Pi: A NEW APPROACH TO FLEXIBILITY IN SYSTEM SOFTWARE

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Dinesh Chandrakant Kulkarni, B.Tech.(E.E.), M.S.E.E.

_____

David L. Cohn, Director

Department of Computer Science and Engineering

Notre Dame, Indiana

April, 1995

Pi: A NEW APPROACH TO FLEXIBILITY IN SYSTEM SOFTWARE

Abstract

by

Dinesh Chandrakant Kulkarni

Conventional operating system design makes decisions based on assumptions about applications' usage of hardware and software resources. When the assumptions do not hold, these decisions may create a mismatch between what an application wants and what the implementation provides. This dissertation proposes a design approach called **Pi** which reduces the potential mismatch by enhancing the flexibility of system software. A system built using the Pi approach allows clients to participate in implementation decisions at run-time through **dual interfaces;** the first one for using the basic functionality, and the second one for changing the implementation.

The Pi approach uses a **reflective architecture** for flexibility. It utilizes a self-representation of a subsystem created using **resource objects** and **contracts**, which decide the subsystem's semantics. Contract implementations can be changed by clients through the second interface. The visibility of a change is restricted to a designated **scope** using a mechanism called **scope-based dispatch**.

The Pi approach has been demonstrated by designing the Pi File System (PFS) architecture and constructing its prototype implementation. The Pi approach has enabled clients to control separate components of the implementation of the file system. For example, naming and caching in the PFS implementation can be tailored by clients to their own needs. The approach has allowed clients to make incremental changes to the implementa-

tion and has ensured that the effects of changes are limited to a client-specified scope like a process or a file. Also, the overhead of Pi flexibility mechanisms is limited to a few indirections, and hence the performance penalty is small. Experience with the file system shows that it is possible to design flexible system software which meets the threefold requirements of incrementality, scope control and low overhead.

To my mother

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The work described here and this document would not have been possible without the help and support of a number of people. First and foremost, I would like to thank my advisor, Dr. David Cohn, for providing a great environment for research, allowing the freedom to pursue an interesting topic and his direction. Also, his patient and careful review of multiple drafts amid a hectic schedule made this document possible.

Dr. Peter Bahrs from IBM Corporation deserves a special note of thanks for agreeing to review my work in spite of his very busy schedule. He provided a practitioner's perspective on operating system design and helped in formulating an evaluation strategy for the prototype. Comments from other committee members, Dr. Brockman, Dr. Sha and Dr. Chen helped in focusing the document.

Members of the Distributed Computing Research Lab helped at every stage of this work. Arindam Banerji introduced the area of flexibility in the lab, collaborated on the early development of the Pi approach, and later, provided feedback on many ideas. Discussions with Michael Casey, John Saldanha, John Tracey and Alan Yoder were helpful. A number of other graduate and undergraduate lab members helped me throughout my graduate studies.

Gregor Kiczales (Xerox PARC) convinced me that flexible implementations was an area worth investigating. Jim Waldo (Sun Microsystems Labs) provided a different perspective on flexibility. Lou Bifano (IBM Corporation) made AIX source code, an invaluable learning aid, available for research.

I would like to thank all the above-mentioned individuals for their help. This work was supported by research grants from IBM Corporation. The opinions presented here are those of the author and do not necessarily represent the opinions of the IBM Corporation.

1. INTRODUCTION

System software has to face the challenge of matching diverse needs of applications to different characteristics of the resources that applications access. On the one hand, applications use resources in different ways and hence management of resources must cater to their special needs. On the other hand, resources keep evolving, thus expanding and altering the role of system software. Applications include browsers, databases and simulators while resources may be hardware resources like CPU, memory, disk etc. or software resources like files, or communication streams.

The conventional approach to the design of operating systems emphasizes architectures and implementations that deliver the best performance for 'common usage'. In doing so, the design approach makes assumptions about what is common usage and uses these assumptions to optimize the implementation. There are two problems with this approach. First, an application that does not fit the assumed common case is saddled with inappropriate mapping choices and second, evolution of hardware and software resources puts strains on the architecture and implementation leading to a mismatch between resources and mapping choices. There is a penalty borne by applications as a result of this mismatch; their performance suffers, where performance may be any of a number of qualitative or quantitative measures such as functionality, latency, throughput or reliability. As operating systems have matured and considerable experience has been gained in optimizing for the common case, it is now possible and desirable to look beyond the common case and investigate means to reduce the mismatch.

Consider the virtual memory subsystem as an example. The classic least-recently used (LRU) page replacement policy works fine for applications that show locality of reference. But it does not work well for applications that access data serially such as database applications [Stonebraker, 85] and applications that have large working sets [Franklin, 92]. The problem is that these applications have usage patterns that differ from those of common applications assumed by designers. On the resource side, the policy of faulting in a page works well for a uniprocessor, but in a distributed system, distributed virtual memory based on faulting suffers from thrashing and false sharing [Nitzberg, 91].

Such problems of mismatch between applications and implementations and implementations and resources are often addressed by bypassing existing implementations. In other words, the application or some proxy for the application takes over resource management to the extent possible. But this approach defeats the role of system software as a shared resource manager. It requires substantial additional effort and the workarounds may often be ad-hoc, unstructured and constrained by their very nature.

An alternative approach is to allow applications greater control over the implementation of system services [Anderson, 92], [Harty, 92], [Young, 89]. Instead of defining an Application Programming Interface (API) that insulates applications from mapping decisions made by the implementation, it may be possible to let the application participate in making those decisions and thereby make the implementation flexible. This dissertation claims that it is possible to do so and addresses the problems faced by a designer who adheres to this approach.

Current design of operating systems relies on *abstractions* to present hardware and other resources to applications [Loepere, 92]. Classic abstractions include processes for CPU and memory resources, files for data and storage and sockets for communication. They are implemented in the *subsystems* of an operating system; e.g. sockets are implemented in the communication subsystem. These abstractions are accessible to client appli-

cations or simply *clients*, through an *interface*. Implementation details are hidden behind the interface. In the case of object oriented approaches, the implementation is said to be *encapsulated*. As a consequence, an application writer is supposed to rely only on the description (sometimes formal but most often informal) of the *behavior*[1] of the abstraction and not its implementation. In other words, an instance of the abstraction is treated as a *black box* that can be accessed only through its interface which is implementation-independent.

However, in practice, transparency of the implementation does not hold [Sandberg, 85]. The observed behavior of an implementation goes beyond the abstract behavior. Consider for example, a file abstraction whose interface consists of read and write functions. A read preceded by a write at the same offset in a given file is expected to return the previously written value. Typically, a client knows this aspect of the behavior of the file *abstraction* and expects good performance from its *implementation* for small amounts of data read or written. But an implementation that flushes the written data to disk is likely to be noticeably slower than an implementation that buffers the written data in memory. This is a client-visible difference in the behaviors of the implementations. It exposes the implementation in a way not documented by the interface and not covered by the behavior of the abstraction. Thus, it violates the black box assumption about the implementation. Such client-visible features of an implementation are often important for making an abstraction usable and hence they should be controllable.

Recent work in languages has shown the value of converting black-box implementations into more controllable forms by exposing some of the implementation decisions through a *second* interface [Kiczales, 91]. As before, the first interface presents the functionality of the abstraction in a reasonably implementation-independent form and the second interface provides an *option* of participating in specific implementation choices. This

1. The term, semantics is sometimes used in literature in place of behavior.

approach is depicted in Figure 1.1. This approach is called the *dual-interface* approach [Kiczales, 92a]. The resulting implementations are called *open implementations* and the architectures that they are based on - *open architectures*. Alternatively they are also called *flexible implementations and architectures*. In the rest of the document, we will use the terms *openness* and *flexibility* interchangeably to denote a property of an architecture or implementation that allows client participation.

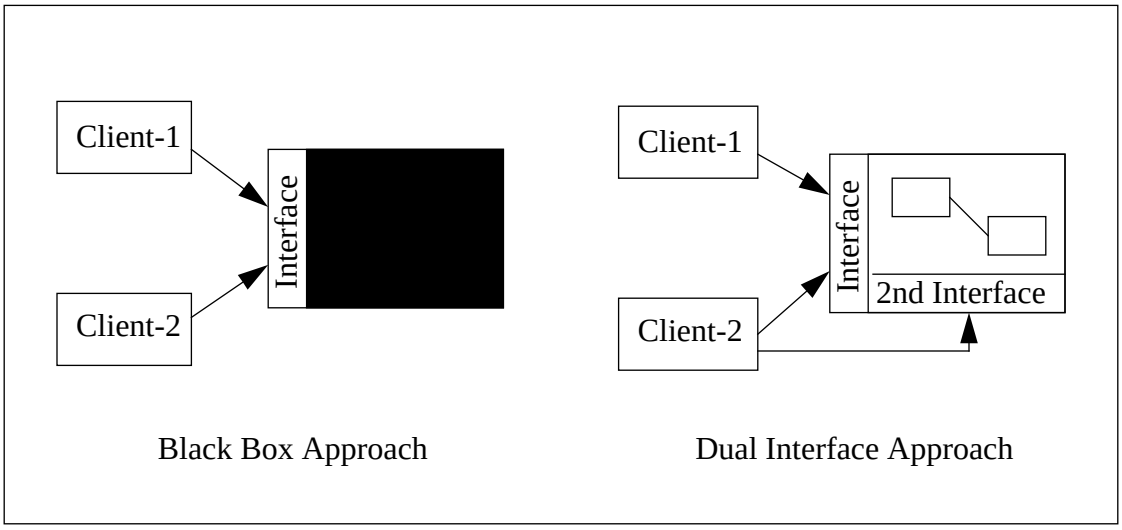

Figure 1.1: Black Box and Dual Interface Approaches

Of course, as shown in the figure, a client can choose to not worry about the additional control and just use the *default implementation*. The default implementation would be similar to the implementation obtained in case of the conventional approach which makes the common case assumption and applies relevant optimizations. Thus, the second interface is optional and not a mandatory burden for application writers. The idea of realizing flexibility through dual interfaces forms a basis of the work described in this document.

The goal of this work is to develop *an* approach to building flexible system software and to demonstrate it in an example subsystem - the file system. Towards this end, we will discuss some of the problems encountered in designing flexible system software and cer-

tain techniques to address them. In that discussion, we will consider flexibility as the main design goal or figure of merit and explore its costs and trade-offs with respect to more traditional design goals such as latency and protection. Further, we will investigate the file system as an example subsystem for openness. We will propose an open file system architecture and discuss its advantages over existing file system architectures. We will also present the experience gained from implementing its prototype. A file system has been chosen as the subsystem to concentrate on because it is important to a wide range of applications, it manages a diverse set of resources and it is convenient for prototype implementation. This work is conducted in the context of existing operating systems and widely used systems programming languages - C and C++. As such, we will take for granted basic operating system services provided by commercial operating systems and the ensuing constraints.

The next section discusses the problems faced in designing an open system software subsystem. Chapter 3 outlines some of the related work in languages and operating systems. It is followed by a section proposing a basic architectural approach called Pi and discussing issues related to the problems described in Chapter 2. A new file system architecture called the Pi file system architecture is described in Chapter 5 and its prototype in Chapter 6. Chapter 7 addresses some of the caveats and concerns expressed by the operating system community. Chapter 8 describes some directions for future work and concludes.

2. PROBLEM OF FLEXIBILITY

A designer who has accepted the tenet of open implementations faces a number of problems. In this chapter, we will discuss the key problems and their ramifications to build a foundation for the Pi approach. But we will start with a simple example to shed more light on open implementations.

Consider a file abstraction. Here we will restrict our focus to files as persistent byte streams. A simple interface to a file abstraction could consist of `open()`, `close()`, `seek()`, `read()` and `write()` calls. While this set of calls does provide the core functionality of a file, it leaves a lot of aspects to design choices. Here are some examples -

- When does a write become persistent, when the write call returns or at some later point?
- If there is a crash, under what circumstances is the data recoverable?
- If the file is normally resident on a remote server, does the `read()` call go over the network or does it use a cached copy? Under what circumstances does the local file system contact the server?

File system designers have grappled with these questions and made reasonable decisions based on measurements and tradeoffs. However, even good design and engineering judgement by the designer cannot compensate for what clients know about their usage. Usage information is mostly client-specific and is sometimes available only at run-time. The basic five-call interface does not allow a client to convey enough usage information to influence implementation. But a *second* interface would allow a client to control aspects of the implementation that were considered the prerogative of the designer. A `cache()`

call could be added to request that a temporary copy be stored in memory in anticipation of read/write calls by clients, and a `flush()` call could be added to override data buffering. This second interface is shown schematically in Figure 2.1 Thus, the addition of a few calls seems to provide a solution for the questions raised above.

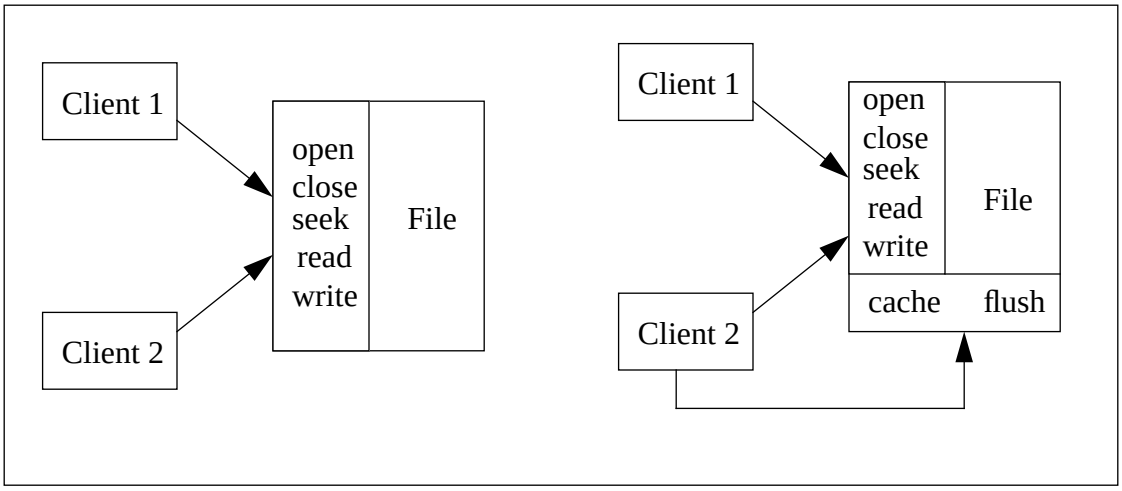

Figure 2.1: File Abstraction and Client Control

However, this solution leads to the following questions:

- How are the client-controllable aspects separated from the rest of the implementation?
- How should an application exercise control over the implementation? Should it instruct the implementation or should it just declare its mode of usage?
- What should be the form of an application's participation. Should it only choose between alternatives or provide its own custom implementation for parts of the system? In the latter case, how much does the application need to provide?
- How should the conflicts between multiple clients sharing a subsystem be handled?
- What is the cost of flexibility?

We will use these five questions to explain the problem of flexibility in this chapter and to guide the development of the Pi approach described in Chapter 4 and the evaluation of the approach in Chapter 6. The discussion of these questions will help us in defining the scope and emphasis of our approach and in relating it to the approaches used in other systems.

2.1 Separating Client-Control

There are two basic approaches to allowing clients to change a subsystem implementation. The first is simpler and more conventional hands-off approach. Here, abstractions are defined at a low-level so that applications can add their own layers of functionality on top of the subsystem that realizes the abstractions. The second approach is more complex and uses a second interface. It allows a client to change existing implementation instead of adding new layers and permits the use of a complex, non-layered structure for subsystem implementation.

The first approach leads to a clean and simple layered structure as in communication protocols. Lower layers provide basic services such as sending and receiving datagrams or byte streams. Clients add layers on top for additional services like encryption or compression. In such cases, applications make decisions that cannot be made reasonably by lower layers [Saltzer, 84]. Specifically, a layered structure allows an application to implement the policy of its choice in an additional or substitute layer. Such a layer uses the mechanisms provided by a lower layer of the subsystem as shown on the left hand side in Figure 2.2. In the figure, Client 2 is shown using the layer L2' instead of L2. For example, an application can encrypt only a small and sensitive portion of data while avoiding the overhead of encryption for majority of packets exchanged using a UDP socket.

Unfortunately, a layer-based separation of application controlled decisions from system software mechanisms may not be possible. First, a higher layer can *do* what is not

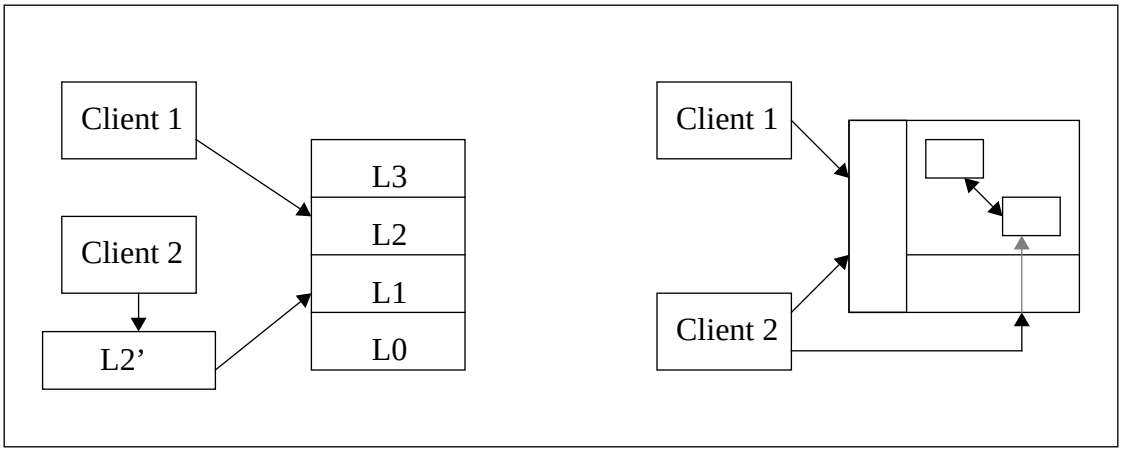

Figure 2.2: Layered and Alternate Approaches

done in the lower layers, but it cannot *undo* what is done in the lower layers. If the file abstraction from our example is implemented such that persistence is delayed till some point after the completion of the write call, a layer on top cannot provide a write call with guaranteed persistence in its behavior. Second, layering can introduce significant overhead, especially when the cost of crossing from one layer to another is substantial. For example, performance problems have been traced to inappropriate layering in communication protocols [Crowcroft, 92]. Third, it may not be possible to cleanly separate policy layers from mechanism layers due to subtle interactions between them. A good example of such interaction is recovery policies that cannot be combined with certain concurrency mechanisms used in transactions [Weihl, 89].

Thus, in general, the parts of an implementation subject to client control are not easily separable from the rest of the implementation. Hence, it may be necessary to build flexibility in different components of a subsystem and then present it to the clients through a second interface. Accordingly, in Chapter 4, we will propose that a subsystem be built as a framework of contracts and that clients be allowed to control contract implementations separated from the framework, through the second interface. Then, in Chapter 5, we will illustrate the separation in a file system architecture.

The inability to separate client-controllable parts using simple solutions like layering makes the first question from our list non-trivial. It also leads to the second question: if clients cannot add new customized layers, then how do they use flexibility and convey information about the right implementation choice?

2.2 Nature of Client-Control

The second interface may contain functions to declare the desired implementation choice. Alternatively, or in addition, it may contain functions to explicitly instruct the implementation to take certain actions. These two approaches present different tradeoffs for the subsystem developer as well as for client developers.

2.2.1 Declarative Approach

In some cases, a client can declare its mode of usage and let the subsystem take care of effecting appropriate implementation decisions. An example would be the call `open()` with an argument indicating that the file is going to be accessed in read-only mode or in sequential mode. It is then up to the implementation to decide the implications of the declaration. There are two obvious benefits of such a declarative approach. It is easy for the programmer developing a client, and the separation of responsibilities between the client and the subsystem is clean. An easier approach is likely to get used more often and a clean separation is likely to produce more robust implementations.

Moreover, the clean separation allows considerable lattitude to the subsystem implementor. A declaration can be considered in the overall scheme of the subsystem and then acted upon. For example, if the client declaration indicates that it would be beneficial to preallocate a large set of buffers, a communication subsystem can set aside buffers according to the available resources and the demands from other clients.

However, the declarative approach is limited in its capability. Only those options that are explicitly designed and implemented by the subsystem developer are available to

the client. Hence the approach is constrained by knowledge of what actions need to be taken in response to client declarations and the burden of implementing all those actions. For example, a virtual memory implementation may allow an application to declare page access policy to be sequential and consequently favor an MRU replacement policy. But if an application wants additional functionality not expressible as a page-access policy, then the virtual memory system would be unable to cater to it; e.g. an application cannot have the virtual memory system decompress pages on the fly if they are stored in compressed form in the backing store. Thus, such a virtual memory subsystem would be open only to the extent of the different declarations that it can interpret.

Further, declarative style of control is not generic. The implementation required for supporting a set of declarations may not apply to other declarations for the same subsystem, let alone to other subsystems. For example, an implementation supporting multiple paging policies does not necessarily simplify the implementation for encrypting pages; nor does it help in implementing a flexible communication subsystem. Hence, it is difficult to develop a coherent design approach if the declarative style is the primary focus.

2.2.2 Procedural Approach

Imperative or procedural control over the subsystem is an alternative to the declarative approach. In procedural control, an application can instruct the subsystem to take a certain action. A subsystem developer supporting procedural client-control can provide a set of mechanisms and let the application decide how to use the mechanisms to obtain the desired policy. For example, a distributed file system could provide `flush()` and `log()` calls as mechanisms; the former to transfer data from buffers to the remote file-server's disk and the latter to append an entry to a log maintained on the local disk. Consider a case where the `write()` call has a default policy of buffering writes in memory before batching them together to send to the server. Here, the reliability of writes can be improved by

using either of the mechanisms. For infrequent writes, `flush()` may be more suitable while for a rapid sequence of writes, `log()` may be more appropriate [Ousterhout, 88].

Now consider a client that needs the extra reliability not provided by the default implementation. A client can implement a different policy like ‘persistence upon return from write’ or ‘log before write’ using the subsystem-provided mechanisms. Such policies can be obtained by a client through its own policy routines `flush_write()` and `log_write()` as shown in parts (a) and (c) of Figure 2.1 respectively. Corresponding code for declarative style is shown in parts b and d respectively. Code fragments are shown in C++ - like pseudo code without type annotations.

(a)

```
flush_write(file, buffer, size)
{
  write(file, buffer,size);
  flush(file);
}
// use flush_write to write
```

(b)

```
// client code
{
  // declare usage pattern: need persistence
  declare(file, PERSISTENCE_ON_WRITE);
  // use write as usual
}
```

(c)

```
log_write(file, buffer, size)
{
  log(logfile_name, file, buffer, size);
  write(file, buffer, size);
}
// use log_write to write
```

(d)

```
//client code
{
  // declare usage pattern: frequent writes
  declare(file, WRITE_INTENSIVE);
  // use write as usual
}
```

Figure 2.3: Procedural and Declarative Client-control

As shown in this example, procedural control can be a reasonable alternative for declarative control. Further, procedural control can provide additional power to clients by allowing them to compose more powerful control strategies using the basic subsystem-provided mechanisms. In the above example, if procedural control is available, a client could combine the `flush()` and `log()` calls to achieve extra reliability. Such a combination may not be possible in the declarative style.

The additional power of composition gives the procedural approach a substantial advantage over the declarative approach. In fact, declarative control could be built on top of procedural control. Further, procedural control is more generic. A small set of powerful mechanisms could be applied to different parts of a subsystem and to different subsystems. On the other hand, procedural control is somewhat complex for a client. An application exerting procedural control over a subsystem needs to know how the *subsystem* works. Whereas an application following the declarative approach need only know how the *application* works. Specifically, in the procedural approach, a client programmer needs to know what actions should be taken to ensure the behavior suitable for the client.

Overall, the power and genericity of the procedural approach outweigh the additional potential responsibility for clients. For the generic flexibility approach that we are interested in, procedural rather than declarative control is more appropriate. In the Pi approach discussed in Chapter 4, the second interface provides procedural control through operations for changing contract implementations. Procedural control provided using contracts also fits well with different strategies for client participation.

2.3 Client Participation in Implementation

The next step beyond procedural control is to allow a client to provide its own implementation of parts of the subsystem. There are three main issues related to client participation, the extent of participation, the flow of control and run-time changes.

2.3.1 Extent of Participation

In an open architecture, implementation responsibility can be split between the subsystem and the client. Partitioning of implementation responsibility is in fact crucial even without concern for client implementations. So far we have treated the designers and implementors of a subsystem as one entity. In practice, multiple implementations are often provided for an architecture. An architecture typically defines the interface for clients, the basic components and the rules for interactions between those components. Some of these implementations may be built long after the initial design, and perhaps under constraints that are not shared by other implementations of the same architecture; e.g. Little Work [Huston, 93] is an implementation of the AFS file system architecture [Howard, 88] that has to address disconnected operation not handled by regular AFS implementations. Unlike AFS, which can be used only on workstations that remain connected to the network, Little Work can be used on machines that switch between networked and standalone mode. Such a change involves a substantial implementation effort beyond what would be required for developing a client that controls the implementation. Hence, an open architecture needs to allow customized implementations, perhaps derived from generic implementations through appropriate partitioning of responsibility for core components and customizable components.

But client-participation increases this need further since client developers may have neither the expertise nor the time that a subsystem implementor has. Thus, a flexible architecture should allow implementations that not only cater to different client needs and resource constraints but entail a *range of participation*; from simple substitution of a component to a major reimplementation. The actual implementation of the architecture can be viewed as a collaborative effort between the subsystem implementor and the client implementor. One extreme would be the black box approach in which the subsystem is entirely implemented without client participation and the other would be a subsystem integrated

by a client out of components. The Pi approach allows three distinct levels of usage for clients as will be discussed in Section 4.2.2

Client components can take on partial responsibility if it is not very burdensome. In other words, the subsystem must not abdicate large parts of implementations as client-responsibility. This leads to the key property of *incrementality* as a goal and a criterion for comparison and evaluation of client participation [Kiczales, 93]. An example of client participation with coarse granularity is the external pager in Mach discussed in the next chapter. Instead, the design proposed in [Krueger, 93] which allows fine-grained control over page-fault handling and replacement page selection fares better according to the incrementality criterion. In the Pi approach, contract implementations allow incremental modifications to a subsystem.

2.3.2 Callbacks and Frameworks

Altering the flow of control within a subsystem implementation is fundamental to flexibility. But with client participation, control may have to be transferred from the subsystem code to the client code. When a client determines the flow of control, the subsystem implementation takes the simple form of a library or a server that returns control to the client after completing a request.

Callbacks may be necessary when the client does not entirely determine the flow of control. Using callbacks, the subsystem can fill in gaps in its implementation using components supplied by the client. These components provide the client an opportunity to customize the subsystem to suit its needs. An example would be a virtual memory subsystem that allows a client to install its own page-fault handler. Such a handler for a client could provide a service like compression and decompression if desired. However, integrating a client-component with the rest of the implementation is often difficult. The rest of the subsystem implementation critically relies on the client component to do a task that is nor-

mally done by a part of the subsystem. Hence fewer assumptions can be made and the coupling between the client-component and the rest of the system has to be low and well-defined. In case of callbacks, additional provisions are necessary to bind symbols imported by the subsystem to code and data exported by the client. This function can be performed by the second interface of the subsystem.

In the object-oriented approach, client-supplied components could simply be instances of classes derived from the classes defined in the subsystem architecture. The resulting structure is referred to as a *framework* [Deutsch, 88] and provides a more manageable way of integrating client-components. The difference between the library approach and the framework approach is shown in Figure 2.1. We will show how to build such frameworks using substitutable contract implementations in Chapter 4 and then design a framework for file systems in Chapter 5.

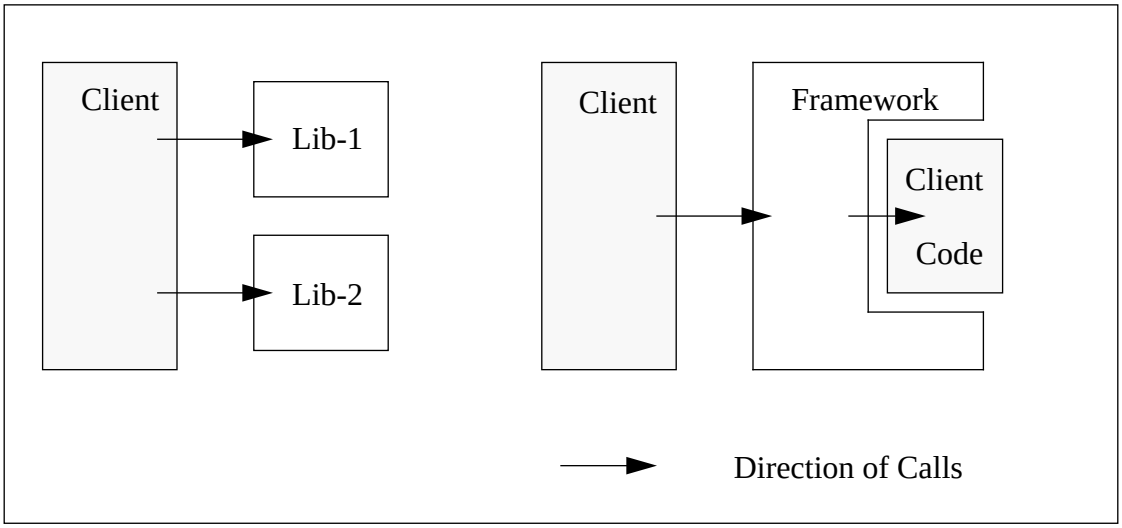

Figure 2.4: Library and Framework Approaches

2.3.3 Run-time Participation

A third important aspect of client participation is whether the client can affect the behavior of the subsystem while it is running. Allowing changes in the behavior of a system after it has been compiled and linked is important for several reasons. First, it fits well

with the practice of shipping only binaries for most system software and second, it allows clients to make changes at run-time, possibly based on external inputs that they receive. But most of the system software is developed using compilation oriented languages like C and C++ that perform static type checking and lack significant run-time support. This is in contrast to applications that use run-time environments like Lisp and Smalltalk. Besides, the overhead of a run-time environment, even if it were to be available, is often an impediment to performance. So it is important to selectively use run-time support for integrating client components. The Pi approach proposes dynamic loading of object modules for this purpose (Section 4.3.2.1).

So far we have discussed how a client can change a subsystem implementation. But a change in the subsystem implementation is likely to affect multiple clients. Hence, there has to be some way of isolating the effect of changes to the requesting client. Scope control is required to restrict the visibility of changes.

2.4 Scope Control

The most important characteristics of system software are that it is shared and that it arbitrates access to resources for multiple applications. As such, it is responsible for allowing a client to request a change for a certain *scope* and then ensuring that the change is visible within and only within the scope. This responsibility is called *scope-control* [Kiczales, 92a].

For example, if a database application asks for MRU page replacement, a word-processor client that expects to see an LRU should not be switched to MRU. In other words, the change of policy should be visible in a scope containing the database; it should not be effective in a different scope containing the word processor. While the requirement of controlling the scope wherein a change is visible seems fairly simple and obvious, the implications for the designer are not. Current design approaches assume that all clients are

equal and provide the same implementation. In our approach, it is necessary to treat clients not as just as multiple entities but also as distinguishable entities. Further, different implementations have to be somehow associated with different clients. The Pi approach uses scope-based dispatch for this purpose as discussed in Section 4.1.2.2.

