operating system extensions independently, there can be a number of additional problems. There is a chance that the extensions will not work together. This problem can arise for a variety of reasons. Of course, if the conceptual designs are incompatible, there is no hope. Even if they are conceptually compatible, their implementations may not be. If the two extensions are *similar* in purpose, they are more likely to tinker with the same facilities, though perhaps in different and incompatible ways. Source changes may be incompatible or there may be programming name clashes. The extensions may not be separable from a binary distribution, thus only allowing one to run at a time. If they can run on the same system, then the question becomes whether they can cooperate rather than merely run separately.

From the user point of view, using different extensions may mean learning totally different interfaces and programming paradigms. This lack of coherence and added training cost may prevent the combined use of extensions, even when technically feasible.

1.4 Dissertation Roadmap

In the remainder of this dissertation, we describe the Frigate[1] object-oriented file system that seeks to address the problems we have previously described. Frigate is specifically designed for easy, modular file system extensibility. We use a framework that provides extensions in user-level servers with an object-oriented

[1]Our system originally started out with the name SLOOP, which had an obvious "OO" connection. When we discovered another project had already used the name, we switched to a "sister" ship class. Despite rumors to the contrary, Frigate is not an acronym.

interface and a version system.

In the next chapter, we describe our design goals and the limits of our design space. In particular, we require a system usable by ordinary users with incremental compatibility. We also advocate object-orientation to help solve some of the software engineering difficulties inherent in any such system.

The third chapter describes the architecture of the Frigate system. There, we first present some background material on the Stackable Layers system upon which Frigate is built. The rest of the chapter details the architecture of Frigate.

In the following chapter, the programming interface for the system is described. Some background material on the CORBA implementation used by Frigate is presented as well.

In chapter five, the uses and potential uses of the system are described. Frigate functionality is summarized and evaluated.

The following chapter reviews related work.

Finally, in the last chapter, we give our conclusions and discuss future work.

CHAPTER 2

Design Goals

The previous chapter listed a number of challenges for a system to confront. A variety of approaches to solving these problems are possible. Below, we outline the goals and principles that guide our design choices.

2.1 Ordinary Users

Most systems desire ordinary users as clients, but here we would like to allow such users to be extension writers as well. By "ordinary", we mean users without special access and privilege. The motivation for this goal arises out of our previous discussion. Distant experts are unlikely to be able or willing to provide what users and their applications need.

This goal means a special mechanism must be provided to allow modification or installation of operating system components. To extend the file system, we do require extension writers to be programmers familiar with user-level system programming. Beyond that, we would like to introduce as few new concepts as possible. The implication here is that no special training (e.g., OS internals class) or proprietary knowledge ought to be required. We also rule out access to operating system source or special tools. This requirement almost demands that the extensions be done as user-level code.

Using user-level code frees us from most of the onerous problems of operating system programming described previously. In particular, programming is free of the special constraints on operating system code (regarding threading, memory allocation, synchronization, etc.) and can be debugged readily with ordinary debuggers. The system would not have to rebooted for installation or version changes. In short, this is the program development environment that users already know. The question, though, remains how to integrate this with the existing file system.

Opening up the filing service this way may pose its own problems of scale. We may have to deal with many extensions being written. This demands a large namespace. We would like to avoid name or version clashes (e.g. MS-DOS [Wol94] interrupt vectors), especially when independently developed extensions are brought together on the same system without any prior coordination. Independent (and possibly) isolated development means a centralized name registry cannot be used. Thus, we will need a distributed naming solution to generate unique identifiers for extension names.

During system operation, there may be many extensions running simultaneously. In this regard, we would like to be limited only by the runtime resources of the system used by the extensions. We do not want arbitrary limits like the size of the operating system address space or fixed preallocated table space.

If anyone can add extensions, then buggy modifications can also be added. The question of trust would seem to be exacerbated. Extensions must not crash the system. In fact, an extension failure must have minimal effects on other users. We contend that some type of firewall is necessary to limit negative effects, since no cheap method of code examination or verification is available.

2.2 Object-Orientation

Some of problems we attributed to ad hoc extensions can be addressed by using an *object-oriented* approach. While a general discussion of object-orientation is beyond the scope of this document, the salient characteristics can be described here. Our description is partly based on OMG CORBA terminology [Sol90]. For a more complete exposition, see [CN91], [Mey88] or [FRS96].

Object-orientation is modeling by using abstractions of real world objects of the problem domain. In so far as possible, all relevant real world objects of the problem domain are explicitly represented at some appropriate level of detail. This direct one-to-one correspondence between model objects and real world objects is what distinguishes this from other approaches. Everything is organized on the basis of objects and what actions can be performed on or by an object. The intent is to make the model intuitive and easily understandable. In computer science, we generally speak of an object as a self-contained software module with private internal state and mechanism and a well defined public interface. For example, in a mail-order business, an object-oriented model might represent the mail-order goods, order forms, factory, warehouse, bank and post office as objects.

The interface is composed of *operations*, corresponding to the actions that can be taken on the object's real world counterpart. The syntax of an operation (i.e., name, arguments, types of arguments) is called its *signature*. A *result* is a piece of information returned to the invoker of the operation, such as a return value. An *exception* is an indication of an error condition. Signatures may also specify the results, exceptions and their types. An invocation of an operation is called a *request* or *message* to the object. In response to the request, an object selects implementation code corresponding to the operation specified by the request. The implementation is called a *method*. After processing by the method, results or exceptions are returned. In our example, operations on the factory and warehouse objects would update their internal state. Thus, when goods were moved from the factory to the warehouse, an accounting program would make requests on the objects to reflect the new state. Messages might also be sent to the goods, themselves, if their state changes during the move.

In many object-oriented programming languages, a request is essentially mapped into a procedure call. In distributed object systems, however, requests often involve interprocess communication (IPC). Thus, requests must be marshalled and transmitted on the client side. At the remote object location, the request is received and unmarshalled before a method can be invoked. Results and exceptions must likewise be transmitted back to the client. It also possible for remote objects to be in a *passive* state and not ready for execution. In such cases, the object needs to be *activated* before method execution. Later, when the object no longer needs to be *active*, it may also be desirable to *deactivate* it.

The separation between public (interface) and private (implementation) parts of the object is called *encapsulation*. The only way to manipulate an object is through its methods. External code sees the object as a black box. Since all object interaction is explicit, rather than through "side-effects", analysis of object behavior is simplified. External code cannot become dependent on behavior not represented in the interface. If the interface remains the same, the implementation is free to change without affecting external code. Objects provide a natural unit of modularity. Changes tend to be localized to objects, which eases software maintenance problems.

When many objects exist with common characteristics, they may be generalized into a *class*. A class is a common pattern to create objects, usually with identical interfaces, mechanism and structure, but with different state. Each object created from a class is called an *instance*. Depending on the system, a class, itself, may be an object. Classes serve to simplify analysis of large numbers of objects and in some cases can imply the sharing of common code and data. In some systems, specially designated *factory* objects are responsible for creating instances. In our business example, we might organize our system to have a class for each product model. Individual instances would represent actual units of a particular product.

Object-oriented systems deal with variants or specializations of an object with the notion of *inheritance*. When object B inherits from object A, it takes A as a basic pattern for itself. It may modify this base structure to add, change or remove components. Only differences need be described. The advantage here is that parts of an object can be reused whether in the design process or by actual sharing of code. This results in saving of design effort and, perhaps, a smaller, more efficient implementation. In some systems, inheritance enables the reuse of interface elements. In others, implementation code can also be reused and possibly shared during runtime.

Most commonly, inheritance is married with classes and not directly available on objects themselves. A derived class is called a *subclass*. The original class is called a *base* or *parent* class. (The "parent" analogy can be taken further by speaking of *ancestor* or *descendent* classes.) Sometimes a derived object may have an operation that has the identical signature of an operation in the base class, but a different method. This is sometimes called an *overloaded* operation. Overloaded operations allow subclasses to be invoked the same way but at the same time have different, more appropriate or specialized implementations. In our business example, when a model variant is introduced, we might define a subclass of the base model to represent it. If a family of products is introduced, there might be several subclasses of a common parent class. The parent class would define those characteristics shared by all models. The subclasses would only have to define their distinctive features.

If different objects have methods with the same signature or signatures that can be made to conform to each other, a powerful type of abstraction, called *polymorphism*, can be used. Programs can be written generically to use the common method transparently without regard to the specific object implementation, even though the method implementations may be radically different. Most commonly, this situation arises through class inheritance, where various derived classes share the same method signatures of a common ancestor class via overloaded methods. For example, our product family parent class may define a method to calculate an approximate life-cycle costs. Each subclass would refine that method for its particular configuration.

The key point is that specialized code can be called in each case without having to change the generic program. New classes can be added latter, or method implementations can change without having to modify the program. (Depending on how binding of interface to implementation is accomplished, it may still be necessary to recompile or relink the program.)

In object-oriented approaches, analysis and design focus on identifying the objects and their methods. Most of the functionality tends to be in the object methods. In contrast, the traditional imperative programming model instead concentrates on the process or control flow (i.e., the statements) as the basis for program structure. While separating interface from implementation is a practice advocated by good software design in general, object orientation goes beyond the notion of abstract data types in such areas as classes, inheritance and polymorphism.

In file system extension work, object-orientation can help us in several ways. It provides a powerful model for new extensions. If a new type of file or file object is introduced, it is natural and intuitive to describe it as an object. It provides a single coherent interface style for extensions. Users of extensions need not worry that each extension brings a new paradigm with it. This makes the system easier to use. Polymorphism provides a powerful way to write generic programs that do not have to change with each new kind of object. Programs do not have to change to use objects that were unknown at the time the program was written.

From an implementation point of view, objects are a natural unit of incremental modular extension. Variants of extensions are dealt with by using inheritance, potentially reusing many components and saving time and effort. We would also like to offer *complete* encapsulation for our extension objects. This will allow our extensions to provide higher levels of security and integrity. It will not be possible to accidently or maliciously get around our extensions.

In general, object-orientation is a stepping stone towards ease of use. We would like a system that is easy to use from both a client and an implementor point of view. Implementors also include those who maintain current extensions, add new versions and add new derived extensions. As stated earlier, ordinary users should be able to assume the roles of both client and implementor. The effort in using the system should be low enough that “small” extensions are practical to write and use.

2.3 Versions

Version control is not an object-oriented concept per se, but it is related. In polymorphism, new objects or classes use the same interface to transparently substitute implementations. The version control problem is somewhat the reverse. We want to have control over implementations despite having the same interface. In some cases, we want objects to be transparently upgraded to the latest version. In other cases, though, we want a particular old version for reasons of compatibility, testing or bugginess of the current version! In other words, we would like to make relevant distinctions between implementations.

This does not have to destroy encapsulation. Rather version control demands that we can control the interface to the implementation binding process. We also require that old and new versions must be able to run on the system at the same time. Our programs can remain generic. We merely supplement our applications with control over binding. Our choices in binding are driven by knowledge external to the application itself, and may vary with each case.

2.4 Compatibility

We would like to be as compatible as possible with old code and programs, because it is not always possible to change them. In essence, we would like to preserve as much of our current software investment as possible. To require someone to port everything currently available to a new paradigm (even an object-oriented one) is too burdensome.

At the same time, we would like to remain compatible with future software allowing us to take advantage of the work that others do. We don't want to saddle the user with the task of having to port all new developments to stay abreast.

Finally, we don't wish to *force* a choice between the user's old environment and our system. We would like both to coexist side by side. The user can incrementally choose to use only as much of the new system as he wants to. Also, we would like the migration of old applications to our new framework to be as easy as possible.

2.5 Other Goals

Of course with any system, there must be adequate security. Since we allow ordinary users to add extensions to the system, it is impossible to assure that extension code will behave properly. Thus, it must be possible to limit the effect of insecure code has. As with safety, security also requires a firewall strategy. It should not be possible for an insecure extension to compromise any other part of the operating system or other extensions. No one other than the users of an extension should be affected by it.

We would also like to minimize the "surprise" factor. We would like for users to be aware of untrusted extensions and not to run them by mistake. This is a difficult goal to reconcile with transparent operation.

Performance also needs to be adequate. Lower performance will limit the space of practical applications. In particular, we would like to keep the overhead low enough that it is practical to use "small" extensions. A poorly performing system will not be popular. A particular extension may be badly written, but we do not want that to affect other users or extensions. The extension framework itself may have a cost separate from that of any individual extension. We require that performance cost to be minimal, especially when *not* being used.

CHAPTER 3

Architecture

Frigate has several components. At the lowest level, Frigate uses *typed* files. File System extensions are stored in an external *Repository*. When an extension is used, a server process is instantiated. Within the operating system, Frigate provides a *Dispatcher* module that manages the servers and intercepts file system calls, passing them out to the servers. This module is built on the Stackable Layers framework, which is described below. Finally, an object-oriented programming interface is provided using the ILU system.

3.1 VFS and Stackable Layers

Our implementation platform is SunOS 4.1.1 [Sun90], which is a variant of UNIX [RT74]. As with most modern versions of UNIX, SunOS uses the Virtual File System (VFS) [Kle86]. The key concept of VFS is the *vnode* (virtual node). The vnode is an attempt to provide an abstract data type for the basic kernel representation of a file. Each vnode has a vector of file system operations associated with it. Regardless of what the vnode actually represents, a particular operation is called the same way. The name "vnode", itself, is meant to suggest the idea of generalizing from the UNIX *inode*.

This interface provides a clean way to separate the generic aspects of the filing system from specific file system implementations (e.g., UFS [MJL84], NFS [Sun89],

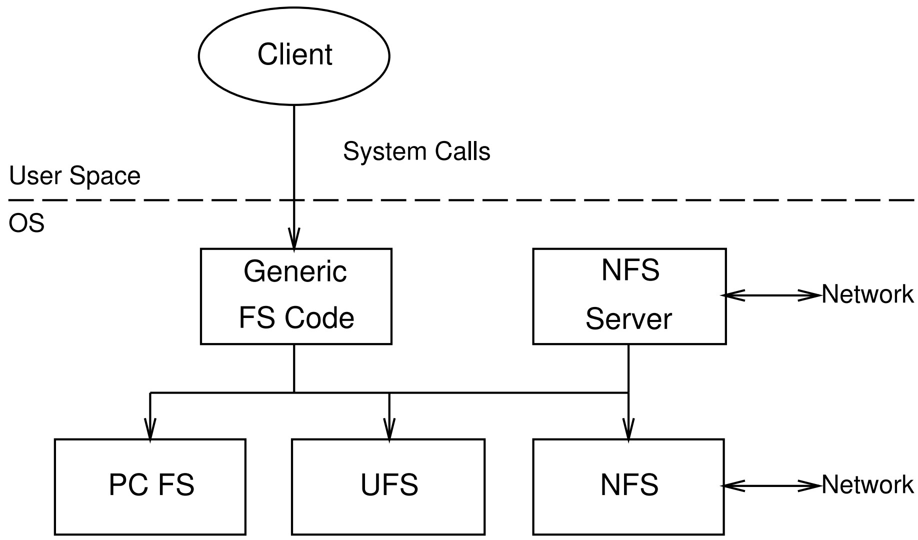

Figure 3.1: VFS Example

procfs [FG91], etc.). This allows multiple file systems to exist side by side in a modular fashion with the same interface. The UNIX mount mechanism provides the means to bind a point in the file namespace to a file volume with a particular implementation. While simple in implementation, this also means that implementations can only specified on a volume rather than individual file basis. VFS is illustrated in Figure 3.1.

Stackable Layers [Hei91, HP94, Hei95] adds another dimension to this structuring paradigm. VFS file systems were still essentially monolithic and generally very large. Stackable Layers allows file systems to be decomposed into smaller layers, which can be composed in a variety of ways. Each layer has a symmetric vnode interface. That is to say, it uses the vnode interface when called and when calling other layers.

This allows layers to be arranged into *protocol stacks* similar to that in System V UNIX Streams[Rit84]. Layers are mounted with the same volume mech-

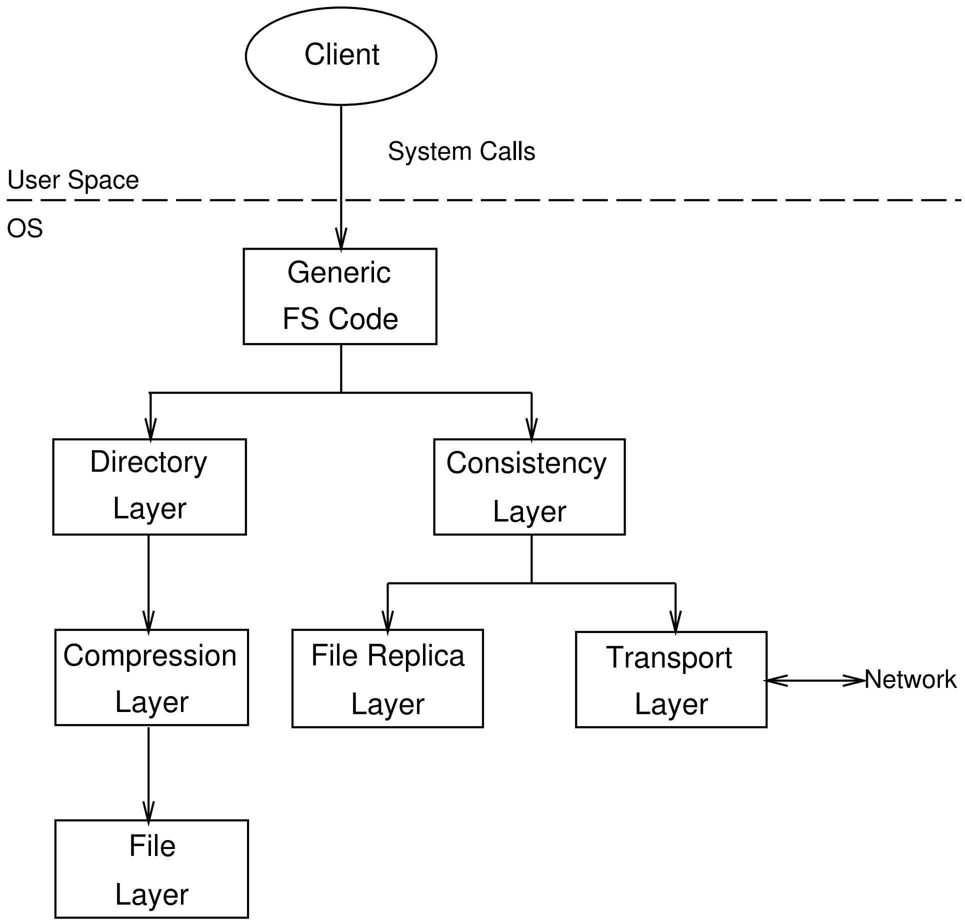

Figure 3.2: Stackable Layers Example

anism used in VFS. Instead of a single vnode, however, a stack of vnodes is constructed for each referenced file. Each vnode in the stack corresponds to a layer.