It is important to note that scope-control is not necessarily equivalent to complete isolation. Whenever resources are shared, a change in resource usage for one client could indirectly affect other clients even if they get the original implementation. For example, if a client's request for an implementation that uses additional buffers for fast communication is granted, other clients using the default implementation could suffer a performance penalty. The total pool of buffers is shared by all implementations. Hence, the privilege to change implementation may have to be restricted. Further, the issue of protection from malicious or errant clients is an important one, but one that is beyond the scope of this work.

The discussion of problems associated with flexibility would be incomplete without considering the performance costs that flexibility imposes. In fact, since system software is required to be efficient, flexibility is often limited by the amount of performance penalty that clients are willing to bear.

2.5 Flexibility and Performance

Flexibility may entail a performance penalty in two ways. First, the very existence of the mechanisms and checks that a flexible implementation requires involves some overhead. Second, the use of certain flexibility features have their own associated costs.

Clients that utilize the default behavior of a subsystem should see as small a performance penalty as possible. Since they are not using the benefits of controlling the subsystem behavior, they should not be expected to pay for it. Additionally, since the default

behavior is often adequate or even appropriate for a large number of common clients, this requirement is necessary for good overall performance.

Clients often utilize flexibility features for better performance. If the overhead of changing the implementation itself is significant, the resulting performance benefit may not be worth the extra programming effort. As an example, consider a virtual memory subsystem that allows a client pager to select a page for replacement when a page fault occurs. If the overhead of calling the relevant client routine is comparable to the cost of saved page faults, the flexibility feature will not be used.

Thus, the cost of flexible mechanisms themselves must be kept under control. This is consistent with our choice of C and C++, languages that emphasize avoiding hidden costs. After careful consideration, we decided not to use more flexible but overhead-prone languages with extensive run-time environments. Specific design decisions and language and operating system features that contribute to overheads in flexible implementations will be discussed in Chapter 4. In Chapter 6, we will present the overhead observed in a prototype implementation that uses the Pi approach.

2.6 Overview

Based on the above discussion of five questions we can refine the goal of the thesis. In Chapter 1, we stated that our objective is to design an approach for flexibility in system software. Now we can take it one step further and say that our goal is to design an approach that

- delineates the client-controllable parts of the implementation;
- provides procedural control;
- allows incremental modification to an implementation;
- enables scope control for client-requested changes; and

- incurs little or no overhead when the flexibility features are not used and only a modest overhead when the features are used.

We will refer to these five key points in the following chapters to assess the effectiveness of flexibility approaches and implementations.

So far, we have discussed the issues affecting flexible implementations independent of each other. In practice, there is a strong interaction between them. It is possible to design an extremely flexible system and allow clients fine-grained control over its functioning at the cost of heavy overhead and poor protection. Instead, appropriate tradeoffs need to be made in specific implementation and where necessary in the architecture itself. We will discuss tradeoffs made in the Pi approach and in the file system architecture in Chapters 4 and 5.

In the next chapter, we will discuss specific approaches used to address some of these problems in languages and operating systems. Building on some key ideas employed by language designers, we will outline the Pi approach for adding flexibility to operating system components. We will also discuss some implementation issues that arise in a commercial operating system. However, some of the problems discussed in this section have a strong subsystem-specific component. Hence we will investigate the file system as an example subsystem, and go through the process of addressing the need for client control and then building it in the architecture and implementation of a file system.

3. BACKGROUND AND RELATED WORK

The issues related to flexibility have been studied from many angles. Recently, language designers have proposed important structuring methods for flexibility in functional and object-oriented languages. Reflective architectures which are explained below, have proved particularly useful in languages and will be used in the Pi approach. On the other hand, operating system designers have focused more on specific partitioning of functionality and performance tradeoffs involved in partitioning. Once a subsystem is partitioned into user-level servers, the servers can be replaced with custom servers by a client; a technique that will be emphasized less in the Pi approach. We will review selected related work in these two areas to provide a basis for the approach proposed in the next chapter. First however, specific terms will be defined to simplify the discussion of related work as well as the proposed approach.

3.1 Terminology

Some of the terminology in this section has been commonly used in areas like artificial intelligence and functional programming for a number of years. Its usage in operating systems is very recent. Many of the terms are discussed in detail in [Maes, 87b] and [Wand, 88].

3.1.1 Reification

Reification is the process of making an implicit entity explicit. By reifying an entity, it can be accessed through function calls, replaced or otherwise manipulated. For example,

in a virtual memory subsystem, a kernel can reify the page-fault handler and then allow a client to select an alternative page-fault handler implementation.

3.1.2 Self-representation

The *Self-representation* of a system is the reification of that system's static structure and dynamic behavior. For example, a communication subsystem can have a self-representation in the form of protocol stacks or graphs. The self-representation may or may not be modifiable. Self-representation subsumes meta-data; i.e. data about the system such as the schema of a relational database. We will deliberately avoid any mention of sufficiency or minimality of the self-representation to avoid getting into intractable issues.

3.1.3 Metacomputation

Metacomputation is a computation performed on the representation of the system rather than the subject of the system. For example, generating a page fault is object computation (or simply computation) for a virtual memory subsystem while installing a replacement-page selector is a metacomputation. Obviously, the distinction between metacomputation and computation depends on the definition of the system and its subject. In object-oriented parlance, the objects representing a system are called metaobjects and metacomputation refers to invoking functions supported by metaobjects. On the other hand, computation refers to objects on which the system acts; e.g. reading a file is computation while changing the caching policy is a metacomputation.

The object-metaobject relationship can extend beyond a single level; i.e. a metaobject can have another object as its metaobject. Thus, in principle, an infinite tower of metaobjects can be constructed. In practice, the tower is normally restricted to a few levels.

3.1.4 Reflection

In case of languages, a distinction between metacomputation and reflection is particularly important. *Reflection* is a special case of metacomputation where the language used for metacomputation is the same as that used for computation. Since we will almost always use the same programming language in both the cases for our work, we will use reflection and metacomputation synonymously. However, the distinction is important for reviewing related work. An example of reflection is a CLOS interpreter that can be modified using CLOS while a non-reflective metacomputation would be modifying a Prolog interpreter written in Lisp using Lisp.

The result of a metacomputation on a system is said to be *reflected* into a change in the behavior of the system.

3.1.5 Causal Connection

Causal connection refers to a connection between the self-representation of the system and the system wherein any modification of the self-representation is *reflected* in a corresponding change in the structure and/or behavior of the system. Thus, with causal connection, the self-representation is not simply information about a system but a determinant of the behavior of the system. For example, consider a system that stores the functions it uses in a table. In such a system, if a metacomputation changes one of the entries in the table, the system would start using the new function instead of the old one. In this case, a causal link exists between the system and the table which is its self-representation.

In summary, a systematic way of making a system flexible, or open, consists of two steps: Develop a self-representation of the system that has a causal connection with the system, and then provide an interface to the self-representation for metacomputations. We will employ this approach in the next chapter.

3.2 Flexibility in Languages

This section illustrates the reflective architectures in three languages and the efficacies of those architectures. It also shows the distinction between the results obtained in languages with run-time environments and those obtained in compiled languages. This distinction is important for our approach since we are using a compiled language. The key points in these languages that build a background for the Pi approach are:

- Reflection in 3-KRS: Basic structure of a reflective system composed of self-representation, causal connection and metacomputation.
- CLOS metaobject protocol: Metaobjects for changing the implementation and separate metaobject interfaces for client-control.
- Open implementations for C++: Use of indirection and additional run-time information.

3.2.1 Reflection in 3-KRS

Maes [Maes, 87a] has presented a coherent theory of reflection in computational systems and in particular in languages. Her approach relies on reflective architectures and an explicit separation between computation and metacomputation. She has also applied the approach to an object-oriented language, 3-KRS which is implemented in Lisp.

The basic structure of a reflective system as proposed by Maes in [Maes, 87b] is shown in Figure 3.1. There are two domains shown as rounded rectangles in the figure. A domain containing the system and the other one containing the set of subjects of the system labelled ‘some part of the world’. For example, if the reflective system is a drawing program, ‘some part of the world’ would be lines, circles and other shapes. The box labelled ‘Data’ contains information about both the domains as indicated by two outgoing arrows. The information about the system’s own domain is the self-representation. It is causally connected to the system; i.e. it represents the dynamic state of the system rather

than providing a static description of the design of the system. A change in the reflective system is accomplished through metacomputation which affects the self-representation of the reflective system. The fact that a reflective system can affect its own domain as a result of metacomputations, is indicated by an arrow labelled ‘acts on’.

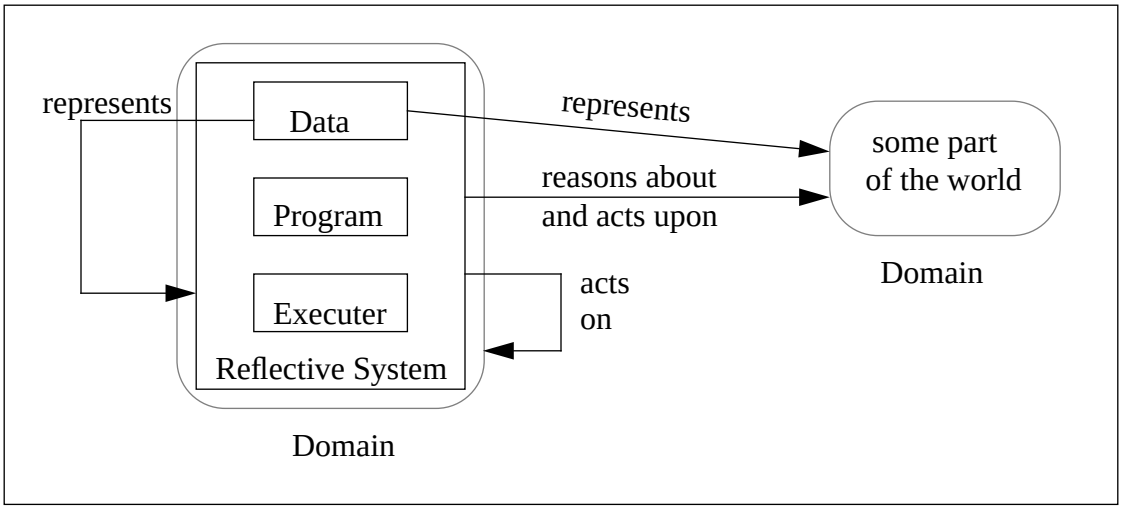

Figure 3.1: A Reflective System

A concise and precise self-representation is crucial for a reflective architecture. If the behavior of a system can be captured in terms of a few reifiable entities, then the task of establishing a causal connection and providing an interface for metacomputation is simplified. Hence, a reflective system must be designed to allow effective self-representation. 3-KRS language is based on objects and enables an effective self-representation.

3-KRS programs deal with objects and can themselves be represented as objects for the purpose of reflection. Each object consists of *slots*; slots refer to other objects or *fillers* that can be defined in the implementation language - LISP. All objects inherit from a top-level object called `Object`. Thus, an object is the main semantic entity and hence the self-representation consists of metaobjects.

When a session with a 3-KRS interpreter starts, the interpreter creates a set of objects called *primitive objects* to bootstrap the language system. Primitive objects include

`Object`, `Slot`, `Number`, `Message`, `Object-Definition` etc. All objects used in 3-KRS programs are specializations of these primitive objects. The primitive objects, typically hidden in implementations of other object-oriented languages, are reified as primitive metaobjects.

All metaobjects inherit from an object called `Meta-Object`. Metaobjects allow a program to read and change the structural information and behavior of the objects used by the program. For example, a class browser can inspect the slots contained in an object - a structural aspect, while a program could alter the process of instance creation - a behavioral aspect. Thus, 3-KRS provides self-representation by reifying the interpreter for the language. The problem of infinite tower of metaobjects is avoided by using a lazy mechanism for their creation.

3.2.2 CLOS Metaobject Protocol

The *C*ommon *L*isp *O*bject *S*ystem *M*eta*o*bject *P*rotocol (CLOS-MOP) was designed to ensure efficiency while retaining elegance in CLOS [Kiczales, 91]. *Metaobject protocols* are interfaces to the language to allow users to incrementally modify the language's behavior and implementation. The metaobject protocols are accessible from within the language. Thus, the distinction between language designers and language users is blurred. The CLOS implementation itself is structured as an object-oriented program and thus, is subject to manipulation.

CLOS objects are instances of CLOS *classes* which consist of *slots*. Classes contain *methods* that can be invoked by clients. CLOS supports multiple inheritance for specialization and *generic functions* for polymorphism. It also supports *multi-methods*; i.e. methods whose implementation is selected on the basis of multiple arguments. A feature called *method combination* allows a sequence of methods to be automatically invoked to allow preprocessing and postprocessing.

CLOS MOP incorporates these language features into its self-representation through a basic metaobject class for each of the program elements. The basic metaobject classes are: `class`, `slot-definition`, `generic-function`, `method` and `method-combination`. A metaobject class is a subclass of exactly one of these classes. A metaobject is an instance of a metaobject class. Each metaobject encapsulates information about the corresponding program entity either directly or through a reference to another metaobject. The class of each metaobject supports an interface or a protocol to provide or change the encapsulated information. Let us consider an example to see how such a protocol can be used.

Consider a class `f` that inherits from multiple superclasses `d` and `e`. The superclasses `d` and `e` in turn, inherit from their superclasses as shown in Figure 3.1 which is adapted from [Kiczales, 91]. When a method is invoked on an instance of class `f`, the CLOS implementation searches the classes in the inheritance graph for the invoked method. The search proceeds in an order defined by the class precedence list. A client can change the class precedence list by providing its own `compute-class-precedence-list` method. In the case of multiple inheritance, a programmer can change the order in which superclasses are searched for a matching method. For the inheritance hierarchy shown in Figure 3.1, a programmer can change the order from the default order to a different one, say depth-first preorder traversal. Interestingly, the client-requested order shown in the figure was in fact in use in another implementation of object-oriented Lisp, different from CLOS, and thus, control over the class precedence list was a need that had to be accommodated.

The CLOS MOP has been available in commercial implementations and its usage has demonstrated that metaobjects are an effective way of making implementations open and that open implementations indeed address the needs of client developers.

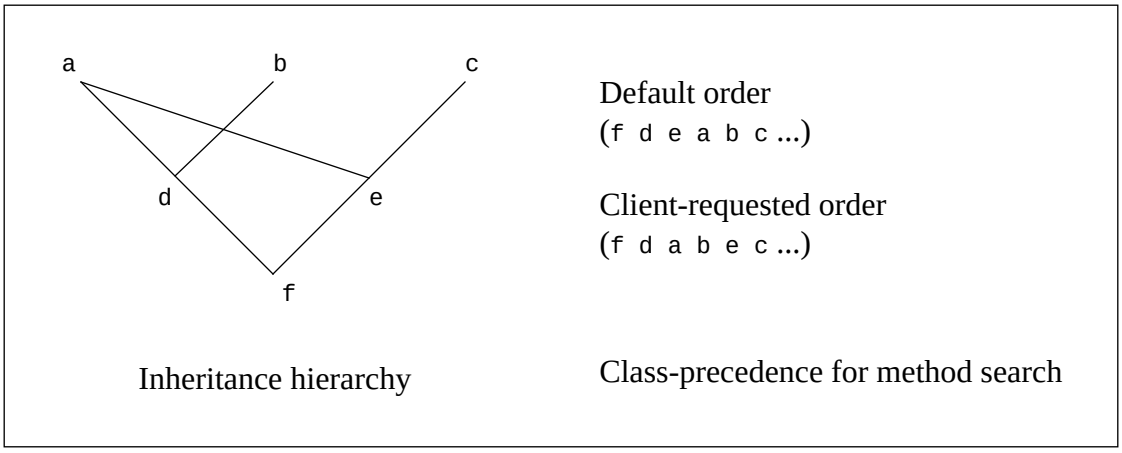

Figure 3.2: Client-control in CLOS

3.2.3 Open Implementations for C++

There are two open implementations of C++, Open C++ which aims at run-time flexibility and ΔC++ which aims at software evolution without recompilation.

Open C++, implemented at the University of Tokyo, has attempted to apply metaobject protocols to C++ [Chiba, 93]. It reifies a C++ function call through a metaobject as shown in Figure 3.1. An object together with its metaobject is called a reflective object. A reflective object can change its behavior in response to metaobject calls. A reflective object has two distinct interfaces: one for object computation and another for metacomputation, which is supported by the metaobject. Open C++ also allows an ascending tower of metaobjects which are generated by a custom translator. The translator uses directives in C++ source code, and special conversion methods provided by the programmer to generate appropriate C++ metaobjects. Figure 3.1, which is adapted from [Chiba, 93] shows a metaobject generated by the Open C++ translator. It intercepts a call for `func()`, a member function of the object in the figure. Such metaobjects can then be used for applications like synchronization and communication among distributed objects.

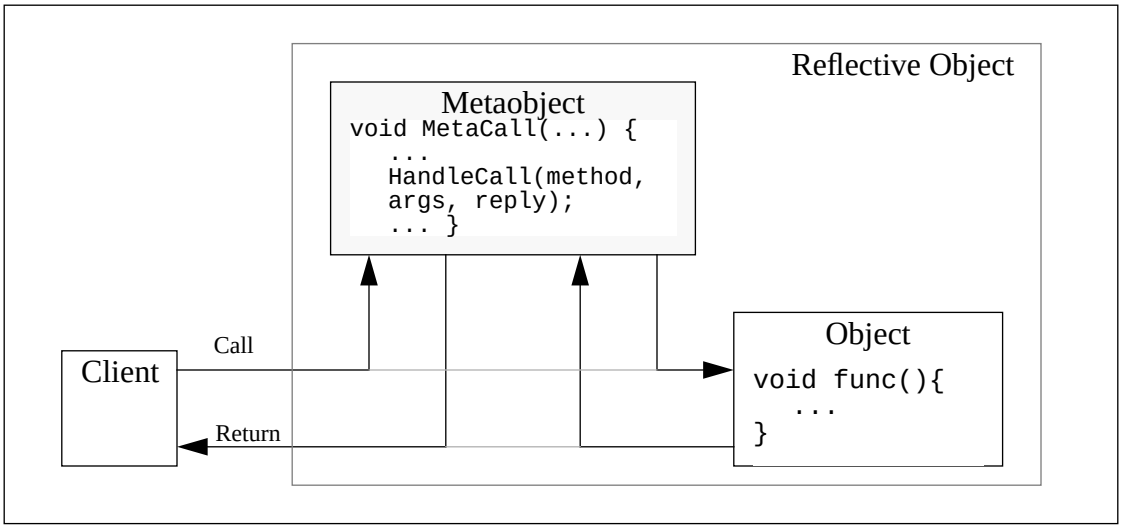

Figure 3.3: Open C++ and the Use of a Metaobject

ΔC++, on the other hand uses a modified C++ compiler instead of a translator to provide much additional control [Palay, 92]. Its goal as a compilation system, is to support class changes with minimal recompilation so that software updates can be distributed in binary form. It provides the following facilities:

- Member extension: new member functions or variables can be added to a class.
- Class extension: a new base class can be added to an existing class.
- Member promotion: functionality can be moved from a derived class to a base class.
- Override changing: altering overriding of a base-class function in a derived class.

Unlike 3-KRS and CLOS, C++ does not have an interpretive environment. Hence implementors of ΔC++ had to build substantial run-time support into the generated code. This support is provided as a set of offset variables (one per member) to resolve references to class members. Hence, the code generated by the ΔC++ compilation system leads to a substantial run-time performance penalty over that generated by an ordinary C++ compiler. Most importantly, from our point of view, ΔC++ illustrates the difficulty of obtaining

flexibility in a compilation-based environment and the resulting costs compared to the its less flexible counterparts.

3.3 Flexibility in Operating Systems

In this section we will discuss some of the operating systems that have emphasized flexibility. The following operating systems have been selected to cover different approaches and constraints. Their overall goals and scopes are *not* necessarily comparable to the work in this document. Hence, we will focus only on aspects that are relevant to flexibility. The most significant flexibility-related points are:

- Hydra: Policy/mechanism separation.
- Alto single user system: Replacement of system modules with client-supplied modules.
- Mach family: Microkernel approach relying on user-level, multiple servers for operating system functionality.
- *x*-kernel: Flexible and efficient decomposition of the communication subsystem.
- Spring: Strongly-typed interfaces and flexible object communication.
- Apertos: Metahierarchies for maximum client-control.

At the end of the description of each operating system, we will briefly relate the system to the design goals stated in Section 2.6. Then in the last section of this chapter, we will discuss the remaining flexibility issues addressed by the Pi approach.

3.3.1 Hydra

Hydra was implemented at Carnegie Mellon University in the early 1970s as a kernel for a multiprocessor system [Wulf, 74] [Wulf, 81]. The project had two goals - to provide an environment for effective utilization of the multiprocessor and to facilitate

construction of such environments. It is the latter that is of particular interest to us. The design of Hydra was based on the following principles:

- Separation of mechanism from policy: A set of mechanisms was considered appropriate for a kernel but policy decisions were carefully avoided to the extent possible.
- Rejection of strict hierarchical layering: Building blocks were typed procedures.
- Protection: Protection was uniformly provided by the kernel using capabilities [Fabry, 74] They were allowed to have any semantics defined by the higher-level software.

The Hydra kernel supported three kinds of basic entities - *procedure, local name-space* (LNS) and *process*. Each procedure contained a list of references, each of which specified an object and the actions that the procedure could perform on those objects in terms of capabilities. The kernel provided a mechanism - `CALL`, to examine the actual parameter capabilities specified by the caller and to create a new LNS for the duration of the call. Access rights provided by a caller-specified capability could even be expanded for the callee to allow it to provide privileged services associated with system software. Since such protected procedures were not limited to kernel-developers, the system could be easily extended by applications. For example, a kernel defined object called POLICY allowed user-level schedulers to communicate with the kernel scheduling mechanism.

Hydra was a pioneer system in delineating client-controllable parts through policy/ mechanism separation and in emphasizing extensibility. It provided procedural control and a limited amount of incrementality in restricted areas like scheduling policy. It also had rudimentary scope control in the form of client-designated procedures for performing some operating system functions. The overhead imposed by the CALL mechanism was considered significant but the problem was attributed to implementation rather than design [Wulf, 81]. Since Hydra was built on hardware, rather than an existing operating system,

the mechanisms and implementations are not directly comparable to those in this document. But we have built on their idea of encapsulating resources into fine-grained objects.

3.3.2 Alto Single User System

Lampson and Sproull [Lampson, 79] implemented an open operating system for the Alto personal computer. Like DOS (which was developed later), their system did not provide preemptive multitasking or protection. However, it used abstract objects with multiple implementations to provide disciplined flexibility and treated user-defined modules the same way as the system-defined modules. The system was implemented in BCPL - a typeless language.

Two procedures called `InLoad` and `OutLoad` were provided to save and restore the operating system state to the disk. Two additional procedures called `Junta` and `CounterJunta` were used to replace and restore operating system modules with an application's customized modules. These two procedures used the contexts saved by the `InLoad/OutLoad` procedures. The system was organized into several levels of services and the layout in the memory corresponded to those levels such that the most ubiquitous service was at one end of the physical memory and the least ubiquitous ones at the other end. The number corresponding to the highest level to be retained was used as an argument by `Junta` to selectively replace all levels above that corresponding to the number.

This system provided separation of client concerns and procedural control but without protection. Incrementality was supported within the limits of a primitive logical and physical organization of the replaceable modules. The issue of scope-control was limited to saving and restoring an application's environment to and from disk. As in the case of Hydra, the Alto single-user system was also implemented directly on hardware and hence the implementor's concerns and mechanisms were at a much lower level than those proposed in this document.

3.3.3 Mach Family

The Mach microkernel [Rashid, 91] is designed to run operating systems at user-level, running on top of the microkernel. The emphasis is on cheap inter-task communication and implementation of traditional kernel services in one or more servers. To accomplish this, Mach provides the following key abstractions [Loepere, 92]:

- Task: A unit of resource allocation that encapsulates an address space and port rights. Unlike a Unix process, a task is not a unit of scheduling.
- Thread: A unit of CPU allocation which is lightweight compared to a Unix process. Multiple threads can coexist in a task.
- Port: A communication end-point which is accessible through send and receive capabilities. Ports are useful in the implementation of object-based services accessed through message passing. Sharing of resources such as memory objects, can be accomplished by sharing port rights.
- Message: A typed collection of data objects. Exchanging messages is the primary method of cooperation between multiple tasks.
- Memory object: A unit of memory management recognized by the microkernel. Memory objects encapsulate virtual memory functionality.

Many traditional operating system services such as the communication subsystem and the file system have been moved from the kernel-level to the user-level and implemented as one or more servers. Single-server [Golub, 89] and multi-server [Julin, 91] implementations of operating system services have been constructed on top of the microkernel.

One example of a crucial service implemented at the user-level is the external pager [Young, 89]. It exports virtual memory services that are traditionally hidden inside the kernel. Hence, it enables a user-level operating system to implement its own paging policy

and backing store through an entity called a *memory manager.* It provides three interfaces to achieve these features:

- a mapping interface for a client to map memory objects into its address space;
- a memory object interface used by the microkernel for upcalls to the user-level memory manager; and
- a memory cache interface to allow the user-level memory manager to manipulate the main memory cache.

These three interfaces have proved very useful in extending the virtual memory services. For example, the *NetMemoryServer* is a user-level task which allows sharing of memory objects over the network. NetMemormoryServer is notified by the kernel when there is a page fault and it retrieves the required page from a remote machine, if necessary [Forin, 89].

The IBM Microkernel Services further enhance the flexibility of configurations supported by the Mach microkernel [Golub, 93]. They allow multiple operating systems to coexist on top of the microkernel and include object-oriented frameworks for device drivers and generic services like loading, naming and default paging [Bahrs, 94]. They also further partition the services into multiple servers to allow easy replacement and configuration of the service providers. A simplified conceptual view of a resulting system is shown in Figure 3.1. The main contribution of the IBM Microkernel Services design lies in extending modularization of operating system components and providing object frameworks for the components to interoperate. The design allows mix-and-match combinations of system software components.

The separation of functionality in separate units Mach and its descendants is defined by the distinction between the microkernel and a user-level task. As such, it does not directly provide for separation of client concerns. Incrementality is pegged at the coarse

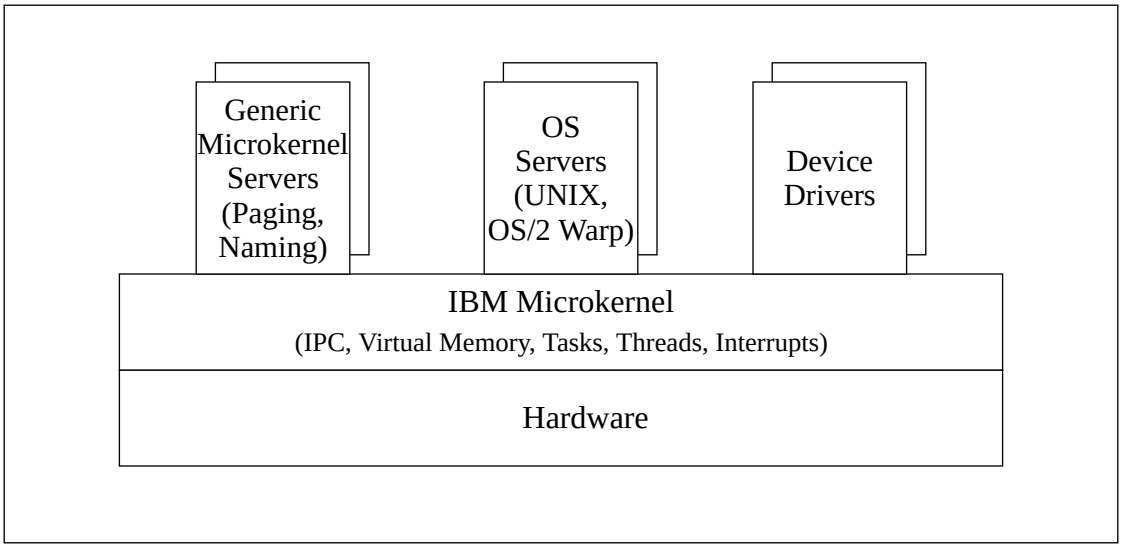

Figure 3.4: Microkernel Approach and Multiple Servers

granularity of a task. Scope control facilities are limited to certain predefined abstractions like memory objects. However, due to the emphasis on IPC and separate protection domains for operating system services, Mach and systems derived from Mach would form an excellent platform for further experimentation in flexibility.

3.3.4 *x*-kernel

x-kernel is an operating system designed at the University of Arizona for implementation of efficient communication protocols [Hutchinson, 91]. The flexibility approach used in x-kernel is derived from and oriented towards communication subsystems. While many operating systems focus on a single protocol like RPC for communication, *x*-kernel focuses on allowing the construction of arbitrary communication protocols. Instead of using a layered structure which is found in many communication subsystems, *x*-kernel relies on three primitive communication objects: *protocols, sessions* and *objects*. It also provides a symmetric call mechanism between the kernel and the user-level; system call for user-level to access kernel modules and upcall in the opposite direction [Peterson, 90].

In addition to an elegant architecture for protocol composition, *x*-kernel has provided substantial performance gains over the normal Unix implementations. It has essentially demonstrated, at least in communication protocols, that performance and flexibility are not opposite but can in fact be mutually reinforcing. It provides excellent incrementality in its communication protocol architecture. However, the same does not hold for file systems and other subsystems. Also, scope control has not been a focus in the *x*-kernel architecture.

3.3.5 Spring

Spring is a distributed operating system recently developed at Sun Microsystems Laboratories. Like Mach, it uses a microkernel structure and object-oriented components with strongly typed interfaces [Hamilton, 93a]. These components or *modules* are treated as substitutable parts and are assigned dynamically to address spaces called *domains*. Modules are composed of objects which may support inter-domain calls. Inter-domain calls use an IPC mechanism called *doors*. The implementation of doors thoroughly optimizes IPC and combines scheduling of the called thread with the transfer of arguments from one domain to another for better performance.

In its virtual memory subsystem, Spring goes beyond Mach by separating memory objects, objects that encapsulate access rights, from pager-objects, objects that support paging operations for fetching the contents of a memory object. This separation has allowed the combination of file system services with virtual memory services in an efficient but flexible way.

An interesting and particularly relevant innovation introduced by Spring is the idea of *subcontracts* for object invocations [Hamilton, 93b]. Since object invocations are fundamental to a Spring implementation, the architecture allows substitution of communication mechanisms on both client and server sides. The substitutable entity is encapsulated

into a subcontract. On the client side, it provides an interface consisting of the following operations:

- marshal to package and transmit an object to another address space;
- unmarshal to unpack an object received from another address space;
- invoke to actually execute an object call after marshalling has been done;
- invoke_preamble to write some subcontract level information into the communication buffer or to adjust the buffer otherwise to influence future marshalling; and
- marshal_copy for copying arguments that are passed with copy semantics.

The server-side interface provides operations that support instantiation of Spring objects, processing of incoming calls and reclaiming a Spring object by revoking the rights to that object held by clients. Thus, subcontracts focus on separating communication mechanisms from the task of implementing Spring objects.

Many of the comments in the last paragraph about Mach also apply to Spring due to its emphasis on the microkernel approach. However, subcontracts provide greater incrementality to object communication than is available in Mach IPC. Scope control is again on a per-object basis and does not address restricting visibility of a change to one of many clients.