Layer components can be reused by other file systems. Improved layers can substituted without affecting other layers. Generic modules such as measurement layers could be inserted at will into stacks instead of hard coding them to fit only in one place. While the composition strategy is called "stacking", the technique is not limited to linear stacks. Tree like structures with fan-in and fan-out are possible. The basic structure of Stackable Layers is illustrated in Figure 3.2.

Stackable Layers also allows new file operations to be defined. Each operation signature is described in an interface description language. Every layer does not have to implement each operation, nor anticipate what operations might be added later. If an operation is not implemented, it simply passed down to the layer beneath it through a delegation mechanism called "bypass".

Stackable Layers also has the ability to use transport layers. This allows stack configurations to traverse address spaces without alteration. The interface description language is used to provide the information necessary for marshalling (and demarshalling) of arguments and transport of data between address spaces. This facility is used for remote operating system to operating system communications and to provide user-level file servers. The later facility is used mostly as an auxiliary development environment.

The Frigate Dispatcher module is packaged as a stackable layer inside the operating system. However, the Frigate extensions, themselves, are outside the operating system address space. Extensions are implemented as a modified version of the Stackable Layers user-level file servers. They are not added as layers inside the operating system, because doing so would go against our requirements for ordinary user access and user-level development. Our approach also means that the full functionality of Frigate can be achieved by the addition of a single, unchanging module to the operating system.

3.2 Typed Files

Frigate uses typed *file entities*, which we call *documents*. A file entity is anything that can be named in a UNIX file system. This not only includes ordinary files but directories, pipes, symbolic links, etc. This is a natural consequence of using building on top of the vnode oriented model of Stackable Layers. Thus, while documents are most commonly implemented on top of UNIX files, they do not have to be; they can use any of the other kinds of file entities.

A document is an object. As discussed earlier, an object has a public interface and private mechanism (method code) and state. Classes are templates for producing objects that share the same interface and mechanism. In Frigate, document classes define the interface for document objects. Document objects are tagged with a *class identifier*. All documents of the same class share the same interface. The private state of a document object is the content of the file entity. However, unlike many object-oriented systems, the mechanism is defined separate from the class. As we will see later, this is done to facilitate version management. Document objects are associated with particular implementation of mechanism by a *version identifier*. This identifier may also be a distinguished value that indicates that the *preferred* implementation should be used.

The document class and version information are new attributes added to UNIX file entities used by Frigate. UNIX, of course, already stores a number of attributes: file size, protection, owner, etc. However, our UNIX had no provision for storing additional attributes. Thus, we use a stackable layer that provides extensible attribute services [Wei97] to store the Frigate attributes. File entities without Frigate attributes are simply ignored by Frigate services and continue to have normal behavior.

The class and version identifiers need to be globally unique. This allows classes, versions and documents to be freely exported without the possibility of name clash. The identifiers also need to be generated in a distributed fashion, in order to facilitate independent development without imposing the overhead of a central registry. We use NCA Universal Unique Identifiers (UUIDs) [ZDL90] to fill this role. Each UUID is a combination of host address and a 48-bit timestamp on a 4-microsecond granularity. The host address can be any address up to seven bytes long with another byte describing the protocol family of the address. With reserved fields, this identifier is 16 bytes long. For both class and version, 32 bytes need to be stored, though in the typical case only about 20 bytes are really necessary. Our implementation includes a SunRPC [Sun88] server that generates UUIDs on demand.

VFS and Stackable Layers are limited in scope to only UNIX file entities. Frigate, on the other hand, has an object model that encompasses more than just document objects. However, documents do have some special properties in Frigate. Documents correspond on a one-to-one basis to UNIX file entities and can be named by a file system path. While document contents can be persistent, they need not be. In some cases, they are generated on demand. At the same time, Frigate objects that are not documents can also be persistent, if implemented that way. Most commonly, persistent objects that are not documents, themselves, though, are usually contained in documents.

Documents have two kinds of operations. As with any Frigate object, documents have operations that are defined through the object-oriented ILU interface described later. Documents can also respond to vnode operations. In order to incorporate vnode operations into the ILU model used by Frigate, ordinary UNIX file entities are all considered to be part of an implicit class. That class responds only to vnode operations and thus has exactly the normal UNIX file interface. Frigate document classes implicitly inherit from that class. This inheritance, though, is somewhat different from normal ILU inheritance. Frigate documents effectively inherit both interface and methods for vnode operations. Normally, inheritance in ILU is limited to interface. The inheritance of implementation is a natural consequence of the Stackable Layers bypass mechanism. Of course, inherited vnode methods can be overloaded and redefined as desired. Inheritance for any other (non-vnode) operation follows the normal ILU convention.

3.3 Repository

The repository stores the interfaces and implementations for extensions. For every class, a record in the repository describes its interface. The interface description record is keyed by *interface identifier*. The interface for a document is noted by its class identifier. The class identifier is used to look up the interface record that has an interface identifier of the same value. In the current system, class and interface identifiers are essentially the same. However, a more sophisticated mapping is possible. The interface description consists of identifiers for vnode operations 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 the server. ILU operations are not explicitly described in this record. All ILU operations object are automatically transmitted to the server via a special vnode operation that is always intercepted.

Implementations stored in the repository are the actual servers that implement objects. The implementations are indexed by the combination of interface and *implementation identifiers*. One implementation for each class may be marked as *preferred*. Applications can specifically request the preferred implementation. By convention, a preferred implementation is usually the latest or most stable version.

Stored with the implementation is information about the server capacity and startup arguments for the server. Server capacity indicates how many documents may be assigned to a server by the operating system. (An implementation may decide on its own to assign documents to itself.) A distinguished value indicates that any number of document objects of the proper class and version may be assigned to the server. Server capacity is usually used for load balancing purposes. Startup arguments allow optional features to be selectively activated in a server without recompilation (e.g. debugging code).

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. This feature is used to aid in version management. Because of alias records and preferred versions, version identifiers and implementation identifiers are not necessarily the same.

The repository is implemented by a repository server and database. The repository server provides three types of services: repository management, database query and server management (starting, stopping and listing running servers). All services are provided in a single repository server process to simplify the design and avoid the need for a database locking protocol. Our current system uses a local repository server process. However, a relatively modest change to our implementation would allow the server to be hosted on a remote machine.

Should the repository server crash, it can be restarted. The restarted server process will register the new service address. Clients that interact with the repository are designed to timeout on the cached server address and search again for the service address. In this way, more resilience is provided for the vital repository service.

The repository command set provides all the expected operations to manage the repository: commands to add, remove and list interfaces, implementations and aliases. Another operation designates the preferred implementation. Unless an implementation has been explicitly designated, the implementation with the highest timestamp is considered preferred. Repository management functions are requested by user interface programs. All updates from repository management functions take effect immediately.

Database queries are made by the Dispatcher Layer in the operating system. Queries are made to determine the interface of a document class and to provide a mapping from version to implementation. The latter is necessary because the document version identifier is not always the same as the corresponding implementation identifier of the implementation server. In the case of a preferred implementation, the mapping operation gives the actual implementation to be used. In the case of an alias or chain of aliases, the mapping provides the identity of the "real" implementation.

The Dispatcher also makes requests for implementation servers to be started and stopped. The implementation servers are forked from the repository server. The repository server sets up and tears down a rendezvous point for operating system and implementation server. The directory constructed in this process is mounted to provide an NFS protocol [Sun89] communications link between operating system and implementation server. Since our current repository server is a local process, implementation servers and file system rendezvous points are also local.

Since our kernel can only communicate directly via UDP [Pos80], a reply cache is provided in our repository server for the start implementation (server) operation to ensure that the operation is not repeated on a retransmit. This feature is necessary because the start operation can be relatively expensive and because server initialization procedures may not be idempotent. A reply cache is also provided for the shutdown service to ensure proper status code returns.

Modifications to an interface record and its related implementations are only allowed by the user that initially registered the class. This is verified by the ability of the updating user to write a file with a matching owner identity. In essence, file system authentication mechanisms were borrowed to avoid adding another subsystem. The server will only honor start and stop requests from the operating system itself. Such requests must come from protected, privileged ports. With modest effort, a richer security mechanism could be added, but it is beyond the scope of our current work.

3.4 Dispatcher

The Dispatcher is a stackable layer that stacks in the operating system above the layers that provide storage and Frigate attributes. This layer manages the vnodes for documents and directs the management of the servers that implement document objects. (The repository server provides the actual mechanism to start and stop servers.)

From the view of Stackable Layers, this is just another layer (see Figure 3.3). All active files in a mounted volume use the same vnode stack configuration. Ordinarily, this means that each file in the volume exhibits the same type of behavior. The vnodes of the Dispatcher layer, however, do not have the same behavior for all files. Instead, the behavior of a Dispatcher vnode varies according to the class and version of its document. Thus, identical stack configurations that contain the Dispatcher do not have to behave identically.

When a file is newly referenced, its stack of vnodes is constructed. When the Dispatcher vnode is constructed some special actions occur. First, it is deter-

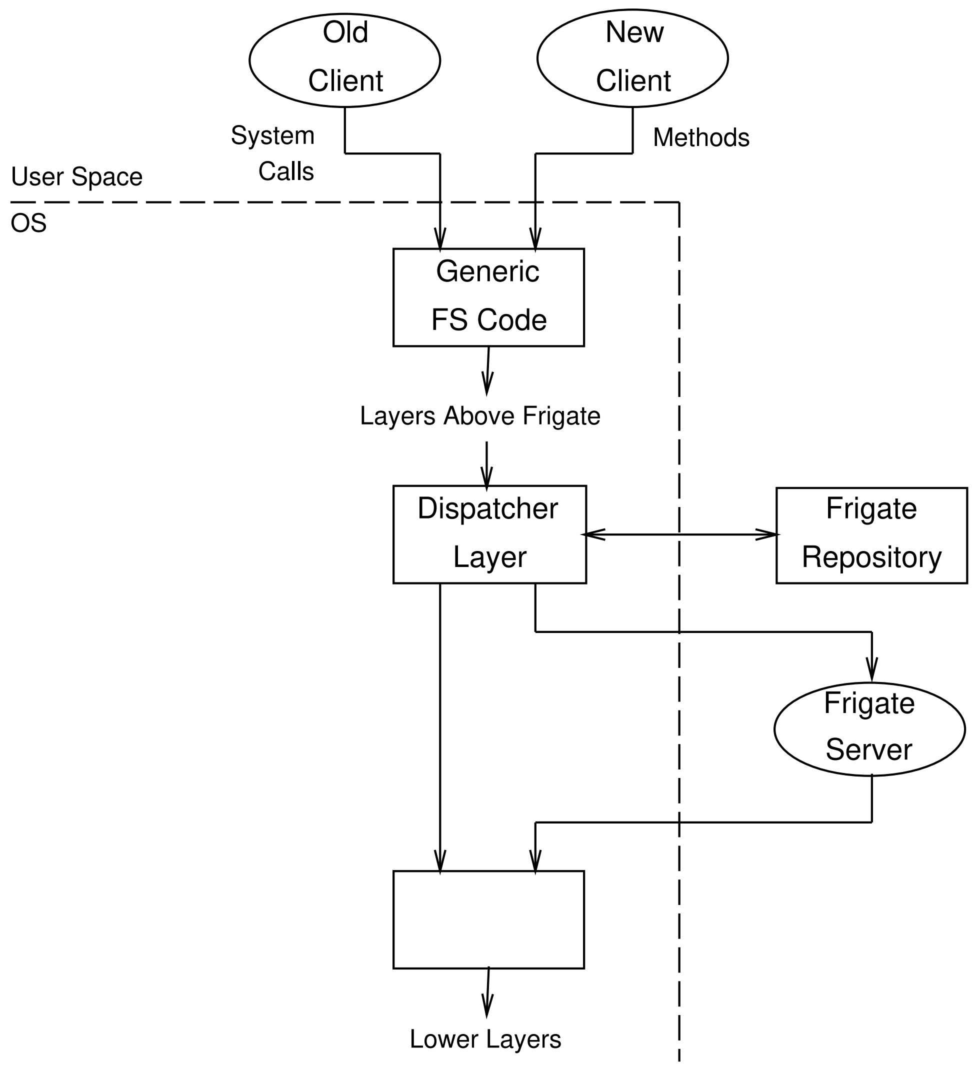

Figure 3.3: Frigate Architecture

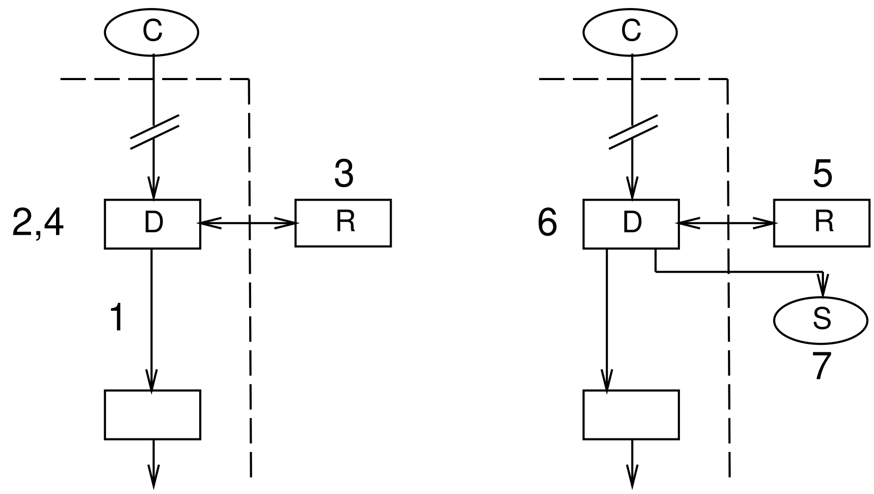

Figure 3.4: Server Startup - 1. Attribute Lookup, 2. Check for Existing Class Record, 3. Lookup Class Record, 4. Build VOP Vector, 5. Map Version to Implementation, 6. Check for Existing Server, 7. Server Startup. (C: Client, D: Dispatcher, R: Repository, S: Server)

mined 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.

If a vnode belongs to a document, then the class attribute is used to find its interface. If the operating system already has an active document for that class, the information is already available. If not, a query must be made to the repository server. Since the request involves a remote procedure call, it is quite possible the requesting process will sleep and another process scheduled. To prevent consistency problems, the vnode is locked until construction is complete or fails. Unfortunately, this is not the only synchronization problem that the Dispatcher must contend with in our operating system environment. Two different strategies are used depending on the weight of the operation. For cheap operations, usually a check is done before starting the operation and just before committing the change. If superseded, the operation is undone. For expensive operations, locking is used.

The interface information from the repository is used to construct a vnode operations vector. First, a default vector is obtained. The default vector is similar to the vector used for files that are not documents, except that there are some additional operations to manage servers. For each vnode operation redefined by this class, the vector is patched to a handler that intercepts the operation and passes it to an implementation server. Some operations require special handling and thus need to be patched to alternate routines that do some additional processing before sending to a server. For example, operations that affect the directory name lookup cache in the operating system must also perform certain cache operations. Operations not redefined by this class are simply set to be bypassed down within the operating system to the the layer beneath.

Servers are lazily started; no attempt is made to start one until some ILU operation or redefined vnode method is called. In this way, no unnecessary server starts are performed. When an operation that requires a server is invoked, the version is mapped to the actual implementation by querying the repository server. 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 those which service that implementation. If a matching implementation server has spare capacity, then the document is assigned to that server. If several candidate servers exist, then an idle server is preferred over one that is currently servicing a call or initializing. If no server is available, then one must be started.

When a server needs to be started, the operating system sends a command to

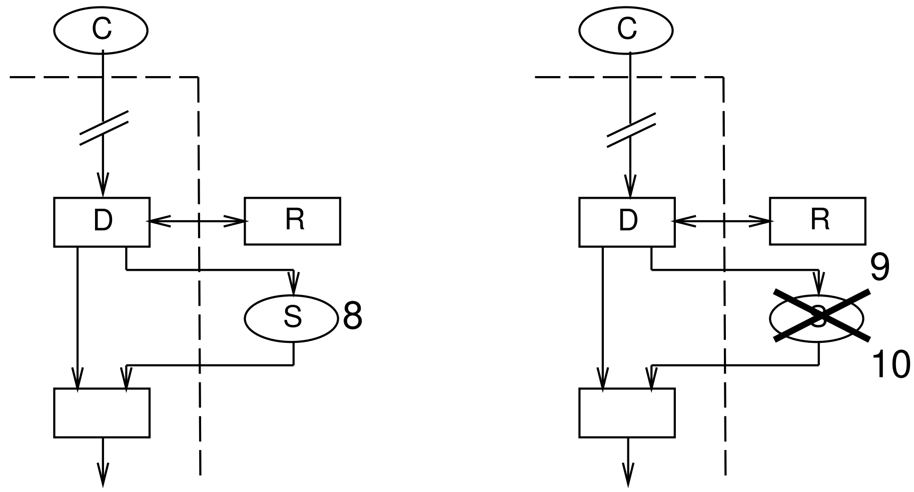

Figure 3.5: Server Life & Death - 8. Service Methods & Overloaded VOPs, 9. Drop Operation, 10. Server Shutdown. (C: Client, D: Dispatcher, R: Repository, S: Server)

the repository server. Using the information returned from the repository server, the operating system sets up a mount directory and uses the mount mechanism to rendezvous with the new server forked by the repository service. This server mount operation, though, is somewhat unusual. Usually a mount results in a server handling vnodes that are part of a single volume corresponding to particular portion of the file namespace. However, in this case, the resulting file system mount does not imply the vnodes handled by the server come from any particular volume. The documents serviced by this mount are those assigned to the server by the Dispatcher, and they may reside in *any* volume whose stack contains the Dispatcher.