3.3.6 Apertos

Apertos is a reflective operating system designed at the Sony Computer Science Lab to provide a flexible mobile computing environment [Yokote, 91]. It uses metaobjects and their hierarchies as building blocks to provide flexibility. A *meta-hierarchy* consists of metaobjects and in turn, the second-level metaobjects for the first-level metaobjects and so on. Each meta-hierarchy provides a customizable virtual machine for applications [Yokote, 92a]. A virtual machine is called a *metaspace* in Apertos. At the most basic level, Apertos implements a tiny microkernel called *MetaCore* which is the terminal metaobject;

i.e. it has no metaspace. The other important abstraction in Apertos is a *reflector*; a reflector is a metaobject which represents metacomputing defined by a group of metaobjects. Like metaobjects, reflectors are also organized in a hierarchy and certain reflectors are provided by the Apertos implementation.

The logical organization of objects and their metaspaces is shown in Figure 3.1 which has been adapted from [Yokote, 92a]. The lightly shaded areas in the figure are the first-level metaspaces while the darkly shaded areas are the next level metaspaces. Objects whose execution is supported by a metaspace are shown astride the boundary of the metaspace while the objects contained inside are metaobjects and constitute the metaspace. Application objects can change the behavior of the operating system by invoking operations supported by reflectors or by migrating from one metaspace to another. Object migration is particularly critical for the heterogeneous mobile computing environment that Apertos has targeted. Hence, mechanisms are provided for checking compatibility between metaspaces. Further, Apertos also provides kernel transactions and support for real-time applications by limiting the MetaCore to operations whose execution time is predictable [Yokote, 92b].

Apertos is a pioneer in applying reflection to operating systems. It separates client-controllable aspects from the rest of the implementation using metahierarchies which also provide incrementality. Scope-control is again effected on a per-object basis and there is no special mechanism for per-client scope control. Like Mach and Spring, it is implemented directly on top of hardware; the approach is revolutionary rather than evolutionary.

3.4 Summary and Remaining Issues

Research in languages like CLOS and 3-KRS has clearly shown the value of reflective architectures for flexibility by allowing clients to participate in implementation deci-

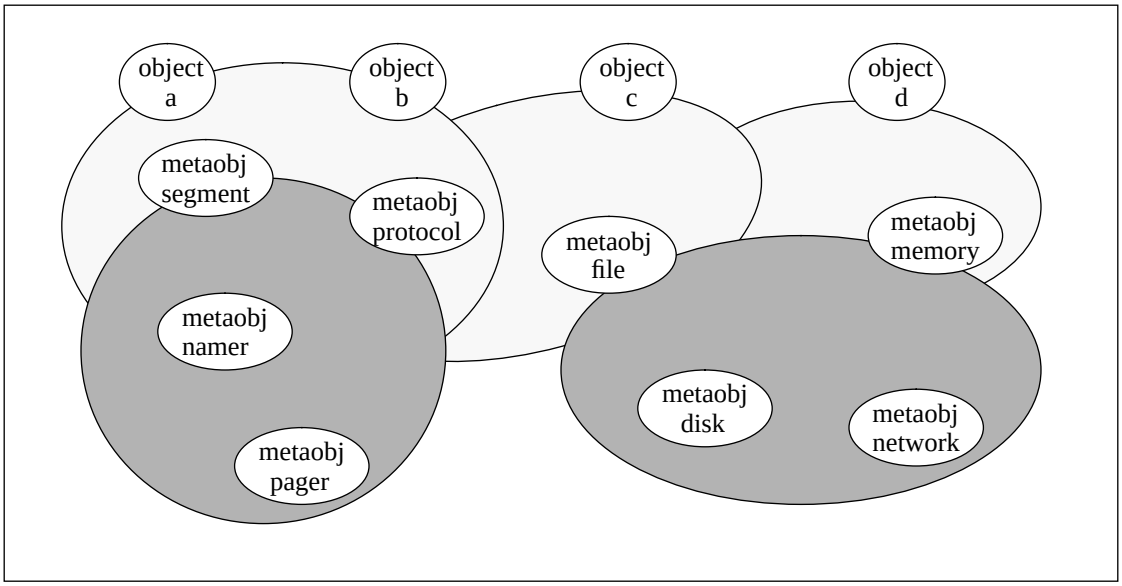

Figure 3.5: Metaobjects and Metaspace Hierarchies in Apertos

sions in a disciplined manner. The architectures have used concise self-representation and causal connection to allow a client to choose a custom language from a broad spectrum. For example, as described in the subsection on CLOS, the order in which classes are searched can be redefined by a client. Both CLOS and 3-KRS are functional languages and their implementations have extensive run-time support from the interpretive environments. On the other hand, the problems encountered by similar efforts in C++, an imperative and compilation-oriented language, illustrate the difficulty posed by the lack of a good self-representation and run-time support. The loss of information during compilation further complicates the task of constructing a self-representation. Also, the performance expectations of programmers using a language like C++ often doom flexible but expensive features. These lessons are critical for the design of our approach to operating systems. The Pi approach relies on the basic reflective model of 3-KRS and dual interfaces in CLOS while providing a few efficient mechanisms for self-representation.

Early operating system efforts indicate the value of policy/mechanism separation and customization. However, unlike Hydra and Alto-OS, we are not designing a kernel

from scratch but are interested in an evolutionary approach for existing operating systems. Also, due to the emphasis on distributed systems and legacy constraints imposed by resources used, the issues have changed substantially since those efforts in the 1970s. Alto-OS in particular, does not address the issue of multiple simultaneous clients and protection; issues that are considered important in this work.

More recent efforts like Mach and Spring have concentrated on partitioning of functionality across protection domains. Hence they have emphasized optimization of inter-domain communication and construction of operating systems at the user-level. While partitioning is valuable and fast communication mechanisms are essential for partitioning, they do not by themselves address the issue of incrementality which is a critical aspect of flexibility. For example, as pointed out in [Kiczales, 93], Mach allows the replacement of a whole pager or a file system at user-level but does not address how a file system or a pager can be modified to suit an application's need. Also, these systems address the issue of scope control only to a limited degree. They do not solve the problem of different client scopes.

On the other hand, our effort is more focused on the semantics of operating system services and the need for incrementality and scope control. In that sense, our approach is orthogonal to that of a system like Mach and hence can in fact be considered the logical step after achieving the partitioning proposed by Mach. We can also benefit from the technologies like fast IPC, produced by these systems. In the same vein, like *x*-kernel, we emphasize the semantics of operating system services but unlike *x*-kernel, address scope control and do not propose a new kernel. We owe much to *x*-kernel for their insights into decomposition of communication subsystem into a set of flexible abstractions.

Apertos has pioneered the idea of reflection is operating systems. Unlike Apertos, we do not provide a metaspace and reflectors for every object and instead focus on key mapping decisions made by operating system designers. We feel that while the meta-hier-

archies can enable extreme flexibility, they do not by themselves capture the behavioral aspect of services provided by operating systems. They may also entail significant overhead in absence of extensive programming environment support which is unavailable in existing operating systems. Also, as mentioned above, we are interested in adding flexibility rather than building a new kernel from scratch. Moreover we believe that scope control problem is important enough to merit explicit facilities. In summary, the choice of an evolutionary approach, emphasis on capturing the decisions of operating system designers and scope control are the main distinguishing characteristics of our approach presented in the next chapter.

4. Pi APPROACH

There are a number of ways to make an implementation open. In this chapter we will start with two of the more obvious options, and then refine them to obtain what we call the *Pi approach*. We will use distributed shared objects as an example to illustrate the key aspects of the approach. This description is followed by a discussion of the programming model consisting of guidelines for implementors and client developers. Finally, we will discuss some of the implementation issues that arise while applying the Pi approach.

There are two obvious approaches to obtain flexible implementations. One based on existing implementations and the other on metaobjects. The former provides a starting point for adding flexibility to well-established and optimized implementations while the latter capitalizes on programming tools that can automate behavioral change at object granularity.

Existing subsystem implementations have a number of unexposed parameters that affect their behavior. An example is the number of preallocated buffers on the receiver side of a communication channel. Preallocating more buffers in a communication subsystem would improve latency by saving the time required to allocate additional buffers. On the other hand, increasing the number of preallocated buffers for receiving large messages could lead to reduced buffer space for senders who may have to be blocked. Another example is the delay between successive flushing of file buffers. While reducing the delay in a file system would increase the crash-resilience by decreasing the duration for which data is volatile, it would reduce the overall throughput of the file system. Thus, it may be

desirable to determine the values of parameters such as these at run-time, according to client-needs.

These and many similar parameters can be identified by carefully examining existing implementations. A client could then select the values for them. In fact, this approach is already used to some extent in `ioctl` calls provided in UNIX [Leffler, 89] and OS/2 [Deitel, 92]. However, consistent with the bad reputation earned by `ioctl` calls, this approach has several limitations. First, the approach is very unstructured; it does not address how clients can participate in crucial, non-parametric decisions that substantially alter the implementation. For example, in a scheduler implementation, the time-slice parameter could be exposed and adjusted but the scheduling policy cannot be changed. Second, the approach is not disciplined; a change in a certain parameter can have effects on other parts of the implementation and other clients. For example, in a scheduler implementation, a long-time slice may be undesirable if there are many processes waiting in the scheduler queue. Hence, we will have to look beyond exposing parameters in existing implementations.

Building a system out of objects augmented by suitable metaobjects is the second solution. Programming tools like SOM [IBM, 93a] and Open C++ [Chiba, 93] provide varying degrees of support to generate metaobjects that can provide run-time control over structure and behavior of an object. Structural control includes obtaining type-information while behavioral control includes altering function dispatch.

In principle, a system composed of objects and their metaobjects is very flexible. In such a system, it is possible to achieve two desirable aspects of disciplined flexibility discussed in Chapter 2; metaobjects allow incremental changes, and, with additional scaffolding, may permit scope-control. The combination of known object-oriented subsystem designs and tool-generated metaobjects provides a bewildering range of control. But therein lies the drawback of this approach. Extensive control over individual implementa-

tion objects is difficult to translate into client participation in choosing options that affect the behavior of the overall subsystem. Conversely, choices that a client would like to make are difficult to translate into metaobject calls throughout the implementation. Also, a uniform use of metaobjects leads to substantial performance penalties while providing flexibility that may not be of use. Hence, we need to look beyond a cookie-cutter approach based on metaobjects.

The Pi approach is designed to combine some of the advantages of the two approaches. Like the legacy implementation-based approach, we will exploit the subsystem-specific semantics and like the metaobject approach, we will pay attention to disciplined use of client-control. Our approach concentrates on designers' decisions and uses the reflective system model discussed in the previous chapter. We will provide run-time representation for entities like an interface, that are normally only a part of the program text and then look for ways to manipulate them. The approach consists of a reflective architecture and a programming model that can be followed by implementors and clients.

4.1 Reflective Architecture

Recall that a reflective system consists of a *self-representation*, a *causal connection* between the representation and the system and facilities for *metacomputation*. Clients perform metacomputations to change the self-representation of the system and, through the causal connection, the system itself. Accordingly we will discuss these three aspects.

4.1.1 Self-representation

Operating systems manage *resources*, like memory and locks, on behalf of applications. They respond to *events* such as interrupts, traps and system calls. They provide *interfaces* for clients and for internal use, and perform certain sequences of steps, which we call *protocols*, involving resources, in response to events. Many of the resources,

events and interfaces are inaccessible to clients. We can use this simplistic description to guide us through the development of a self-representation.

4.1.1.1 Resources

Resources can be encapsulated as *resource objects*. A resource object is an instance of a *resource class* and provides an interface to a resource at appropriate granularity. For example, a memory object encapsulates a portion of main memory possibly consisting of one or more pages, and provides operations for allocation, deallocation, reading and writing [Loepere, 92]. This simple view of a memory object containing three pages is depicted in Figure 4.1. The object as viewed by a client is shown as a rectangular box drawn in dashed lines, with calls in its interface marked by arrows. The supporting implementation, a resource object, is shown as a circle hidden behind the interface. Page boundaries, which are omitted in subsequent figures, are marked by dotted lines.

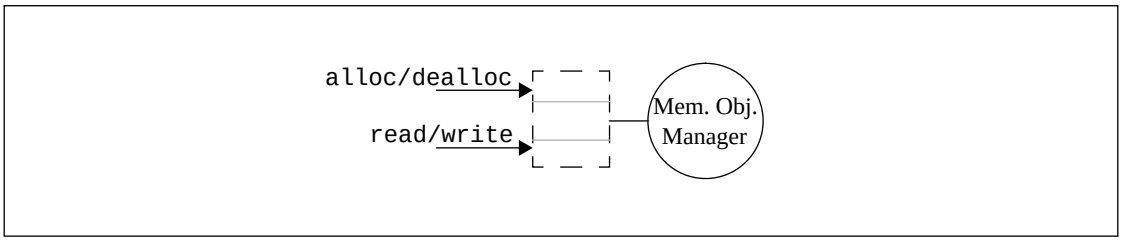

Figure 4.1: Memory Object

Resource objects may encapsulate non-hardware resources as well. A lock for ensuring mutual exclusion is significant not for the underlying storage occupied by a semaphore but for its functionality of avoiding conflicts between readers and writers or multiple writers. Resource objects can be grouped together to provide a composite resource object. For example, a lock can be associated with a memory object to make it available for concurrent usage by multiple processes as shown in Figure 4.2.

The implementation of a resource object may be fairly complex, in which case, it is desirable to reuse the implementation. Consider a persistent object as shown in Figure 4.3.

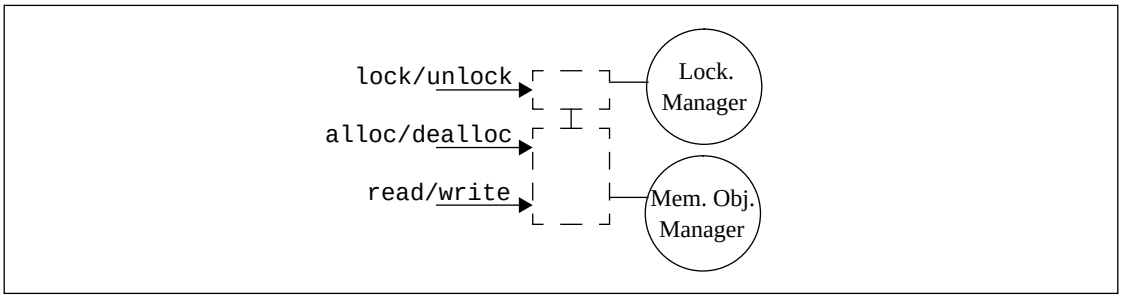

Figure 4.2: Resource Object for Locking

To support the calls like `store()` and `retrieve()`, a persistence manager implementation has to provide a disk storage implementation. Such a persistent object would have a fairly complex implementation to take care of allocation of disk blocks, garbage collection, ensuring high throughput, etc. Hence, it is undesirable to expect a client to provide its own implementation. However, encapsulation ensures that, if necessary, the implementation can be changed. Thus resource objects form the basic building blocks of a subsystem; they will be typically reused and occasionally replaced.

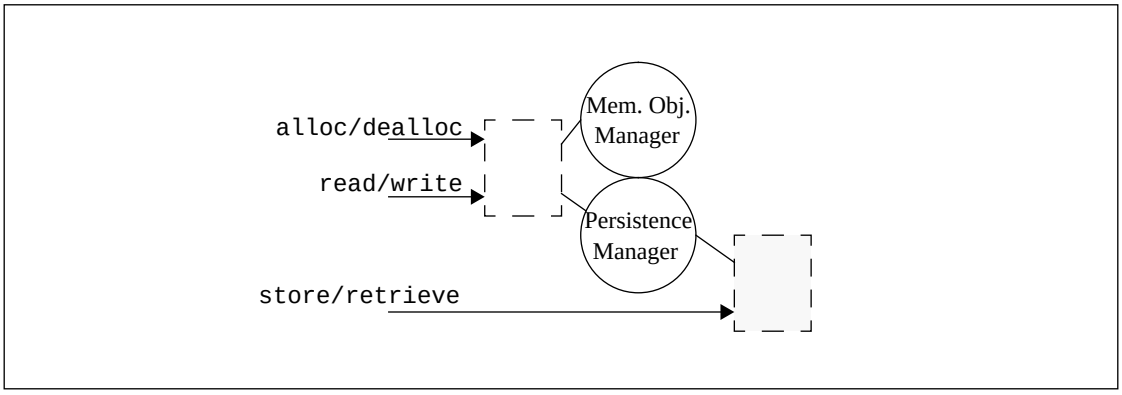

Figure 4.3: Resource Object for Persistence

The implementation of a resource object may span multiple machines. Consider converting the simple memory object of Figure 4.1 into a *distributed shared object* (DSO) as in Figure 4.4 [Kulkarni, 93]. A DSO would be shared by multiple processes, possibly running on different machines. A *coherency controller object* could be used to ensure that processes on different machines which have different physical copies of the DSO have the

same logical copy. For example, a process on machine M1 has a separate physical copy of a DSO that it shares with a process on another machine M2. Yet, the coherency controller must ensure that both the processes see the same data. Such a coherency controller object would then decide which is the latest copy and provide the most up-to-date contents upon request. Typically, the coherency controller would have at least some part of its implementation on each of the participating machines. While the parts on different machines may be independent in terms of failure, installation and administration, they still collaborate to present a single resource object. Thus a self-representation for a subsystem may span multiple machines.

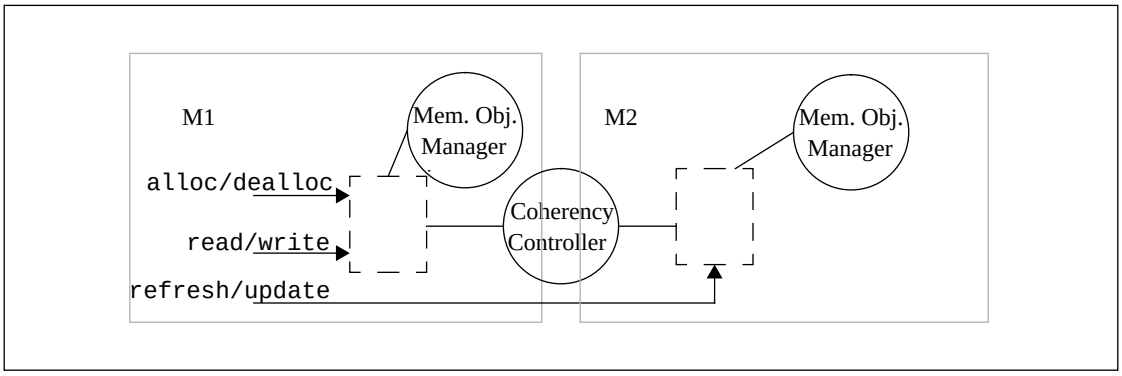

Figure 4.4: Resource Object for Coherency

The memory object embellished with additional functionality of locking, persistence and coherency is shown in Figure 4.5. In this final form, it could be supported by a monolithic subsystem, or it could be a composite of separately controllable resource objects. Such a decomposition will be discussed in the context of DSOs later in this chapter and in the context of file systems in the next chapter. Thus, the identification of the separate constituent resource objects is the first step in the development of a self-representation.

We will use the DSO example to explain the other aspects of a self-representation and the Pi approach. The rich set of functionality shown in Figure 4.5 rests on a number of

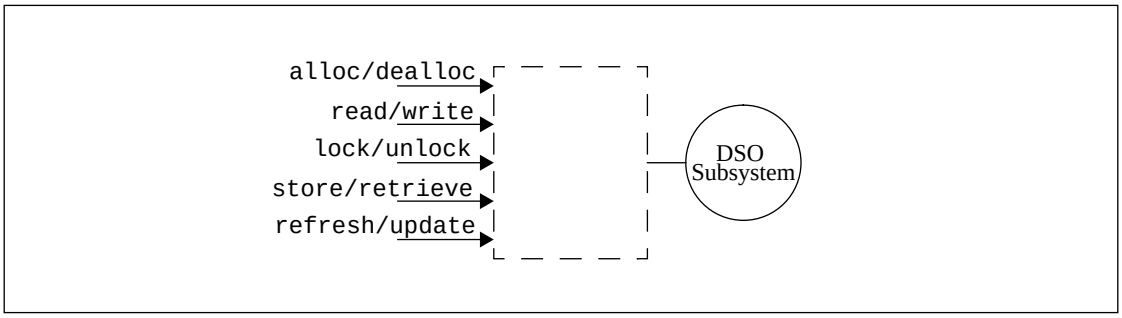

Figure 4.5: DSO Subsystem

design decisions which will be opened for client control as we develop our approach. After resources, events is the next important item in self-representation.

4.1.1.2 Events

The internals of an operating system reveal a second key aspect - events. Operating systems handle a number of events created by hardware and by applications. On the hardware side, they handle interrupts from devices like timers, network cards and disk controllers; on the application side, they handle system calls for memory allocation, signals and inter-process communication.

Applications are often shielded from events. For example, in a typical implementation of distributed shared objects, page fault events are not visible to a client. A page fault is generated by a client process when it accesses a page belonging to a memory object, but the page is not mapped in the process' address space. However, it may be the processing of such events that a client would like to alter. Thus, some critical events may have to be included in the self representation. The occurrence of an event leads to an invocation of an operation supported by a resource object which is documented in the resource object's interface.

4.1.1.3 Interfaces

As mentioned earlier, resource objects have *interfaces*. These interfaces define the *signatures* for operations supported by the resource object. Signatures document the name of the operation, the types for the arguments taken by an operation, and the type for the return value of the operation. A signature for the lock operation supported by a lock object associated with a memory object could be

```
int lock (int process_id, int read_or_write_mode);
```

Thus, the signature tells us that an operation named `lock` takes two integers as arguments and returns an integer.

Interfaces export a set of operations and hide the implementation details. Thus, it is possible to have multiple implementations for a given interface. But all the implementations must *conform* with the interface; otherwise changing the implementation would affect the clients of the interface. Conformity has two aspects: *signature compatibility* and *functionality compatibility.* Signature compatibility can be checked using a tool like a compiler while functionality compatibility must be ensured by implementors. For an implementation of a lock object, taking two integers as arguments for the `lock()` call and returning an integer ensures signature compatibility, while actually ensuring mutual exclusion when a write lock is successfully acquired, provides functionality compatibility. Unfortunately, there is no systematic way of checking functionality compatibility and hence it will remain implicit in our self-representation. Consequently, we will use interfaces in our self-representation with the understanding that implementors will ensure functionality compatibility for each implementation option for a given interface.

The separation of interface from implementation is a good design principle [Meyer, 88]. However, we take it one step further by incorporating the separation in the self-representation. This allows a client to bind to different implementations while using the same interface. The separation is ensured by evolving a resource object into a pair of objects: an

interface object which provides indirection and an *implementation object* which provides the functionality. In the previous discussion of resource objects, we considered an interface to be a purely program-text entity with no run-time existence. But in this next step, we are reifying the interface at run-time as an interface object as shown in Figure 4.6. The figure shows a resource object, depicted as a circle, and its decomposition into an *interface object* is shown as a rectangle and *implementation object*, as a rounded rectangle.

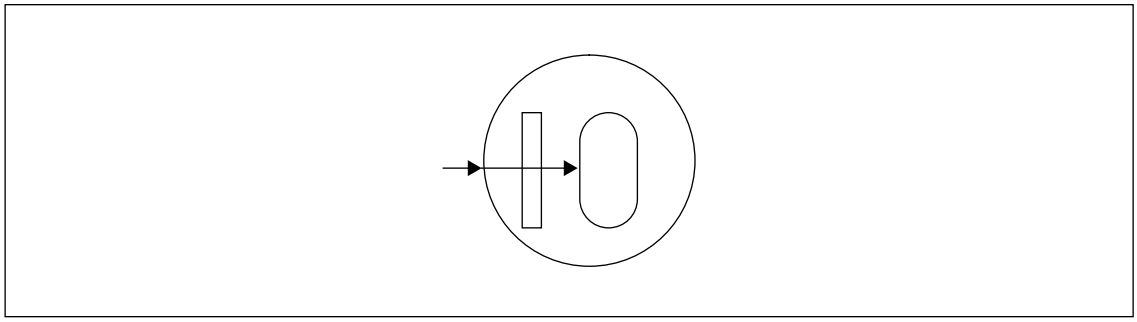

Figure 4.6: Interface and Implementation Objects

A *subsystem* is composed of a number of resource objects. A default implementation object is provided for each resource object. But the implementation object associated with the interface object can be changed at run-time. Clients can use a different implementation object if the default implementation does not meet their needs. Such a change would result in the interface object redirecting service requests to the new implementation object. Thus, an interface object allows a client to control the processing of events that are normally processed by the subsystem-supplied, default implementation object.

Invocations on the interface object can be intercepted and redirected to other objects. In effect, a different implementation can be interposed between the client and the original implementation. This allows the addition of pre- or post-processing code for one or more operations documented in the interface. A number of the possible changes to the implementation structure are shown in Figure 4.7. An invocation can be redirected, another object can be notified of an invocation, or invocations can be intercepted through

interposition. The shaded rounded rectangles depict the implementation objects used for the changes.

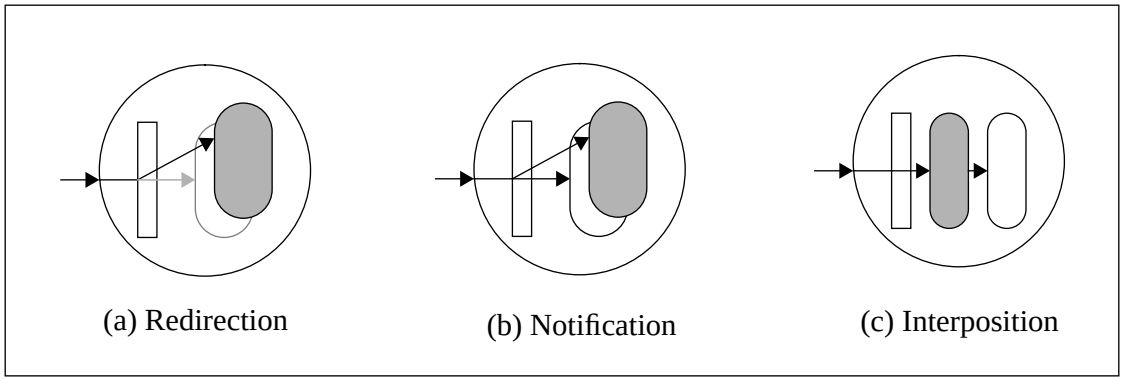

Figure 4.7: Modification of Event Handling through Interface Objects

Consider the distributed shared object example. The coherency controller exports two operations - `refresh()` and `update()`. A call to `refresh()` ensures that the latest version is obtained while a call to `update()` ensures that the current version is declared the latest version. For simplicity, assume that the signatures are:

```
int refresh();
int update();
```

It is possible to have multiple implementations that support this interface. Upon invocation of `update()` on machine M1, an implementation on M1 could either send out invalidation notices declaring all other copies of the DSO stale or it could ship the latest copy to all sharing sites. The choice depends on factors like how often `update` is called and how often the latest copy is required elsewhere. With a flexible implementation, a client could instruct the coherency controller interface object at run-time as to which implementation to use.

Resources, events and interfaces seem to provide a good handle on the self-representation of a subsystem. Resource objects encapsulate reusable code that a client-supplied implementation can selectively use. Exposing events allows a client to change the

processing inside a subsystem and interfaces provide some amount of safety and structure for using alternative implementations. However, we need to take a further step to bring the controllable aspects closer to the design decisions. The notion of *protocols* will help us do that.

4.1.1.4 Protocols

Consider the case of a client that wants to share an object with a process on another machine as shown in Figure 4.8. The client calls `share()` which leads to the following sequence of events. The resource object handling the call invokes `refresh()` on the local coherency controller to ensure that the local copy is up to date; then, `register()` is invoked on the remote DSO subsystem to provide a copy of the object to the remote process. While all three functions are documented in one interface or another, the *sequence* of invocations performed as a result of a `share()` call is not represented anywhere.

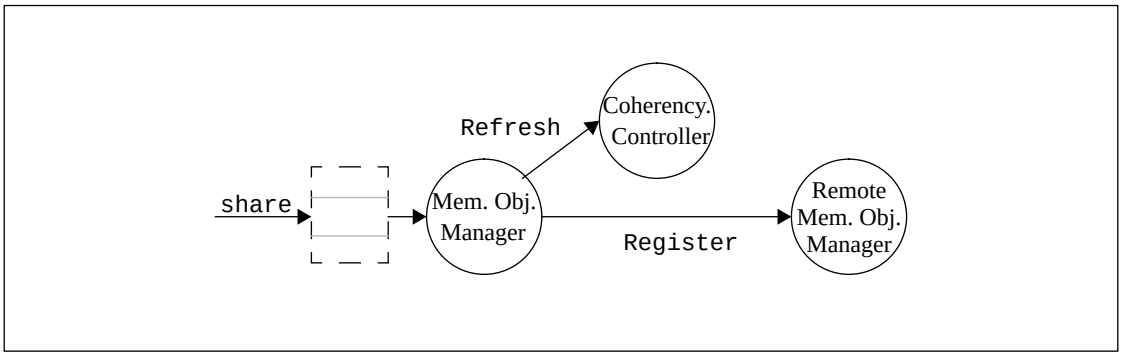

Figure 4.8: Incoming and Outgoing Invocations in a Protocol

An interface provides a static description of the associated implementation. A *protocol*, on the other hand, describes sequences of invocations. A sequence includes invocations on the interface that the protocol supports and the resulting invocations on the interfaces of other protocols and resource objects that the protocol uses[1]. Thus, protocols implicitly capture the states of the interacting objects. In addition to encapsulating a

1. Our use of the term protocol is similar to that in communication protocols [Holzmann, 91]. It is markedly different from that in CLOS or Smalltalk wherein a protocol is similar to our interface.

sequence of invocations, the protocol for sharing also specifies the participating objects in its informal description: the coherency controller on the machine where the operation is invoked and the DSO subsystem on the remote machine. Thus protocols also bridge multiple components through their invocations.

More importantly, protocols represent key decisions made by a designer. In the DSO example, consider the protocol used for providing access to a page in the DSO. When a process on machine M1 accesses a location in the memory object, a page fault is generated if the corresponding page is not available in a valid state on that machine. The page may either be absent or, if present, the data contained may be suspect due to concurrent writes to the corresponding logical page on other machines. In response to the page fault, a DSO implementation may do one or more of the following:

- Request access to the latest copy of the page by sending a message to another machine M2 or by sending a broadcast if the machine owning the latest copy cannot be easily determined.
- Request a shared or exclusive lock depending on the access. An exclusive lock may be requested even for a read access if write accesses are likely to occur in near future.
- Prefetch adjacent pages in anticipation of accesses in the near future or even fetch the entire DSO.

Clearly, this sketch of possible page-fault protocols shows multiple choices that may be important to *some* clients. Yet, all these protocols are providing the same basic service to their clients: handling a page fault; i.e. they support the same interface for clients. This interface is called the *protocol interface*. On the other hand, one can envisage distinct *protocol implementations* behind the same interface corresponding to each of these choices. We will represent different protocols as distinct implementation objects. A client should

be able to select one at run-time and perhaps even provide a new one if none of the existing ones is suitable. *Contracts* are used in the Pi approach to provide this facility.

4.1.1.5 Contracts

The concept of contracts is a key to the Pi approach. Contracts combine separation of interface from implementation with protocol implementations. As mentioned before, interface objects allow a client to change the implementation and a protocol implementation is what a client would like to be able to select or change. Thus, a contract object is an evolution of a resource object; not only does it have a well-defined interface but that interface is reified as an interface object. We will now look at the classes that define a contract and the instances of those classes.

A contract has a name and supports an interface. Its run-time instances are called *contract objects*. A contract object consists of an interface object, called a co*ntract interface object* or CIO, and an implementation object called a *contract protocol object* or CPO. Since the interface and implementation objects in a contract are separate, the CPO can be changed. The leftmost circle in Figure 4.9 shows a contract object C1, with two possible implementations. The CIO is shown as an 'L' shaped figure and the two alternate CPOs as rounded rectangles. However, a contract object can have only one CPO at a time and hence one of the CPOs is shown in dotted lines.

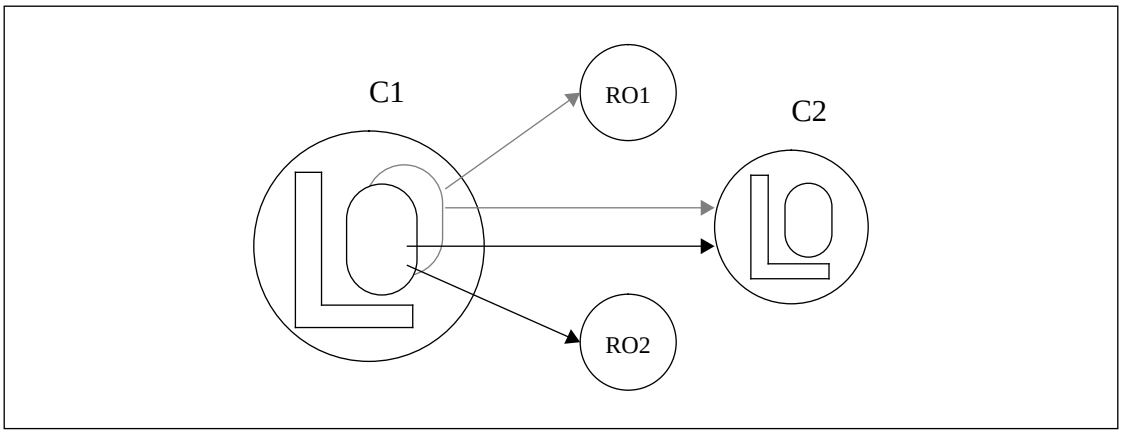

Figure 4.9: Contracts and Protocol Implementations

A CPO implements a protocol using resource objects and other contract objects. As shown in the figure, the protocols implemented by two CPOs may be different. The CPO drawn with a dotted line calls resource object RO1, while the other CPO calls resource object RO2. But multiple protocol implementations may reuse the same object. The two CPOs in the figure, both invoke functions supported by the same contract object C2.

Now let us turn our attention from objects to classes and the interfaces of the classes. It is convenient to visualize a CIO as an instance of a *contract interface class* CI, and CPO as an instance of a *contract protocol class* CP. While classes and inheritance hierarchies are not essential for the self-representation, they are convenient for implementation and useful for exposition.

A class has an interface and all the instances of the class support that interface. There are two interesting parts of the interface of a CI class corresponding to the basic idea of dual interfaces. The first part, called the functionality interface, lists calls supported by a CPO and the second part called the control interface, allows a client to change the implementation object, i.e. the CPO. A CI gets the two parts of its interface from two distinct classes as shown in Figure 4.11. The protocol interface class provides the functionality interface while the metacontract class provides the control interface. The interface of the CI class is a composition of the two interfaces. The figure shows a protocol interface class with two calls for the page fault protocol discussed in the preceding section: `handle_page_fault()` and `pin_page()`. The two calls associated with the metacontract class, `query_impl()` and `set_impl()` allow a client to check which implementation is used for the CPO and to switch implementations.

Now let us take the figure one step further to include the classes for CPOs. We saw in Figure 4.9 that two CPOs may implement two different protocols. The different protocol implementations can be easily effected if we have different contract protocol (CP) classes. Figure 4.11 shows two CP classes. Both inherit from the same base class - Proto-

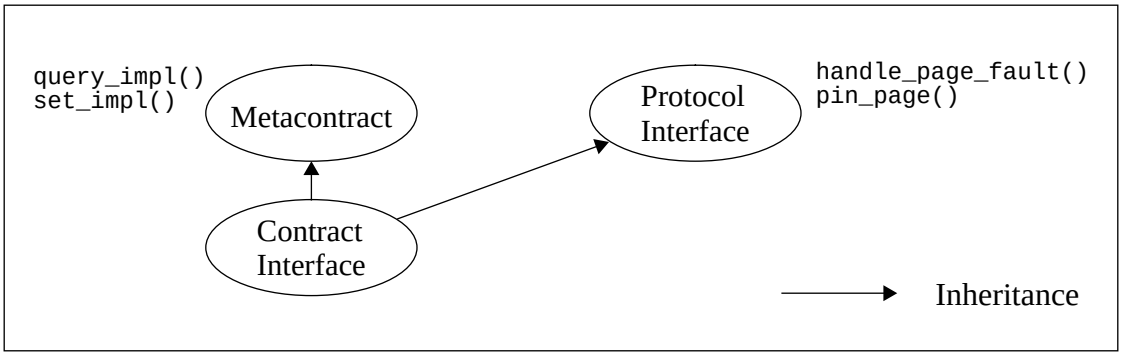

Figure 4.10: Composition of a Contract Interface Class

col Interface. Hence, both support the same protocol interface. But since their implementations are different, their instances in Figure 4.9 implement different protocols.

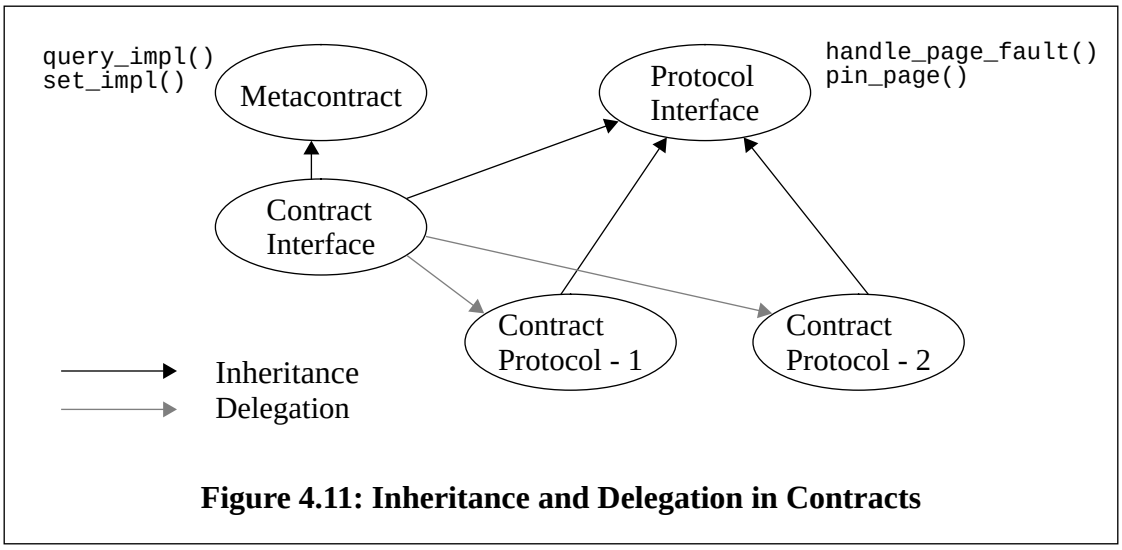

Figure 4.11: Inheritance and Delegation in Contracts

There are two kinds of relationships portrayed in Figure 4.11: inheritance and delegation [Lieberman, 86]. The inheritance relationship is between classes and delegation between objects or instance of those classes. A Contract Interface (CI) class inherits from a Protocol Interface class while an instance of a CI class delegates to an instance of a Contract Protocol class. Inheritance relationships cannot be manipulated at run-time and hence a CIO, an instance of a CI class, delegates to a CPO, an instance of a CP class. The delegation can be readily changed at run-time. The change is effected in response to a `set_impl()` request from a client. The top two classes in the hierarchy are abstract

classes; i.e. they are not directly used for creating instance objects. The derived classes on the other hand are concrete classes that have corresponding instances at run-time.

With this information about the contract class hierarchy in mind, we can revisit the issue of identifying a contract. If a client wants to change the protocol implementation used by the subsystem to service client requests, there has to be a way of identifying the contract and the implementations. *Contract descriptors* allow us to do that. A contract descriptor consists of two parts: a name for the contract, which identifies a CI class, and an id for the protocol implementation which identifies a CP class. For example, a descriptor for a page fault contract could identify the page fault CI class and a page fault CP class that prefetches all the pages in a DSO. The descriptor can be used by a client to refer to a contract and a specific implementation for the protocol in that contract. A client can supply a contract descriptor to a subsystem and have the subsystem create the correct CIO and CPO. The descriptor may also be used to replace a CPO at run-time. In effect, a client selects the implementation it wants.

Implementation selection has to be constrained to enforce some discipline. A CI class maintains compatibility information for deciding which options for CP are permissible and when. In certain cases and in certain states during execution, an existing CPO may not be replaced by another CPO, if the replacement endangers the integrity of the overall contract. For example, in the DSO object, a page-fault contract may not be replaced once DSO implementations on different machines have agreed to a certain coherency protocol for a DSO.

4.1.1.6 Subsystem as a Composition

So far, we have looked at the components of a subsystem. We defined contracts in order to allow a client to participate in some of the decisions made by an implementor. Now we can look at how a subsystem is composed of resource objects, some of which are contract objects.

A run-time view of a subsystem composed of multiple contract objects is shown in Figure 4.12. The two interfaces to the subsystem are discussed in Section 4.1.3. The overall functionality of the subsystem is provided by a set of collaborating contract objects as shown in the figure. As before, CIOs are shown as 'L' shaped structures to emphasize dual interfaces, CPOs as rounded rectangles and resource objects as circles. The arrows indicate potential operation invocations at run-time. Since the figure shows a snapshot, only one CPO is shown for each contract object. As shown in the figure, a CPO interacts with another CPO of a different contract through a CIO for that contract.

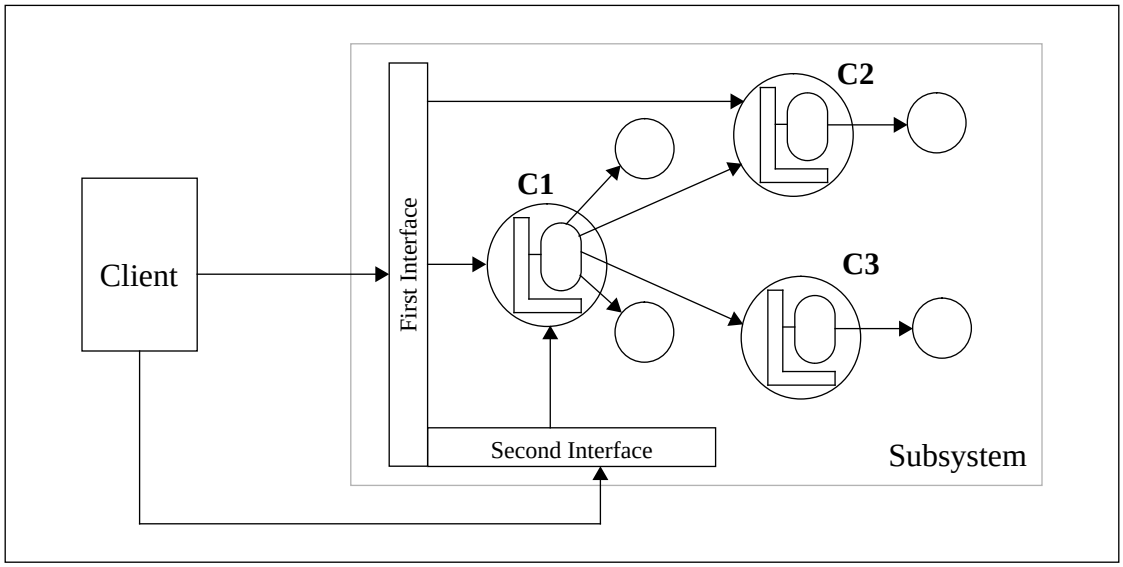

Figure 4.12: Schematic of a Subsystem with Contract and Resource Objects

Together, these contract[2] and resource objects define the subsystem as a graph of objects. Alternatively, the subsystem can be viewed as a *framework* of contracts and resource objects. Each contract has a well-defined interface, and a well-defined position in the framework. Multiple

2. The term contract was introduced in [Helm, 90] for specifying behavioral compositions of interacting objects. Helm et. al. use contracts written in a specification language to describe behavioral dependencies beyond inheritance. The contracts in our approach are similar to Helm's contracts in that they specify interaction between objects that are not necessarily related through an inheritance hierarchy. But we provide explicit separation between interface and implementation and reify the interface as CIO to make contracts suitable for self-representation. There is another difference, perhaps the most important one, that has to do with the way contracts provide causal connection when coupled with scopes. Although there are several differences in goals and structure, Pi contracts were inspired by the discussion in [Helm, 90].

implementations of CPOs that conform to the corresponding CIO interfaces, fit into the framework. The framework of contract objects has an interesting dynamic property. A caller contract object needs to either find an existing callee contract object or construct a new one. In Figure 4.12, **C1** must find or construct **C2** and **C3**. Construction of a new contract object opens several opportunities for flexibility. A contract object with a specific CPO can be constructed, or even the graph of the subsystem can be altered. For example, it is conceivable that some CPO for contract object **C1** may not need a contract object in place of **C3**. The CPO may itself provide the functionality of two contract objects, **C1** and **C3**. Thus, the decomposition of a subsystem into contracts is not rigid and makes several implementation options available.

Since a contract descriptor uniquely identifies both a contract interface and a protocol implementation, a list of contract descriptors describes a part or whole of the subsystem. A list of all default contract descriptors would describe the entire subsystem while a shorter list containing descriptors for contracts that have been changed could describe the deviation from default implementation. This completes the discussion of self-representation in the Pi approach.

4.1.2 Causal connection

Recall that a self-representation is useful for reflection only if it determines the behavior of the subsystem it represents. Also, as discussed in Chapter 2, it is extremely important to ensure that any changes made to a subsystem are visible within a restricted scope. We will use scope control to ensure these two requirements.

4.1.2.1 Scopes

A simple example of scope control is when a client process wants a change to affect only itself. The client process is the *scope* for that change. Likewise a client may want to change the memory allocator for all large DSOs. Then the set of all large DSOs is the

scope for that change. Notice that there are two kinds of scopes: those related to clients and those related to the units in the subsystem. A *client scope* is a set of entities that want to see a change while a *unit scope* is a set of entities in the subsystem that are affected by the change. These scopes, viewed as sets, are abstract entities. The Pi approach provides a run-time representation for them within a subsystem as objects. Henceforth, we will refer to the run-time representation of the abstract entity as scope.

The Pi approach relies on two properties of scopes shown in Figure 4.13. First, scopes are instances of *scope types*; and second, scope types are required to be *nested*. *Process* is a scope type while a specific client process (e.g. *Process 1*) is an instance of that scope type. Likewise, *DSO* is a scope type while a specific DSO is an instance of that scope type. The relationship between scopes and scope types is similar to that between an object and its class. The former (scopes and objects) implement a running system while the latter (scope types and classes) are a part of the design of a flexible system.

Scope types are required to be nested. For example, an application consists of one or more processes. Hence, the scope type *Process* is nested inside the scope type *Application*. In the Pi approach, this nesting property defines the rules for visibility of a change made by a client. If a client requests that a change be made visible within the scope of an application, then the subsystem makes sure that the change is seen by all processes belonging to that application. We will see a different example of scope types and scopes in the next chapter.

A client can apply a change to a scope. In order to do so, scopes must be identifiable. Like contracts, scopes have descriptors for identification.

In the previous section we saw that a change is made by a client through the mechanism of contracts. A client can effect a change by requesting a different contract protocol object (CPO) from the default one. Now if the scope of that change is to be controlled, we

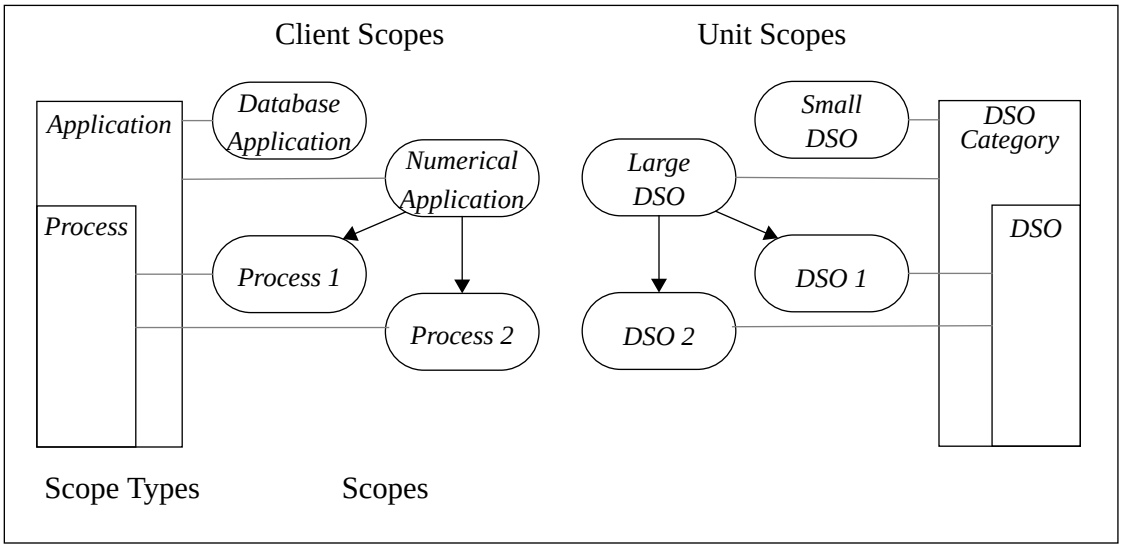

Figure 4.13: Scope Types and Scopes

must associate contracts with scopes. We use contract descriptors lists which were discussed in the last section, for this purpose. A scope has a list of zero or more contract descriptors associated with it.

Figure 4.14 shows the scopes from the previous figure with the associated contract lists. The contract lists are shown as slotted rectangles with varying number of slots. The figure also shows the outermost unit scope named *The Subsystem* which represents the entire subsystem. It has a contract descriptor list marked *Default CD List* which contains descriptors for default implementations for all the contracts in the subsystem. Typically, functionality for an abstraction like a DSO is implemented by multiple contracts, and the default implementations for all the necessary contracts are described by such a list. Other scopes have possibly shorter or even empty lists that only contain descriptors for modified contract implementations. These lists come into existence because a client requests that a non-default CPO be used for a certain contract and that the change be applied to a specific scope. Both the contract and the scope are identified through their descriptors. The contract descriptors are used in scope-based dispatch which enforces scope-control.

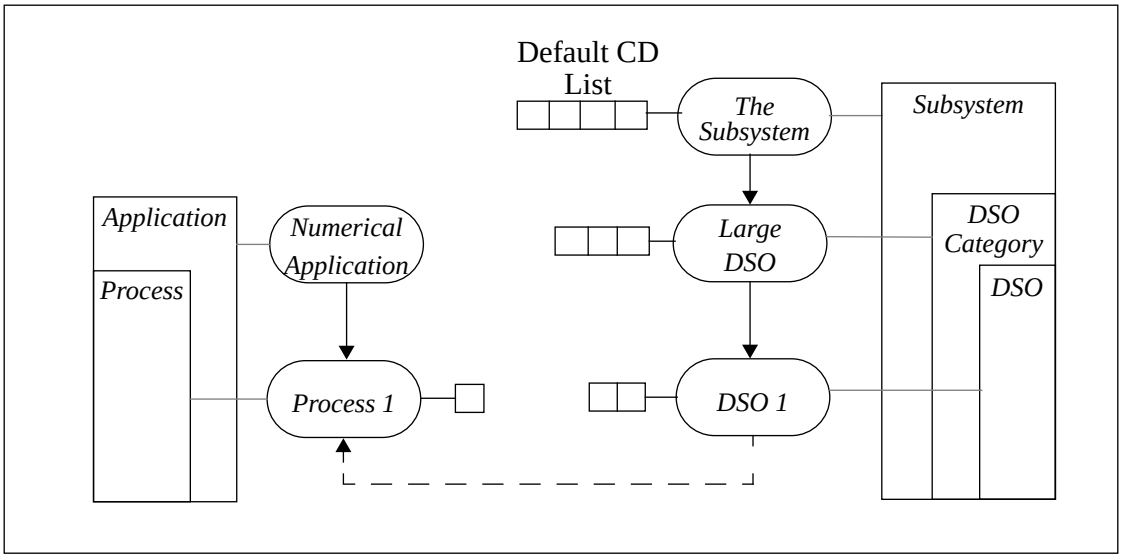

Figure 4.14: Scopes and Contract Descriptor Lists

4.1.2.2 Scope-Based Dispatch

Let us consider a DSO constructed for a specific client. The relevant scopes are *DSO 1* representing the DSO and *Process 1* representing the client process. Since *DSO 1* is created at the request of *Process 1*, an arrow is shown in Figure 4.14 from *DSO 1* to *Process 1*. Our objective is to design an algorithm that will construct contract objects that are appropriate for a given scope. In this case, we have to construct the contract objects to implement the functionality for *DSO 1* that satisfy the selections made by *Process 1*. Since the selections for each scope are defined by the associated contract descriptor list, that is our starting point.

The selection of contract objects relies on nesting of scope types. If a contract with a certain name is not present in the associated list, a contract of the same name from the list associated with the containing scope is used. For example, if we cannot find a descriptor for the page fault contract in *DSO 1* scope's list, *Large DSO* scope's list is searched for an entry corresponding to the page fault contract. There are two hierarchies, one for client scopes and another for unit scopes. The unit scope hierarchy can always provide a descriptor for any contract used in the subsystem. If none of the inner scopes have a descriptor for

a certain contract (say page fault), the outermost scope, *The Subsystem*, will provide the descriptor corresponding to the default implementation of that contract. On the other hand, the client hierarchy may not provide a descriptor. In the figure, if no client has requested any change in the page-fault contract, the search through the client scope hierarchy will not produce a descriptor for page fault contract.

After searching the two scope hierarchies, a CIO is constructed with the descriptor(s) as argument(s). If there is no client side descriptor, the CIO constructs a CPO corresponding to the unit-side contract descriptor. If two descriptors are found, then the CIO decides which of the two descriptors should be used for constructing the CPO. If possible, the client-side descriptor is used. However, the client-side descriptor may not describe a valid choice for a CPO. For example, if a client asks for a CPO for the page-fault contract that relies on broadcasts in an interconnection that does not support broadcasts, the client-request cannot be granted. In this case, the unit-side descriptor prevails and a corresponding CPO is constructed. Internally, the CIO maintains information to decide whether a descriptor indicates a valid choice or not.

Thus, the specific set of steps followed for constructing a contract object for *DSO 1* is as follows:

1. The contract lists associated with unit scopes are searched starting from the innermost scope (*DSO 1*) to the outermost scope (*The Subsystem*), until a contract descriptor with the required contract name is found. A contract descriptor is always found by the end of the search.
2. A contract descriptor is obtained, if one exists, for the client scope (*Process 1*) using a similar search in the client hierarchy.
3. The descriptor(s) found in the first two steps are used to construct a contract object. First a CIO is constructed, which in turn, creates a CPO, thus completing the construction of the contract object.

We call this process of deciding and constructing the contract object to be used, *scope-based dispatch*. It is used every time a contract object is created.

Scope-based dispatch implements *multimethods* for object construction [Kiczales, 91]. While in an object oriented language like C++, the function dispatched is determined exclusively by the target of an invocation, multimethods allow the selection of a function based on additional information. Here, multimethods are implemented for selecting an appropriate CPO constructor. The additional information is provided by client scopes and is determined in step 2 above.

Further, even after a contract object has been constructed, the client represented by *Process 1* can request a change in the CPO. The CIO handles such requests and changes the CPO if appropriate as discussed in the previous section.

Scope-based dispatch ensures the two critical constraints mentioned at the beginning of this subsection. It maintains a causal link by guaranteeing that the contract associated with a scope in the self-representation gets used at run-time and it ensures that changes are confined within a scope by using the scope-containment relationship for deciding which contract to use. Of course, it allows clients to change the self-representation through metacomputation.

4.1.3 Metacomputation

Metacomputation is accomplished by invoking operations provided by the second interface of the subsystem. There are three basic categories of operations in the second interface. The first includes operations to set and query the contract implementations for a scope; the second allows construction and destruction of scopes and the third, which is optional, allows addition of new contract implementations.

A client can select a contract implementation and indicate a scope to which the new implementation should be restricted. If the client request is accepted, the contract descriptor list associated with the scope is modified accordingly. Further, if a contract object with a matching name has already been created, `set_impl()` is invoked on the corresponding

CIO. Subsequent computations can then use the new CPO. A CIO may refuse to honor a client request for a change if the current state of computations makes it unsafe to change the CPO. A special and potentially interesting case is where a CIO does not allow any change in the implementation once the contract object has been created. This limits metacomputation to a safer subset of changes, viz. those effected through scope-based dispatch.

New units, and therefore scopes representing those units, may be created and deleted using explicit calls in the second interface. For example, the unit *Large DSO* shown in Figure 4.13, and the corresponding scope could be created in a DSO subsystem. When a scope is created, its outer scope and an optional contract list are specified. If a contract list is not specified, the contracts will default to those used by the outer scope. Scope creation and deletion may also result due to calls in the first interface. This is particularly relevant for the innermost scopes like communication endpoints and files that are created in response to well known calls like `socket` and `creat` in the first interfaces of their respective subsystems.

Finally, a subsystem may permit clients to request that new contract implementations be added to the subsystem. This is a somewhat more complicated part of metacomputations due to the fact that a client has to be trusted not only to provide a working implementation but also for specifying to what extent it is compatible with other implementations.

In order to perform these metacomputations, a client may be required to have certain privileges. For example, a subsystem in UNIX may restrict metacomputation privileges to processes with superuser privileges. The privilege policy and its enforcement are not covered by the Pi approach and are left to the subsystem developer’s discretion.

In the Pi approach, the operations required for the second interface are deliberately kept simple. However, a subsystem designer can add more operations to the second interface that use the basic facilities for changing contract implementations. The choice of implementations available to a client can be packaged as *policies* that can be selected procedurally or obtained declaratively. In the case of distributed shared objects, an implementation that propagates changes to a data object on demand can be packaged as an update-on-demand policy implementation while another implementation that propagates changes as soon as they are available can be packaged as immediate-update policy implementation. A client of such a DSO subsystem need not know anything about the implementations except the desired policy.

Thus, there are three ways of changing the implementation. For example, consider the memory management policy for a DSO. A client can request a specific page-fault contract implementation by calling `set_impl()`, select a subsystem-supported coherency policy, or simply declare that a DSO is going to be used in a particular way, say heavily write shared. The Pi approach requires support for the first method and leaves it to subsystem implementor to decide whether the other two methods should be supported.

Our approach avoids an infinite, or even a multi-level, tower of metaobjects that is possible in a reflective system [Wand, 88]. Certain aspects of a subsystem are considered immutable. In particular, scope types and contracts are defined at design-time and are built-in. A client may not create a new scope type, although new instances of a given scope type may be created. Similarly, the approach does not recognize new types of contracts although new implementations for existing contracts may be provided.

4.1.4 Summary

So far we have discussed three components of a reflective architecture designed using the Pi approach: self-representation consisting of resource objects and contracts,

causal connection implemented through scope-based dispatch and metacomputation effected through a second interface. Now let us look at how the approach can be applied.

4.2 Application of the Approach

Effective application of the approach to a subsystem requires developers of the subsystem and its clients to cooperate to ensure the integrity of the subsystem and the appropriate granularity for changes. The next two sections delineate the roles of subsystem and client developers.

4.2.1 Subsystem Developers' View

A subsystem constructed according to the Pi approach consists of a framework and a set of implementations that fit into the framework. The following steps summarize the application of the Pi approach to a subsystem design and implementation:

- Define the two interfaces of the subsystem: The standard subsystem functionality should be provided through the first interface. The second interface must support `set_impl()` and `query_impl()` calls. It may support additional calls for declarative control and/or loading new contract implementations (Section 4.1.3).
- Define scopes and their implementations: Hierarchies of client and unit scopes should be defined to restrict visibility of changes (Section 4.1.2.1). Efficient scope-based dispatch implementation should be supplied based on the two hierarchies.
- Define a framework of contracts: The overall functionality of the subsystem should be decomposed into contracts. For each contract, a protocol interface and then a contract interface should be defined (Figure 4.11). In this key step, the extent of client control permitted by the subsystem is determined. Once the protocol interfaces are defined, a client cannot change them at run-time. Also, all CPO implementations are required to conform to the contract interfaces.
- Define contract protocol classes: At least a default CP class must be defined for

each contract. CPOs are created at run-time as instances of CP classes (Figure 4.11). Each CP class makes a specific implementation decision related to the contract.

- Define resource classes: Functionality used by multiple CP classes is provided as resource classes (Section 4.1.1.1). Resource objects which are instances of resource classes, help CPOs in providing the functionality of their contracts.

The decision about whether a certain functionality should be provided through a resource object or a contract object can be quite tricky. In principle, any resource object can be replaced by a contract object with the same protocol interface. But if it is not desirable to allow a client to change a functionality as an independent entity, it is better to present that functionality as a resource object. Similarly, a crucial design decision that affects clients, is best included in a CPO rather than a resource object. A rule of thumb is to implement *mechanisms* as resource objects and *policies* as CPOs. But the distinction between mechanisms and policies has to be made on a per-subsystem basis.

4.2.2 Client Developers' View

There are several levels at which clients can use the subsystem. At the simplest level, a client would use only the first interface. At this level, a flexible subsystem would look no different from its conventional counterpart. If the subsystem provides for declarative reflection, a client can disclose its usage pattern and let the subsystem use it as a hint for making changes internally.

A more sophisticated client would use the second interface to set the contract implementations. Such a client would need to know which of the available CPOs is most appropriate for its needs. The simplest way to select a protocol is to do so at the time of creating a new unit scope such as a socket, or a file. Typically, selecting a non-default implementation for the smallest scope that does not interfere with other clients or other unit scopes is

a safe strategy. However, with additional knowledge of the CPOs, a client can attempt to change the protocol associated with other scopes.

Finally, the most sophisticated type of clients could provide additional protocol implementations. This level of participation is fairly demanding. The client would need to know the contract interface, the complete semantics of the protocol that it wants to implement, and the resource objects that it is going to use to simplify the task. Also, since this level of flexibility is optional in the Pi approach, not all subsystems designed with the approach would support it. In fact, at this level, the client is almost on par with a subsystem implementor.

4.3 Implementation Issues

There are several implementation issues to be considered in a flexible subsystem and in particular in the Pi approach. These issues fall into two main categories: language issues and operating system related issues. As mentioned in Chapter 2, we will consider C and C++ as the implementation languages and AIX, an implementation of UNIX, as the operating system of choice. The discussion in this section will illustrate implementation strategies that can be used to apply the Pi approach and provide some information about potential overheads. Further discussion of this topic based on the file system prototype appears in Chapter 6.

4.3.1 Language Issues

We explained contracts in Section 4.1.1.5, using the basic ideas of inheritance, delegation and abstract classes. C++ provides adequate support for implementing these features. The most important C++ feature for this purpose is *virtual functions* [Stroustrup, 91]. Virtual functions allow specialization of functions declared in a *base class* so that a C++ compiler automatically invokes the correct version of the function based on the type of the object supporting the function. Hence, a base class pointer can be used to point to an

object of a derived class while ensuring that the version of the function dispatched corresponds to the object that the pointer points to, rather than the type of the pointer variable. In the case of contracts, this facility is used in conjunction with delegation to substitute implementations.

A skeleton of C++-like pseudo-code is shown in Figure 4.15 for the class hierarchy of Figure 4.11. The top-level classes are abstract classes; i.e. they are not used for creating instances. The virtual functions defined in these classes are implemented in the derived classes. Due to the virtual function mechanism, the operations invoked by the delegator on the delegatee use the appropriate implementations provided by contract protocols.