After mounting, the operating system uses special vnode operations (on the mount "root") to build the corresponding transport vnode (essentially an NFS rnode) for the document in question. This provides the mechanism to actually pass operations to the particular corresponding vnode in the server. Using special Frigate extensions to Stackable Layers, special care is taken so that the server's vnode for the document is stacked over the same vnode that is beneath the Dispatcher in the operating system. Essentially, the vnode stack for a document divides into parallel paths, one of which goes through the server. They both join back together at the layer beneath the Dispatcher. The server path is taken by ILU operations and redefined vnode operations. The other path is taken by other vnode operations.

This is different from Stackable Layers, which sends all operations to its user-level servers even if not implemented by the server. (This allows Stackable Layers to not have to worry about obtaining the same lower level vnode.) Similar operations are used to match underlying vnodes, when the server returns a reference to vnode. The significance of our split path strategy is that it allows non-server operations to bypass directly to lower layers. This is more efficient than having to take a detour through the server.

After the server path for a document has been established, ILU operations and redefined vnode operations are passed to the server. ILU operations pass through special vnode operations that pass ILU information to and from the server. Due to the limitations on UDP, it also necessary to fragment and reassemble large requests. Redefined vnode operations are passed through normal Stackable Layers transport mechanisms.

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 invoker. 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 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 final *drop* operation is sent to inform the server. When a server no longer has any documents assigned to it, the operating system unmounts the server and destroys the mount directory. Finally, the repository server is directed to terminate the server. The life cycle of a server is illustrated in Figures 3.4 and 3.5.

3.5 Servers

Frigate servers are a modified and enhanced version of the Stackable Layers user-level servers. The Stackable Layer servers allowed one or more layers to be moved outside of the operating system address space. Frigate servers are also in user space. In addition, they have special support for flexible configuration management. Code for ILU objects can be added and removed from the configuration in modular fashion. In the overall stack, servers essentially occupy the same position as the Dispatcher layer, but servers do not have to be implemented just as a single layer. More than one implementation can run side by side for any particular class. Also, multiple servers can run with load distributed on the basis of document capacity. The possibilities in server configuration are illustrated in Figure 3.6.

In our current implementation, servers are forked from the repository server and, therefore, are on the same machine as the operating system Dispatcher layer. On server start up, the operating system uses the mount mechanism to establish communications. The server, itself, initializes its ILU classes and ILU

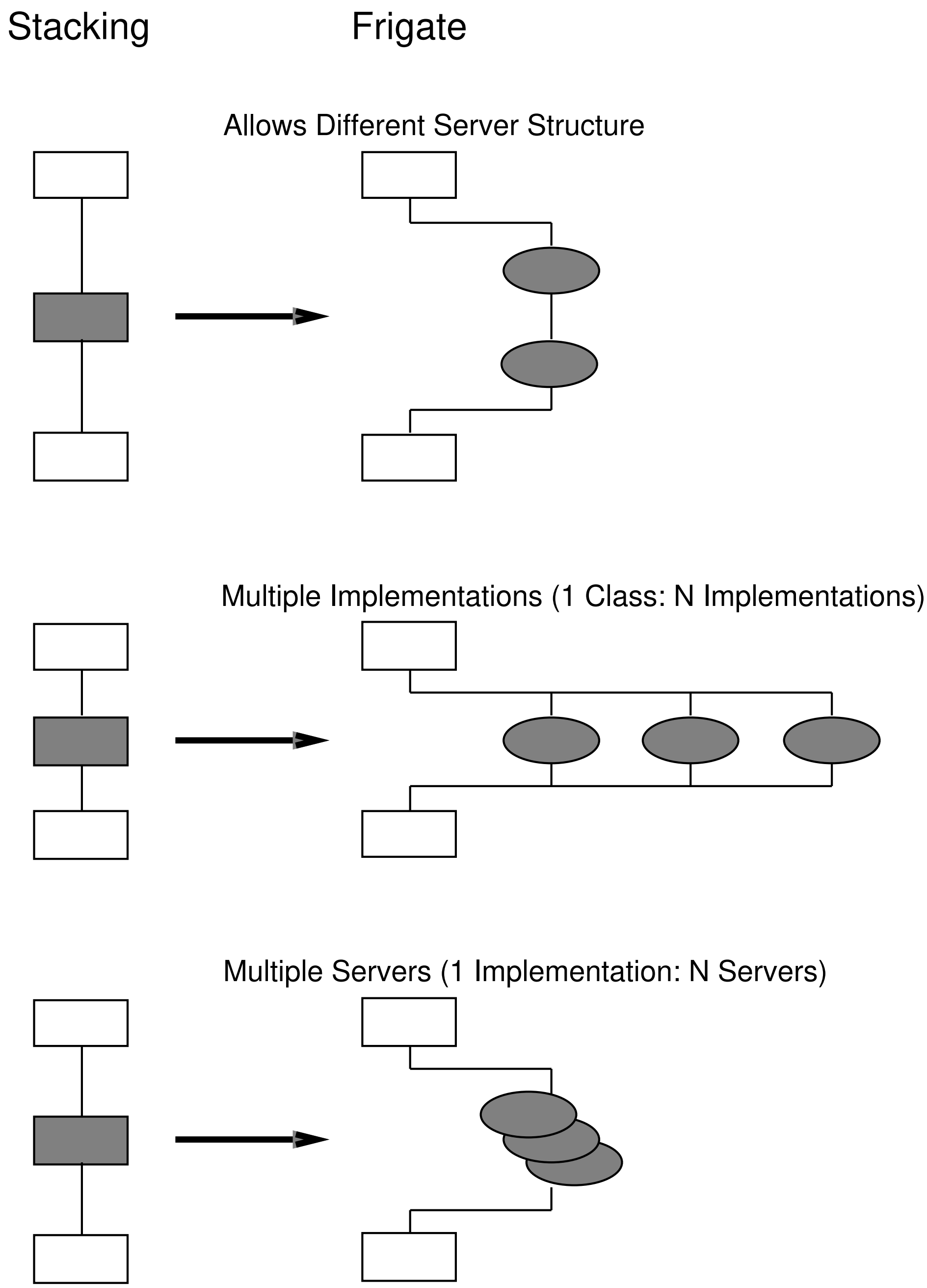

Figure 3.6: Server Variations

communications module. Each document implementation within a server also initializes its vnode operation vector.

This initialization of vectors is slightly more complicated than the patching done for the Dispatcher. Vnode methods require some further processing both before and after the actual ILU method handler to map back and forth from vnodes to ILU document objects. For each redefined vnode method, the vnode operations vector is patched to a *wrapper* function to accommodate this need. In most cases, this a generic function, though some operations that have irregular structure require alternate functions. The wrapper must in turn invoke the actual method handler. The address of the actual handler is patched into a second parallel operations vector.

Stackable Layer user-level servers (which are similar to NFS servers) are stateless. Between calls, all vnodes except those necessary for mounting are deallocated. Server side deallocation occurs even when the operating system still holds corresponding vnodes. However, this is not acceptable for documents, which potentially may be heavier weight objects. The model of document activation and deactivation is meant to match the lifetime of the corresponding operating system vnodes. Thus, document server vnodes and associated state must remain active for as long as the operating system holds corresponding vnodes. Therefore, document server vnodes are withheld from deallocation. This continues until the operating system is finished with a document and sends the server a drop message for the document.

Sometimes servers can handle more than one kind of document. When a server document vnode is first created, the server must assign a type from the available document types it knows about. A hook for a decision procedure is provided to allow the server to choose appropriately (e.g., by examining class and implementation attributes). Otherwise, a default choice is provided by the framework.

When a document vnode is first created, the document *create* method is automatically invoked. Usually this method provides a means to activate and initialize the document from its persistent, passive state. Similarly, when the document vnode is being dropped after its last reference, the *destroy* method is invoked to deactivate the object and potentially update its persistent storage.

The actual implementation of a method lies in its handler code. If necessary, the handler may call on services from the underlying stackable layers. Calling on lower layers leads back into the operating system, using the Stackable Layers "utok" transport service, rejoining the stack just below the Dispatcher layer. ILU operations use a reply cache to avoid problems with UDP retransmits.

In the case of Stackable Layers user-level servers, only one volume is supported per server and all file references implicitly use this fact. As mentioned above, Frigate servers handle requests for assigned documents rather than those of a particular volume. Thus, Frigate cannot assume all file references implicitly belong to a particular volume. Our file reference decoding scheme must also use explicit volume identifiers to map to a particular vnode. At the same time, operations on volumes, so-called "VFS" operations, have relatively little meaning for a Frigate server, because of the lack of volume correspondence. The directory name lookup cache is also turned off by default in the server, because document assignment does not necessarily imply name locality. (In fact, turning it on would slow down some operations.) In addition, some vnode operations specify multiple vnodes (e.g., rename operations) from the same volume. Frigate attempts to assign such vnodes to the same server, if possible. However, there is no assurance that this is always possible (e.g. due to server capacity) or when the vnodes in question do not correspond to documents.

3.6 Installation and Version Management

Initial installation of the Frigate system infrastructure *does* require privileged access. A copy of the enhanced operating system must first be installed as well as the repository server and database. However, after infrastructure installation, no further privileges are required. The extensions, themselves, do not require special privileges to install.

The mechanisms provided by Frigate allow multiple models for version management. In all cases, the repository accommodates installation and update without any disruption of system operation. While currently active documents will continue to execute with old implementations, any new use will immediately use new versions. Both new and old can run side by side. New implementations can be installed and tested in isolation by proper management of version identifiers.

Like most systems, though, Frigate can still have a problem when an entire set of interrelated implementations should be updated as a single transaction. If all such implementations are combined in a single server binary, then the update can be done as a transaction. If this is not the case, though, it is possible for a temporary inconsistency to exist.

Documents marked with preferred versions always use the implementation so marked. Exactly what this designation means can vary according to convention. In general, it 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 automatically use updated versions without having to search for and change the versions on existing documents. New versions are not 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. Again, there is no need to hunt down existing documents. 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 [Com90] and TIFF [Ald92] formats. Since mechanism (implementation) is divorced from class definition in Frigate, the version system can play this role instead. 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 interface to implementation binding process. By using aliases, the evolution along each line of development can be transparently managed. If future implementations merge lines of development, aliases chains can be made to converge as well.

CHAPTER 4

Programming Interface

The architecture of Frigate provides a means to add functionality into the filing services. This enhanced functionality is built in an object-oriented framework based on the ILU system. Clients can access this new functionality either transparently or by using a new object-oriented interface. Class implementors use the object-oriented framework to add their enhancements in a structured way.

4.1 ILU

The Inter-Language Unification (ILU) system [JSS95] from Xerox PARC is used by Frigate to provide an object-oriented programming framework based on the CORBA standard [Obj96]. (It was chosen partly because complete source code was available.) Our implementation used version 1.8 of ILU. We limit our discussion to the features used by Frigate.

ILU is a distributed object programming system that supports multiple programming languages. ILU generally follows the OMG CORBA architecture [Sol90] and standards [Obj96], though it is not fully compliant with the standard. An ILU application consists of a collection of ILU objects and clients. Objects are implemented by servers and may be distributed. Clients programs use object services by requesting operations on them. Requests are transparent with respect to object location; the same forms are used to make a request regardless of whether the object's server is on the same host as the client program or not. Objects can also be clients of other objects.

4.1.1 ILU Interfaces

The signatures of object operations are specified in an interface language, either OMG Interface Definition Language (IDL) or Xerox PARC's Interface Specification Language (ISL). The interface language provides a standard means for specifying for operations. The interface language is then compiled into one or more target languages. The target language mapping provides the programming language constructs for a client to make requests. Essentially, it provides a means to convert from programming language forms to request transmission format. On the server end, the mappings provide the interface between the actual method code and the request format (which is also the reply format). The use of an interface language is the key to supporting multiple programming languages. Since the request format is the same no matter what programming language is used, the client and server do not have to use the same programming language. They only have to use the same interface.

Interface language descriptions are packages of related data types including object operation definitions. In IDL, these packages are called *modules*. ISL calls such a package an "interface". (Unfortunately, this term is also used in IDL to denote another construction. To prevent confusion, reference to ISL use of the term is explicitly noted.) Typically, a package is a single file. A module may import definitions and define exceptions and types. Exceptions define a named error with possibly a value and documentation information. Types include builtin types for primitive data types (long, character, etc.) constructed data types (record, arrays, unions, etc.) and object interfaces.

```
module Service { // example module

typedef long Element;

exception NoSuchElement {};
exception EmptyQueue {};

interface Queue {
   void      add_back(in Element elem);
   Element remove_front()               raises(EmptyQueue);
   void      delete(in Element elem)     raises(NoSuchElement, EmptyQueue);
   Element front()                      raises(EmptyQueue);
   long      size();
   boolean  member(in Element elem)  raises(EmptyQueue);
   Queue    copy();
   };

// double ended queue, inherits all Queue methods
interface DQueue: Queue {
   void      add_front(in Element elem);
   Element remove_back()                raises(EmptyQueue);
   Element back()                       raises(EmptyQueue);
   };
};
```

Figure 4.1: Example CORBA IDL

```
INTERFACE Service; (* example interface *)

TYPE Element = INTEGER;

EXCEPTION NoSuchElement;
EXCEPTION EmptyQueue;

TYPE Queue = OBJECT
  METHODS
    add_back(elem: Element),
    remove_front(): Element     RAISES EmptyQueue END,
    delete(elem: Element)       RAISES NoSuchElement, EmptyQueue END,
    front(): Element            RAISES EmptyQueue END,
    size(): INTEGER,
    member(elem: Element): BOOLEAN RAISES EmptyQueue END,
    copy(): Queue
  END;

TYPE DQueue = OBJECT (* double ended queue *)
  SUPERTYPES Queue END (* inherits all Queue methods *)
  METHODS
    add_front(elem: Element),
    remove_back(): Element      RAISES EmptyQueue END,
    back(): Element             RAISES EmptyQueue END
  END;
```

Figure 4.2: Example ILU ISL

The definition of an object interface is called an IDL *interface* (or an ISL "object type definition"). More than one IDL interface may be defined in an IDL module. Object interface definitions include operation signatures and inheritance information. (ISL also allows some additional object flags, e.g., automatic garbage collection.) Objects that use the same interface essentially belong to the same class. (In fact, an earlier version of ILU actually used the word "class".) Operation signatures specify the operation name, the arguments, results and exceptions. Arguments are individually described by type and mode (input or output).

In IDL, inheritance is specified by giving a list of inherited IDL interfaces (called "supertypes" in ISL). Inheritance implies that all operations of inherited interfaces are also available. An object can treated as if it was an object of an ancestor type. IDL does not allow inherited operation signatures to be redefined or for two different operations with the same name to be inherited. In IDL (and ISL), inheritance is restricted to the interfaces of objects; object implementations are not inherited. Each object implementation must provide methods for both new and inherited operations. An example IDL module is shown in Figure 4.1. The equivalent ISL definition is shown in Figure 4.2.

After writing the IDL (or ISL) specification of the interface, it is compiled into one or more target programming languages by using the ILU "stub generator" program. ILU provides concrete language mappings for a variety of languages including ANSI C [KR88], C++ [Str91], Modula-3 [CDG89], Python [Ros95] and Common LISP [Ste84]. The mappings are usually fairly straight forward; though some special provisions must be made for languages that are not object-oriented. For example, the ANSI C mapping prefixes names with the interface and object type names.

```
...     /* some definitions omitted for clarity! */

typedef ilu_integer Service_Element;

extern ILU_C_ExceptionCode ex_Service_NoSuchElement; /* Exceptions */
extern ILU_C_ExceptionCode ex_Service_EmptyQueue;

...     /* Queue object definitions omitted */

typedef ILU_C_OBJECT Service_DQueue; /* DQueue object */

/* Client Definitions of Methods */
void Service_DQueue_add_front(Service_DQueue, ILU_C_Environment *,
        Service_Element);
Service_Element Service_DQueue_remove_back(Service_DQueue,
        ILU_C_Environment *);
Service_Element Service_DQueue_back(Service_DQueue, ILU_C_Environment *);

/* Server Definitions of Method Handlers */
void server_Service_DQueue_add_front(Service_DQueue, ILU_C_Environment *,
        Service_Element);
Service_Element server_Service_DQueue_remove_back(Service_DQueue,
        ILU_C_Environment *);
Service_Element server_Service_DQueue_back(Service_DQueue,
        ILU_C_Environment *);
```

Figure 4.3: Example ILU ANSI C Language Mapping

The compilation provides data type mappings, calling stubs and skeletons for method implementations in the target language. In the ANSI C mapping, the client program makes requests by simply calling a stub function corresponding to the desired operation. The stub function signature is a straight-forward translation from the operation signature with arguments for the object handle and CORBA environment added. The signatures for the client stub routines and those for the server method implementations are provided in slightly different forms for ILU clients and servers. (The server versions for ANSI C have "server_" prefixed to their names.) This prevents them from clashing should a single process be both client and server. The mapping is illustrated in Figure 4.3.

4.1.2 ILU Programs

A complete client program is constructed by compiling the language mapping files along with the program main body. Servers are constructed by compiling the mapping files along with method implementation code. When link editing the executable client or server binary, a few ILU libraries must also be included. The program development process is show in Figure 4.4.

ILU applications can be flexibly configured. An ILU application consists of at least one client and server. It is possible there will be several of each. Processes may act as clients or servers or both. Thus, an ILU application may consist of one process or many. However, any particular object instance is only provided by one server. Objects may be implemented by local or distributed servers. Objects may be accessed locally or remotely with no change in calling format. ILU takes care of the details of inter-process communications through its available transport implementations. Local proxy objects are provided when references to remote objects are returned. It is possible for there to be multiple proxies for the same

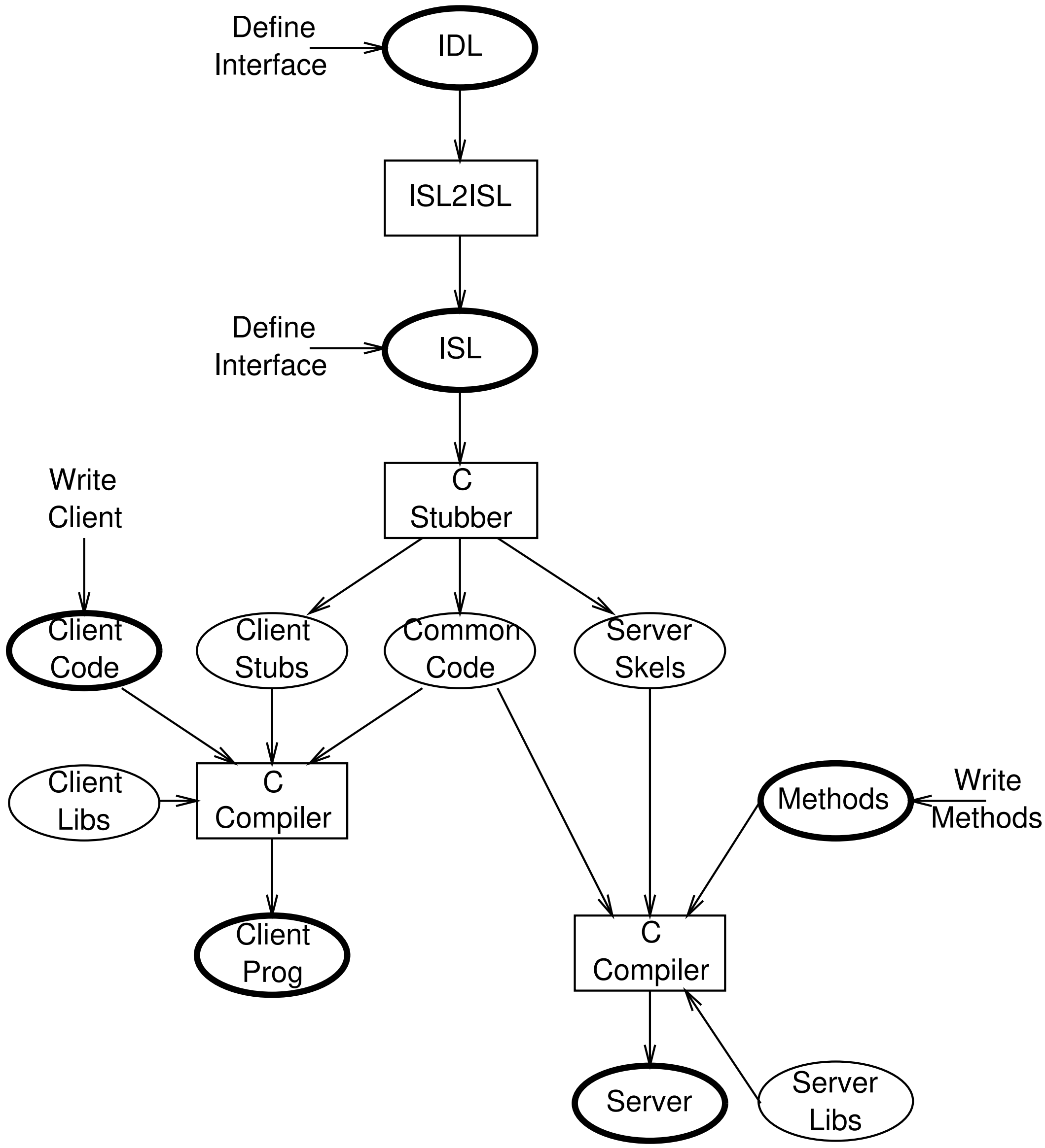

Figure 4.4: ILU Programming

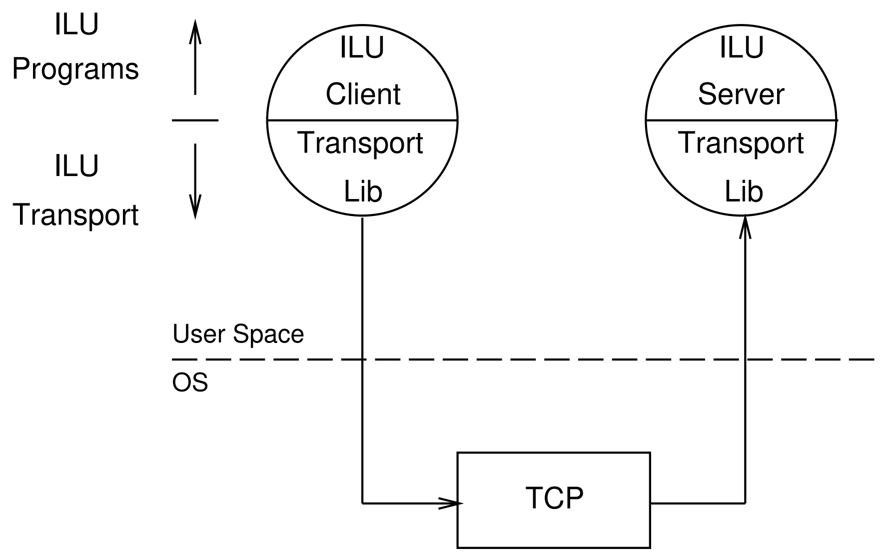

Figure 4.5: Example ILU Application

remote object instance.

As noted before, interfaces can be mapped into a variety of languages. All mappings use the same remote request format. Thus, there can be more than one client programming language used in a single application. The client languages do not have to be the same as the server programming language. All servers in an application do not have to be written in the same language either.

At runtime, ILU programs may make use of identifiers called object *handles* that can be used to identify objects and the location of the servers where they are implemented. A registry is provided to rendezvous with servers and the objects they provide. Unfortunately, the registry, which uses file directories, is rather primitive. However, hooks are provided for using a more sophisticated name service.

A simple example ILU application (illustrated in Figure 4.5) might include a server and client running in separate processes. The server publishes information about available objects in a registry. The client reads the information from the registry and contacts the server by TCP [Pos81] transport. It obtains a local proxy object for the remote server object by resolving the published object handle. Thereafter, the client sends messages to the object by merely calling the functions defined in the language mapping on the local proxy. Calls on the proxy are translated automatically into remote calls. Exceptions can be detected by interrogating the CORBA environment data. Inherited operations are invoked by treating an object as if it was an ancestor type. For example, in the sample interface shown in Figure 4.1, we could use the Queue call stubs on not only Queue objects but DQueue objects as well.

ILU is very similar to most other implementations of the OMG CORBA [Obj96]. However, ILU 1.8 does fall short of complete compliance to the standard. For example, it does not have an OMG Dynamic Invocation Interface or OMG style Repository. Also, it provides some of the Basic Object Adapter services in a different form.

4.2 Documents

Frigate integrates document objects and vnode methods with ILU. Currently, Frigate implements support for document interface definition in both IDL and ISL and provides support for the ANSI C target language.

Frigate document classes appear in the ILU framework as IDL interfaces (or ISL object types). They are distinguished from other ILU objects by inheriting interfaces from the implicitly defined *vop* (vnode operation) IDL module. This approach was taken to avoid any syntactic change to IDL or ISL. However, the semantics of vnode operations are unusual from the point of view of ILU. Thus, the

```
module ACL { // Access Control List

typedef sequence<long> IDList; // list of IDs

interface PFile: vop::access { // file protected by ACL
   IDList   GetACL();
   boolean  SetACL(in IDList list); // only owner or root
   };
};
```

Figure 4.6: Example Frigate Document IDL

interpretation of this special inheritance is somewhat different to accommodate the marriage of these two paradigms.

Depending on stack configuration, vnode operations may pass through Stackable Layers above the Frigate layer. The semantics of such operations may be affected before reaching Frigate, assuming they get to Frigate at all. Also, Stackable Layers includes the notion of "bypass", where all undefined (or more properly unimplemented) vnode operations are simply passed further down the stack, thus receiving any default or lower-level implementation. Therefore, unlike other operations, calling a *vnode* operation without an implementation at the Frigate layer does not automatically result in an error.

In effect, any vnode operation not implemented in a document (except for a few special ones) automatically receives an implicit inherited implementation. This is different from ILU methods, where only interfaces but not implementations are inherited. The functionality of the inherited implementation is that

```
INTERFACE ACL; (* Access Control List *)

TYPE IDList = SEQUENCE OF INTEGER; (* list of IDs *)

TYPE PFile = OBJECT (* file protected by ACL *)
   SUPERTYPES vop.access END
   METHODS
      GetACL(): IDList,
      SetACL(IN list: IDList): BOOLEAN (* only owner or root *)
   END;
```

Figure 4.7: Example Frigate Document ISL

provided by layers other than Frigate. Since stack composition may vary, this inherited functionality may vary on a volume by volume basis.

Inheriting an interface from the vop module indicates that a document type implements the corresponding vnode operation. The name of the inherited interface is the same as the vnode operation. Thus, a document implementing the "access" vnode method, would inherit from the vop::access interface, using the fully qualified name. (The ISL equivalent is vop.access.) An example document definition in IDL is shown in Figure 4.6. The equivalent ISL is shown in Figure 4.7.

All vop module interfaces have their single vnode operation defined for them under the name "op". Since this is the same for all vop module objects, provision has been made to allow this name to be omitted. One can call vop_access rather than vop_access_op. The client name space for vnode operations is naturally separate from that of ILU methods by virtue of being inherited. (One invokes vnode operations on documents as if they are objects from the vop module.) On the server side, the server skeletons use the "vop_" prefix rather than "server_" to prevent name clashes with ordinary ILU methods. Thus, a Frigate document implementation can have an ordinary ILU method and a vnode method with the same unqualified name.

There are some documents that do not actually implement any vnode operations. This presents a problem as vnode operation declarations are used to distinguish the interfaces of documents from other ILU objects. To mark these interfaces as belonging to documents, they inherit the vop "spec" (special) interface, which has the fully qualified name of vop::spec (or vop.spec in ISL). Of course, no actual vnode methods are defined by this object type.

Vnode operations are, by design, inherited only. The signatures of vnode operations are externally fixed and so cannot be redefined. Frigate also prevents the vop module from being explicitly defined or imported. This eliminates the possibility of inconsistent definition and ensures full compatibility with Stackable Layers. In Frigate, Stackable Layer vnode operation definitions are fed directly to the stub generator by specifying the appropriate INTGEN (".c" or "_uuid.c") files on the command line.

The Frigate extended ANSI C language stub generator generates an additional file for each document class. This document class information is used when installing the class description in the repository. Additional information items for documents and their vnode operations are also added to the standard ILU files. Frigate programs must use some additional include files beyond the standard ILU ones to add Frigate and Stackable Layer definitions. When linking programs, some additional libraries must also be used. The Frigate program development process is shown in Figure 4.8.

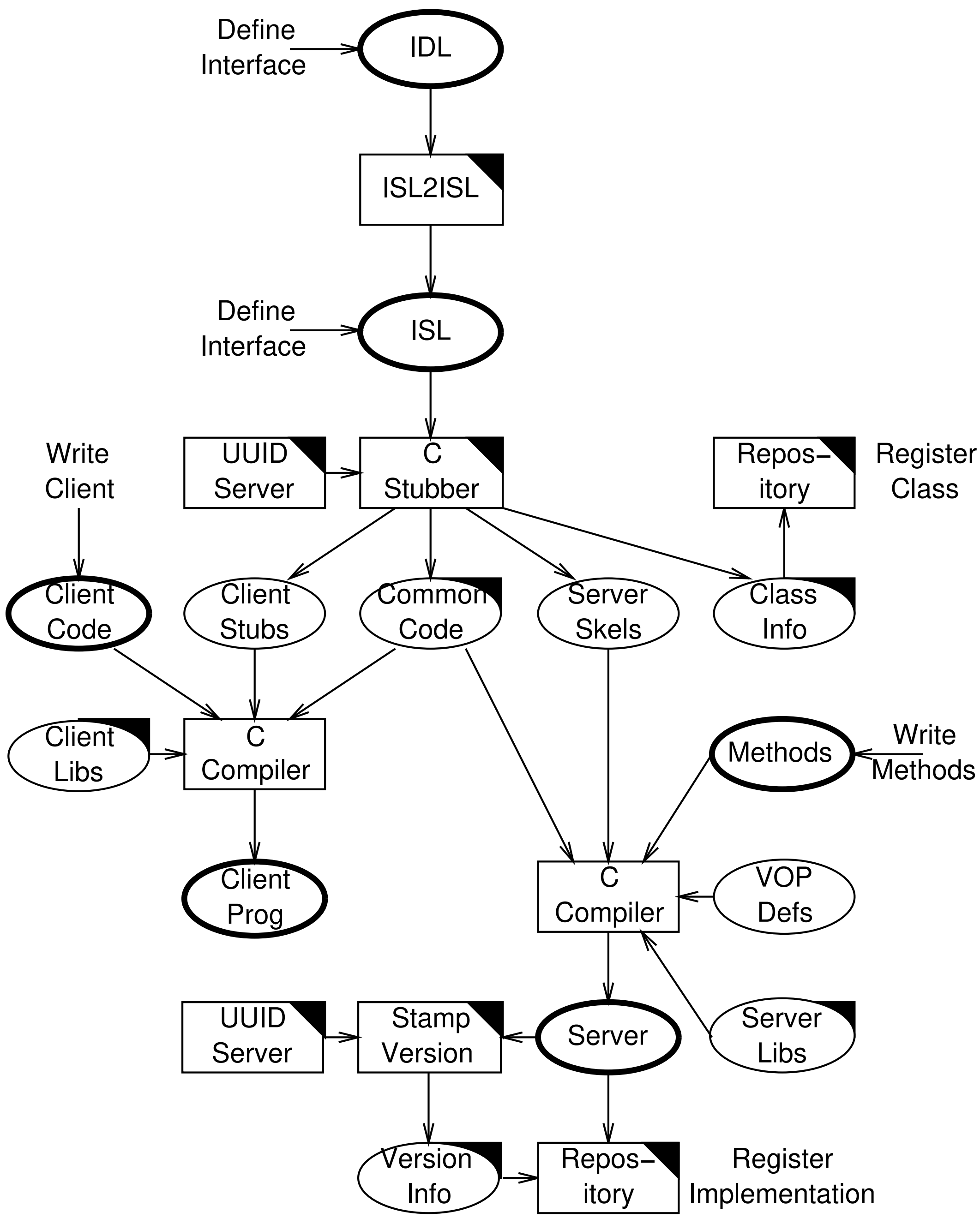

Figure 4.8: Frigate Programming - Elements with marked corners are added or modified for Frigate.

```
...     /* only some Frigate related definitions shown! */

/* Client Definitions of Methods */
typedef struct nvop_access_args vop_access_args; /* type of arg struct */
int vop_access(ACL_PFile, ILU_C_Environment *, vop_access_args *);

/* Client Convenience Macros */
#define uses_vop_access .../* define instance of arg struct */
#define the_vop_access_args .../* name of arg struct */
#define vop_access_op vop_access /* the full name */
#define VOP_ACCESS(obj, env, ...) .../* call inline with all args */

/* Server Definitions of Method Handlers */
int vop_ACL_PFile_access(ACL_PFile, ILU_C_Environment *,
        vop_access_args *);

/* Server Builtin Method Handlers */
int vop_ACL_PFile_Create(ACL_PFile, ILU_C_Environment *,
        struct vnode *);
int vop_ACL_PFile_Destroy(ACL_PFile, ILU_C_Environment *,
        struct vnode *);

struct FF_Class ACL_PFile_FF__ClassRecord = .../* Frigate Class Record */
```

Figure 4.9: Example Frigate Document ANSI C Language Mapping

The ANSI C language stubs for all (directly invoked) vnode operation have three arguments. The first two are the standard ILU object and environment. The third argument is the address of the standard Stackable Layer argument structure for that vnode operation. An instance of the structure can be defined globally or locally using the appropriate macro definitions. Since it can be tedious to fill out the fields of these structs individually, macros that assign the structure fields inline are also available. An example of the Frigate ANSI C language mapping is shown in Figure 4.9.

At runtime, Frigate documents are located by using file system paths. The file system namespace, itself, is 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. Calls made on proxy objects are automatically converted to remote calls. Routines also exist to destroy local proxy objects and end document access.