A C implementation would have to use explicit function pointers in the absence of a virtual function mechanism. Although less elegant and less type-safe, a vector of function pointers can be used to invoke the functions supported by the protocol interface. The contract protocol can be changed by replacing the function pointer vector.

In both C++ and C, the separation between the interface and the implementation of a contract is achieved using an indirection. This indirection has an associated cost of an extra pointer dereferencing operation. Secondary performance penalties could also be caused by possible cache misses. Additional costs are incurred due to the search required for scope-based dispatch when a contract is instantiated. But typically the number of levels of scope types (three in Figure 4.14) is likely to be small and hence the search overhead is likely to be low. Typically, these costs would be very small in comparison to the costs of common codepaths in subsystem implementations. We will revisit this issue in Chapter 6 in the context of a file system implementation.

4.3.2 Operating System Issues

Two important operating system issues need to be considered while using the Pi approach: addition of code within a protection domain at run-time and cross-domain inter-

```
// An abstract class documenting operations in the second interface
// of the contract interface class
class Metacontract
{
   public:
     // just provide signatures; no implementation
     virtual int set_impl(Impl_Id) = 0;
     virtual int query_impl(Impl_Id*) = 0;
};

// An abstract class for the interface of the page fault contract
class Page_Fault_Protocol
{
   public:
     int handle_page_fault(Page_Id) = 0;
     int pin_page(Page_Id) = 0;
};

// This is a contract interface class that provides implementation
// for metacontract operations and delegates the other operations
class Page_Fault_Contract_Interface
{
   public:
     // constructors and destructors not shown
     int set_impl(Impl_Id);          // impl. in a separate class
     int query_impl(Impl_Id*);       // impl. in a separate class
     // delegate the next two operations
     int handle_page_fault(Page_Id id)
        {delegatee->handle_page_fault(id);};
     int pin_page(Page_Id id) {delegatee->pin_page(id);};
   private:
     // the delegatee is a contract protocol object
     Page_Fault_Protocol* delegatee;
     // more info. about compatibility, current state etc.
};

// One implementation of the protocol
class Page_Fault_Protocol_1
{
   public:
     // constructors and destructors not shown
     // the following functions have concrete impl. in this class
     int handle_page_fault(Page_Id);
     int pin_page(Page_Id);
   private:
     // implementation-specific information
};

// Another implementation of the same protocol
class Page_Fault_Protocol_2
{
   public:
     // constructors and destructors not shown
     int handle_page_fault(Page_Id);
     int pin_page(Page_Id);
   private:
     // information specific to this implementation
};
```

Figure 4.15: Pseudo-code for a Page-Fault Contract

action. In the following sections, we will discuss dynamic loading, use of multiple protection domains and the overhead of cross-domain interaction. These issues are discussed in the context of the platform used for the work described in this document: AIX 3.2.5 running on 25MHz RS/6000 model 530 machines based on the Power architecture and connected by 10 Mbps ethernet.

4.3.2.1 Dynamic Loading

Like most flavors of UNIX, AIX provides facilities for dynamic loading of object modules [Hook, 93a] [Hook, 93b]. The calls `load()` and `loadbind()` can be used respectively to load an object module and resolve the symbols imported by the module. Subsystem implementations that provide the facility to add a protocol implementation to a running system would have to use this facility. A protocol implementation loaded at runtime exports the functions supported by the corresponding contract interface. It imports the functions supported by resource objects used by the protocol implementation and other contract interfaces.

4.3.2.2 Protection Domains

Protection domains provide a number of benefits, inevitably occur in subsystem implementations and can be effectively accommodated in the Pi approach. Let us look at these three issues one by one.

Subsystem implementations can greatly benefit from the use of multiple protection domains. Partial implementations provided by clients can be used much more effectively if the rest of the subsystem is insulated from them through the protection domains implemented by the underlying operating system. Protection domains act as firewalls and improve the robustness of the subsystem while allowing the flexibility that Pi approach seeks to provide. Also, the development of such implementations is greatly simplified by

the fact that they can be tested and debugged separately [IBM, 94]. In AIX, a protection domain is either the kernel or a user-level process with its own address space.

Apart from implementation convenience, multiple protection domains are important because they are unavoidable. In a distributed system consisting of autonomous machine, implementations often have to cross machine and administrative boundaries to provide services like communication and file systems. The DSO coherency controller shown in Figure 4.4 is one example of such an implementation. There are three basic types of cross-domain interactions as shown in Figure 4.16: system call, upcall and RPC. RPC involves processes that may or may not be on the same machine.

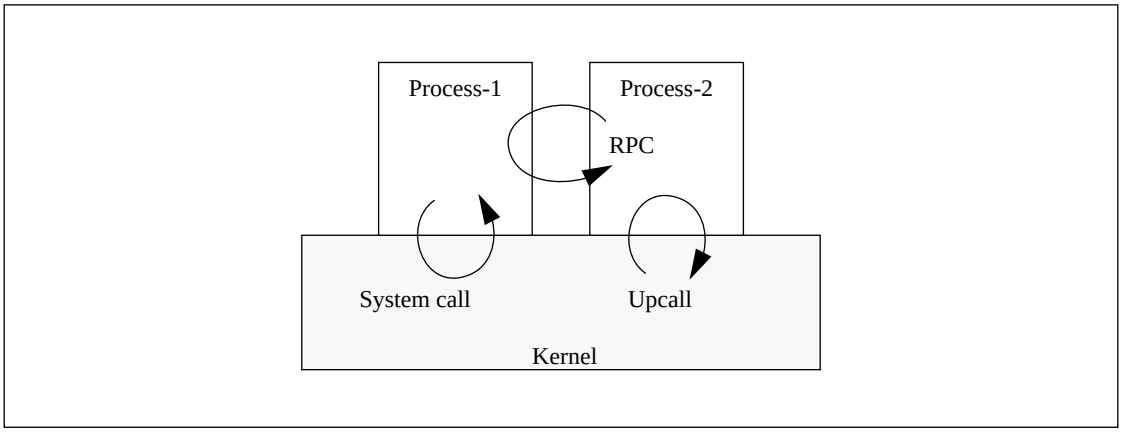

Figure 4.16: Calls Across Protection Domains

Cross-domain interactions can be accomplished through *proxies* [Shapiro, 86]. As shown in Figure 4.17, a proxy **P** in address space **A1** represents a resource object **O** in another address space **A2**. It forwards all calls to **O** and collects results on behalf of the caller. Because of **P**, all invocations made on **O** look local to a caller in **A1**. Proxies can be easily introduced in contracts as shown in the figure. A CPO can actually be a pair of a proxy and a remote protocol object as shown in Figure 4.17. Proxies are shown as hatched shapes in the figure.

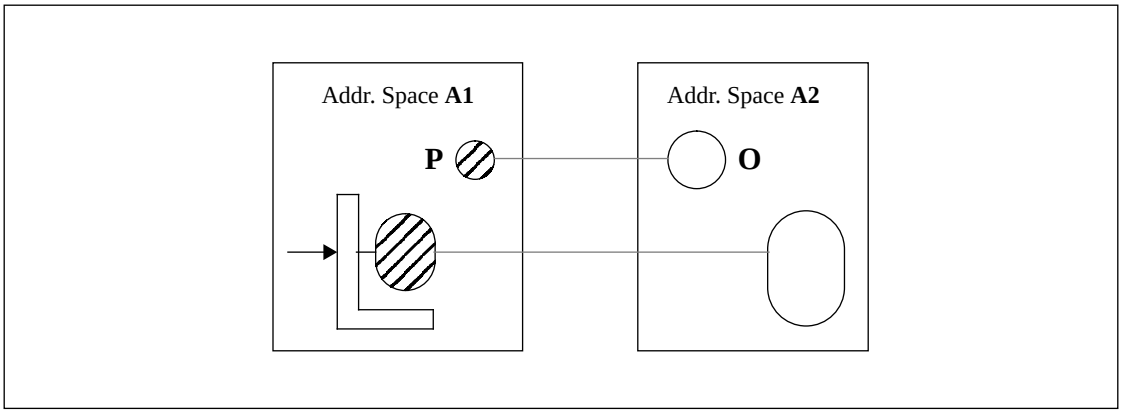

Figure 4.17: Use of Proxies in Contracts

4.3.2.3 Cross-Domain Interaction Costs

The opportunities discussed above entail a substantial cost. We want to measure the costs, explore alternate inter-domain communication mechanisms and understand their implications for the application of the Pi approach to a subsystem implementation.

Interaction between protection domains is several orders of magnitude slower than a function call within a protection domain. Table 4.1 shows the average cost of cross-domain calls in AIX. The tests measured the time for one roundtrip performed using null-RPC from one domain to another and then back. Each measurement consisted of 10 roundtrips (1000 in case of function call) and the mean was computed over a set of 10 measurements. The first row shows the time for a minimal function call within an address space as a reference value. The second row shows the time taken by a null system call; i.e. the cost of crossing the user-kernel boundary both ways. System V IPC numbers are shown in the third and sixth rows. System V IPC, which is restricted to a single machine, can be used for IPC as well as an upcall from the kernel domain to a user-domain. The fourth and fifth rows show the costs when sockets are used with UDP as the underlying protocol.

Table 4.1

Costs of Cross-Domain Calls

Nature of Interaction	Time in μsec.
Function call	0.25
System call	22
RPC using System V messages - single machine	870
RPC using UDP sockets - single machine	1600
RPC using UDP sockets - two machines over ethernet	3100
Upcall using System V messages	700
Pi custom upcall	180

Since the cost of an upcall using System V IPC is substantial, a custom upcall package was developed for collaboration between kernel-level and user-level components. It uses kernel-level facilities for minimizing the delays in scheduling the process containing the called code and shared memory for parameter passing. The last row in the table shows the cost of the Pi custom upcall. The details of the implementation and extensive performance measurement can be found in [Banerji, 94a].

The numbers for cross-domain interaction costs are strongly dependent on the hardware platform and the operating system. However, the ratios of the costs of cross-domain interaction to a function call are comparable across multiple platforms [Anderson, 91]. Hence, the relative costs are more important than the absolute numbers and should be taken into account while implementing CPOs. Also, developing more efficient implementation of cross-domain interaction primitives as in the custom upcall, is only one of the two critical steps.

The second step is reducing the interaction between modules in different protection domains. When the costs of calls in Table 4.1 are compared with the costs of a function call within a protection domain, it is clear that they need to be factored into the design, not

just the implementation. Hence, contract interfaces need to be designed assuming that some of the protocol implementations may be remote. An example of such a need will be discussed in Section 5.3.4.9 in the context of name resolution in file systems.

Finally, there is another important implication of the high cross-domain interaction costs. So far, we have assumed the synchronous RPC model for object invocations; the caller waits for the callee to finish. But there exists an opportunity to reduce this wait and exploit concurrency using a separate process (or a thread where available). In our experiments with the Pi approach, we have developed a tool called a *work queue* for exploiting concurrency [Banerji, 94a]. A work queue consists of two queues; one for synchronous calls and another for asynchronous. Both queues are shared by two or more processes running on behalf of callers and callees. The caller decides which queue to use for a particular invocation *on a per-invocation basis*. The asynchronous queue can be used to improve the throughput of the subsystem while the synchronous queue can be used for urgent requests. This choice between the two styles of invocation can be easily incorporated into CPOs to obtain different performance characteristics. While the work queue tool can also be used in conventional design, contract objects offer a better granularity for its use.

4.4 Overall Perspective

The Pi approach uses the reflective model to provide flexibility in key places in a subsystem. It encapsulates design and implementation decisions into protocol objects and makes them substitutable through contract objects. It can effectively utilize multiple protection domains through proxies which can be used to build contract object and resource object implementations.

At this point, it is worthwhile to visit the five points mentioned at the end of Chapter 2. We will revisit these points in Chapter 6, after the discussion of the file system prototype.

- Separation of client-control: Contracts provide the separation; framework of contracts and protocol interfaces are defined by a designer while CPO implementations are selected by a client.
- Procedural control: The second interface allows clients to select an implementation.
- Incremental modification: Contract objects provide incrementality; CPO for one or more contracts can be changed while using default CPOs for the other contracts.
- Scope control: Scopes can be used to define the visibility of a change. Scope-based dispatch uses client-input in selecting the implementation dispatched.
- Low overhead: Simple indirection used in contract objects and small number of scopes in a subsystem aid in limiting the overhead.

The Pi approach makes a conscious tradeoff in the amount and nature of flexibility it affords. Unlike conventional design approaches, it allows clients to change CPOs which often define the policies used by the subsystem. In a clear departure from the metahierarchy approaches like Apertos [Yokote, 92a] that make every aspect of an implementation subject to client control, flexibility is restricted, thereby ensuring subsystem integrity. While making this tradeoff, special attention was also paid to minimizing the overhead imposed on clients that do not use flexibility features.

We have discussed the run-time changes made possible by the Pi approach in detail. But the approach is also conducive for the more conservative, and simpler, compile-time flexibility. In fact, wherever possible, designers should resort to compile time flexibility features and use the run-time features only where there is no compile-time option. Contracts, which can include a set of precompiled CPO implementations and allow run-time selection from among them is one example of such a strategy. Emphasis on compile-time flexibility improves the overall performance by minimizing run-time indirections, and enhances safety by catching type errors at compile time. It also makes a subsystem more reliable since multiple implementation options can be tested for interoperability during

development. Also, entities in the Pi approach that are not subject to run-time metacomputation can be easily subclassed for compile-time flexibility. For example, a new scope type which provides persistence for contract lists can be created by inheriting from an existing scope type. Thus, the approach does not forego any of the advantages of compile time flexibility associated with framework based object oriented approaches [Bahrs, 92] [Campbell, 91].

Structures similar to Pi scopes have been in use in operating systems for purposes other than changing the implementation at run-time. For example the `u` and `proc` structures in UNIX [Leffler, 89] are used for access control, accounting and garbage collection. However, the Pi approach takes scopes one step further by using them for self-representation and controlling the implementations used. Delegation, which is used in contracts, is a key mechanism in prototype-based languages [Agha, 87], [Ungar, 87] where it is often used as a substitute for inheritance. Coplien [Coplien, 92] uses it in the envelope-letter idiom in C++, where an envelope object forwards calls to a letter object.

In order to illustrate the use of the Pi approach, we will discuss a flexible architecture for a file system. We will show how the Pi file system architecture overcomes some of the drawbacks of an architecture that is currently used widely and how it benefits from the basic structures like contracts and scopes proposed in the Pi approach. We will then describe and implementation and evaluate its effectiveness. Later we will expand on the discussion in this subsection by drawing from the file system development experience and addressing concerns about flexibility raised in the operating system community.

5. Pi FILE SYSTEM ARCHITECTURE

A file system is an important subsystem of an operating system. It provides access to a variety of data sources, and serves clients with different usage patterns and varying views of the data. Hence it is an ideal candidate for verifying and refining the Pi approach. In this chapter, we will describe a way of making file systems flexible using the *Pi File System (PFS) Architecture* which utilizes the Pi approach. But to motivate the development of PFS, we will first discuss what we would like a flexible file system to be able to do, and what a currently popular file system architecture provides.

5.1 Desirable Features

The following file system features have become desirable due to the increasing importance of accessing data resources spread over a wide network, and the variety of access methods required.

- Different APIs for accessing data of different types.
- Support for multiple, possibly autonomous namespaces.
- Control over storage and retrieval of data.
- Tailorable support for caching.
- Support for multiple I/O protocols to obtain data from heterogeneous repositories.

Files have different models and corresponding APIs. Different operating systems use different, sometimes multiple file models. UNIX uses only the byte-stream model; it assumes that a file is a stream of bytes without any additional structure like records and fields. Hence, UNIX provides facilities for creating and closing a stream (`open` and

`close`), reading from and writing to a stream (`read` and `write`) and changing the offset within the stream (`lseek`). Non-byte-sized records of fixed or variable lengths constitute another model. This model is very important for a number of business applications and are directly supported by non-UNIX systems such as the Distributed Data Management facility in IBM's SAA [Demers, 88], and VMS [Shah, 93]. Atomic objects is yet another file model; data units in ARCADE [Delaney, 89] and files accessed through FTP [Postel, 85] in their entirety are two examples. Atomic objects support calls to get and put data but no calls for opening, closing or changing offsets. More complicated models such as hypertext objects [Halasz, 94] or data units coupled with data unit links [Delaney, 89] provide sophisticated facilities for traversing a graph of data objects. Currently, many such models are supported by layering functionality on top of the byte-stream model. This precludes clients from conveying model-specific information to the file system implementation. The implementation simply assumes a byte-stream mode and foregoes model-specific optimizations.

In a distributed system, especially in one spread over a wide area network, accessible data resources are in multiple and autonomous physical and administrative domains. Hence, the *namespaces* through which the data resources are accessible are also separate. Also, due to the large number of data resources available, namespaces that are independent of the data resources have emerged. Archie [Emtage, 92] and X.500 [Hunt, 93] are two such name services. Finally, users often like to customize their namespaces, while applications expect to access resources through names of their choice. The need for custom namespaces has been acknowledged in two recent systems: Plan 9 [Pike, 93], a successor of UNIX which provides per-process namespaces, and Prospero [Neuman, 92], a naming system which allows users to customize namespaces. The key consequence of the need for separate and possibly customizable namespaces is that treating naming and data access together, as in the UNIX file system, is no longer a good proposition.

Like naming, *data access* should also be customizable. Compression and encryption of data are two examples of functionality that might be desirable while storing and retrieving data. For example, an encryption facility has been added to NFS [Blaze, 93] by changing the NFS source code. However, consistent with our goal of providing client control, the addition of such services should not require the source code for the entire file system or knowledge of its internals.

Caching is an important implementation aspect in distributed file systems. Much effort has been put into designing caching algorithms for file systems and several effective algorithms have been implemented. However, the choice of an algorithm is dictated by the file system implementation and clients cannot change the architectural decision. For example, typical NFS implementations cache files for a very brief period, for about 30 seconds. Since this decision about caching is built into an implementation, clients cannot improve the caching performance even when the file access patterns allow caching for a much longer duration. Therefore, a caching interface and perhaps, multiple built-in caching protocols should be provided in a file system architecture. Thus, we agree with Carl Hauser's plea for explicit caching support [Hauser, 92].

Finally, support for *multiple protocols* is very desirable. Already, resources retrieved on the popular World-Wide Web are accessed using multiple protocols [Berners-Lee, 93]. Even outside the Web, NFS and AFS protocols are used for accessing files in small and large workstations clusters in cohesive administrative domains; FTP and HTTP are used for transferring files between administratively independent domains while IMAP and POP are used for email. Administrative independence of domains and different underlying resources make a strong case for supporting multiple protocols, at least in the architecture of a file system. Different implementations can then support different sets of protocols.

So far, we have focussed on the part of the file system that runs on the same machine as the client of the file system. Even in the development of PFS, we will restrict our atten-

tion to this part and mention file-server issues only as and when relevant to client-side design. The design of a full-scale distributed file system with servers and a new protocol between client-side and server-side file system is beyond the scope of this work. More importantly, we want to use existing servers and ascertain the degree to which they can be accommodated in our architecture.

Before discussing how these requirements can be addressed in a flexible file system architecture, let us look at a current file system architecture, the *vnode architecture*. We have chosen the vnode architecture because it is used in popular file systems and is supported in most if not all flavors of UNIX. Also it is the architecture used in AIX, which is our platform for experimentation. We will describe the basic vnode architecture and the modifications and adaptations of that architecture in some of the file systems. We will use this description to demonstrate the need for an improved architecture, to simplify the discussion of PFS and to serve as a basis for comparison.

5.2 Vnode Architecture

Traditionally, the file system has been an integrated part of the UNIX kernel [Bach, 86]. However, the vnode architecture was introduced in SunOS [Kleiman, 86] to support multiple kinds of file systems[1], and especially remote file systems. The original UNIX architecture used an inode to represent a file inside the kernel. The vnode architecture introduced a new abstraction called a virtual node or *vnode*, for the file system independent part of the inode. It also split the file system functionality into two parts; a generic part sometimes called the Logical File System or *LFS*, and a non-generic part called a Virtual File System or *VFS*. A new VFS can be added to the kernel and multiple VFSs can be simultaneously used. However, the kernel contains exactly one LFS which is typically

1. In UNIX, the term ‘file system’ is used for the entire subsystem as well as its components. We will follow the same convention but we will explicitly distinguish between the two, where the meaning is not clear from the context.

considered immutable and unsubstitutable. The two parts, LFS and VFS interact through a thin abstraction layer called the *vnode layer.*

The vnode layer acts as a switch that multiplexes different file systems as shown in Figure 5.1. Three different file systems commonly used in the AIX kernel are shown as examples of VFSs; CDRFS is the file system for a CD-ROM; JFS is the Journaled File System, commonly used file system on the local disk, and NFS, a distributed file system.

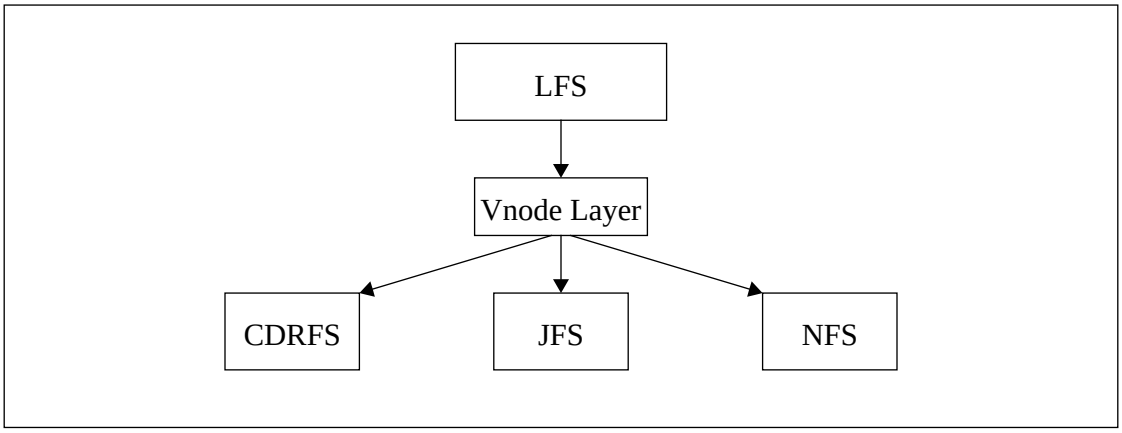

Figure 5.1: Vnode Layer as a File System Switch

The vnode layer abstracts the differences between multiple virtual file systems to form one logical view for the LFS. The LFS on the other hand, interfaces with the rest of the kernel and translates the UNIX file system API into operations supported by the vnode layer. It also manages the combined namespace of the component file systems. The file system namespaces are glued together through the `mount()` mechanism supported by the LFS. Given a pathname, the LFS traverses the system-wide directory tree, one step at a time, to make sure that the file operations requested by a client are serviced by the appropriate file system. Pathname traversal is discussed further in Section 5.3.4.9.

The vnode layer supports two kinds of operations: *vfs operations* that are performed on a VFS and *vnode operations* that are performed on a file object. A new file system that supports these operations can be added to the AIX kernel through the kernel extension

facility [IBM, 92]. VFS operations provide support for creating, mounting and unmounting file systems. Vnode operations provide support for using files, directories and links.

A list of important vnode operations used in AIX is shown in Table 5.1 along with a brief description of what the operations do. The set of operations varies across different flavors of UNIX but the variations are minor and not very relevant to our discussion. Hence we will assume that these operations represent those in a generic vnode architecture. The operations, adapted from the include file `<sys/vnode.h>` in AIX, are divided into two sets in the table. The first set of operations manipulate file names, directories, links and access control lists while the second set deals with the file represented by the vnode. Although vnode implementations do not show such a differentiation between operations, we have intentionally partitioned the operations to facilitate the description of the Pi file system architecture in the following sections.

The vnode architecture is used in a number of distributed file systems. Commercial file systems like NFS [Sandberg, 84] and AFS [Howard, 88] and research file systems like Ficus [Guy, 90] and Coda [Steere, 92] use the vnode architecture. Likewise, it is used for supporting different devices like disk and CD ROM. Hence, it is a good reference architecture for comparison.

5.2.1 Extension of the Vnode Architecture

The primary extension proposed for the vnode architecture is *stacking*. Stacking of vnodes was first proposed in [Rosenthal, 90]. The basic idea is to use a stack of vnode objects instead of a single vnode. Each vnode has the same interface but provides different functionality. New functionality can then be cascaded on to existing one by pushing a vnode on the stack for the file system. The result is a classic layer-based system at the vnode level. The multi-layer structure does indeed provide greater flexibility. For example, a compression layer could be added to a vnode stack. Such a layer would modify the

TABLE 5.1

VNODE OPERATIONS

`vn_access`	Checks if a user has permissions to access a file
`vn_create`	Creates a new file
`vn_getacl`	Gets the access control list
`vn_link`	Creates a new hard link
`vn_lookup`	Gets a vnode corresponding to a name component
`vn_mkdir`	Makes a new directory
`vn_readdir`	Reads a directory in standard format
`vn_readlink`	Reads the contents of a symbolic link
`vn_remove`	Removes a file
`vn_rename`	Rename a file
`vn_revoke`	Revokes access
`vn_rmdir`	Removes a directory
`vn_setacl`	Sets access control information
`vn_symlink`	Creates a symbolic link
`vn_close`	Closes a file
`vn_fclear`	Clears portions of a file by setting the contents to zero
`vn_fid`	Provides a file identifier
`vn_fsync`	Flushes memory-resident data to disk or other storage media
`vn_ftrunc`	Truncates a file
`vn_getattr`	Gets the attributes of a file
`vn_ioctl`	Device-specific operation
`vn_lockctl`	Controls locks associated with files
`vn_map`	Maps a file to a memory segment
`vn_open`	Opens a file
`vn_rdwr`	Reads from or writes to a file
`vn_select`	Polls a vnode to check if it is ready for I/O
`vn_setattr`	Sets attributes for a file
`vn_strategy`	Reads or writes blocks
`vn_unmap`	Unmaps a file from a memory segment

`vn_rdwr()` operation (Table 5.1) so that the client always sees uncompressed data while the underlying vnode always sees compressed data. Provision for adding a vnode layer in a different protection domain has also been implemented in several systems [Heidemann, 94]. However, as mentioned in Chapter 2, a layering technique like stacking cannot undo what is done in a lower layer; in our case, the single-layer vnode architecture. The name resolution problem in vnode architecture which is discussed in detail in Section 5.3.4.9, is one example of a limitation that cannot be undone by layering.

5.2.2 Limitations of the Vnode Architecture

Vnode architecture is a good file system switch. But with reference to the desirable features discussed in Section 5.1, it suffers from the following shortcomings:

- LFS only supports the byte-stream oriented UNIX file system API
- There is no clear separation of namespace management from data management
- LFS, the irreplaceable and unmodifiable layer of the vnode architecture performs a substantial amount of namespace management, thus reducing the freedom for a virtual file system.
- There is no explicit support for caching

The byte stream API, while fairly powerful, does not express a rich enough set of operations. For example, it does not allow an atomic file to be fetched in one step; rather a file must be first opened and then read. This shortcoming of the API hinders the implementation of the first desirable feature discussed in Section 5.1.

Lack of separation between naming and data access functions is an impediment to customization of namespaces. In the vnode architecture, a file is a named data object where the name and the data are managed by the same entity. Hence, an entity providing name services for files is also expected to support operations for reading and writing files.

LFS manages a machine-wide namespace on behalf of multiple file systems. It keeps track of all the mount points in the file system name-tree; i.e. directories where two namespaces are connected to each other. Since the LFS cannot be changed, the semantics of mounting is severely limited. The LFS traverses a pathname internally while disclosing only a part of the name at a time to a VFS. Hence, it also preempts opportunities for optimization in component file systems. We will discuss this issue in detail in Section 5.3.4.9.

Finally, caching algorithms are beyond the scope of the vnode architecture. Component file systems like NFS and AFS use their own caching algorithms. But the fact that vnode architecture does not recognize caching as an important file system activity cuts off the component file systems from most user input regarding caching. An exception is the `fsync()` call which instructs a component file system to flush buffers and is invoked by LFS when a client invokes the corresponding system call. But in a distributed system, there is more to caching than a single call for flushing cache. We will revisit this problem in Sections 5.3.5 and 5.3.6.

The vnode architecture also suffers from a more generic shortcoming. It assumes that the granularity of a change is a VFS; i.e. to obtain a different behavior, an entire VFS has to be replaced. Thus, there is no support for changing operations dynamically or for a specific client. Of course, this level of flexibility is beyond the vnode architecture's design goals and hence, it is not supported. But since we are starting from a different set of desirable features and the explicit goal of flexibility, we will address these shortcomings in the following description of the PFS architecture.

5.3 PFS Architecture

The Pi file system architecture is an application of the Pi approach discussed in Chapter 4 to file systems. Hence we will discuss the scopes, contracts and interfaces that constitute the architecture. We will refer to the vnode architecture as a source for evolution

and as a basis for comparison. As mentioned in Section 5.1, we will focus on the client-side part of the file system.

5.3.1 Scopes

There are two kinds of scope types in PFS: client-based and unit-based. Based on the entities used for accounting and access control in UNIX, the three client-based scope types are *Group*, *User* and *Process*. On the other hand, there are five unit scope types: *Subsystem, FileSystemType, FileSystem, File* and *OpenFile*. A process always belongs to a user and a user always belongs to a group, the primary group in UNIX. Likewise, a file must belong to a file system which in turn must belong to a file system type. Thus, these scope types form two containment hierarchies as shown in Figure 5.2.

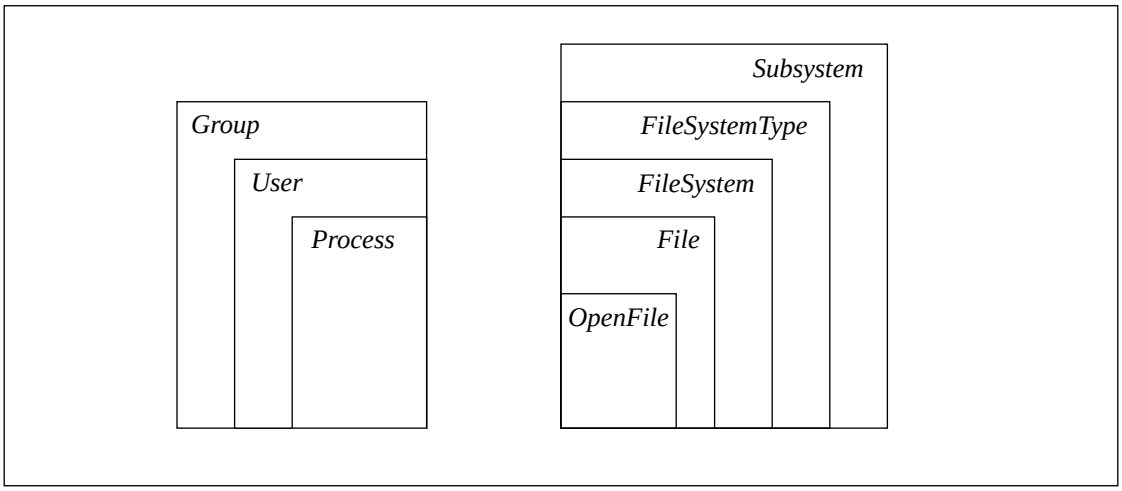

Figure 5.2: Scopes in the PFS Architecture

Recall from Section 4.1.2.1 that each scope type can have multiple instances. For example, a PFS implementation can have NFS and FTP-FS as two instances of *FileSystemType*, multiple file systems corresponding to FTP_FS, and hundreds of files in each file system. In general, there may be multiple groups, users, processes, file system types, file systems, files and open files. However, there is only one instance of the outermost scope

type in a subsystem. These instances or scopes are used to maintain information like selections of contracts, pertaining to the corresponding entity like a process or a file system.