4.3 Transport

A typical distributed ILU application has client and server processes communicating via some type of ILU transport module. The transport is based on some underlying inter-process communications facility such as TCP. From an ILU perspective, the Frigate architecture has a similar structure. User programs are clients. Frigate servers act as ILU servers. The path in between is handled by Frigate's transport module. The Frigate transport encapsulates a path through the Stackable Layers system that goes through the system call interface of UNIX, down the stack to the Frigate layer and out of the operating system to

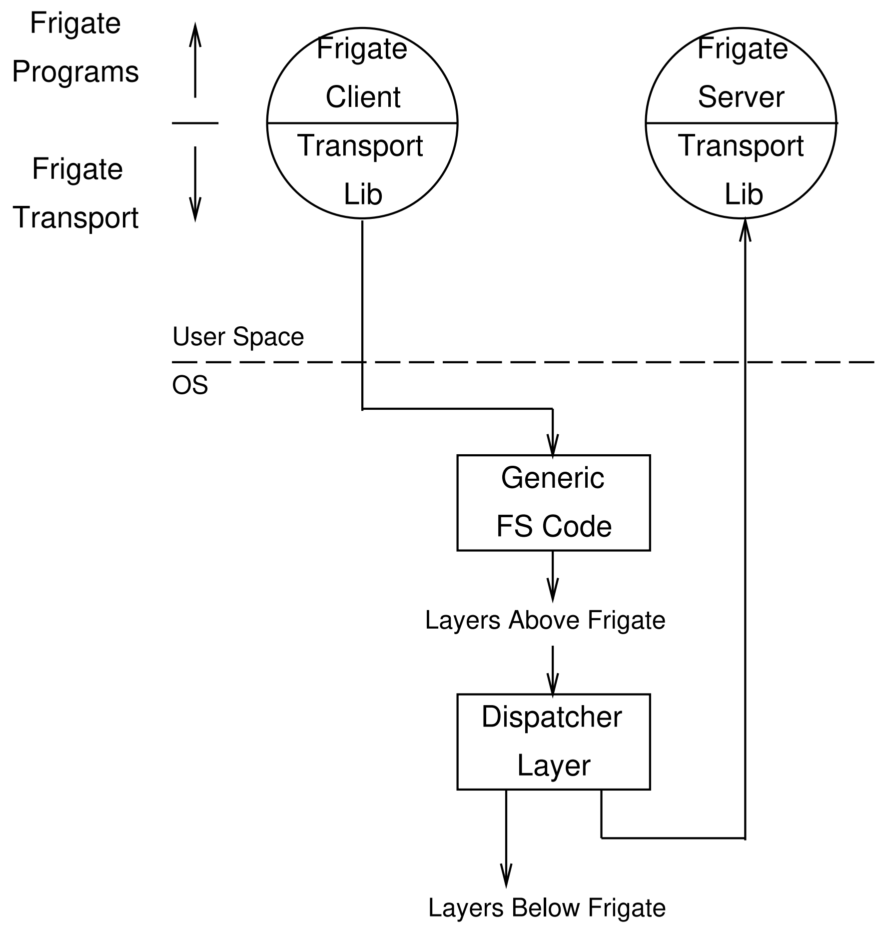

Figure 4.10: Example Frigate Application

a Frigate server using the Stackable Layers transport layer. Any errors returned from Stackable Layers are mapped into ILU exceptions.

Document objects are the only ILU objects that have vnode operations. Stackable Layers provides a so-called "out-of-kernel" interface that allows vnodes to be referenced outside of the operating system. On the client end, document object references are translated to vnodes to use this interface. The translation uses internal information generated from the initial filename to proxy object resolution. The Frigate calling stub for a vnode operations invokes the corresponding

Stackable Layers operation.

The available Stackable Layers transport has optimizations designed for NFS [Sun89]. As a result, the transport alters the semantics for many standard UNIX vnode operations. The result is that some operations are not faithfully transmitted to the server. For some applications, these NFS optimizations are acceptable (e.g. servicing requests from caches).

In other cases, however, it is not acceptable. For these situations an alternate set of vnode operations is provided. Each alternate operation has the same signature as the basic operation it replaces, except in name. Alternate operations have the "new" string prefix added to the name of the corresponding basic operation. If a document implements an alternate operation, any calls on the basic operation will be automatically mapped in the Frigate layer prior to transport. This avoids any semantic alteration of the request.

ILU methods on documents use the normal ILU argument (de)marshalling designed for SunRPC. The resulting ILU request/reply packets are sent to the server through a special Frigate vnode operation. Because the Stackable Layers transport uses unreliable UDP [Pos80], some special measures must be taken. All transmissions are fragmented into packets of 8K bytes or less and reassembled as necessary. In addition, a reply cache is provided in the server for method requests. This prevents needless or dangerous repeated execution when retransmissions are received.

Each document has a one-to-one relationship to a vnode. Other ILU objects implemented in the server have no correspondence to a vnode. Thus, they are prohibited from receiving vnode operation requests. They may, however, still have conventional ILU methods. As noted above, transmission of ILU operations through Stackable Layers, however, requires the use of a vnode. To allow transmission through Stackable Layers, some additional mechanism was added to the process of resolving object handles to local proxy objects. The only way to *initially* communicate with a Frigate server is through a document. Whenever any object handles are returned from a Frigate server, they are marked with information about the document vnode used to transmit the request. Calls on a local proxy object with a marked handle will be transmitted on the associated document vnode. This scheme does require that the transmitting document be available for the lifetime of associated object handles. Since documents are often the persistent containers for these other ILU objects, this requirement is usually easy to satisfy.

Vnode references also introduce an anomaly. In conventional ILU, a client has a unique handle based on a single communication path between client and server. In Frigate, it is possible that there will be more than one out-of-kernel vnode for a particular document. Thus, it is possible that a client may possess multiple different handles for the same object. This possibility required making a small localized fix to ILU. Frigate client code should not assume documents have unique ILU object handles.

4.4 Servers

A particular server may be capable of serving more than one type of document object and may serve many other types of objects. To aid in server construction, Frigate uses a modular server structure. In addition to ordinary method handler implementations, Frigate servers use two configuration lists and some special methods.

One configuration list contains links to initialization routines for the included

IDL modules. This list is used at server startup. The second list has links to Frigate class records that are automatically produced by the stub generator. The class record contains information about how to activate or deactivate documents of that type.

For each document type, two implicit methods must be provided, Create and Destroy. These methods are referenced in the Frigate class record. When an object is activated, the Create method is automatically invoked. Typically, this method reconstructs a memory representation of the object from persistent storage. When the last reference to an object is released, the Destroy method is invoked, usually saving state to persistent storage. Each server also requires a definition for the "FF_ChooseClass" function. This function determines what type of document to activate when a document vnode is created.

Frigate servers must be able to service some "housekeeping" vnode operations that are unrelated to method invocation. For this reason, the top level event loop of Frigate servers still services Stackable Layer vnode operations. ILU event servicing is called from within vnode operation handlers for the Frigate transport. Vnode operations bypass the usual ILU event loop. However, the Frigate infrastructure still maps the vnode to an ILU document when a vnode method handler is invoked.

Method handlers often have to use filing services. Handlers can call vnode operations on lower layers. In addition, a special interface is provided to convert lower layer out-of-kernel vnodes into ordinary file descriptors. This allows the use of regular UNIX system call programming in method handlers.

After a server executable has been constructed, it is assigned an implementation identifier. This is accomplished by "stamping" the server with a special Frigate utility. The utility also generates a file that contains the necessary information for installation into the repository.

When debugging a server, the standard debugging facilities of both Stackable Layers and ILU can be used. Also, any custom debugging features may be activated by altering the server startup command line stored in the repository. This allows selective alteration of behavior without having to recompile.

CHAPTER 5

Evaluation

Frigate provides a new file system infrastructure. It allows the file system to be extended to provide powerful new functionality in a consistent framework. Moreover, this ability is given to the unprivileged user. We describe each of these aspects and illustrate them in more detail below.

5.1 Design Philosophy

Frigate is a concrete running system. At a more abstract level, it is also a case study in system design. Understanding the design philosophy of a system is important to evaluating the goals and tradeoffs made in a system.

Frigate is a system built for extensibility. The design advocated is one where power is exposed safely by using a suitably chosen interface. Services are presented in a framework rather than as a fixed menu of choices. Extensions are modular in design and at runtime, but there is a cost for using containers capable of running untrusted code. Because systems are always changing, Frigate emphasizes many software engineering concerns. Frigate's focus on ease of use is rather unique for system software. Also, compatibility is provided to avoid the prohibitive costs of porting the large existing environments.

Systems that can only be modified by small numbers of privileged experts are not malleable enough to suit large communities of users with diverse needs.

Frigate presents a view that local users and programmers, rather than distant experts, are best equipped to add value to their application. This is the expected point of view for much application programming but not usually for system software. System software, and particularly the operating system, is thought to be too complex, too dangerous and sometimes too proprietary to be entrusted to users. Certainly, there is some kernel of truth behind these assertions. The problem of exposing the underlying power of the operating system in a useful and safe manner is a difficult one. The need to extend the operating system has been recognized by several related research projects (described in the next chapter). However, as noted before, most systems still make extension the exclusive domain of privileged gurus.

Frigate and a few other systems view the operating system more as an open pluggable framework. The operating system does not have to inevitably fail in anticipating all needs. Rather, the operating system must provide a way to add new functionality in a structured manner. For Frigate, the way to do this is to expose the VFS operating system interface through Stackable Layers. In addition, the VFS model is fitted into the wider object-oriented context of ILU to provide a single coherent programming model. Open frameworks have been popularized in other areas, but not generally in operating systems, which are often still seen as the domain of a small group of specialists.

Extension code, like other program code, has the dual goals of modular design and efficient invocation. Programming languages provide the user with procedures, classes, modules, packages, etc. as structuring tools. With most compiled languages, it is hoped that most of the benefits of structured design can be realized with little actual runtime cost. Nevertheless, even at the lowest level, machine instructions that implement procedure calls do impose a cost in terms of stack manipulations, saving registers, etc. At a higher level, threads, processes and other constructs can be used to structure a larger system. Processes provide a well-protected runtime environment, but the container is bulky enough that invocation and communication efficiency can suffer.

Some other systems have taken a view similar to programming languages. Extension behavior is restricted in various ways on the front end to get sufficient security or efficiency. For example, some systems attempt to provide enforcement of safety in the language compiler. When actually running, it is hoped that only very simple protection schemes are required. Frigate chooses to use processes to contain extensions. This is a consequence of having very few restrictions on extensions and expecting *untrusted* code to be a normal case. A price, however, is paid to create servers and to communicate with them.

Our design recognizes the high costs of software development and engineering. Many systems tradeoff costs here for better performance (when actually working correctly). Many times, this view implicitly assumes only experts attempt system modification. The argument often used is that development costs are one-time and that any loss of efficiency is paid repeatedly by users. The reality in many systems, though, is that development is never done. There are always new features to add, bugs to remove, versions to upgrade, etc. Multiple versions on a single system is a normal situation. We believe that most systems are not static, rather they are *always* changing and evolving. Nevertheless, disruptions in service are not acceptable and should be minimized even in the face of constant change. The Frigate architecture is designed with this view in mind. Frigate strives for ease of use. An attempt is made to avoid a steep learning curve. As described previously, the development environment is user-level, object-oriented and uses familiar existing tools. Frigate is compatible with the current UNIX environment and supports incremental use. Frigate provides support for multiple versions. When version changes are made, there is no interruption in service. Failed servers do not disrupt other users or crash the system. Independently developed extensions do not interfere with each other.

5.2 What is Extensibility?

Before evaluating the extensibility of Frigate in detail, we must settle on some operational definition of extensibility. Up to this point, we have relied on a somewhat intuitive notion of its meaning. Extensibility is the ability to add extensions. An extension is an addition to an existing service framework. If no reference is made to an existing service, then new code is simply a standalone piece of work or a “new feature”. On the other hand, an extension chooses to take advantage of an existing paradigm for conceptual or architectural reasons. This notion is fairly close to our original concept of “adding value to the file system”. The notion of the file system and an object-orientation provide our ordering structure.

Most often the user’s view of a framework is through some call or request interface. Thus, from the user’s view, an extension is usually seen in the context of an existing interface. Extensions can replace an existing service request, add some increment of function to a request or add a completely new type of service request. Even in the last case, the change is still made within the existing interface paradigm.

Extensibility is measured in several different ways. Part of extensibility is measured in terms of the extensions that a system allows. How powerful are the extensions? Do they perform well? What are the limits on the construction, size, number and type of extensions? Are they safe and secure? Another criteria, which often receives less attention, is ease of use. This goes back to our root goal of "adding value *easily*". Ease of use is always relative to some viewpoint. The end user thinks in terms of whether an extension's interface is clear, consistent, simple and intuitive. The end user must also consider how new functionality can be applied to problems. Functional characteristics such as performance and failure modes are also important. The extension implementor may think in terms of effort required to write an extension. But the discipline of software engineering points out that this is not the full story. One must also think about debugging, documenting, adding new features later, managing versions, porting, compatibility, etc. These costs are often far more than those of the original implementation effort.

In the following sections, we examine the different aspects of Frigate's extensibility.

5.3 Examples

Frigate has a variety of possible uses. Here we outline some example applications of Frigate to illustrate the power and flexibility of the framework.

Frigate can be used to provide all of the traditional "advanced" file system features such as compression, encryption, migration. For the most part, these services are ones that continue to use the ordinary system call interface. The new functionality is provided transparently and automatically. In Frigate, such services are provided by redefining vnode operations. The new definitions for vnode operation enhance the behavior of 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 [Ste91] 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 functions to manage access control lists, provide different levels of information to different users, read logs, etc. The ordinary system call interface is still used to provide basic read/write capabilities. In general, Frigate could be used to construct a model of file security that is far richer than the simple UNIX protection scheme. Similarly, a file system that provides multiple versions of a file entity is possible through Frigate. The set of versions provide multiple views of the entity or a revision history. In many cases, old programs can still be used, if wrappers are provided to properly select the appropriate version before access.

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. 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 [Mic93] or OpenDoc [FM96] to be constructed in Frigate. In this case, a document object is a container for other "content" objects that contain text, images, audio, spreadsheet cells, graphs, etc.

With the addition of persistent handles (ILU handles are not persistent) for content objects, *links* become possible. Links are references to content objects that can be stored in documents. Link references can span files and possibly machines, if some sort of distributed support is available. With a link tracking service, updates of the source can automatically be reflected in any document containing link references. Links are a specialized case of interfile consistency. Frigate can also be used to provide 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 higher level file system abstractions, such as graphical display, printing or digital signature operations. Another example is the so-called *workflow* document. Workflow documents are electronic replacements for the traditional paper form or report. In addition to being structured to hold the necessary information, they also have procedural mechanisms builtin. This assures that they are reviewed, signed, copied, etc. according to organizational policy. Documents may automatically send email to inform parties of their responsibilities in the approval process or to automatically apprise them of changes.

5.4 Easier Extensibility

Our overall goal of making the file system easier to extend has several aspects. First, we provide for extensions that are not resident in the operating system address space. This frees us from the 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 interface and implementation identifiers are assigned in a distributed fashion so independent development is possible with no name clashes. The namespace, based on NCA UUIDs [ZDL90], is sufficiently large to prevent exhaustion for decades. Multiple extensions are not limited by having to fit into a single address or overlay space. Practically speaking, Frigate is limited only by the number of possible available server processes at runtime. If there is very high activity, the repository server might become a bottleneck. That could be easily corrected by using multiple servers with an appropriate synchronization protocol.

Frigate provides an object-oriented interface to the file system. This provides a single coherent client interface paradigm instead of multiple interface styles. It also provides the ability to use polymorphism. 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 end users and implementors are given a structured environment that merges vnode method and ILU method programming into a common framework. Frigate remains compatible with Stackable Layers without duplicating definitions. No effort is necessary to keep the two consistent. The familiar file descriptor programming model is also available to program methods.

Variants are cleanly described through interface inheritance. True encapsulation that cannot be bypassed, accidently or purposefully, is provided around file entities. This encapsulation provides higher levels of data integrity, because there is no way to access objects except through the defined interface. Overloaded vnode operations can enhance the behavior of old programs with no change to the program.

Frigate provides a mechanism for version management to allow both eager and selective version update. When new versions are installed, there is no need to reboot or restart the system. New invocations are automatically assigned the updated versions. They is no need to hunt down every document of a particular version to update it. Multiple implementations can run side by side with no interference.

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.

5.5 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 directly through Stackable Layers transport layer interfaces. Other operating system services may be used in the service of a method. Extensions do not have direct access to the operating system address space. Requests arrive at a server through the NFS-based transport layer. Requests for lower layer services enter the operating system through the "utok" transport layer. Since utok is technically part of the system call interface, it is available to any user process. However, gaining access to lower level layer entities is only possible with the proper "handles" that cannot be obtained without identifier information only granted by the operating system. 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. Except for operating system bugs, it is not possible to use the varying credentials to accumulate access rights. Even if access is granted previously, the next access will be barred on the basis of the submitted credentials. Of course, a server could fork to "copy" its access rights. But, this is no worse than what happens when a directly executed malicious program forks. Information can be read, stored and then passed out through another request; but this is no worse than an ordinary program writing information to an auxiliary file. Overall, malicious code cannot do anything not possible through ordinary 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. For example, an extension could define a malicious vnode method for the getattr operation. That operation could be triggered by the long form of a directory listing on the associated document.

Frigate attempts to mitigate the problem in two ways. First, privileged (root) processes 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 these vnode operations to be overloaded.) In this way, a user can safely determine if a suspicious file entity is an untrusted Frigate document by examining its attributes. If this is the case, the user will know that other operations might trigger document activation.

The NFS-based transport to servers offers some potential problems. While servers do not publish their addresses, it is possible to locate their ports and send false requests. If received during a method service window, the request could operate with inappropriate access rights. This problem can be addressed through the use of encryption. (Currently, we have not added any such protection to remain free of encryption technology restrictions.) Alternatively, this could be addressed by providing a more protected Stackable Layers transport layer.

The files in the Frigate repository are 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. User requests are authenticated by the ability to write a request file owned by the class owner identity.

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.

5.6 Compatibility

Frigate remains compatible with the vnode programming model. In fact, some Frigate extensions provide their new functionality exclusively through vnode operations. Clients can call the enhanced vnode operations directly. Typically, however, the client usage of such extensions is done indirectly through the regular 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 system using Frigate 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 log-structured file system layer for the standard UNIX file system 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 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. OMG IDL support is provided in ILU by automatic conversion to ISL. All inheritance support 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.

5.7 Performance

We measured Frigate's performance in two different ways. First, we took a series of micro-benchmarks. These measurements focused on the time spent along each segment in the execution paths through the Frigate system. The second set of experiments were macro-benchmarks that reflect overall system performance.

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. For all measurements, at least 30 data points were taken. All test runs were made under quiescent conditions.

5.7.1 Micro-Benchmarks

Requests in a Frigate system can follow a variety of execution paths. Each request traverses a number of system components during processing. For Frigate, most of these components are Stackable Layers. In some cases, a request simply passes through a component unchanged with minimal processing (e.g. Stackable Layer "bypass"). In other cases, a request will cause substantial processing in a component.

For measurement purposes, we divided each execution path into segments. Segments were chosen to correspond to system component boundaries. For our micro-benchmark experiments, the system was instrumented to measure the time spent in each segment. The time attributed to a segment includes both that spent processing the request (call) as well as processing the corresponding results (return).

We considered three different representative cases: a Frigate compatibility

Execution Path			
Non-Frigate	Compatibility Operation	To Server	Through Server
System Call	System Call	System Call	System Call
	FFD	FFD	FFD
		XNFS	XNFS
		(server)	(server)
			UTOK
	EAFS		EAFS
UFS	UFS		UFS

Table 5.1: Micro-Benchmark Execution Paths

operation, a server-implemented Frigate operation and a typical non-Frigate request. A compatibility operation is simply one that bypasses down the layer stack to be serviced at the bottom of the stack. Our measurements assumed that all layers in the stack had to be traversed. A server-implemented operation (method or overloaded vnode operation) is one that is intercepted and passed to the server. We considered two different scenarios for server operations. In the first scenario, we assumed an operation could be satisfied by the server without further layer calls. A second scenario assumes that the server implementation calls on lower-level services and therefore issues calls back into the kernel.

For Frigate scenarios, only the layers necessary to support Frigate are considered. These include the Frigate Dispatcher (FFD), an extended file attribute service layer (EAFS) and a base storage substrate consisting of the Unix File-System (UFS) packaged as a Stackable Layer. A server operation must also make use of other layers in between FFD and EAFS. The NFS-based transport layer

(XNFS) is used to send intercepted requests to the server from FFD. The User-To-Kernel (UTOK) layer is used by the server to call into the kernel at the EAFS level. For comparison, the non-Frigate system only makes use of UFS. In all cases, system call overhead is also considered. Table 5.1 illustrates the various execution paths and their constituent segments.

Three representative file operations were chosen for measurement: stat, read and write. The "stat" operation retrieves file attributes and is representative of a class of control-oriented operations that only move small amounts of data. For our experiments, all relevant stat information was already available in memory and therefore did not require disk I/O. Read and write are representative of operations which move large amounts of data. Data copy and caching are important considerations for such data intensive operations. For operations such as stat, data movement costs are not as important.

5.7.1.1 Stat Operation

Measurements for the stat operation are shown in Table 5.2. The top half of the table shows the time spent in each segment. The bottom half adds together the relevant segment times to show a typical total execution path time.

A minimal system call as measured by John Ousterhout's "Bench" package [Ous90] takes about 29 micro-seconds on our test system. This figure represents the overhead of changing to the kernel environment and back. Therefore, the remaining time of 94 micro-seconds represents stat specific processing in the generic file system code. For a non-Frigate system, much more time is spent in system call processing than in actually servicing the UFS specific portion of stat. Very little is done in the UFS for our experiment – the operation is dispatched and the stat fields are copied from inode fields. (Recall, our experiment arranged for

Stat Segment	mean	stddev
System Call	124.9	0.84
FFD	75.8	0.55
XNFS	3974.5	433.97
UTOK	888.1	89.77
EAFS	59.6	0.76
UFS	20.6	0.49
Path	**Total**	**Percent**
Non-Frigate	145.5	100 %
Compatibility	280.9	193 %
To Server	4175.2	2870 %
Through Server	5143.5	3535 %

Times in microseconds.

Percent of Non-Frigate time.

Table 5.2: Stat Segment and Path Times

the stat data to be readily available without any I/O operation.)

For the Frigate compatibility operation case, additional time is spent bypassing through EAFS and FFD. The size of this added bypass overhead is about equal to the entire non-Frigate case. In these experiments, stat is a very lightweight operation. Thus, Frigate bypass overhead is significant. On the other hand, for heavier-weight operations, the significance of this added overhead would be reduced.

Making a trip to the server far outweighs any other costs. It's clear that our particular IPC mechanism (derived from NFS) is expensive. For operation on a single host, this overhead could be cut substantially by using more efficient mechanisms. It should be noted that costs measured here also include a context switch to and from the server. The Bench package shows a context switch to another process and back to take 852 micro-seconds. If our system was extended for distributed operation, a lot of this overhead could not, however, be avoided, since the much of the general mechanism embodied in our transport layer would be required.

Getting back into the kernel through UTOK adds another large overhead of almost 900 micro-seconds. This overhead is surprisingly large, even taking into account crossing another system call boundary. This would appear to be another place where significant optimization could occur in Frigate.

Just how much could Frigate be improved by optimization? We have estimated the magnitude of potential improvement is quite large. Such estimates are useful for showing the intrinsic cost of our architecture, rather than the quality of our current implementation. In essence, they show what an ideal implementation would allow.

For IPC, we use the Spring operating system [HK93] as an example of what is possibly achievable. Spring, which also runs on SPARC-family processors, is able to achieve a simple one-way IPC in 220 machine cycles plus data copying costs [Lie95]. That many cycles (which includes SPARC register window handling) comes to about 9 micro-seconds for our machine. This figure includes a highly optimized context switch. "Bench" figures for data copying, show copying a complete VFS stat (getattr) operation takes about 27 micro-seconds (using less than peak rate due to transfer size). A complete two-way IPC, then would take 72 micro-seconds. This speed is 12 times faster than the current UTOK or 55 times faster than XNFS.

Ideal Stat Segment	Time	
System Call	124.9	
FFD	0.0	
XNFS	18.0	
UTOK	18.0	
EAFS	0.0	
UFS	20.6	
Path	**Total**	**Percent**
Non-Frigate	145.5	100 %
Compatibility	145.5	100 %
To Server	142.9	98 %
Through Server	181.5	125 %

Times in microseconds.

Percent of Non-Frigate time.

Table 5.3: Ideal Stat Segment and Path Times

However, there is still probably room for improvement. At this level of performance, data copying costs have become significant. This cost might be eliminated by using address space mapping techniques. Essentially, a shared-memory segment would be used to hold the operation data structures. Since the operation to establish the address space mapping probably takes longer than the stat call itself, we assume the mapping is allocated ahead of time. By doing so, data copying costs could be totally eliminated for individual IPC operations, leaving a round trip cost of 18 micro-seconds.

If these changes were made, "bypass" overhead would then become the largest additional overhead in a Frigate system. This overhead is particularly distressing, because bypass is only passing the operation through without modification. Stackable Layers are modular, in part, because they need only concern themselves with adjacent layers; individual layers do not require global configuration information. However, since stack configuration does not usually change dynamically in current usage, we could optimize bypass to take advantage of global information. The optimized bypass would be setup at mount time to call the ultimate service layer without calling any of the intermediate layers. Using these assumptions, bypassed layers would not cost any time.

The effect of all of these potential optimizations together is shown in Table 5.3. With optimized bypass, compatibility operations are identical in performance to non-Frigate calls. Optimized IPC gives vastly improved "To Server" costs, making it equivalent to non-Frigate costs. "Through Server" costs are also improved by 28 times, but still show significant overhead due to the light-weight nature of this operation. Nevertheless, we assert that general use would be quite feasible given such optimizations.

5.7.1.2 Read and Write Operations

The times for an 8KB read operation in Frigate and non-Frigate systems are given in Table 5.4. Figures are given for both uncached and cached data. In the uncached case, disk I/O costs in the UFS layer are the largest cost. Our 8KB reads are arranged in such a way as to defeat sequential pre-fetch strategies. However, it should be noted that data is also copied twice during UFS operation, once from disk into the virtual memory system and then to user space. The Bench package shows an 8KB copy to take about 835 micro-seconds. Bypass costs for FFD and EAFS are relatively insignificant in this context.

Read Segment	Uncached mean	Uncached stddev	Cached mean	Cached stddev
System Call	238.0	9.92	238.0	9.92
FFD	57.1	2.30	57.1	2.30
XNFS	12268.4	230.70	880.4	29.14
UTOK	839.9	12.46	839.9	12.46
EAFS	59.4	1.79	59.4	1.79
UFS	24121.5	1914.84	1002.4	41.26
Path	**Total**	**Percent**	**Total**	**Percent**
Non-Frigate	24359.5	100 %	1240.4	100 %
Compatibility	24476.0	100 %	1356.9	109 %
To Server	12563.5	52 %	1175.5	95 %
Through Server	37584.3	154 %	14465.2	1166 %

Times in microseconds. Percent of Non-Frigate Time.

Table 5.4: Read Segment and Path Times

A trip to the server does take significant time. However, if the request can be satisfied by the server itself, the cost is still only about half of an actual disk I/O. The transport layer also contains two 8KB data copies from IPC to virtual memory and then out to user space. If it is necessary to call back into the lower layers, UTOK costs also come into play. Interestingly, the cost here is about the same as the stat case, despite the fact that a data copy also happens here. In fact, UTOK time is very close to data copy time.

In the cached case, time for UFS operation time falls dramatically. When using cached data, there is still one 8KB data copy to user space during UFS execution. That data copy cost represents the minimal possible time for a read

Ideal Read Segment	Uncached Time		Cached Time	
System Call	238.0		238.0	
FFD	0.0		0.0	
XNFS	18.0		18.0	
UTOK	18.0		18.0	
EAFS	0.0		0.0	
UFS	23286.5		167.4	
Path	**Total**	**Percent**	**Total**	**Percent**
Non-Frigate	23524.5	100 %	405.4	100 %
Compatibility	23524.5	100 %	405.4	100 %
To Server	256.0	1 %	256.0	63 %
Through Server	23560.5	100 %	441.4	109 %

Times in microseconds. Percent of Non-Frigate Time.

Table 5.5: Ideal Read Segment and Path Times

operation without resorting to address space mapping techniques. Even in the non-Frigate case, it still takes 148% of the data copy time to perform a complete operation. The added costs for FFD and EAFS in a Frigate compatibility operation are still relatively minor sources of overhead.

The time shown for the transport layer, in the cached case, assumes that no actual transport occurs. Instead, the data is located and copied from the NFS-based caches. The cost for this transport layer operation is very close to just the data copy cost. The cached "To Server" total represents the cost of an operation destined for the server but satisfied from the transport cache. Because the transport cache is slightly faster than the UFS cache, the overall To Server costs are slightly less than the cached non-Frigate case.

The cached "Through Server" case assumes an actual transmission and so assumes the full cost of the transport layer as previously measured in the uncached case. We still do however benefit from the lower cached UFS costs when calling back into the kernel. While a great improvement over uncached times, the cached "Through Server" case does not compare well to the cached non-Frigate time.

The potential effect of the aggressive optimizations mentioned before yield the results shown in Table 5.5. As before, in our ideal implementation, we assume a one-way IPC can be accomplished in 9 micro-seconds with data copying replaced by memory mapping. Bypass overhead is handled by calling the real service layer directly. In this idealized scenario, UFS is also assumed to use mapping techniques to reduce data copying. In the cached case, all UFS data copies are eliminated. For non-cached data, one copy is still necessary during I/O to transfer the data into the virtual memory system. The effect of all of these potential optimizations is to speed up both Frigate and non-Frigate operation. With these speed increases, Frigate is now within 10 % of non-Frigate operation in all cases. In cases, where requests can be satisfied directly by the server without lower layer calls, Frigate does much better than non-Frigate.

Measurements for an 8KB write operation in Frigate and non-Frigate systems are shown in Table 5.6. Times for both synchronous and deferred writes are considered. A "deferred write" is often, but not always, just a write to the cache. After several operations, batched writes are performed and actual I/O occurs. This behavior accounts for the low average time but high standard deviation of XNFS and UFS deferred operations.

The synchronous write case is much like the uncached read case. I/O is even more dominant for writes. Two data copies occur for both XNFS and UFS layers.

Write Segment	Synchronous mean	Synchronous stddev	Deferred mean	Deferred stddev
System Call	240.1	4.27	240.1	4.27
FFD	55.9	2.96	55.9	2.96
XNFS	12796.4	87.86	2810.7	1575.19
UTOK	832.7	8.58	832.7	8.58
EAFS	57.5	2.32	57.5	2.32
UFS	55676.2	8166.61	1651.8	977.14
Path	**Total**	**Percent**	**Total**	**Percent**
Non-Frigate	55916.3	100 %	1891.9	100 %
Compatibility	56029.7	100 %	2005.3	106 %
To Server	13092.4	23 %	3106.7	164 %
Through Server	69658.8	125 %	15634.4	826 %

Times in microseconds. Percent of Non-Frigate Time.

Table 5.6: Write Segment and Path Times

XNFS also includes two context switches. XNFS costs are about the same as the read case. For writes, FFD and EAFS costs are insignificant. Requests that can be satisfied by the server alone have even more advantage than in the read case. UTOK costs are close to data copy time.

For deferred operation, costs are drastically reduced. Most of the time, a deferred operation only results in a single data copy at either transport or UFS level. However, at some point a batched write operation does occur. A deferred "To Server" operation only rarely causes actual IPC. Most of the time, only the transport cache is written. The "Through Server" total does assume an actual IPC, but the cached UFS cost is used. Overall, Frigate operations do much

Ideal Write Segment	Synchronous Time		Deferred Time	
System Call	240.1		240.1	
FFD	0.0		0.0	
XNFS	18.0		18.0	
UTOK	18.0		18.0	
EAFS	0.0		0.0	
UFS	54841.2		816.8	
Path	Total	Percent	Total	Percent
Non-Frigate	55081.3	100 %	1056.9	100 %
Compatibility	55081.3	100 %	1056.9	100 %
To Server	258.1	0.5 %	258.1	24 %
Through Server	55117.3	100 %	1092.9	103 %

Times in microseconds. Percent of Non-Frigate Time.

Table 5.7: Ideal Write Segment and Path Times

better than the synchronous case, but non-Frigate operations improve even more through deferred operations.

Ideal optimization figures are shown in Table 5.7. The effects for write are very similar to the read case. As with read, we still assume there will be one data copy for I/O operations. IPC is a negligible cost in many cases. Frigate is at worst only slightly slower under these assumptions. In cases where the server directly satisfies requests, Frigate shows spectacular performance.

	Layer Configuration					
	UFS		EAFS UFS		FFD EAFS UFS	
Phase	mean	stddev	mean	stddev	mean	stddev
mkdir	4.3	2.28	4.5	0.97	4.8	1.04
cp	11.6	2.01	16.0	1.59	16.9	1.93
find	10.1	1.50	11.9	1.66	11.8	1.84
read	15.4	0.97	15.7	1.09	16.2	0.96
make	103.0	1.85	106.4	1.35	105.6	1.77
user	83.0	0.30	83.1	0.28	83.3	0.28
sys	37.7	0.35	38.8	0.37	39.9	0.36
elapsed	149.8	3.12	160.9	3.47	161.1	2.39

All figures in seconds.

Table 5.8: Non-Use Times

5.7.2 Non-Use 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 (MAB)[HKM88, Ous90] in a file volume with and without the Frigate service. The Modified Andrew benchmark is a macro-benchmark, though there are several distinct phases in the benchmark. A mix of file operations and compute-intensive operations are performed.

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

	Layer		
Phase	EAFS	FFD	Both
mkdir	+ 5 %	+ 7 %	+ 12 %
cp	+ 38 %	+ 8 %	+ 46 %
find	+ 18 %	− 1 %	+ 17 %
read	+ 2 %	+ 3 %	+ 5 %
make	+ 3 %	0 %	+ 3 %
user	0 %	0 %	0 %
sys	+ 3 %	+ 3 %	+ 6 %
elapsed	+ 7 %	+ 1 %	+ 8 %

Table 5.9: Non-Use Times as Percentage Change Over UFS

attributes (EAFS) and the actual Frigate Dispatcher layer (FFD). Additional measurements were made with just the extended attribute layer added to see how much of the overhead came from that support layer. The average of 30 runs in each configuration is shown in Table 5.8. Times for each phase of the benchmark and the breakdown between system and user times is shown. The percentage increase in running times is given in Table 5.9. Figures are given for both EAFS and FFD together and separately.

Overall, we believe the overhead of having Frigate, but not using it, is adequate for research purposes. However for wider general use, the overhead would have to be lowered. In these macro-benchmarks, there is an 8% elapsed time overhead of which 7% comes from our extended attribute support layer. To improve these results, we need not resort to the extensive and aggressive optimizations previously described. A far more modest effort that just improves the extended attribute service could greatly improve this situation. For example, storing Frigate attributes in leftover UFS inode space would eliminate this overhead. In system time, each layer contributes about 3%. This result compares to about 2% for a layer that does nothing but bypass.[Hei91] It is not clear why there is a decrease in find times when FFD is added unless the EAFS cache is less effective for some reason.

It should be noted there are some limitations to using the well-known MAB. Despite the large number of runs in a controlled environment, some phases show relatively large standard deviations. This result is due to the fact that some of the phases of MAB are relatively short and only coarsely measured. Also, MAB tends to be more weighted towards the compute-bound last phase, which is relatively insensitive to file system performance.

5.7.3 Comparison to Library

Our third 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 encryption machine [GS91]. 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 overloaded the read/write vnode operation to provide transparent encryption and decryption. A new operation was added

File Size (KB)	Library		Frigate		Frigate as
	mean	stddev	mean	stddev	% of Library
8	0.3	0.05	1.3	0.44	433 %
80	2.8	0.23	4.7	1.56	168 %
800	26.4	0.18	32.5	0.55	123 %
8000	295.6	5.80	343.2	6.58	116 %

All runtimes in seconds.

Table 5.10: Single Client Times

to provide the session key to the extension. Because of Frigate's architecture, file data could be buffered and shared between clients in the clear. To prevent unauthorized parties from gaining access to clear text data, the access vnode operation was also redefined.