Scopes are linked together according to the containment hierarchy in Figure 5.2. The parent of a scope can be efficiently determined in order to expedite scope-based dispatch. Conversely, for the purpose of garbage collection, all the children of a scope can be efficiently traced. It is crucial to avoid searches in the scope hierarchy since search-oriented solutions would not scale well for a large number of scopes and would lead to unacceptable run-time overhead.

Having discussed scopes, we can now turn our attention to contracts. To design contracts, we will first apportion the overall functionality to six components. Then we will discuss the role of each component and the contract supported by that component.

5.3.2 Functionality Decomposition

Functionality decomposition is the first step in designing contracts so that a client can control smaller parts and specific functions of the file system. In the vnode architecture, there are two components: LFS and VFS; in the PFS architecture, there are six components as shown in Figure 5.3. These components are closely related to the desirable features discussed in Section 5.1. The figure shows a client-side file system based on the PFS architecture enclosed in a dotted box. The PFS file system allows a client to access data from different sources, one of which is shown in the figure. A data source may be a device like a disk or a remote file server. The arrows inside the dotted box show caller-callee relationship between the components. According to the direction of the arrows, we will refer to components as *upstream* or *downstream* with respect to each other.

A PFS implementation has multiple instances of each of the six components, just as a UNIX file system has multiple VFSs. However, for brevity we have shown only one instance of each functional component in the figure. A PFS implementation uses scopes to

manage multiple instances of components. For example, an instance of the name caching component may be used only for a specific file system.

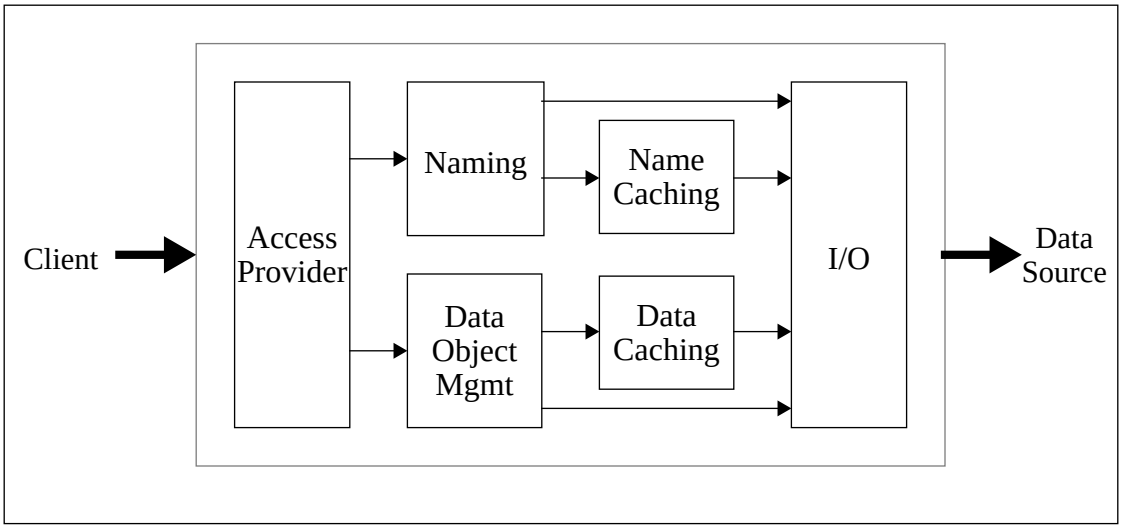

Figure 5.3: Functionality Decomposition in PFS

As a rough analogy, the access provider performs many of the functions of the LFS in the vnode architecture. However, most of the naming-related functions of the LFS are taken over by the naming component. The five components other than the access provider, perform the functions of a VFS in the vnode architecture. The components fit together in a framework, which provides the glue code and miscellaneous services.

In accordance with the Pi approach, there is a contract associated with each of the components. Recall from the previous chapter that a contract object consists of an interface object (CIO) and a protocol object (CPO). The protocol object in a contract can use certain resource objects and other contracts. Hence, in the following sections, we will discuss the six components, their resource objects and contracts, and how the components collaborate to provide the overall subsystem functionality.

In each of the sections, we will discuss two aspects of the component. First, the file system functionality that the component provides and second, the Pi approach-based structure of that component.

5.3.3 Access Provider

The access provider component supports one or more models of files as discussed in Section 5.1. A more elaborate hierarchy of the important file models is shown in Figure 5.4. This hierarchy may be extended further through subclassing. The root of the hierarchy is a generic file type called *File*. Files are named objects. More precisely, there is at least one pathname associated with a file. Pathnames and name resolution are discussed in the next section.

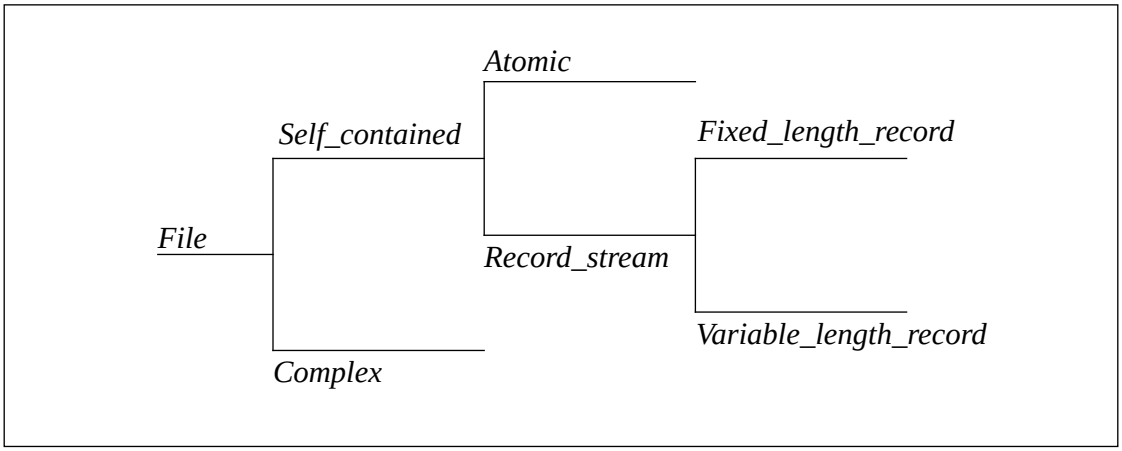

Figure 5.4: Hierarchy of File Models

Self_contained files objects, as the names suggests, do not have external components whereas *Complex* file objects are like hypertext; they may have embedded links to other files which may contain data or executable programs. *Record_stream* file objects have an offset variable while *Atomic* files do not export any variable for internal organization. Record streams may have records of fixed or variable sizes. UNIX files which are byte streams, are a special case of the *Fixed_length_record* files with a byte-long record.

Different file models entail different APIs. An Atomic file can be retrieved without specifying the amount of data while a record stream requires a call to change the offset. A Complex file exports a facility to return the links contained in the file either individually (`get_link()`) or as a list (`list_links()`). Following the Pi approach, PFS architecture

provides a contract for the access provider component. The contract interface documenting the basic calls for the file model hierarchy is shown as C++-like pseudo-code in Figure 5.5. This interface forms the basis for all access provider implementations.

```
// Only the functions derived from Access_Protocol are shown
// Naming-related functions are deferred to the next subsection

class Access_Provider_Contract_Interface
{
   public:

   // constructors and destructors - not shown

   // for record streams only
   int open(Pathname* this_path, int option);
   int close();
   int seek(int offset, int from);

   // for record streams and atomic objects
   int read(void* buffer_ptr, int max_size);
   int write(void* buffer_ptr, int size);

   // for atomic objects only
   int get(void**);

   // for complex objects only
   int get_link(Pathname* link);
   int list_links(Pathname_list* link_list);
};
```

Figure 5.5: Access Contract Interface

We have intentionally omitted one detail from the contract interface for brevity: a parameter containing the *credentials* of the client. The credential parameter is passed for almost all calls in this and other contract interfaces. It allows the concerned component to determine whether the client has permissions to receive the requested service, and to request services from other components on behalf of the client. The exact form of the credentials is implementation-dependent but the credentials should provide enough information for authentication and accounting. Credentials could contain process, user and group ids or they could be more complex and contain a ticket obtained from a security service like Kerberos [Steiner, 88]. However, since the issue of security is orthogonal to the decomposition of functionality, we will not explicitly state the parameter for credentials in the discussions of access provider and other components.

At run-time, the contract interface is supported by a contract object. When a client makes the first call related to a file, a contract object is created. All subsequent requests related to that file and that client are then serviced by that contract object. For example, in the case of a stream-based file, an `open()` call leads to creation of an access provider contract object. For other file models, similar actions are taken.

Unlike LFS, which cannot be changed, access provider CPOs are replaceable so that they can cater to different objects in the hierarchy of Figure 5.4. A client may also provide a custom access protocol to exploit specific features of a file or client's access pattern. For example, an implementation handling complex objects may provide a prefetching facility. Prefetching connected files (up to a limit of course) would substantially reduce the latency of accesses by a client. Such a customization would be useful for a hypertext browser.

Notice that the contract interface contains calls for multiple file models. This enables a client to make a run-time selection of a protocol implementation that supports the necessary subset of the calls. An access contract would consist of a CIO that supports the interface in Figure 5.5, and a CPO that effectively implements a subset of the same interface. For example, a client that wants to access atomic files would use a protocol implementation that supports `read()`, `write()` and `get()` calls from the contract interface. The implementation of other calls in the CPO can simply return an error indicating that those operations are not supported for an atomic file. Thus, we have delayed the decision about the file model until run-time and thereby allowed a client to participate in the decision.

In order to provide the functionality discussed in this section, an access provider relies on two downstream components realizing naming and data object management functions. The access provider invokes the appropriate functions on the naming component and then the data object manager, using the results returned by the former. Naming functions are discussed next.

5.3.4 Naming

In PFS, a name provides information used by a client to identify a data object. It is important to distinguish a name from an address or an identifier used for data storage. A name has a meaning for a client while an address or a storage identifier has a meaning for the storage implementation. For example, the name /bin/ls denotes to a client, an executable file for listing directory entries. On the other hand, the addresses for disk blocks where the file is stored are meaningful for the VFS implementation that stores and retrieves the file. The naming component bridges the distinction between the two.

In this section, we will first define a number of resource objects like bindings, directories, namespaces and pathnames. Then we will describe two key activities performed by the naming component: resolution and binding. The resource objects and activities will help us in defining a naming contract. The flexibility features provided by the naming contract distinguish the PFS architecture from the vnode architecture as described in Section 5.3.4.9.

5.3.4.1 Name

A *name* is a member of a language over some alphabet. A corresponding *primitive name* belongs to a subset of that language. For example, `a/b/c` and `abc/abc` are names drawn from the language L = {`(a|b|c|/)`*} defined over the alphabet A = {`a,b,c,/`} while `a`, `b`, `c` and `abc` are primitive names drawn from the language {`(a|b|c)`*}. Typically, some characters or character sequences are reserved as *separators* and a name is a concatenation of one or more primitive names separated using the separators. In the above-mentioned example, '/' can be considered a separator. But the PFS architecture does not specify any separators and leaves their specification to the implementation[2]. An address or a storage identifier, henceforth called an *id*, can also be thought of as a name drawn from

2. A similar strategy has been used in the IBM microkernel [IBM, 94b].

some language which may be defined over a different alphabet from that used for primitive names. For example, a 32-bit virtual address is drawn from the language $\{(0|1)^{32}\}$.

Since names and ids are syntactic entities, we will deal with them through resource objects like bindings which encapsulate the syntactic entities and provide precise semantics or behavior. This allows us to replace implementations and thereby change the behavior without having to specify the syntactic details in the PFS architecture. Consequently, a wider range of data sources and namespaces can be accommodated by the architecture. Next, we will discuss the important semantic entities in naming which are implemented as resource objects.

5.3.4.2 Binding

A *binding*[3] is an association of a primitive name and an id, syntactically represented by the two-tuple <name, id>. The id in a binding is quite flexible. It can identify a file, a directory, a link or a namespace. A binding is an opaque or encapsulated object. Its internal storage format is not known to its users. A given id may appear in multiple bindings; i.e. there may be aliases for an id. Conversely, a name may appear in multiple bindings; i.e. there may be a mechanism to disambiguate which is external to a name. A name is *bound* to an id to create a binding and the name is *resolved* to obtain the associated id. Thus, `bind()` and `resolve()` are the basic operations supported by a resource object implementing a binding.

5.3.4.3 File-Binding

A *file-binding* is the simplest kind of binding. The id in a file binding, called a *locator*, is used by other components of the PFS architecture to support operations on the identified file. For example, when the pathname corresponding to a file is resolved, the *naming* component returns a file-binding which is used by the data object management component

3. The word ‘binding’ is used both as a noun and as a verb; the former for denoting an object and the latter for denoting the process of creating that object.

for locating the file. In this sense, it is different from other bindings which are manipulated by the naming component.

5.3.4.4 Link

A *link* is an association between two names; a source and a destination. As such, it is a binding that relates two names. However, it provides an additional operation for reading the destination of the link. When the operations `resolve()` and `bind()` are invoked on a link, the link automatically invokes the same operations on the binding corresponding to the destination name. Further details of the semantics of a link are deliberately left unspecified in the PFS architecture to allow lattitude to implementors. For example, when a link is created, the link implementation may verify that the destination exists; it may also collaborate with the corresponding binding implementation to ensure that the link is deleted when the binding is deleted to avoid dangling links.

5.3.4.5 Directory

A *directory* is a named container containing entries which are bindings. It is itself a binding between a name and a container and it can recursively contain bindings for other directories. The id in a directory binding is the identifier for the container; it may be the address of the disk block where the directory is stored or some other identifier that can be used to construct a directory object at run-time. It is important to note that a directory need not be coupled to data on a storage device. For example, a directory may be constructed at run-time through queries to a database. Like all bindings, a directory supports `resolve()` and `bind()` operations; in addition it provides a `list()` operation for listing entries.

5.3.4.6 Namespace

A *namespace* is a named, structured collection of bindings. It too, is a binding whose id identifies the collection. Specifically, a namespace is a directed graph of bindings where file bindings are denoted by leaf nodes, and directories and links are denoted

by interior nodes. An empty directory may be denoted by a leaf node. As shown in Figure 5.6, an edge in the graph denotes a name[4]; a primitive name in case of an edge originating from a directory node. In the figure, let us assume that $\mathbf{S_0}$, $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_4}$ are directories, $\mathbf{S_3}$ is a link while $\mathbf{S_5}$ is a file-binding. Like the three bindings described above, a namespace also provides operations for resolution and binding of names. Each namespace has an associated scope in the PFS architecture which is an instance of the file system scope type shown in Figure 5.2

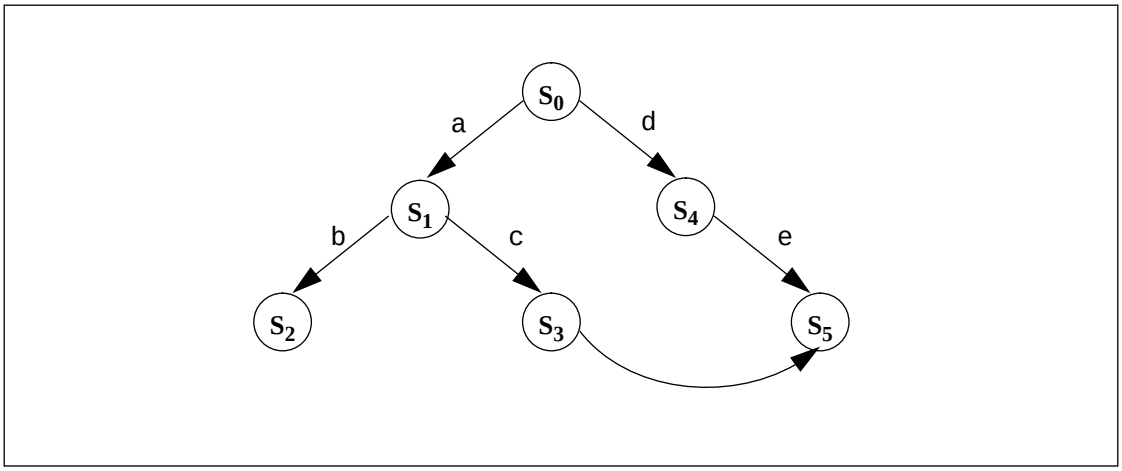

Figure 5.6: An Example of a Namespace

5.3.4.7 Federated Namespaces

Multiple namespaces may be glued together to create a single *federated* namespace. Each namespace retains full control over its portion of the federated namespace but agrees to invoke `resolve()` and `bind()` functions on other namespaces when a pathname crosses its boundaries. In order to glue together namespaces, a namespace may support operations to *mount* and *unmount* another namespace. We will revisit the different behaviors that are possible for mount in the next chapter.

---

4. In this figure, following [Comer, 89], we have used names to label edges rather than nodes. A less precise but more popular depiction associates names with nodes.

5.3.4.8 Name Resolution and Binding

Name resolution is the process of translating a name into an id. There are two ways of looking at the resolution process: the classic approach of syntax-based translation [Comer, 89] or, as we do in PFS, as a semantic process of a sequence of invocations on resource objects. The PFS process models syntax-based translation, and it requires two resource objects, pathname and context.

There are two simultaneous activities involved in resolution of a name in a namespace. The namespace graph is traversed and as the graph is traversed, portions of the name are considered resolved. A *context object* corresponds to the just visited node in the graph. A *pathname object* encapsulates the name, and keeps track of the portion of the name that has been resolved. If we are at node **S1** in Figure 5.6 while resolving the name `/a/c`, then resolution would proceed further with a context object corresponding to node **S1** and the pathname object reflecting the fact that `/a` has been resolved. The resolution process can be conveniently formulated as a finite state machine.

Consider a finite state machine (FSM) corresponding to the namespace in Figure 5.6. The nodes in the graph correspond to the possible states of the FSM. So we will use the same names $\mathbf{S_0}$ .. $\mathbf{S_5}$ for denoting the current state. If we think of the labels on the edges (names) as the inputs to a Moore machine[5], context objects can be thought of as the outputs of the Moore machine. As the resolution process progresses, it moves from one state to another by modifying the pathname object (consuming the input) and producing context objects (producing the output).

A context object is a run-time representation of a binding. Hence, in the PFS architecture, name resolution proceeds by invoking the `resolve()` operation on a sequence of

5. A Moore machine is a finite state machine wherein an output is associated with each state and each transition is labelled to indicate the input symbol(s) consumed by the machine in making the transition [Carroll, 89].

context objects. Each invocation produces a new context object and causes a change in the state maintained by the pathname object. For example, when the name `a/b/c` is resolved in the namespace shown in Figure 5.6, `resolve()` operation is invoked on the sequence of context objects corresponding to states $\mathbf{S_0}$, $\mathbf{S_1}$, $\mathbf{S_3}$, $\mathbf{S_5}$ as the three primitive names, `a`, `b` and `c` are parsed in the pathname object.

A context object may be implemented as a contract object or simply as a resource object. In the former case, its implementation could be changed at run-time while in the latter case, even the minimal overhead of an indirection required in a contract object can be avoided. Note that a context object is not the same as a binding. A binding is an object that exists independently of any run-time invocations. A context object on the other hand is a result of the resolution process; an object created at run-time based on a specific client-request.

The process of binding is similar to the process of resolution. The difference is that when a directory or a binding is not found in the process of traversing the namespace, a new binding is created. For example, if the name `a/b/f` is being bound to a file with id `I`, then a new binding <`f`, `I`> would be created in the directory $\mathbf{S_2}$. The implementation of PFS may impose further restrictions on binding. For example, it may return an error if a request to bind `a/b/g/h` to some id is received when a directory corresponding to `a/b/g` does not exist.

The naming functionality is documented in a *naming contract interface* which is shown in Figure 5.7. This functionality is provided by a context object, which is the contract protocol object (CPO) for the naming protocol. Resolution and binding are the two most important and generic aspects of the protocol; other aspects shown in the figure are specific to directories, links and namespaces.

```
class Naming_Contract_Interface
{
   public:

   // constructors and destructors - not shown

   // functions common across all contexts
   int resolve(Pathname* this_path, Context** returned_context);
   int bind(Pathname* this_path, Context* bind_to_this);

   // file-supported function - provide id for data object manager
   int get_id(id* returned_id);

   // link-supported function - read the name pointed to
   int readlink(Pathname* source_path, Pathname* destination);

   // directory-supported function - get a list of directory entries
   int readdir(Pathname* dir_path, direntry_array* entry_ptr);

   // namespace-supported functions
   int mount(Pathname* mount_here, Context* mount_this);
   int unmount(Pathname* unmount_from, Context* unmount_this);

   // more functions to deal with attributes - not shown
};
```

Figure 5.7: Naming Contract Interface

5.3.4.9 Comparison with Naming in the Vnode Architecture

In the vnode architecture, a substantial part of the resolution and binding process occurs in the LFS portion of the subsystem which is not replaceable or modifiable (Section 5.2). In the PFS architecture, most of the naming functionality has been moved into a replaceable naming component. Hence, we can easily accommodate multiple simultaneously active namespaces with different implementations. In the vnode world, this would be analogous to moving the naming functionality into the VFSs. Of course, the contract mechanism also provides the ability to select or replace the implementation for a given namespace. These advantages will be discussed in greater detail in the next chapter.

The difference in the name resolution approaches can be explained with an example of resolution of the pathname `/a/b/c`. In the PFS approach, `resolve()` is invoked on the root namespace with the pathname as an argument. The implementation of the namespace then internally performs resolution as discussed above and returns a context object corre-

sponding the binding of the pathname and id. On the other hand, the following steps are taken by LFS in the vnode architecture to resolve the pathname `/a/b/c`

1. Find the vnode for the root, represented by the first '`/`' character in the pathname.
2. Invoke `vn_lookup` on the root vnode to get a vnode for `a`.
3. If there is a VFS mounted on top of the vnode for `a`, get the root vnode for that VFS, otherwise use the vnode obtained in the previous step.
4. Invoke `vn_lookup` on the vnode obtained in step 3.
5. Repeat steps 2 through 4 for primitive names `b` and `c`.

Note that LFS resolves the name one primitive name at a time; so the VFS implementation that actually implements the namespace sees only one primitive name at a time; first `a`, then `b` and then `c` in the above example[6]. In contrast, in the PFS architecture, a namespace gets an entire pathname at once. This is a significant difference when we consider the cost of cross-protection domain calls (Table 4.1). Vnode-style resolution works fine when both the LFS and the namespace(s) needed for resolution are within the same protection domain. However, when the LFS and a namespace needed for resolution are in separate domains, this resolution process introduces unnecessary overhead since a cross-domain call is required for *each* primitive name in the given name. For longer names, this overhead can be substantial. Researchers have tried to reduce the overhead by introducing name caches in the same domain as LFS. For example, NFS uses a cache called Directory Name Lookup Cache (DNLC) so that if the same name is resolved again, the overhead is not incurred again [Goodheart, 94]. But the overhead is unnecessary in the first place and can be avoided by providing the entire name to a namespace.

5.3.4.10 Naming and Scopes

Recall from Section 5.3.1, that there are multiple scopes in a subsystem. In particular we discussed FileSystem and FileSystemType scopes. Since naming is separated from data management, there are corresponding scopes related to the naming component. A NameSpace and a NameSpaceType. For example, X.500 would be a NameSpaceType

6. This problem has also been pointed out in [Welch, 93].

scope while a particular X.500 namespace would be a NameSpace scope. These scopes are at the same level as the FileSystem and FileSystemType scopes.

The other interesting scope-related aspect of naming is related to client scopes. One of the common debates in operating systems is the scope of a federated namespace. Sprite [Welch, 90] implements a uniform namespace for a cluster of workstations, UNIX [Ritchie, 74] provides a uniform namespace for all the processes on one machine and Plan 9 [Pike, 93] provides a per-process namespace. In the PFS architecture, there is a namespace associated with a process scope. However, that namespace may delegate all the functions to a common namespace associated with the Subsystem scope. Hence, the decision of the scope of a namespace is left to the implementor who may even make it available to clients. The architecture provides the capability for per-process namespace without mandating it.

Name management is one of the two important parts of a file system. The other one is data object management. It uses the id supplied by the naming component and allows operations on the corresponding file for reading and writing data. Next, we will discuss the data object management component.

5.3.5 Data Object Management

The Data Object Management (DOM) component is responsible for storage and retrieval of data. Like the access provider component, it supports multiple file models. However, the access provider and naming components take care of converting the file names into suitable locators, i.e. ids in file bindings. Hence, the DOM component deals exclusively with file locators.

The DOM component shields the access provider from the details of the I/O protocol and caching. Like VFS, it presents a generic view of data objects through a contract. The basic functions provided by DOM contract are similar to those provided in the access

provider contract shown in Figure 5.5. In most cases, when a particular function is invoked on the access provider contract, the corresponding function can be invoked on the DOM contract after name resolution. However, occasionally, the access provider may have to translate the calls to convert from one file model to another. Translation would be necessary when the file model supported by an instance of the DOM component is not the same as that supported by the access provider. For example, if an instance of the DOM component supports the atomic model, while the file system client expects to see a byte-stream model, the access provider would be provide the translation.

Apart from providing a generic view of data storage and retrieval to the access component, the DOM component implements caching protocols using the mechanisms provided by the cache management component. A client that wants to change caching would select an appropriate implementation of the DOM protocol. The selection facility is provided by the DOM contract.

The DOM component provides another important operation, `get_file()`. It is provided on a per *FileSystem* scope basis; i.e. each file system has its own implementation of the operation. The operation takes the locator of a file as an argument and searches for the corresponding *File* scope. If the scope is not found, a new *File* scope and a DOM contract object for that scope are constructed.

5.3.6 Data Caching

The caching component provides performance improvement for storage and retrieval of data. It hides the high latency that is typical of may storage devices and remote communication. At the same time, it must also ensure the integrity of data. There are three distinct aspects of integrity: persistence, mutual exclusion and coherency.

Persistence can be ensured by transferring data to non-volatile storage; typically a disk. The need for persistence is offset by the desire to provide low latency for writes and

achieving high overall throughput for the file system. Hence, the caching component only provides certain mechanisms and leaves it up to the DOM component or a client to decide when data should be transferred to non-volatile storage. There are two calls in the caching contract described in Figure 5.8 for persistence: `move()` and `copy()`. In general, these two calls effect the movement of data between storage levels including, but not limited to volatile memory and disk. The same mechanism may be used for moving data to other storage devices like tapes.

```
// Functions in the caching contract are used by the DOM component

class Caching_Contract_Interface
{
   public:

   // constructors and destructors - not shown

   // persistence-related functions
   int move(Storage_Id from_storage, Storage_Id to_storage);
   int copy(Storage_Id from_storage, Storage_Id to_storage);

   // mutual exclusion-related functions
   int lock(int mode, int* returned_lock_id);
   int unlock(int lock_id);

   // coherency-related functions
   int refresh();
   int update();
};
```

Figure 5.8: Caching Contract Interface

Mutual exclusion is accomplished in the caching component using locks. Locks may be acquired in read or write mode either directly by a client or by the DOM component. A read lock provides shared access and assurance that no writer is allowed to access the file. A write lock provides exclusive access. Locks may be used to provide transactional semantics [Bernstein, 87] for file system operations.

Coherency is necessary in a distributed environment due to the presence of multiple potentially modifiable copies. While mutual exclusion assures that there is at most one writer, it does not assure that the copy being read is the latest or that an update made to a

copy will be visible in other copies. Hence, as in the case of distributed shared objects discussed in the last chapter, the caching component provides two calls: `refresh()` to get the latest copy and `update()` to propagate changes.

A natural follow-on of mutual exclusion for single copy and coherency for multiple copies is the combination of the two called *distributed locks*. A distributed read lock assures not only that there are no writers changing the local copy but that the local copy is the latest and that none of the remote copies are being modified. Likewise, a distributed write lock assures exclusive access as if there is only one copy and that updates will be propagated after the lock is released. Such a distributed locking mechanism is provided in the architecture as an implementation option. However, it is not required and may not be available in some implementations.

There is an important caveat about the caching mechanisms. As mentioned in the beginning of this chapter, the PFS architecture does not deal with the design of file servers in a distributed system. Instead, it provides an architecture for client file system. Hence, the semantics of the distributed locking calls discussed above depends on the caching protocol implemented by the server. For example, FTP and NFS servers do not maintain information about the state of the client. Hence, a cache implemented using FTP [Postel, 85] and NFS [Sun, 89] protocols is also limited in its capability. The client file system is not informed when the file changes. Such limitations further increase the importance of explicit calls like `refresh()` and `update()` since they allow a client of the file system to decide when multiple copies should be made coherent.

The caching calls are not limited to individual files. They also apply to a file system. For example, the call `copy()` can be used to transfer all the modified files in a file system to the disk. Thus, a caching contract is associated with a file as well as a file system.

5.3.7 Name Caching

Name caching component deals with similar issues as the caching component described above. However, there are subtle differences between the two components. Directories and files differ in how they are modified and what semantics is expected of them [Kistler, 93].

A file is normally treated as an indivisible object. When a file is modified, an old copy is considered useless. On the other hand, directories are slowly changing objects with more predictable patterns of change. They differ from files in three respects. First, directories change more slowly than files. Second, the coherency constraints on an operation like `readdir()` may not be the same as those on `read()` operation performed on a file. Third, and most important, modification to a directory typically occurs through insertion or deletion of an entry or a change in attributes rather than as a complete change of contents. The name caching contract exploits these differences, and especially the third one. Apart from the calls in the caching contract, it provides facilities for propagating changes to a directory to another storage level or a remote machine through the calls `insert()` and `delete()`.

5.3.8 I/O

The last component of the PFS performs input and output operations for the upstream components. It manages devices and communication with file servers. As such, it has to deal with issues beyond reading and writing of files and directories. For example, an ftp protocol implementation has to perform a handshake with a remote ftp server by logging in, and it has to manage the connection with the ftp server. Hence, unlike in other components, most of the details of the I/O component must be left to the implementation. They cannot be fully specified in the PFS architecture. The PFS architecture requires an I/O contract to support at least one of the several file models discussed in Section 5.3.3.

Additional implementation-specific calls may be provided in the contract for authentication and setting of parameters like timeout period.

5.3.9 Interfaces

The PFS architecture provides two interfaces for clients. The first interface provides the basic functionality and is backed by default implementations of contracts. The second interface provides additional control to clients and allows them to change contract implementations.

From a client's point of view, files, directories and links are the basic entities in a file system. The common operations are creating, deleting, reading and writing these entities. Hence, the first interface provides calls for performing these operations. Most of these operations have already been documented in the access provider and naming contracts shown in Figures 5.5 and 5.7.

A client seeking more control would like to deal with aspects of self-representation of the file system. The common aspects would be scopes like namespaces and file systems, internally used contracts like the caching contract and the implementations for contracts. Hence, the second interface provides calls for the following tasks which have been discussed in the preceding sections.

- Rearrangement of the federated namespace through mounting and unmounting of namespaces
- Creation and deletion of scopes like namespaces, file systems, namespace types, and file system types.
- Effecting caching directly through calls provided in the caching contract
- Manipulating existing contract implementations using `get_impl()` and `set_impl()`.
- Providing a new contract implementation for one of the predefined contract inter-

faces.