Our first experiment used a single client which iterated over each of block of data in a file. Each block was read, modified, and written back into another location in the file. This client provided a mix of file I/O operations and compute processing. The block size (8 KB) was chosen to match that of the underlying UNIX file system. The results for the elapsed time of the library and Frigate implementations is shown in Table 5.10. Each result is calculated from 30 runs. The table also shows Frigate runtimes as a percentage of the library implementation times.

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 gets larger, Frigate overhead decreases. For an 800 KB file, the overhead is 23 %. At 8000 KB, the overhead falls to 16 %. As the file size gets larger, the server startup time is amortized over the entire run. The vast majority of

File Size	Repetitions							
	1		10		100		1000	
(bytes)	mean	stddev	mean	stddev	mean	stddev	mean	stddev
1	0.2	0.06	0.2	0.05	0.4	0.11	2.4	0.14
8 K	0.8	0.19	5.5	0.22	52.9	0.18	526.5	0.57
80 K	5.4	0.09	52.4	0.97	526.1	1.38	5263.7	3.24
800 K	57.2	17.23	556.3	109.77	-	-	-	-

All runtimes in seconds.

Table 5.11: Shared File Performance for Library Implementation

file operations are reads and writes and thus very dependent on total size of data transferred. (For this experiment, the amount of data moved in file operations is twice the file size.) 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 IPC transport, and the need to context switch to the server process to handle requests. Again, it appears that far less than our idealized optimizations could close any performance penalties shown here.

Our second experiment involved two clients sharing access to an encrypted file. One process iterated over each block of the file. Each block in the file was read, modified, and written. Before proceeding to the next block, the second client also read, modified, and wrote the same block. The entire file was processed repeatedly.

The elapsed runtimes for the library and Frigate implementations are shown in Table 5.11 and Table 5.12. The file size and the number of times the file was

File Size	Repetitions							
	1		10		100		1000	
(bytes)	mean	stddev	mean	stddev	mean	stddev	mean	stddev
1	1.1	0.54	1.0	0.23	1.3	0.25	4.5	0.31
8 K	1.2	0.25	1.7	0.26	6.5	0.46	54.7	1.24
80 K	4.2	0.35	8.4	0.51	49.1	0.40	456.6	1.51
800 K	34.5	0.64	74.1	0.20	-	-	-	-

All runtimes in seconds.

Table 5.12: Shared File Performance for Frigate

File Size	Repetitions			
(bytes)	1	10	100	1000
1	550 %	500 %	325 %	188 %
8 K	150 %	31 %	12 %	10 %
80 K	78 %	16 %	9 %	9 %
800 K	60 %	13 %	-	-

Table 5.13: Shared File Frigate Performance as Percentage of Library Times

processed were varied in our experiment. Some of the largest cases were not measured due to excessive runtimes for the library implementation. Each result in the tables is calculated from 30 runs. The runtimes for Frigate as a percentage of the library implementation are shown in Table 5.13.

The server startup cost again makes the library implementation faster in small cases. Since so little data is actually transferred in the one byte file case, it essentially only measures the overhead of the framework. Again, it is confirmed that Frigate is slower than the UNIX file system.

However, in the larger cases, Frigate does significantly better. The ability to share text in the clear reduces runtime substantially. In some cases, Frigate runs in less than one-tenth 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. Frigate also has architectural advantages in some other cases. For example, Frigate has access to the operating system's internal identifiers, allowing it to uniquely identify files.

For a fixed amount of I/O, we find also that Frigate does slightly better on smaller files processed many times rather than larger files processed fewer times. This result is the expected behavior of Frigate's buffering. Smaller files take fewer I/O operations to be "faulted" into the buffer cache. Thereafter, requests can be served from the cache without needing to perform actual disk I/O operations.

5.7.4 Other Performance Examples

Overall performance of Frigate for an interactive web CGI application was acceptable in our test environment. However, this judgment is subjective. There are, of course, other applications where server starts and interprocess communication can make Frigate significantly slower.

Since Frigate makes use of Stackable Layers, one might naturally wonder how Frigate's performance would compare to a pure Stackable Layers implementation. Extensions that can be implemented equally well under Frigate or Stackable Layers can only make use of vnode operations. Stackable Layers, of course, does not support any general object model. In these cases, a pure in-kernel Stackable Layer probably offers higher performance because the layer-to-layer communications cost to and from the extension layer is reduced. The overall performance cost will vary depending on the relative importance of this layer-to-layer communication.

On the other hand, Stackable Layers suffers the various disadvantages of development and debugging inside the operating system. In some cases, the coding and debugging may be significantly more difficult because of the programming restrictions inside the operating system.

5.7.5 Architecture and Performance

Overall, the relative performance of Frigate's current implementation is dependent on server startup times and IPC costs. The server startup time can be amortized by long-lived file use or by sharing of servers. It is clear that our current IPC implementation, while general, is not very efficient. The IPC cost can be dealt with in a number of ways including the strategies used in our "ideal" optimizations. Some further discussion of other near-term, strategies for startup and IPC appear in our discussion of future work in the final chapter

Most of Frigate's performance issues come from transition between components during IPC with extensions or layering overhead. It is clear that IPC overhead is inherent to Frigate architecture. This is true because of the simple fact that extensions are placed in separate processes. Even with our ideal optimizations, some IPC cost will always be incurred. Some other approaches do not place extensions in separate processes and thus need not perform IPC. Because of this, when it comes to communicating with an extension, Frigate's architecture is inherently slower than these other approaches. Whether this overhead is significant depends on the application as shown in our macro-benchmarks.

Frigate chooses to to wall off extensions in another process to provide safety even with untrusted code. Some other approaches either use alternate forms of runtime protection or use restrictions on extension construction to assure *some* level of safety. A more extensive discussion of the semantic and functionality differences between these approaches appears in the chapter on related work. Here, we concentrate on performance differences between the strategies.

Other forms of runtime protection include restricting memory reference ranges. Though semantically very limited, this approach adds no overhead, if hardware support is available. If semantically richer tests are added that include code execution, then performance starts to approach IPC levels, except that no context change need occur. Restricting construction of extensions relies on safety enforcement by a compiler or by proof of program properties. Compiler and proof strategies limit what can be expressed in an extension. However, the resulting extensions can be called with no more than simple function call overhead.

Frigate also makes use of Stackable Layers. While layering improves modularity, overhead costs are imposed by Stackable Layers. For Frigate, they appear primarily in bypass operations. As we described in our ideal optimizations, much of this overhead can be avoided. In optimizing bypass, so that it calls the ultimate service layer directly, global information must be used. As a result, this strategy violates runtime modularity. However, it is judged acceptable for Frigate to do this, since the flexibility of dynamic stack reconfiguration is rarely exercised. Also, modularity is still preserved for layer construction and static configuration. From Frigate's point of view, this case is different from extension IPC, where module boundaries must remain intact at runtime, since extension code is untrusted. We already must assume that Stackable Layers code is trustworthy.

Can a system have all the benefits of modularity without imposing any of the runtime costs? Current approaches can only accomplish this by restricting expressive power (or alternatively by using exponential sized proofs). Ultimately, these approaches rest on empirical claims that they are useful enough in practice. Frigate takes a more conservative approach and does not try to understand extension behavior or restrict construction. But to maintain safety, Frigate then must pay an inherent performance cost. Of course, Frigate and these other systems lie in a design space continuum, and other intermediate trade-offs are possible.

Frigate, also, must ultimately rest on an empirical claim that it performs well enough for enough applications. Our benchmarks suggest that even the current implementation of Frigate does perform well enough in some cases. Further, it appears modest optimizations can practically increase the range of use. Our "ideal" optimizations suggest that, in the limit, the overall approach can be made feasible for general use.

CHAPTER 6

Related Work

Frigate does not exist in a vacuum. As object-orientation became popular, inevitably it was applied to operating systems. Also, for reasons previously described, there is a natural interest in being able to extend the operating system. We examine a representative, though not exhaustive, selection of object-oriented filing interfaces and file system extension mechanisms and compare them to Frigate.

In general, we will classify systems according to their structure and functionality. A key question is where the extension is added to the overall system. In object-oriented terms, this is equivalent to asking where the method code actually runs. 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 inter-process communication 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.

6.1 Language Systems

Object-orientation first arose as a programming language concept. Of course, programs written in object-oriented languages can use objects with support for persistent storage. Typically, there are methods to instantiate new objects from files and to save object state to files. When instantiated, the objects have a high-level method interface appropriate to the application instead of the primitive system call interface. Some object-oriented interfaces to files are even available in languages that are not themselves object-oriented. For example, Smart-Files [HMV95] provides generic object-oriented data file access in the C programming language.

Some object-oriented languages have support for similar functionality as a builtin feature. An elementary example is found in the Eiffel programming language [Mey88]. The "storable" class provides methods to read and write an internal language representation of an object to a file. It also finds any objects referred to directly or indirectly by the specified object and stores representations of them as well. Other classes gain persistence by merely inheriting the storable class.

When languages have support for persistent objects, they can offer an natural enhanced filing interface that is fully integrated with the rest of their language environment. Unfortunately, their models of file system use are usually limited and cannot use any file entities but ordinary files and perhaps pipes. Their main limitation, though, is that their rich world is not available outside of their own language environments. Persistent objects are only encapsulated when used in the language environment. They are not integrated with the operating system. They have no ability to intercept file accesses from other programs.

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. Multiple links prevent file pathnames from doing this; because generation numbers are not available at the system call level, even inode numbers are not sufficient. As we saw in the performance tests, it can also be a performance liability. Frigate attempts to get the best of both worlds. It still has the full power of an intercept mechanism below the system call level. At the same time, it avoids kernel address space development and provides an object-oriented programming environment.

There are a few systems where the entire system, sometimes including hardware, is built to support a high-level language environment. In some cases, the target language is object-oriented. In this type of system, extensions can be closely integrated with the operating system. Potentially, complete customization of the file system is possible. Practically speaking, though, extensibility depends on whether the system is designed to allow it. An example of this approach is the Symbolics LISP Machine [MSW83]. However, it proved impossible for most of these systems to keep up with the performance increases of general purpose solutions. As a result, this approach has largely died out.

6.2 Object Frameworks and Libraries

Most CORBA [Obj96] based object frameworks are designed to integrate with target language environments. Thus, for the most part, they share the characteristics of language-based solutions. These systems are typically built on top of the operating system and provide the ability to call object services in a client-server fashion. Commercial systems of this type are used to provide interoperability in a suite of applications or application components. For example, OpenDoc [FM96] is meant to provide compound document architecture and user interface. Object frameworks with "applet" and "serverlet" features offer a novel form of modularity and may be uniquely mobile or downloadable. However, when actually executing, they too share the characteristics of language-based solutions. Some other object framework cases, such as OLE 2 [Mic93] and ActiveX [Cha96], are less clear, since it is difficult to determine how much has been subsumed by the operating system.

ILU [JSS95] is a CORBA style object system. In Frigate, however, ILU is integrated with the operating system rather than using the operating system merely as an operating substrate. This is natural direction of development, since Frigate's services are specialized towards providing a object-oriented file service. But it also means that Frigate provides complete encapsulation that is not limited to boundaries of a particular object framework. All access to files, including ones outside the object model, are potentially subject to Frigate's control.

Another file system extension approach uses libraries. Commonly used system libraries are modified to intercept file accesses. Often these libraries are shared and dynamically loaded. This allows existing compiled programs to use the modified library without recompilation. In theory, this has the ability to intercept most file accesses. However, it may still be possible to circumvent encapsulation depending on the details of dynamic linking. 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 [KK89], COLA [KK92] and IFS [EP93]. These 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. Also, there are some questions of about who should be able to modify common shared system libraries. Bugs in some critical system libraries may be very dangerous. Often, there is little provision for selective use and configuration.

6.3 Servers

Another form of operating system extension is the server. The server executes at user-level outside of the operating system (or at least outside of privileged execution modes). The intercept mechanism can be above or below the system call interface. The code to actually do call redirection may be inserted as a modified library, as an operating system patch or via some other method. The extension code, itself, runs in the server instead of in the operating system. Intercepted calls are passed to the extension server by some form of IPC. The need to start servers, and the use of IPC can cause some loss in performance.

The Object-Oriented File System (OOFS) [Sum94] 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 a true object-oriented programming interface. There are no classes, methods, inheritance, etc.

The Distributed Object-Based System (DOBS) [DV89] 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 define 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 of files is not enforced.

Some operating systems offer special support to intercept system calls, sometimes as an extension of debugging mechanisms. As long as the intercept can be properly set up, complete encapsulation is possible. Interposition Agents [Jon93] 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. The User-Level File Organizer (UFO) system [AIS97] is another example of using debugging facilities. UFO intercepts system calls and redirects them by using the UNIX "/proc" facility.

There are also some systems that intercept file system calls within the operating system. The File Monitor Interface (FMI) [Web93] provides a user-level server facility based on the VFS interface. Vnode operations are either passed to the standard VFS-based file system or to a server. Information in special file attributes is used to determine if server redirection occurs. In this way, the intercept mechanism is similar to the Frigate Dispatcher. However, there is no provision for an object-oriented programming model.

Similar facilities were also built for other operating systems. Pseudo-File-Systems [WO89] were developed for Sprite [OCD88]. Userfs [Fit95] provides this service for LINUX [WK95]. 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 [BP88] and Extensible Streams [RLM86] go somewhat further in that servers are started automatically, if none is running. Both also use type identifiers similar to Frigate's version attributes. This allows specifying servers on a file level granularity.

All of the systems that redirect from inside the operating system (and Frigate as well) can provide enhanced behavior for existing UNIX system calls by overloading vnode operations or their equivalent. This allows existing as well as new programs to benefit. Only Frigate, though, exports any object-oriented interface or can operate on anything but file entities. Frigate offers a richer model with an object-oriented interface integrated with the VFS model, the ability to also operate on all file system entities, a file level granularity, selective intercept based on attributes and servers assigned by document type, version and server capacity.

Micro-kernels attempt a more radical reorganization of the traditional monolithic operating system based on the server approach. The entire operating system is reorganized along these lines. 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 [ABB86] and Chorus [RAA90]. Exokernel [EKO95] and Cache Kernel [CD94] attempt to push this organization as far as possible. The file system is usually implemented by one or more such servers. In the process of reorganization, many micro-kernels have also acquired an object oriented structure. An example is Mach 3.0 [GJ91, JCS91, SJ95]. While this object-orientation is of great use in the development and specialization of the operating system, it is not generally exported to users. Instead, it is used as an internal structuring tool. Users continue to the see the same unextended system call interface. Frigate, of course, offers a single integrated model to both method programmer and client programmer.

6.4 Extensible Operating Systems

A number of approaches to extensibility have been tried. As described previously, early attempts to add extensibility interfaces to the operating system included streams [Rit84] and dynamically loaded device drivers. The later is included in systems such as SunOS 4.x [Sun90] and Chorus [Arm91]. Both streams and loadable drivers originally arose from the need to deal with hardware heterogeneity. These two examples are illustrative of somewhat different approaches to extension.

Streams are an attempt to modularize and restructure serial I/O. Similar initiatives were taken to reorganize other parts of the operating system. As previously described, the VFS [Kle86] interface allowed complete file systems to be added in a modular fashion to UNIX. Applications of VFS include NFS [SGK85], the Automounter [CL89] and Union Mounts [PM95]. The NFS interface provides a natural way to intercept calls in a manner similar to that already described for other server systems. For example, the Translucent File Service [Hen90] used this strategy to provide custom views of the file system and copy-on-write semantics. These features were used in Sun's Network Software Environment (NSE) to provide configuration and version management services. A similar approach was taken in the Rational (formerly Atria) Clearcase [Rat] product.

Vnode "stacking" allowed VFS file systems to be decomposed into layers. Stackable Layers [Hei91, HP94, Hei95] has been used to implement the Ficus replicated file system [GHM90, Guy91] and is used by Frigate and other systems. An alternative model of stacking is described in [Ros90]. The Spring operating system [KN93, NKM93, RMP93] also offers stacking with additional object-oriented features. Instead of whole file systems, stacking allow modules of a smaller granularity to be added to the file services. They also offer significant configuration flexibility by being composable into stack and tree-like structures. However, without other features, Stacking will not satisfy our requirements for ordinary user access. Frigate attempts to bridge this gap.

The dynamically loadable device driver is one example of *patching* the operating system. By "patching", we mean any sort of alteration of post link time operating system code. In its most extreme form, patching is literally changing various bits of assembly code by use of a debugging or overlay tool. Traditionally, this was done to work around bugs, when source code was not available, or to provide customized behavior by modifying interrupt vectors or trap handlers. For example, MacOS [App92] has used patching to add fixes at boot time to correct problems with "toolbox" code. In some operating systems, the loadable device driver interface is the only available way to modify certain parts of the system. As a result, the driver interface is sometimes used as a way to load code that can in turn modify other portions of the operating system that have nothing to do with device handling. Unfortunately, in some systems, viruses have also used patching to affect system behavior.

Safer, controlled patching can be provided by downloading "safe" modules into the operating system. The modules are made safe by restricting address references [WLA93] or by using safe languages. An example of the later is the SPIN operating system [BSP95], which uses Modula-3 [CDG89] as its extension language. Another method is to use "proof-carrying code" (PCC) [NL96]. PCC contains a formal proof that the extension does not violate security policy. Unfortunately, to keep proofs to reasonable size, PCC must be fairly small.

Because patch code resides in the operating system address space, patching can offer high performance extensions. There are no process boundaries to cross. With the use of the techniques mentioned above relative safety can be assured. However, to assure "front end" safety, extensions must be restricted or stylized. Also, they are still difficult to develop and debug in the operating system. In some cases, there is limited space to add patch code and thus large extensions are not possible.

The problem with most of the extensibility interfaces for operating systems is that they are targeted at privileged, expert specialization of the operating system. They are not meant for casual users. Most also require development and debugging in the operating system address space.