Thus, as discussed in Section 4.2.2, there are three levels of usage available to a client. At the first and most basic level, the first interface provides calls that would be enough for most clients. At the second level, a more sophisticated client can use the caching calls or select an implementation. Finally, a sophisticated client may provide a custom implementation. The three programming levels require significantly different and increasing understanding of the PFS architecture and implementation.

This description of a client's view completes the PFS architecture. However, to further clarify how the components fit together, let us consider a small example and walk through the steps taken by different components to provide a service to a client.

5.3.10 An Example

Consider a client that opens, reads and then closes a file. Let us assume that the client wants to use the default implementations and that the file is a stream of bytes. The following sequence of actions is performed by the implementation:

1. The client invokes the `open()` function provided by the PFS's first interface.
2. The PFS framework creates an access provider contract object (Section 5.3.3).
3. The access provider object creates a new *OpenFile* scope and determines the contract implementations to be used by following the scope-based dispatch procedure given in Section 4.1.2.2
4. The access provider object constructs a pathname object out of the client-supplied name and calls `resolve()` on the naming contract object associated with the client process scope.
5. The naming contract object initiates the process of resolution given in Section 5.3.4.8. The process produces a sequence of context objects, and may span multiple namespaces. If a binding corresponding to the pathname is found, the resolution process is considered successful and a file context object is returned by the naming component. Otherwise an error code is returned.
6. The access provider uses the locator provided by the file context object to request a DOM contract object.
7. The DOM component searches its internal data structures for a contract object corresponding to the locator and, if found, returns the contract object. If a contract object is not found, a *File* scope is created and a new DOM contract object is con-

structed using scope-based dispatch.

8. The access provider object invokes `open()` on the DOM contract object. The DOM contract object calls downstream components if it is necessary to interact with the data source.
9. Control is returned to the client indicating the result of the `open()` call.
10. Client invokes `read()` on the PFS's first interface.
11. The corresponding `read()` function from the access provider contract is invoked, which in turn invokes `read()` on the DOM contract object obtained in step 7.
12. The DOM contract object reads data with the help of the caching and/or I/O component.
13. Control and data are returned to the client indicating completion of the read request.
14. The client invokes `close()` on the PFS's first interface.
15. Actions similar to those in steps 11 through 13 are performed for `close()`.

This sequence illustrates how the PFS components work together to provide file services to clients. It also illustrates how the basic features of the Pi approach, viz. contracts and scope-based dispatch, get used. At this point, we will conclude our discussion of PFS architecture with a summary and then proceed to a discussion of its implementation and evaluation in the next chapter.

5.4 Summary

We started with the purpose of applying the Pi approach to a subsystem. We chose the file system and discussed the features that a new architecture should support. Support for different APIs and data resources as well as tailorable naming and caching are some of the desirable features for a new file system architecture. We discussed the vnode architecture as a basis for comparison and evolution. We then proposed the Pi File System (PFS) architecture and discussed its components in detail. The PFS architecture is more flexible than the vnode architecture due to fine-grain decomposition and control over naming and caching. It uses scopes and contracts, two key ideas from the Pi approach to provide greater control to clients. Finally, we saw how clients can use a PFS implementation through dual interfaces. Now we can discuss an implementation of the architecture and evaluate the effectiveness of the Pi approach for providing flexibility.

6. PROTOTYPE IMPLEMENTATION AND EVALUATION

In the last chapter, we discussed the Pi File System (PFS) architecture. In this chapter, we will present a prototype implementation of that architecture and selected experiments performed with it. We will also analyze our experience with the prototype to evaluate the effectiveness of the Pi approach.

The goal of the prototype was three-fold: to demonstrate the feasibility of constructing a flexible file system using the PFS architecture (Section 6.2.2); to implement a testbed for evaluating the effectiveness of the Pi approach with respect to the goals discussed in Chapter 2 (Section 6.2.3), and to understand the capabilities and limitations of the Pi approach beyond the agenda of Chapter 2 (Section 6.2.4). But before discussing the results in the light of these goals, we will explain the prototype implementation.

6.1 Implementation Overview

An implementation of the PFS architecture was constructed mostly at the user level for easy experimentation. A kernel-level implementation in AIX would have been more efficient (Table 4.1) but more cumbersome and complex to modify. Also, as described in Section 4.3.2.2, we wanted to allow extensions that use multiple protection domains. Hence, an implementation consisting of one or more AIX processes was a natural choice. Further, to avoid reimplementing the low level storage management facilities provided by AIX, the prototype was designed to use the AIX file system for storing data on disks.

The implementation discussed in this chapter, called PFS, closely follows the PFS architecture. However, it does not currently provide facilities to dynamically load protocol implementations and to dynamically add new file system types. However, clients can select a protocol implementation at run-time and they can create new instances of the supported file system types. Also, protocol implementations and new file system types can be added at compile-time as separate modules.

In this section, the overall organization of the implementation is discussed and protection domain issues are briefly addressed. Additional implementation details are given in the appendix. As in Chapter 4, the platform for experimentation was AIX 3.2.5 running on 25 MHz, RS/6000 model 530 workstations.

6.1.1 Organization

The structure used for the prototype implementation is shown in Figure 6.1. It separates the framework from the protocol implementations. The framework implements the key ideas of the Pi approach like dual interfaces, scopes and contracts, while the protocol implementations provide file system functionality like naming and caching.

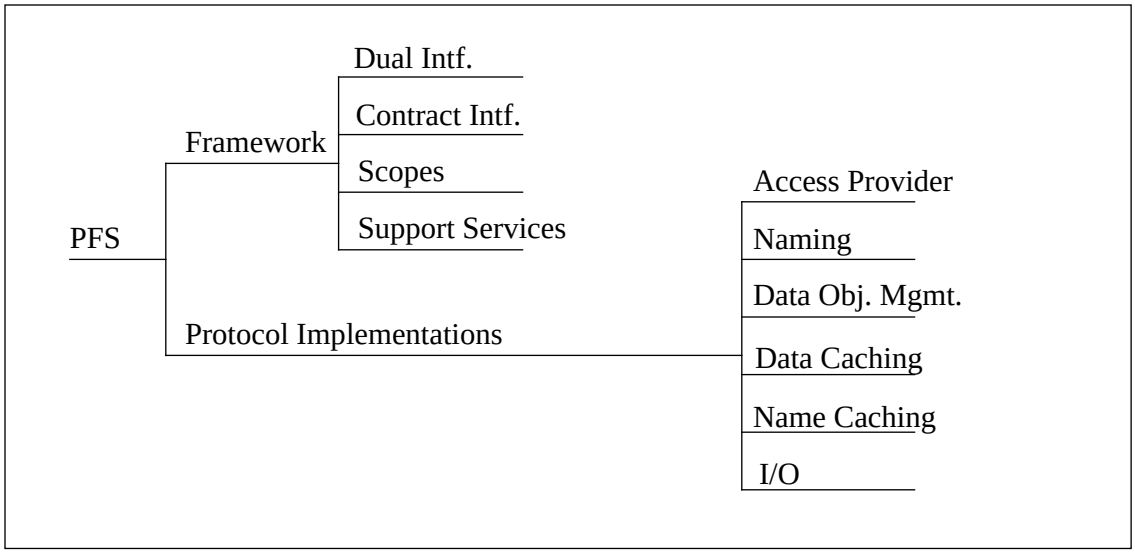

Figure 6.1: Organization of the PFS Implementation

Many of the entities shown in Figure 6.1 are realized as C++ classes. The classes may have multiple run-time instances in a PFS implementation. For convenience, we will start a class name with an upper-case later, e.g. `Process`, while using a lower-case letter to start the name of an object of that class, e.g. `process`.

6.1.1.1 Framework

The framework relies on two interface objects to field client requests: one for the first or functionality interface and the other for second or control interface. The two objects are instances of separate classes. Each interface object receives client-calls, performs some preliminary processing and then invokes the functions supported by scopes and contracts to provide the service requested by a client.

Scopes are implemented as instances of classes that represent the scope types described in Section 5.3.1. We will use the same names for classes as those used for scope types in Figure 5.2. All scope classes inherit from a base class called `BaseScope`. The instances of class `BaseScope`, which are contained in every scope through C++ inheritance, keep track of the contract descriptors selected by a client for that scope. Scopes are organized in a hierarchy according to the nesting property shown in Figure 5.2. There are two subhierarchies, one for client-scopes and another for unit scopes. The top-level scope named `subsystem_scope` is the only instance of class `Subsystem` in a PFS. It provides contract descriptors of default implementations to ensure that scope-based dispatch algorithm of Section 4.1.2.2 terminates successfully. In fact, it also implements scope-based dispatch for the entire subsystem. Given an instance of a class like `Process` or `OpenFile`, `subsystem_scope` traverses the corresponding sub-hierarchy of scopes and provides a contract descriptor. The order of traversal is: instances of classes `Process`, `User`, `Group` on the client side, and `OpenFile`, `File`, `FileSystem`, `FileSystemType` on the subsystem side.

The PFS implementation uses a separate contract interface class for each contract. Each class uses an inheritance relationship similar to that shown in Figure 4.10 to support two interfaces. It maintains information about constructors for contract protocol objects (CPO) and the compatibility relationship between them. A contract interface object (CIO) is given two descriptors obtained from the two scope sub-hierarchies. The CIO then checks compatibility and creates an appropriate CPO. Once a CPO is created, the CIO delegates calls to the CPO.

The fourth part of the framework provides a set of support services. It includes functions for initialization, proxies for allowing clients to communicate with the file system and a few miscellaneous functions. When the file system process is started, the initialization code reads the configuration of the file system, creates the *Subsystem* scope and dual interface objects, initializes component file systems and namespaces and then waits for client requests. When a client request is received through a proxy, the support services code invokes the relevant operation on one of the interface objects. The proxy code for the client requests resides in the same address space as the client, and communicates the parameters of a client call to the file system process.

6.1.1.2 Protocol Implementations

While the framework provides flexibility-related services, protocol implementations provide the file system specific functionality like name resolution, caching and I/O. As shown in Figure 6.1, protocols related to the six functional components of the PFS architecture are provided. The current PFS implementation includes default as well as alternate protocol implementations. The latter are selected by a client explicitly through a `set_impl()` call supported by the second interface. We will discuss a subset of protocol implementations in the next section to illustrate how a client can change a PFS implementation.

The PFS implementation relies on the AIX file system for storing files on the disk rather than creating its own partition. The basic storage functionality is orthogonal to the investigation of the Pi approach and the PFS architecture, so it was decided that the extra effort was not worthwhile. There are three advantages of this decision:

- PFS can be easily started; the executable files as well as the configuration files can be accessed using the standard API.
- PFS so started can provide access to the files stored by AIX. In this case, the AIX file system is used as a data source accessed through PFS. A client can use naming and caching features of PFS while accessing those files.
- The caching protocol implementations in PFS can use AIX files as local persistent storage for remote data sources like FTP and IMAP servers. A remote file can be cached in memory or as an AIX file on a local disk.

PFS currently supports the atomic and data stream file models. It provides I/O protocol implementations to access files stored by the AIX file system, files available on FTP servers and email managed by IMAP servers [Crispin, 90]. A thin veneer of routines was developed for accessing AIX files; an FTP implementation was developed from scratch to access FTP servers and a freely available IMAP client written by Crispin [Crispin, 93] was adapted for email.

6.1.2 Process Configuration

PFS is realized as one or more AIX processes separate from the client processes. Two different configurations of the implementation are shown in Figure 6.2. The top one is a simple single-threaded version with all the protocols implementations in the same process. The bottom one is a multi-process version created for greater concurrency. It implements a CPO for the I/O contract in a process separate from the rest of the PFS. It shows

that a CPO can be implemented in a separate process as discussed in Section 4.3.2.2. However, in the rest of the chapter, we will focus on the single process version.

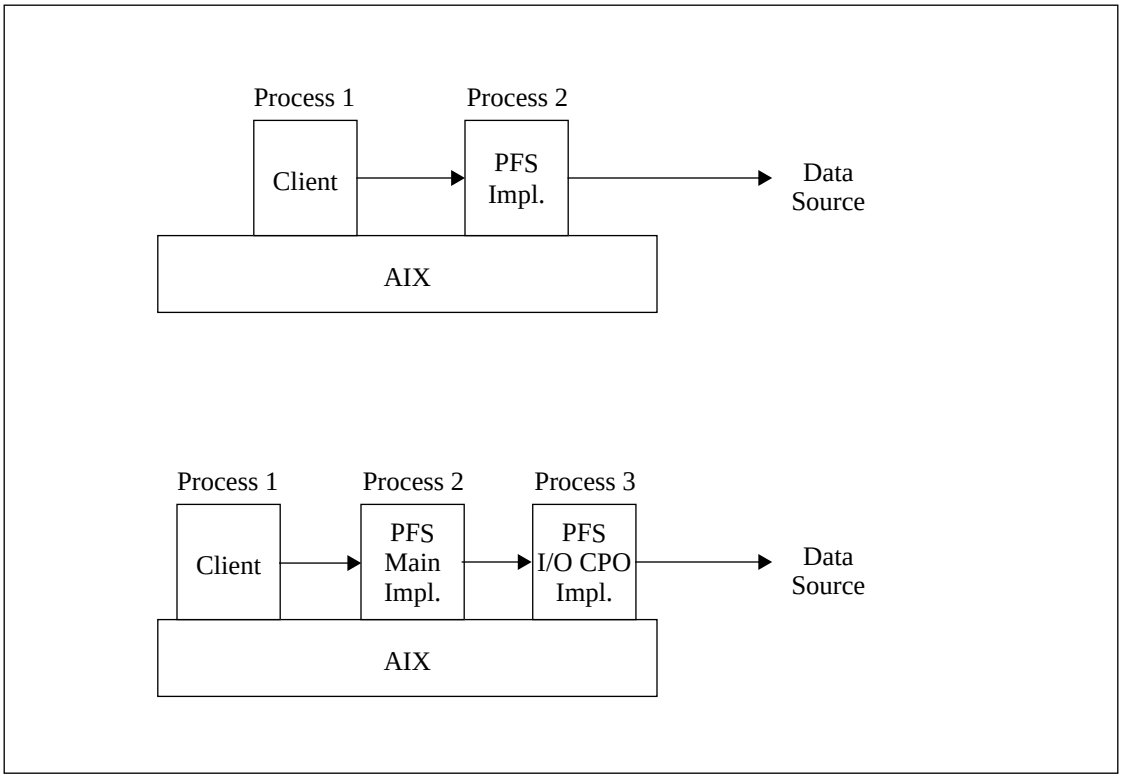

Figure 6.2: Process Configurations of PFS Implementation

6.2 Evaluation

We have shown that a file system can be constructed using the Pi approach and the PFS architecture. Now we can turn to the evaluation of the Pi approach through a set of experiments with the prototype. After discussing the experiments, we will examine the effectiveness of the Pi approach.

6.2.1 Evaluation of Contracts and Scopes

We have selected three simple experiments that use PFS. These experiments shed light on the desirable flexibility features discussed in Chapter 2. Out of the five features discussed in Section 2.6, we have already covered the first feature in Chapter 4, viz. separation of client-controllable aspects from the rest of the implementation through contracts.

Contract implementations can be selected by clients while the framework implementation cannot be altered. Now, we can look at the other four aspects: procedural control, incrementality, scope-control and overhead. One or more of these aspects are addressed in each of the experiments. An analysis of the overhead involved in using the Pi approach follows the discussion of the three experiments.

The description of each experiment contains the following:

- The difference between the default and the non-default behaviors.
- The reasons for the client-selected implementation not being the default implementation.
- The procedure for changing the PFS implementation.
- The results achieved due to the change.

6.2.1.1 Union Mount

The first experiment concerns the naming component discussed in Section 5.3.4. In this experiment, a client changes the behavior for the `mount()` call which glues together multiple namespaces (Section 5.3.4.6). The default and client-selected behaviors for `mount()` can be explained using Figure 6.3. The default implementation provides a UNIX-like behavior; the mounted namespace shadows the part of the namespace that it is mounted over. Hence, files `a.ps` and `b.ps` become visible in the namespace while files `c.ps` and `d.ps` disappear. The client-selected implementation realizes what is called a *union mount*[1] [Pike, 93]. A `mount()` call causes a union of multiple namespaces at the mount-point instead of shadowing. Hence, all four files become visible under the mount-point `papers`. Even multiple namespaces can be mounted at the same point without one namespace shadowing another.

---

1. Union mount is also implemented to a very limited extent in most shells through the mechanism of environment variables like PATH. However, such a mechanism is ad-hoc and remains outside the purview of file systems.

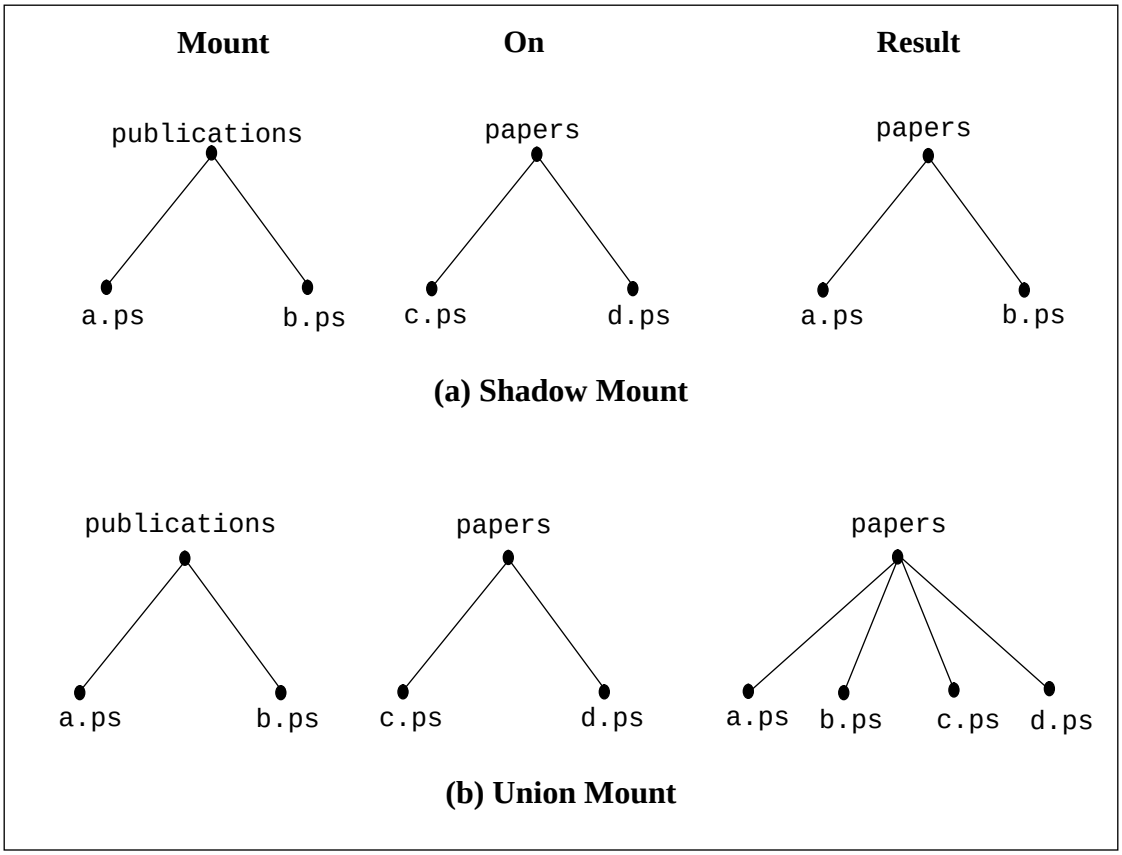

Figure 6.3: Shadow and Union Mount

Union mount helps a client create a customized namespace. A client can create a single logical view out of multiple physical data resources. For example, if the two directories in the figure reside at different locations, then papers on the same subject stored at different sites can be made visible within the same directory. Such a logical view may be presented to a user by a browser client. The location information need not clutter up the logical view of the combined namespace. Thus, users can obtain truly location-independent names.

However, union mount behavior is not necessarily appropriate for the default implementation due to the following reasons.

- Most clients in UNIX are implemented assuming shadow mounts.
- Union mounts can lead to collisions between names; directories mounted at the

same point may have logically different files with the same file name. Most clients are not equipped to handle name collisions.

Thus, in this case, a client has to instruct the subsystem implementation to use the union mount implementation.

PFS includes a union mount implementation option for the naming contract. A client selects the union mount implementation for itself by invoking `set_impl()` on PFS's second interface. In this experiment, the client requests that the change be applied to the corresponding *Process* scope. In Section 5.3.4.9, it was mentioned that each *Process* scope has its associated naming contract to implement a per-process namespace. In this case, the new implementation of mount() is made available to the requesting client by setting an entry in the contract descriptor list. The entry specifies that union mount implementation be used for naming contract for that process. Thereafter, the client can use the union mount functionality by simply invoking the `mount()` operation.

This experiment shows that a client can obtain a desirable functionality even if it is not deemed appropriate for the 'common case'. Client participation in a significant design decision, viz. semantics of `mount()`, is possible at run-time and can be accomplished without affecting other clients.

6.2.1.2 Prefetching

The second experiment concerns the data object management (DOM) component discussed in Section 5.3.5. In this experiment, a client selects a non-default implementation which performs a whole-file read whenever a file is opened for read-only access. The default implementation does not read the file until `read()` is called by the client. When the file being accessed is retrieved from a remote source, an FTP server in this experiment, the two implementations show different performance. The default one has to contact the server twice, first for servicing `open()` and then for `read()`; each time a connection has to be set up with the server which causes a delay. By contrast, the client-selected implemen-

tation connects to the server once and prefetches the entire file, avoiding a subsequent connection for reading. In addition to saving the delay of an extra connection, this implementation reduces the latency for a `read()` call. Since the data is prefetched, the `read()` call can be quickly serviced.

However, the client-selected prefetching implementation is not appropriate as the default implementation. A designer optimizing for the 'common case' will not select this implementation for the following reasons:

- A file may be updated between the calls `open()` and `read()`. All clients may not want to miss the interim update and a file system designer has no way to predict whether such an update is possible.
- A file opened in read-only mode may never be read; the client may just want the meta-information such as the time of last modification or size. In this case, the overhead of prefetching is unnecessary.
- The prefetching implementation increases the latency for the `open()` call; the call does not return control to the client until the entire file has been read.

Hence, again in this experiment, a client has to instruct the subsystem implementation to perform the prefetching optimization.

PFS includes a prefetching CPO implementation for the DOM contract. A client selects the prefetching implementation through the `set_impl()` function provided by PFS's second interface. In this experiment, the change is requested by the client for a *Process* scope corresponding to the client program. With the help of the PFS library, the client communicates the appropriate scope and contract descriptors to the PFS. The PFS validates that the client has made a legitimate request. Since the indicated scope is the client process, the request is accepted and an entry is created in the contract descriptor list for the client process scope inside PFS. Then, the prefetching implementation is automatically dispatched when the client opens a file.

The prefetching experiment run using the FTP protocol for I/O yielded the following results. The average cost of the sequence of calls, `open()` followed by a `read()`, was reduced from 1.85 seconds to 0.71 seconds for a one kilobyte file. The numbers were obtained on a platform consisting of two RS/6000 model 530 machines connected by a 10Mbps ethernet. PFS ran on one machine while a standard FTP server resided on the other. The experiment was repeated ten times to obtain the mean. Such a substantial improvement in performance is possible because of the high overhead of connecting to an FTP server. But the magnitude of the performance improvement is perhaps less crucial. The important implication is that a client can achieve significant gains through the ability to change an implementation to suit its needs.

6.2.1.3 Caching

The third experiment also pertains to PFS's DOM component (Section 5.3.5). In this experiment, a client requests a caching implementation for one of the component file systems, a file system providing access to an FTP server. The default implementation gets a file from the FTP server in response to a `read()` request but purges the copy of the file once the file is closed. Hence, subsequent requests by the same or other clients require that the file be fetched again. On the other hand, the caching implementation retains the copy of the file for a certain period of time so that subsequent `read()` requests received within that period, can be efficiently serviced. As a result, the latency for those `read()` calls is reduced and the load on the server is also reduced.

However, the caching implementation may not be appropriate as the default implementation for the following reasons:

- A cached copy could become stale if the master copy on the FTP server is updated during the caching period. Stale data may not be acceptable for some applications using the file system.
- A file may not be read a second time during the caching period.

- Caching files in this fashion can drain memory resources from other component file systems and/or other processes running on the same machine.

Thus, as in the two previous experiments, a client has to instruct PFS to use the caching implementation. In doing so, the client indicates that the caching optimization is acceptable in spite of the possibility of stale data and that it is worthwhile even with the required memory consumption.

PFS includes a caching implementation for the DOM contract used for FTP file system. A client selects the caching implementation by calling `set_impl()`. In this experiment, the client requests the change for a *FileSystem* scope rather than the *Process* scope, and has the necessary privilege for doing so. In response to the `set_impl()` request, the PFS creates an entry in the contract descriptor list for the FileSystem scope. Thereafter, due to scope-based dispatch, a caching CPO is created for the DOM contract.

The machine configuration and the measurement method for this experiment was the same as that for the previous one. Caching improved performance of `read()` call from 416 milliseconds to 2 milliseconds for a 1kb file and from 629 milliseconds to 6 milliseconds for a 32 kb file for subsequent `read()` calls after the file was cached. Again the substantial difference is attributable to the overhead of accessing the data resource; it includes the delay caused by the FTP protocol and the cost of reading the file from a disk on the FTP server. But as before, the key implication is that client participation can provide substantial gains for certain client-resource combinations.

6.2.1.4 Overhead

The three experiments discussed above show how a client can change the PFS implementation. But there is an additional aspect of PFS's evaluation not revealed in these experiments. We need to measure the overhead introduced by the flexibility mechanisms proposed in the Pi approach and implemented in PFS. Contracts and scopes are the two basic mechanisms in the Pi approach which can add overhead.

A contract object relies on delegation from the interface object CIO to the protocol object CPO. Delegation introduces an overhead that would not be otherwise incurred in a subsystem. The other source of overhead is scope-based dispatch. Whenever a new contract object is created, the scope hierarchies need to be searched for scope descriptors for non-default implementations. The overhead for these two mechanisms is shown in Table 6.1.

The granularity of the timer available on the RS/6000 platform is two microseconds. Hence, the numbers in the first two rows were obtained by extracting the relevant code from the PFS implementation, repeating the respective operations 10000 times with the extracted code, and timing the run. The measurements were averaged over ten runs to obtain the numbers in Table 6.1. The variance was observed to be less than 0.2% of the mean. However, in a large subsystem, scope-based dispatch may introduce a little more overhead due to hardware cache misses.

Table 6.1

The Overhead of Flexibility Mechanisms

Mechanism	Time
Delegation in a contract object	80 nanoseconds
Scope-based dispatch in PFS	5 microseconds
Cost of a `set_impl()` call	2 milliseconds
Cost of a `query_impl()` call	2 milliseconds

The table also shows the costs of two calls introduced by the Pi approach. These calls are used only by the clients using the second interface. Clients using only the default implementations do not have to pay the price of these calls. On the other hand, the overheads in the first two rows are borne by all the clients regardless of whether they desire the

extra control enabled by the Pi approach. Hence, we have been careful about keeping the overhead low for those mechanisms.

The overhead numbers in Table 6.1 become more meaningful when compared with the costs of other operations inside the PFS. Table 6.2 shows the costs of common file system-related operations like opening, reading and writing a file. It shows times for two data sources[2] with significantly different overhead characteristics. The two tables indicate how the costs arising out of the Pi approach compare with costs of subsystem origin.

Table 6.2

The Overhead of Internal File System Operations

Operation	Time in milliseconds	
	JFS	FTP server
Opening a file for reading	0.5	460
Reading a 1kb file	17	416
Reading a 1Mb file	1252	1321
Writing a 1kb file (flushed to disk/server)	46	512
Writing a 1Mb file (flushed to disk/server)	1400	1513

This completes the description of selected experiments performed to demonstrate client control and to measure overheads. Now we can consider the three-fold goal stated at the beginning of this chapter.

6.2.2 Feasibility

The PFS implementation has shown that the PFS architecture is a sound and reasonable file system architecture. Thus, it has shown that the Pi approach can be effectively

2. The column labeled JFS pertains to the Journaling File System, the file system for local disks in AIX. The specific label is used to indicate that the numbers are not for NFS or CD ROM file system which are also accessible through the standard AIX file system interface.

applied to a subsystem. It has employed contracts and scopes, two key ideas from the Pi approach in an existing operating system, AIX, using a mainstream language, C++. Next, we will consider the five points mentioned in Section 2.6.

6.2.3 Assessment Based on the Evaluation

- Separation of client-controllable parts of the implementation: The Pi approach separates client control using contracts. Contract interfaces are defined by the subsystem architecture while clients can replace protocol implementations. For example, in PFS, the framework is defined by the architecture while individual contract implementations like the prefetching implementation are subject to client-control.
- Procedural control: Clients can make calls to select a specific implementation for any of the contracts in a subsystem. In general, they can provide their own implementation for custom needs. The `set_impl()` call used in the file system realizes procedural control.
- Incremental modification to an implementation: The granularity of a change in an implementation is small. A whole subsystem need not be replaced; only specific contract implementations can be changed. For example, only the naming implementation can be changed while reusing the rest of the file system that manages data objects, caching and I/O.
- Scope control for client-requested changes: A change made by a client can be restricted to a certain scope. Scope-based dispatch ensures that a client-selected implementation is used if appropriate. Changing the naming implementation for one client without affecting others is an example of scope-control.
- Negligible overhead when the flexibility features are not used and a modest overhead otherwise: Delegation in contracts and scope-based dispatch impose a negligible overhead. Since they are used by all clients, the Pi approach has emphasized

low overhead in their design. The numbers obtained for the PFS implementation shown in Table 6.1 support this design principle. The cost of procedural control is also modest when compared to the cost of subsystem-specific calls.

6.2.4 Capabilities and Limitations

The design and implementation of PFS helped in understanding the value of the Pi approach beyond verifying the goals discussed above. The Pi approach clarified file system architectures by emphasizing the different design decisions in various architectures. The notion of a second interface borrowed from [Kiczales, 92b] also proved useful. It lead to a small set of generic calls that can be carefully implemented to ensure integrity of the implementation and reasonable performance. Replacement of implementations at the protocol object level is a strength of the approach. The object-sized granularity helps in keeping the implementations internally consistent; e.g. the whole DOM protocol implementation is replaced rather than just the implementation for the read or write operation. But the granularity is also a substantial improvement over server-sized granularity seen in the early microkernel approach.

On the other hand, the Pi approach still leaves several important questions unanswered. It does not provide policies for restricting access to the second interface nor does it deal with authentication issues required to implement those policies. Also, it does not address the issue of protection for client-supplied implementations. Finally, it does not directly provide declarative control to simplify the task of client developers. Many of these limitations form the motivation for future work which is discussed in Chapter 8. But they also relate to some of the concerns expressed about flexible operating systems in conferences and USENET news groups. That is the subject of the next chapter.

7. CAVEATS AND CONCERNS

So far, we have discussed how a client can change a subsystem's implementation. In doing so, we have assumed that the client developer understands the consequences of making a change. In fact, we even explicitly stated that clients are at different levels of sophistication. Some will just use the first interface, others will select an existing implementation, and still others may provide an implementation. Still, it is important to at least briefly discuss the need for caution in the use of flexibility. These areas for caution can be treated as caveats for a client developer and concerns for a flexible subsystem user. This chapter discusses three such areas: *interoperability, dependability* and *complexity.*

Interoperability refers to the ability of a flexible subsystem to function with other subsystems of an operating system in spite of changes. Dependability reflects its capacity to provide certain essential services to clients in spite of changes. Complexity measures the additional cost incurred in making a subsystem flexible. These three properties are further explained below.

The goal of this discussion is to provide a more balanced view of flexibility and to relate it to some of the experience gained during the course of this work. The issues are important in their own right and merit a much more detailed investigation which is outside the scope of this work.

7.1 Interoperability

Consider traversing links in a directory structure in a flexible file system like PFS. Suppose that a client has requested an implementation that keeps track of the symbolic

links being followed so that moving *up* the directory tree (as in `cd ..`) causes the file system to backtrack along the link. For the namespace in Figure 7.1, the client would want to be back in directory **B** after traversing the link **E** (which goes to C**)** and then changing directory to backtrack. Clearly, this behavior is different from the normal behavior of a UNIX file system which would take the client to A rather than B. Navigation up a directory tree is normally independent of the links followed down and is decided exclusively by the parent of the current directory. Now if the client has requested that the change be applied to its user scope, an application using relative path names and running on behalf of the same user would fail. For example, a `make` utility will not be able to interoperate with the flexible file system changed in this fashion. It often uses relative file names and hence expects the parent directory to be independent of the links.

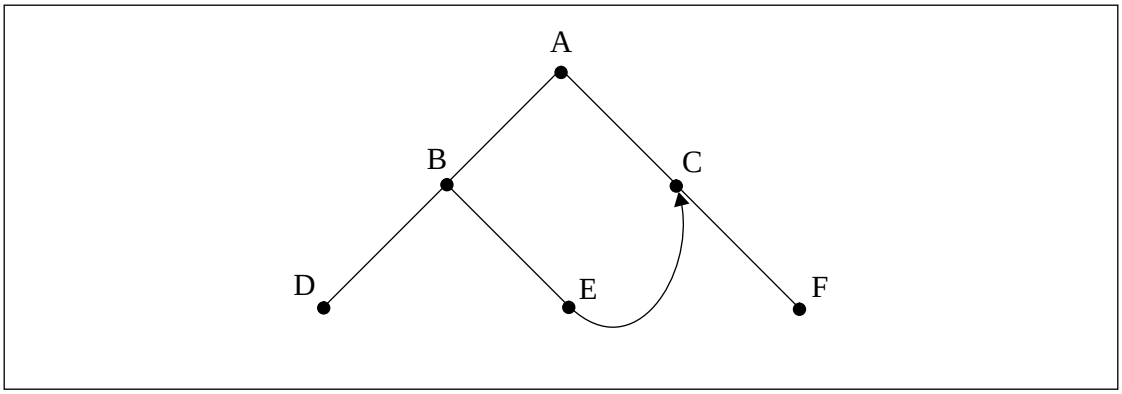

Figure 7.1: Links and Parent Directories in a Namespace

This example illustrates the potential perils of a change. Of course, in this case it is obvious that the client chose a wrong scope for applying changes. But more subtle problems can occur in practice. The gist of the problem is that an application (`make` utility) expected a certain behavior from a subsystem (file system) which was modified; the invariance of certain expected behavior was lost due to flexibility features that allowed a fairly fundamental change. The very fact that many invariances can be challenged by client-control makes the interoperability problem an important one.

7.2 Dependability

Dependability is a particularly important property of a subsystem that is expected to be invariant and hence we will treat it separately. Again, consider a file system that users depend on for longevity of their data. When a machine crashes, the file system on the machine is expected to recover from the crash and restore data to a consistent state. It is widely known that the choice of a caching strategy determines how much data is likely to get lost when a crash occurs. For example, write-back caching delays propagating the changes made to a file to the server for better performance. If there is a crash between the time a `write()` call is successfully completed and before the changes are propagated to the server, the newly written data is lost.

Typically, application writers are aware of this danger implicit in a file system that does caching and take measures to limit the potential damage. However, if the caching implementation can be changed at run-time, say by increasing the period for which updates are not propagated, applications may be subjected to unexpected risks. Thus, the caching implementation selected by a client could adversely affect the dependability of the file system. Similar concerns arise in a communication subsystem regarding reliable message delivery and in the scheduler regarding meeting critical deadlines for processes. Client-control can challenge the fundamental property of dependability in these subsystems.

7.3 Complexity

The third concern is about the complexity[1] of implementations. The functionality demanded from subsystems leads to complex implementations. Flexibility features further add to the complexity. The run-time machinery required for selecting and/or adding implementations in a file system makes it more complex than its non-flexible counterparts. For example, in the PFS implementation, several thousand lines of code are devoted to flexi-

1. Measures such as lines of code, development time are often used to quantify complexity of large subsystems. However, we will refrain from restricting to a specific quantitative metric here.

bility features alone; the code does not implement any file system functionality. Likewise, clients that use the flexibility features also entail greater complexity.

More complexity means greater development effort and more opportunities for bugs and performance bottlenecks. Hence complexity becomes a concern in itself. On the other hand, flexibility requires modularity which improves reliability of an implementation and makes maintenance easier. For example, a new name caching implementation can be easily developed for PFS without worrying about rest of the file system components. Also, some of the flexibility code is likely to be reusable. Thus, complexity is more of a tradeoff concern where the advantages gained through modularity should be used to offset the cost incurred in providing flexibility. The tradeoff in the Pi approach is that scope types cannot be changed by a client and the granularity of implementation change is pegged at contract protocol objects.

7.4 Remedy?

The concerns mentioned above are often raised in the operating system community during the discussion of flexibility. Hence, it is important to pay attention to the caveats and understand the potential problems that flexibility can cause. There is no known remedy to completely solve the problems discussed here but there are measures to reduce the risk.

- Restrictions on the use of flexibility. Policies should restrict the changes a client is allowed to make based on client privileges and the impacts of each change.
- Emphasis on scope-control. Any change should be limited to the smallest appropriate scope.
- Restrictions on the extent of flexibility. A minimum granularity should be required for implementation replacement and certain parts of a subsystem could be deemed immutable.

In summary, flexibility mechanisms are effective when coupled with policies for disciplined change. More extensive experimentation with a number of mechanisms in a variety of situations is required before flexibility can be used in mission critical situations.

8. CONCLUSIONS

Design decisions in system software can be more effective if they are based on application input. This dissertation has proposed the Pi approach, a method to make system software more flexible and, by giving applications disciplined control over software implementations. This chapter summarizes the study, discusses its contributions and suggests directions for future work.

8.1 Summary

The Pi approach is based on the idea of dual interfaces for a subsystem: one for accessing conventional functionality and the other for allowing clients to change the implementation. It emphasizes incrementality to reduce the burden on clients, scope-control to restrict visibility of a change and low overhead mechanisms to reduce the performance penalty. It builds on the recent work in reflective languages and provides a few simple design patterns that can be used for flexible architectures and implementations.

The Pi file system architecture uses the Pi approach to address design choices in file systems. Instead of codifying specific decisions, it provides a framework to accommodate multiple decisions. The actual decisions can be made by a file system implementation at run-time, in collaboration with applications using the file system.

Experience with the Pi file system prototype indicates that flexibility is desirable in file systems and realizable through the PFS architecture. The PFS implementation has enabled clients to obtain new functionality and improve performance by selecting the most appropriate implementation. In particular, naming and caching can be tailored by cli-

ents. At the same time, the overhead of flexibility mechanisms is a small fraction of the cost of providing the required service like reading a file.

8.2 Contributions

This work has taken an important step in making system software flexible. Its contributions can be divided into two categories: approach for flexibility and application to file systems.

The Pi approach has provided two key mechanisms for flexibility: contracts and scopes. These mechanisms are useful for decomposing the functionality of a subsystem and controlling run-time changes by clients.

- *Contracts*: A clean separation between interface and implementation can be achieved using contract objects. Hence, implementations can be substituted at run-time while providing the service promised by the interface.
- *Scopes*: A record of changes to the subsystem implementation can be maintained using scopes. Scope-based dispatch ensures that client-specified implementation choices are used in deciding which implementation should be used for contract objects.

The second area of contribution is file systems. Here the Pi file system architecture provides the following benefits:

- *Modular architecture*: Separation of file system functionality into six components using contracts allows fine-grain control which is not possible in other file system architectures. It also facilitates experimentation by allowing incremental changes.
- *Naming*: Separation of naming from the rest of the file system is a particularly important step in providing customized access to a wide range of data sources. Users can create their own namespaces without being shackled to location-based naming. In addition, the name resolution procedure followed in the PFS architec-

ture is an improvement over the widely used vnode name resolution procedure.

- *Caching*: Caching is a critical component of file system functionality. The PFS architecture has explicit facilities for controlling caching. Hence, applications are no longer restricted by file system designers' decisions about caching.

8.3 Directions for Future Work

This work can be extended in a number of interesting ways. Again, there are two main categories for future work. First, the flexibility approach can be significantly enriched by addressing policies for restricting changes, protection and tool support. File system architecture and implementation can be enhanced by adding virtual memory support and evolving the vnode implementations.

- *Policies for client-control*: A thorough investigation of policies for restricting client-requested changes would help in addressing commonly expressed concerns about flexibility. Experimentation with different policies in various subsystems would be required to understand the problems caused by client-control.
- *Protection*: While the Pi approach allows proxy-based implementations, it does not provide sufficient support for safely adding client-supplied code to a subsystem. A current project being conducted in the Distributed Computing Research Lab is investigating this area [Banerji, 94b].
- *Tool support*: Contract objects can be more easily and widely used if their generation can be automated. SOM emitter frameworks [IBM, 93b] could be used to create tools for automation.
- *Virtual memory support*: The PFS architecture could be enhanced to allow memory mapping of files. Use of virtual memory facilities can substantially improve the efficiency of PFS implementations.
- *Native implementation*s: The PFS architecture could be implemented directly on

hardware. Such a prototype would not use the services of another file system and, hence, is likely to provide better performance.

- *Vnode implementation*: Implementations of the vnode architecture which are widely used in commercial systems could be modified to incorporate PFS features. A second interface, a change in the naming component, support for contracts and the addition of scope-based dispatch would be required to accomplish the modification.

APPENDIX A. IMPLEMENTATION SUMMARY

The details of the implementation discussed in Chapter 6 are summarized here. The organization of this chapter is based on Figure 6.1 on page 111. The components of the PFS framework and their interoperation are discussed in the following sections. The discussion uses brief introductory text and C++ code annotated with comments for explanation.

A.1 Framework

A.1.1 Dual Interfaces

As per the PFS architecture, the prototype has two interfaces reified as interface objects. The class definitions for these objects, `first_intf` and `second_intf` are shown below. The first interface object provides access to the basic file system functionality. The second interface object provides services for clients to change implementations and create new namespaces and file systems. It also provides functions to add and remove client scopes which are used by the support services component.

Most functions provided by the interface classes have a parameter for passing client credentials. The type `cred` defines the credentials for a client and contains information about client scopes: group, user and process.

```
class first_intf
{
public:
            // constructor needs a pointer to the outermost scope
            first_intf(SubsystemScope* ssp);

            // data object related functions
```

```
            int open(char* filename, int flags, cred* cp);
            int close(int fd, cred* cp);
            int read(int fd, void* buf, int size, cred* cp);
            int write(int fd, void* buf, int size, cred* cp);
            int seek(int fd, int offset, int from, cred* cp);

            int get(char* filename, void* buf, int size, cred* cp);
            int put(char* filename, void* buf, int size, cred* cp);

            // naming related functions

            // basic naming functions
            int resolve(Pathname*, Context**);
            int bind(Pathname*, Context*);

            // link-supported function - read the name pointed to
            int readlink(Pathname* source_path, Pathname* dest_path);

            // directory-supported function - list directory entries
            int readdir(Pathname* dir_path, dirent_list* entry_list);

            // namespace-supported functions
            int mount(Pathname* mount_here, Context* mount_this);
            int unmount(Pathname* unmount_from, Context* unmount_this);

private:
            // function for getting the AP contract
            APContract* get_APContract(int fd, cred* cp, client_descr*
            crp);
            // use a member instead of a global variable
            SubsystemScope* subsystemscope;
};


class second_intf
{
public:
            // constructor needs a pointer to the outermost scope
            second_intf(SubsystemScope* top_scope_ptr);

            // scope-related functions

            // create/remove client scopes
            int addClient(cred* client_credp);
            int remClient(cred* client_credp);

            // construct a new file system
            int newfs(int fstype_id, void* fs_parm, cred* client_credp);
            // construct a new namespace
            int newns(int nstype_id, void* ns_parm, cred* client_credp);

            // contract related functions
            int set_impl(scope_descr* sdp, contract_descr* cdp, cred*
            client_credp);
```

```
            int query_impl(scope_descr* sdp, contract_descr* cdp, cred*
            client_credp);

private:
            // use a member instead of a global variable
            SubsystemScope* subsystemscope;
};
```

The interface objects, one for each class, are instantiated when the PFS implementation is initialized. Support service component is responsible for their instantiation.

A.1.2 Scopes

The scopes discussed in Section 5.2 are implemented as classes. All scope classes are derived from the base class `BaseScope` which provides the implementation for common functions. The implementations are then reused or specialized in the derived classes.

```
class BaseScope
{
public:
            // constructor sets the default implementations
            BaseScope(int default_impl=INVALID_CID)

            // navigation through the scope hierarchy
            virtual BaseScope* get_parent();

            // identify this scope in the hierarchy
            virtual void id(scope_descr& sd);

            // get the contract implementation id
            int get_contract(int contract_name);
            // set the implementation id for the specified contract
            // cd is the contract descriptor
            int set_contract(contract_descr cd);

private:
            // maintain a list of contract implementations
            // used to implement get_contract()
            ContractList*           clp;
};
```

The derived scope classes are linked together in a tree structure for easy navigation through the scope hierarchy. Client scopes and the unit scope `File`, are also hashed for faster access than would be permitted by the linked list-based tree structure.

A.1.3 Contract Interfaces

Contract interface objects are instances of contract interface classes. The class definitions have been discussed in Chapter 5 in the context of the PFS architecture. The common functionality used by all contract interface classes is captured in the class `MetaContract`. `MetaContract` is an abstract class; a derived class must be used for creating instances.

```
class MetaContract
{
  public:
            // set and get the implementation for this contract
            virtual int set_impl(ImplId) = 0;
            virtual int query_impl(ImplId*) = 0;
};


class ContractList
{
public:
            // default constructor; set all impl. ids to invalid value
            ContractList()
            // set impl. ids to default_impl
            ContractList(int default_impl)
            // set impl. id for a specific contract
            ContractList(contract_descr cd)

            // sometimes the default copy constructor will get used
            ContractList(const ContractList& cl)

            // accessor functions - get/set the contract impl. id
            int get(int contract_name) const
            int set(contract_descr cd)

private:
            // array for maintaining impl. ids; one per contract
            int         descr_array[LAST_NAME+1];
};
```

A.1.4 Support Services

Support services allow the other framework components to utilize AIX services and interact with clients. They are responsible for configuration of a PFS implementation, its

initialization and client communication. As such, they are not defined in the Pi model and hence are very implementation specific.

Configuration refers to the selection of features for a PFS. Several PFS implementation features such as the number of processes used, the mode of communication between the processes, and the presence of a kernel-resident VFS component can be determined while starting a PFS. Support services use a configuration file to determine these features and initialize a PFS accordingly.

Initialization involves the creation of key PFS framework and AIX entities. Dual interface objects, subsystem scope and related data structures are the framework entities that need to be created before any client requests can be serviced. Processes, System V IPC message queues, shared memory and semaphores are the main AIX entities that need to be created.

Communication allows a PFS to interact with its clients. There are two parts of the communication support services. One part is a set of libraries that are linked with the client code; it creates message queues for the client process and provides wrappers for the calls supported by the dual interfaces. It frees clients from the details of routine message exchanges with the PFS server. The other part is linked with the PFS code and manages communication at the server end.

Frameworks based on the Pi model, as in PFS, could be created for communication and process management. The support services would then consist of a much smaller glue code between cooperating frameworks. However, the current implementation does not provide such a flexible structure. Instead, it uses a conventional implementation for support services while concentrating on the file system framework.

A.2 Protocol Implementations

While the framework provides flexibility to choose alternate implementations, protocol implementations provide the file system functionality. They support the interfaces discussed in Chapter 5; these interfaces are exactly the same as the first interfaces of contract interface classes.

Protocol implementations embody the peculiarities of the protocols they support. For example, the I/O protocol implementation for FTP is are quite different from that for IMAP. Hence, they are custom developed for specific design decisions such as caching upon read or reversible traversal of links.

LIST OF REFERENCES

[Agha, 87] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press.

[Anderson, 91] T. Anderson, H. Levy, B. Bershad & E. Lazowska, The Interaction of Architecture and Operating System Design, *Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS IV), ACM, pp. 108-120.

[Anderson, 92] T. Anderson, B. Bershad, E. Lazowska & H. Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM TOCS*, 10(1), Feb. 1992, pp. 53-79.

[Bach, 86] M. Bach, *The Design of the UNIX Operating System*, Prentice Hall, NJ.

[Bahrs, 92] P. Bahrs, D. Moreau & W. Dominick, GO||: An Object-Oriented Framework for . Concurrent Graphics Systems, *Computer Graphics Using Object-Oriented Programming*, Ed. S. Cunningham et. al., John Wiley and Sons, 1992.

[Bahrs, 94] P. Bahrs, IBM Workplace Technology, Workshop on Flexibility in System Software, OOPSLA ‘94.

[Banerji, 94a] A. Banerji, D. Cohn, D. Kulkarni & J. Tracey, *Efficient Kernel-Level/ User-Level Interaction*, Tech. Report 94-19, Dept. of Computer Science & Eng., University of Notre Dame, IN.

[Banerji, 94b] A. Banerji & D. Cohn, Protected Shared Libraries, Tech. Report 94-37, Dept. of Computer Science & Engineering, University of Notre Dame, Notre Dame, IN.

[Berners-Lee, 93] T. Berners-Lee, R. Cailliau, J. Groff, B. Pollermann, World-Wide Web: The Information Universe, Electronic Networking: Research, Applications and Policy, 2(1), Meckler Publishing, CT, pp. 52-58; also available as ftp://info.cern.ch/pub/www/doc/ENRAP_9202.ps.

[Bernstein, 87] P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, MA.

[Blaze, 93] M. Blaze, Transparent Mistrust: OS Support for Cryptography-in-the- Large, *Proc. Fourth Workshop on Workstation Operating Systems*, IEEE, pp. 98-103.

[Campbell, 91] R. Campbell et. al. Choices, Frameworks and Refinement, *Proc. Intl. Workshop on Object Orientation in Operating Systems*, IEEE, pp. 9-15.

[Carroll, 89] J. Carroll & D. Long, *Theory of Finite Automata with an Introduction to Formal Languages*, Prentice Hall, NJ.

[Chiba, 93] S. Chiba & T. Masuda, Designing and Extensible Distributed Language with a Meta-Level Architecture, *European Conference on Object-Oriented Programming (ECOOP*

‘93), Springer Verlag, LNCS - 707, pp. 482-501.

[Comer, 89] D. Comer & L. Peterson, Understanding Naming in Distributed Systems, *Distributed Computing*, 3(2), Springer Verlag, pp. 51-60.

[Coplien, 92] J. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison Wesley, MA.

[Crispin, 90] M. Crispin, *Interactive Mail Access Protocol: Version 2*, Internet RFC 1176.

[Crispin, 93] M. Crispin, IMAP Source Distribution Version 3.0, ftp://ftp.cac.washington.edu/mail/imap.tar.Z.

[Crowcroft, 92] J. Crowcroft, I. Wakeman, Z. Wang & D. Sirovica, Is Layering Harmful? *IEEE Network Magazine*, Jan. 92, pp. 20-24.

[Deitel, 91] H. Deitel & M. Kogan, *The Design of OS/2*, Addison Wesley.

[Delaney, 89] W. Delaney, *The ARCADE Distributed Environment: Design, Implementation and Analysis*, Ph.D. Dissertation, Dept. of Electrical Eng., University of Notre Dame, IN.

[Demers, 88] R. Demers, Distributed Files for SAA, *IBM Systems Journal*, 27(3), pp. 348-361.

[Deutsch, 89] L. Deutsch, Design, Reuse and Frameworks in the Smalltalk-80 System, *Software Reusability, Volume II*, Ed. E. Biggerstaff & A. Perlis, ACM Press. pp. 57-71.

[Emtage, 92] A. Emtage & L. Deutsch, Archie: An Electronic Directory Service for the Internet, *Proc. USENIX Winter ‘92 Conference*, pp. 93-110.

[Fabry, 74] R. Fabry, Capability-Based Addressing, *Communications of the ACM*, 17(7), July 1974, pp. 403-412.

[Forin, 89] A. Forin, J. Barrera & R. Sanzi, The Shared Memory Server, *Proc. Winter ‘89 USENIX Conference*, USENIX Association, pp. 229-243.

[Franklin, 92] J. Franklin, Tiled Virtual Memory for Unix, Proc. USENIX Summer ‘92 Conference, USENIX Association, CA, pp. 99-106.

[Golub, 89] D. Golub, R. Dean, A. Forin & R. Rashid, UNIX as an Application Program, Proc. Summer USENIX, June 1990, pp. 87-96.

[Golub, 93] D. Golub, R. Manikundalam & F. Rawson, MVM - An Environment for Running Multiple DOS, Windows and DPMI Programs on the Microkernel, *Proc. USENIX Mach III Symposium*, Apr. 1993, pp. 173-190.

[Goodheart, 94] B. Goodheart & J. Cox, The Magic Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design, Prentice Hall.

[Guy, 90] R. Guy et. al., Implementation of the Ficus Replicated File System, Proc. Summer USENIX ‘90, USENIX Association, CA, pp. 63-71.

[Halasz, 94] F. Halasz & M. Schwartz, The Dexter Hypertext Reference Model, Communications of the ACM, 37(2), Feb. 1994, pp.30-39.

[Hamilton, 93a] G. Hamilton & P. Kougiouris, The Spring Nucleus: A Microkernel for Objects, Proc. Summer ‘93 USENIX Conference, USENIX Association, CA.

[Hamilton, 93b] G. Hamilton, M. Powell, J. Mitchell, Subcontracts: A Flexible Base for Distributed Programming, Fourteenth ACM Symposium on Operating Systems Principles, Operating Systems Review, 27(5), Dec. 1993, pp. 69-79.

[Harty, 92] K. Harty & D. Cheriton, Application-Controlled Physical Memory Using External Page-Cache Management, *Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 187-197.

[Hauser, 92] C. Hauser, A Plea for Interfaces to Support Caching, Proc. Third Workshop on Workstation Operating Systems, IEEE, pp. 137-140.

[Heidemann, 93] J. Heidemann & G. Popek, File System Development with Stackable Layers, ACM TOCS, 12(1), Feb. 1994, pp. 58-89.

[Helm, 90] R. Helm, I. Holland & D. Gangopadhyay, Contracts: Specifying Behavioral Composition in Object-Oriented Systems, Proc. OOPSLA ‘90, pp. 169-180.

[Holzmann, 91] G. Holzmann, Design and Validation of Computer Protocols, Prentince Hall, NJ.

[Hook, 93a] G. Hook, Linking and Binding, Course Notes, IBM AIX Systems Center, Dallas, TX.

[Hook, 93b] G. Hook, Advanced Linking, Course Notes, IBM AIX Systems Center, Dallas, TX.

[Howard, 88] J. Howard et. al., Scale and Performance in a Distributed File System, ACM TOCS, 6(1), Feb. 1988, pp. 51-81.

[Hunt, 92] R. Hunt, CCITT X.500 Directories - Principles and Applications, Computer Communications, 15(10), Dec. 1992, Butterworth Heinemann Ltd., UK, pp. 636-645.

[Huston, 93] L. Huston & P. Honeyman, Disconnected Operation for AFS, *USENIX Mobile and Location Independent Computing Symposium*, USENIX Association, CA, pp. 1-10.

[Hutchinson, 91] N. Hutchinson & L. Peterson, The x-Kernel: An Architecture for Implementing Network Protocols, *IEEE Trans. Software Eng.*, 17(1), Jan. 1991, pp. 64-75.

[IBM, 92] *Writing a Device Driver for AIX Version 3.2*, Second Edition, IBM Corporation, Part No. GG24-3629-01

[IBM, 93a] *SOMObjects Developer Toolkit Users Guide*, Version 2.0, June 1993, IBM Corporation.

[IBM, 93b] *SOMObjects Developer Toolkit Emitter Framework Guide and Reference*, Version 2.0, June 1993, IBM Corporation.

[Julin, 91] D. Julin et. al., Generalized Emulation Services for Mach 3.0 - Overview, Experiences and Current Status, *Proc. USENIX Mach Symposium*, Nov. 1991, pp. 13-26.

[Kiczales, 91] G. Kiczales, J. des Rivieres & D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press.

[Kiczales, 92a] G. Kiczales, M. Theimer & B. Welch, A New Model of Abstraction for Operating System Design, *Proc. Intl. Workshop on Object-Oriented Operating Syst. (IWOOOS)*, IEEE, pp. 346-350.

[Kiczales, 92b] G. Kiczales, Towards a New Model of Abstraction in the Engineering of Soft-

ware, *Proc. International Workshop on New Models for Software Architecture, Reflection and Meta-Level Architectures*, Research Institute in Software Engineering, Tokyo, pp. 1-11.

[Kiczales, 93] G. Kiczales & J. Lamping, Operating Systems: Why Object-Oriented?, *Proc. Intl. Workshop on Object-Orientation in Operating Systems*, IEEE, pp. 25-30.

[Kistler, 93] J. Kistler, *Disconnected Operation in a Distributed File System*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, PA.

[Kleiman, 86] Vnodes: An Architecture for Multiple File System Types in Sun UNIX, *Proc. Summer '86 USENIX*, USENIX Association, CA, pp. 238-247.

[Krueger, 93] K. Krueger, D. Loftesness, A Vahdat & T. Anderson, Tools for the Development of Application Specific Virtual Memory Management, *Proc. OOPSLA '93*, ACM, pp. 48-64.

[Kulkarni, 93] D. Kulkarni, A. Banerji, M. Casey & D. Cohn, Structuring Distrubuted Shared Memory with the Pi Architecture, *Proc. 13th Intl. Conference on Distributed Computing Syst.* (ICDCS), IEEE, pp. 93-100.

[Lampson, 79] B. W. Lampson & R. F. Sproull, An Open Operating System for a Single-User Machine, *Proc. 7th Symposium on Operating Systems Principles*, ACM, Operating Systems Review, 13(5), pp. 98-105.

[Leffler, 89] S. Leffler, M. McKusick, M. Karels & J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison Wesley, MA.

[Lieberman, 86] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, *Proc. OOPSLA '86*, ACM, pp. 214-223.

[Loepere, 92] K. Loepere, *Mach 3 Kernel Principles*, Revision NORMA-MK12:July 15, 1992, Open Software Foundation, Cambridge, MA.

[Maes, 87a] P. Maes, Concepts and Experiments in Computational Reflection, *Proc. OOPSLA '87*, ACM, pp. 147-155.

[Maes, 87b] P. Maes, *Computational Reflection*, Ph.D. dissertation, Tech Report 87-2: Vrije Universiteit Brussel, Belgium.

[Meyer, 88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall.

[Neuman, 92] B. Neuman, *The Virtual System Model: Scalable Approach to Organizing Large Systems*, Ph.D. Dissertation, Dept. of Computer Science and Eng., University of Washington, WA.

[Nitzberg, 91] W. Nitzberg & V. Lo, Distributed Shared Memory: A Survey of Issues and Algorithms, *IEEE Computer*, 24(8), pp. 52-60.

[Ousterhout, 88] J. Ousterhout & F. Douglis, *Beating the I/O Bottleneck: A Case for Log-Structured File Systems*, Technical Report No. UCB/CSD 88/467, Computer Sciences Division, University of California, Bekeley, CA.

[Palay, 92] A. Palay, C++ in a Changing Environment, *USENIX C++ Technical Conference*, USENIX Association, pp.195-206.

[Peterson, 90] L. Peterson, N. Hutchinson, S. O'Malley & H. Rao, The x-kernel: A Platform for Accessing Internet Resources, *IEEE Computer*, 23(5), May 1990, pp. 23-33.

[Pike, 93] R. Pike et. al., The Use of Namespaces in Plan 9, *Operating Systems Review*, 27(2),

Apr. 1993, ACM, pp. 72-76.

[Postel, 85] J. Postel & J. Reynolds, *File Transfer Protocol*, Intenet RFC 959.

[Rashid, 91] R. Rashid et. al., Mach: A Foundation for Open Systems, *Proc. Workshop on Workstation Operating Systems*, IEEE.

[Ritchie, 74] D. Ritchie & K. Thompson, The UNIX Time-Sharing System, *Communications of the ACM*, 17(7), July 1974, pp. 365-375.

[Rosenthal, 90] D. Rosenthal, Evolving the Vnode Interface, *Proc. USENIX Summer '90 Conference*, USENIX Association, CA, pp. 107-117.

[Saltzer, 84] J. Saltzer, D. Reed & D. Clark, End-to-End Arguments in System Design, *ACM TOCS*, 2(4), Nov. 1984, pp. 277-288.

[Sandberg, 85] R. Sandberg et. al., Design and Implementation of the Sun Network Filesystem, *Proc. USENIX Summer '85 Conference*, USENIX Association, CA, pp. 119-130.

[Shah, 93] J. Shah, *VAX C Programmer's Guide*, McGraw Hill Inc.

[Shapiro, 86] M. Shapiro, Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Proc. Sixth Intl. Conference Distributed Computing Syst.* (ICDCS), pp. 198-204.

[Steere, 92] D. Steere, J. Kistler & M. Satyanarayanan, Efficient User-Level File Cache Management on the Sun Vnode Interface, *Proc. USENIX Summer '90 Conference*, pp. 325-331.

[Steiner, 88] J. Steiner, B. Neuman & J. Schiller, Kerberos: An Authentication Service for Open Network Systems, *Proc. Winter '88 USENIX Conference*, USENIX Association, pp. 191-201.

[Stonebraker, 85] M. Stonebraker, D. DuBourdieux, W. Edwards, Problems in Supporting Database Transactions in an Operating System Transaction Manager, *Operating Systems Review*, 19(1), Jan. 1985, pp. 6-14.

[Stroustrup, 91] B. Stroustrup, *The C++ Programming Language*, Addison Wesley.

[Ungar, 87] D. Ungar & R. Smith, Self: The Power of Simplicity, *Proc. OOPSLA '87*

[Wand, 88] M. Wand & D. Friedman, The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower, *Meta-Level Architectures and Reflection*, Ed. P. Maes & D. Nardi, Elsevier Science Publishers, pp. 111-134.

[Weihl, 89] W. Weihl, *The Impact of Recovery on Concurrency Control*, Tech. Report TM-382.b, Laboratory for Computer Science, MIT, MA.

[Welch, 90] B. Welch, *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*, Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley.

[Welch, 93] B. Welch, A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9, *Computing Systems*, 7(2), Spring 1994, USENIX Association, pp. 175-199.

[Wulf, 74] W. Wulf et. al. HYDRA: The Kernel of a Multiprocessor Operating System, *Communications of the ACM*, 17(6), June 1974, pp. 337-345.

[Wulf, 81] W. Wulf, R. Levin & S. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw Hill.

[Yokote, 91] Y. Yokote et. al., The Muse Object Architecture: A New Operating System Structuring Concept, *Operating Systems Review*, 25(2), April 1991, pp. 22- 45.

[Yokote, 92a] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, *Proc. OOPSLA '92*, ACM, pp. 414-434.

[Yokote, 92b] Y. Yokote, A New Mechanism for Object-Oriented System Programming, *Proc. International Workshop on New Models for Software Architecture, Reflection and Meta-Level Architectures*, Research Institute in Software Engineering, Tokyo, pp. 88-93.

[Young, 89] M. Young, *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, PA.