6.5 Object-Oriented Operating Systems

Another approach is to provide operating systems that are themselves 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 [HKN91] 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 [CIR93, MLR88, MCR89] implements an object-oriented operating system written in a modified C++. 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 casual 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 [DLA88, PD88] 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 [HM90] 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 [BMK96] and Frigate explicitly attempt integrate compatible use with a new object-oriented operating system interface.

CHAPTER 7

Conclusion

We briefly summarize the motivation, key architectural features and contributions of this dissertation. Avenues for future research are also discussed.

7.1 Summary and Contributions

Operating systems and the file systems they contain have generally 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. Some extensibility features have appeared in other systems. However, they do not go far enough in providing open access for extensibility.

Frigate provides such open access to all users, not just to those with system privilege or specialized training. Frigate combines a user-level server framework 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 flexible implementation that does not have to mirror the structure of a corresponding implementation inside the operating system address space. The server process boundary also acts as a firewall for added safety and security.

The object-oriented framework provides a powerful 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 or enhanced. 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 version 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 very large and 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 one or two of these aspects. In short, we believe Frigate uniquely opens the file system to the world by providing an infrastructure for modular extensibility.

We also believe that Frigate and its example applications show the viability of a particular approach to system software. This approach advocates building a modular extensible framework rather than constructing a predetermined fixed set of general services. Given the increasingly complex and diverse demands made on system software, we believe that this style of system building will become norm.

Obviously, such a framework must be flexible, powerful, efficient and safe. But we also believe that strong emphasis must be placed on ease of use and on minimizing software engineering costs. This emphasis is where Frigate distinguishes itself from many other systems. We feel this is essential to current and future systems. Such systems will become increasingly characterized by continuous change, multiple versions and dynamic reconfiguration. It is therefore essential to provide the kind of functionality that Frigate demonstrates: easy extensibility, transparent version control, compatibility and incremental use.

7.2 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.

As with any object-oriented system, Frigate would benefit from an expanded class library. A number of classes have been implemented for purposes for demonstration and experimentation. Further classes and implementations could be used to provide a rich library of available abstractions. These could either be used indirectly as representative examples or directly as base classes to inherit from.

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++ [Str91], Modula-3 [CDG89] and possibly the interpreted Python language [Ros95]. This would require changes in the language dependent portion of the ILU stub generation programs and the addition of language appropriate interfaces to Frigate specific routines. Porting the server side of Frigate would require somewhat more language specific adaptation, since there is a larger interface area involved and considerable Stackable Layers involvement as well.

CORBA [Obj96] is emerging as one of the industry standards for object systems. ILU provides partial CORBA compatibility. Full compatibility would require a number of additional OMG mandated functions such as a full "Basic Object Adaptor", the "Dynamic Invocation Interface" and direct interfaces to the "Object Request Broker". Currently, the best strategy here is to wait for these features to appear and stabilize in ILU and then port Frigate to later versions of ILU. (Some of this is happening already in the current development version of ILU.)

To support true multi-media applications, Frigate needs to provide real-time support support. The best strategy here is to wait for real-time and quality of service support in ILU, CORBA and the Stackable Layers infrastructure.

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 implementations. Currently, nothing is assumed about the integrity of the server binary. A shared, protected server executable would allow some, mostly security, functions to be done in the server at startup and would allow some simplification of server design.

A dynamically loadable design would also allow servers to tailor themselves while running and load implementations 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 their constituent components. Each implementation would be loaded only as needed. New classes and implementations would not require relinking a server executable. In effect, we would be extending late binding to server binary construction as well.

Currently, servers are indexed by interface and implementation. This structure could be generalized slightly to give servers their own identifiers. The proposed change would allow servers to be identified as implementations of several different interfaces. This could be expanded even further to support runtime changes in server abilities. Such a system would track exactly what implementations are currently available in running dynamically loadable servers.

Having a registry of available services is also a desirable feature in its own right. This functionality would allow the use of servers that have lifetimes independent of document access patterns. Such servers would inform the registry as their availability changed. The repository server would not have to explicitly manage this style of servers.

Another possible improvement is to allow servers to be distributed on other hosts. This would allow additional server configuration flexibility and automatic load balancing of servers. To allow this possibility, a distributed alternative to the Stackable Layers "utok" transport layer must be provided. This could be done by providing additional functionality to the NFS transport layer. Frigate servers could then call back into the operating system substrate beneath the Dispatcher from a remote host. To support remote servers, Frigate repository protocols would also have to be extended slightly.

A number of performance improvements are possible as well. As described earlier, it is possible to add multiple repository servers or even multiple copies of the repository to prevent delayed repository requests. Likewise, multithreaded servers could also reduce service request delays at the Frigate server end. This later possibility, however, depends on out-of-kernel Stackable Layers multithreading support. Also, all concurrently executing threads would have to be restricted to the same access credentials.

Probably, the largest payoffs in performance are in improving communication with the server and server startup. If distributed servers become a reality, then something similar to the current NFS based transport is probably required. However, for servers on the same host, considerably better performance is probably possible. More efficient interprocess communication would be used. This approach would probably use strategies similar to that used in micro-kernel designs (see [Lie93]).

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. If another document was assigned to the server before timeout, server startup could be avoided. This would depend on locality in server access patterns. Unfortunately, such locality is an unproven idea.

Another complementary strategy would rely on server *priming*. This approach would start some number of servers ahead of time. The effectiveness of this strategy would rely on being able to guess which servers would be popular ahead of time. Thus, it also ultimately relies on locality. However, if combined with server dynamic loading, as described above, a different picture emerges.

In this scenario, *generic* servers are started ahead of time when the system is idle. As servers are needed, they are assigned from the pool of generic servers. Specific functionality is made available by dynamic loading of classes and implementations. In this way, the server startup time is avoided. However, not all overhead is avoided. Forking and process overlay time is merely replaced by dynamic loading time. However, it is probable, though, that implementations will dynamically load in less time than it takes to fork and overlay. This is because forking is expensive and overlaying the process image includes copying not only the implementation but also the server support structure itself.

Beyond the file system domain, another avenue for research is the application of Frigate's design philosophy to other system software. Here, the key is to identify places where an open modular framework approach would provide the most leverage.

References

[ABB86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development." In *USENIX Summer Conference Proceedings 1986*, pp. 93–112. USENIX Association, June 1986.

[AIS97] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. "Extending the Operating System at the User Level: the Ufo Global File System." In *Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 77–90. USENIX Association, January 1997.

[Ald92] Aldus Corporation. *TIFF Revision 6.0*, June 1992.

[App92] Apple Computer, Inc. *Inside Macintosh*, August 1992.

[Arm91] François Armand. "Give a Process to your Drivers!" In *Proceedings of the EurOpen Autumn 1991 Conference*, September 1991.

[BMK96] Jose M. Bernabeu-Auban, Vlada Matena, and Yousef A. Khalidi. "Extending a Traditional OS Using Object-Oriented Techniques." In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems*, pp. 53–63. USENIX Association, June 1996.

[BP88] Brian N. Bershad and C. Brian Pinkerton. "Watchdogs: Extending the UNIX File System." In *USENIX Winter Conference 1988 Proceedings*, pp. 267–275. USENIX Association, February 1988.

[BSP95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Backer, Craig Chambers, and Susan Eggers. "Extensbility, Safety and Performance in the SPIN Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 267–284. ACM, December 1995.

[CD94] David R. Cheriton and Kenneth J. Duda. "A Caching Model of Operating System Kernel Functionality." In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pp. 179–193. USENIX Association, November 1994.

[CDG89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. "Modula-3 Report (Revised)." Research Report 52, DEC Systems Research Center, November 1989.

[Cha96] David Chappell. *Understanding ActiveX and OLE.* Microsoft Press, Redmond, WA, 1996.

[CIR93] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM*, **36**(9):117–126, September 1993.

[CL89] Brent Callaghan and Tom Lyon. "The Automounter." In *Proceedings of the Winter 1989 USENIX Conference*, pp. 43–51. USENIX Association, January 1989.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, second edition, 1991.

[Com90] CompuServe Incorporated. *Graphics Interchange Format, Version 89a*, July 1990.

[DLA88] Partha Dasgupta, Richard J. LeBlank Jr., and William F. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work." In *The 8th International Conference on Distributed Computing Systems Proceedings*, pp. 2–9. IEEE Computer Society Press, June 1988.

[DV89] Prasun Dewan and Eric Vasilik. "Supporting Objects in a Conventional Operating System." In *Proceedings of the Winter 1989 USENIX Conference*, pp. 273–285. USENIX Association, January 1989.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 251–266. ACM, December 1995.

[EP93] Paul R. Eggert and D. Stott Parker. "File Systems in User Space." In *Proceedings of the Winter 1993 USENIX Conference*, pp. 229–240. USENIX Association, January 1993.

[FG91] Roger Faulkner and Ron Gomes. "The Process File System and Process Model in UNIX System V." In *Proceedings of the Winter 1991 USENIX Conference*, pp. 243–252. USENIX Association, January 1991.

[Fit95] Jeremy Fitzhardinge. *Userfs – Filesystems Implemented as User Processes.* Softway Pty. Ltd., August 1995.

[FM96] Jesse Feiler and Anthony Meadow. *Essential OpenDoc*. Addison-Wesley, New York, NY, 1996.

[FRS96] Peter Fingar, Dennis Read, and Jim Stikeleather. *Next Generation Computing: Distributed Objects for Business*. SIGS Books & Multimedia, New York, NY, 1996.

[GHM90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. "Implementation of the Ficus Replicated File System." In *Proceedings of the Summer 1990 USENIX Conference*, pp. 63–71. USENIX Association, June 1990.

[GJ91] Paulo Guedes and Daniel P. Julin. "Object-Oriented Interfaces in the Mach 3.0 Multi-Server System." In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pp. 114–117. IEEE Computer Society Press, October 1991.

[GS91] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*. O'Reilly & Associates, Sebastopol, CA, 1991.

[Guy91] Richard G. Guy. "Ficus: A Very Large Scale Reliable Distributed File System." Technical Report CSD-910018, UCLA Computer Science Department, June 1991.

[Hei91] John Shelby Heidemann. "Stackable Layers: an Architecture for File System Development." Technical Report CSD-910056, UCLA Computer Science Department, July 1991.

[Hei95] John Shelby Heidemann. "Stackable Design of File Systems." Technical Report CSD-950032, UCLA Computer Science Department, September 1995.

[Hen90] David Hendricks. "A Filesystem for Software Development." In *Proceedings of the Summer 1990 USENIX Conference*, pp. 333–340. USENIX Association, June 1990.

[HK93] Graham Hamilton and Panos Kougiouris. "The Spring Nucleus: A Microkernel for Objects." In *Proceedings of the Summer 1993 USENIX Conference*, pp. 147–159. USENIX Association, June 1993.

[HKM88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[HKN91] Graham Hamilton, Yousef A. Khalidi, and Michael N. Nelson. "Why Object Oriented Operating Systems are Boring." In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pp. 118–119. IEEE Computer Society Press, October 1991.

[HM90] Sabine Habert and Laurence Mosseri. "COOL: Kernel Support for Object-Oriented Environments." In *OOPSLA/ECOOP '90 Proceedings*, pp. 269–277. ACM, October 1990.

[HMV95] Matthew Haines, Piyush Mehrotra, and John Van Rosendale. "SmartFiles: An OO Approach to Data File Interoperability." In *OOPSLA '95 Conference Proceedings*, pp. 453–466. ACM, October 1995.

[HP94] John S. Heidemann and Gerald J. Popek. "File-System Development with Stackable Layers." *ACM Transactions on Computer Systems*, **12**(1):58–89, February 1994.

[JCS91] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. "Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status." In *Proceedings of the USENIX Mach Symposium*, pp. 13–26. USENIX Association, November 1991.

[Jon93] Michael B. Jones. "Interposition Agents: Transparently Interposing User Code at the System Interface." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 80–93. ACM, December 1993.

[JSS95] Bill Janssen, Denis Severson, and Mike Spreitzer. *ILU 1.8 Reference Manual*. XEROX Palo Alto Research Center, March 1995.

[KK89] David G. Korn and Eduardo Krell. "The 3-D File System." In *Proceedings of the Summer 1989 USENIX Conference*, pp. 147–156. USENIX Association, June 1989.

[KK92] Eduardo Krell and Balachander Krishnamurthy. "COLA: Customized Overlaying." In *Proceedings of the Winter 1992 USENIX Conference*, pp. 3–7. USENIX Association, January 1992.

[Kle86] S. R. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." In *USENIX Summer Conference Proceedings 1986*, pp. 238–247. USENIX Association, June 1986.

[KN93] Yousef A. Khalidi and Michael N. Nelson. "Extensible File Systems in Spring." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 1–14. ACM, December 1993.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.

[Lie93] Jochen Liedtke. "Improving IPC by Kernel Design." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 175–188. ACM, December 1993.

[Lie95] Jochen Liedtke. "On μ-Kernel Construction." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 237–250. ACM, December 1995.

[MCR89] 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.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction*. Prentice-Hall, New York, NY, 1988.

[Mic93] Microsoft Corporation. *Object Linking & Embedding Version 2.0 Design Specificaton*, April 1993.

[MJL84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[MLR88] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. "A C++ Class Hierarchy for Building UNIX-Like File Systems." In *USENIX C++ Conference 1988 Proceedings*, pp. 65–79. USENIX Association, October 1988.

[MSW83] David Moon, Richard Stallman, and Daniel Weinreb. *LISP Machine Manual, Fifth Edition*. MIT Artificial Intelligence Laboratory, January 1983.

[NKM93] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. "The Spring File System." Technical Report SMLI TR-93-10, Sun Microsystems, Inc., February 1993.

[NL96] George C. Necula and Peter Lee. "Safe Kernel Extensions without Run-Time Checking." In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 229–243. USENIX Association, 1996.

[Obj96] Object Management Group. "The Common Object Request Broker: Architecture and Specification." Technical Report 96.08.04 (Revision 2.0), Object Management Group, July 1996.

[OCD88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. "The Sprite Network Operating System." *IEEE Computer*, **21**(2):23–36, February 1988.

[Ous90] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" In *Proceedings of the Summer 1990 USENIX Conference*, pp. 247–256. USENIX Association, June 1990.

[PD88] David V. Pitts and Partha Dasgupta. "Object Memory and Storage Management in the Clouds Kernel." In *The 8th International Conference on Distributed Computing Systems Proceedings*, pp. 10–17. IEEE Computer Society Press, June 1988.

[PM95] Jan-Simon Pendry and Marshall Kirk McKusick. "Union Mounts in 4.4BSD-Lite." In *Proceedings of the 1995 USENIX Technical Conference*, pp. 25–33. USENIX Association, January 1995.

[Pos80] Jon Postel. "User Datagram Protocol." Internet RFC-768, Internet Engineering Task Force, August 1980.

[Pos81] Jon Postel. "Transmission Control Protocol." Internet RFC-793, Internet Engineering Task Force, September 1981.

[RAA90] 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, April 1990.

[Rat] Rational Software Corporation. *Clearcase software configuration management product* . http://www.rational.com/products/clearcase/index.html.

[Rit84] D. M. Ritchie. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–1910, October 1984.

[RLM86] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach. "An Extensible I/O System." In *USENIX Summer Conference Proceedings 1986*, pp. 114–125. USENIX Association, June 1986.

[RMP93] Sanjay Radia, Peter Madany, and Michael L. Powell. "Persistence in the Spring System." In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pp. 12–23. IEEE Computer Society Press, December 1993.

[Ros90] David S. H. Rosenthal. "Evolving the Vnode Interface." In *Proceedings of the Summer 1990 USENIX Conference*, pp. 107–117. USENIX Association, June 1990.

[Ros95] Guido van Rossum. "Python Reference Manual." Technical Report CS-R9525, Centrum voor Wiskunde en Informatica, May 1995.

[RT74] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-sharing System." *Communications of the ACM*, **17**(7):365–375, July 1974.

[SGK85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and Implementation of the Sun Network File System." In *USENIX Summer Conference Proceedings 1985*, pp. 119–130. USENIX Association, June 1985.

[SJ95] J. Mark Stevenson and Daniel P. Julin. "Mach-US: UNIX On Generic OS Object Servers." In *Proceedings of the 1995 USENIX Technical Conference*, pp. 119–130. USENIX Association, January 1995.

[Sol90] Richard Mark Soley. "Object Management Architecture Guide, Version 1.0." Technical Report 90.9.1, Object Management Group, November 1990.

[Ste84] Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, 1984.

[Ste91] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Sebastopol, CA, 1991.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

[Sum94] Steve Summit. "Filesystem Daemons as a Unifying Mechanism for Network Information Access." In *Proceedings of the Winter 1994 USENIX Conference*, pp. 63–77. USENIX Association, January 1994.

[Sun88] Sun Microsystems, Inc. "RPC: Remote Procedure Call Protocol Specification Version 2." Internet RFC-1057, Internet Engineering Task Force, June 1988.

[Sun89] Sun Microsystems, Inc. "NFS: Network File System Protocol Specification." Internet RFC-1094, Internet Engineering Task Force, March 1989.

[Sun90] Sun Microsystems, Inc. *SunOS 4.1.1 Reference Manual*, March 1990.

[Web93] Neil Webber. "Operating System Support for Portable Filesystem Extensions." In *Proceedings of the Winter 1993 USENIX Conference*, pp. 219–228. USENIX Association, January 1993.

[Wei97] Jeff Weidner. "A General Purpose Extended Attribute Service as a File System Layer." Technical Report 970024, UCLA Computer Science Department, July 1997.

[WK95] Matt Welsh and Lar Kaufman. *Running LINUX*. O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[WLA93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 203–216. ACM, December 1993.

[WO89] Brent B. Welch and John K. Ousterhout. "Pseudo-File-Systems." Technical Report CSD-89-499, University of California, BerkeleyComputer Science Division, October 1989.

[Wol94] Van Wolverton. *Running MS-DOS, Version 6.22*. Microsoft Press, Redmond, WA, 1994.

[ZDL90] Lisa Zahn, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. *Network Computing Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1990.