An Overview of the Spring System

James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia

Sun Microsystems Inc.
2550 Garcia Avenue, Mountain View Ca 94303.

Abstract

Spring is a highly modular, distributed, object-oriented operating system. This paper describes the goals of the Spring system and provides overviews of the Spring object model, the security model, and the naming architecture. Implementation details of the Spring microkernel, virtual memory system, file system, and UNIX emulation are supplied.

1 Introduction

What would you do if you were given a clean sheet on which to design a new operating system? Would you make the new OS look the same as some existing system or different?

If you choose to make it look like UNIX, for example, then a better implementation had better be a primary goal. Changing the system as seen by application programs would, however, be a very bad thing to do, since you are supposedly making it look the same as UNIX in order to run existing software. In fact, if you take this route, you will be strongly pressured to make the new, improved system *binary compatible* with the existing one so that users can run all their existing software. Any new functionality that you would like to include would have to be done as a strict addition to the system's existing Application Programming Interfaces (APIs).

If you choose to make the new system different than any existing system, then you had better make it such an improvement over them that programmers will be willing to learn a new set of APIs to take advantage of its improved functionality. Indeed, you will have to convince other companies to adopt and support your new APIs so that there will be sufficient future sales of systems with the new APIs to warrant software development investments by application developers.

Because the opportunity to begin afresh in OS design is increasingly rare, the Spring project has chosen to be different and to develop the best technology we could. However, we decided that we would innovate only where we could achieve large increases over existing systems and that we would try to keep as many as possible of their good features.

What are the biggest problems of existing systems? From Sun's point of view as a supplier of UNIX system technology in our Solaris products, the major issues are:

- the cost of maintaining, evolving, and delivering the system, including kernel and non-kernel code (e.g., window systems),
- a basis for security that is not particularly flexible, easy to use or strongly secure,
- the difficulty of building distributed, multi-threaded applications and services,
- the difficulty of supporting time-critical media (audio and video), especially in a networked environment,
- the lack of a unified way of locating things by name (e.g., lookup is done differently for files, devices, users, etc.).

However, we wanted to keep a number of features that have proven themselves in one or more systems; for example,

- good performance on a wide variety of machines, including multi-processor systems,
- memory protection, virtual memory, and mapped file systems,
- access to existing systems via application compatibility and network interoperability (e.g., standard protocols and services),

- window systems and graphical user interfaces.

Sun's belief in open systems means that we would like to include *extending the system* by more than one vendor as an important aspect of *evolving* it.

When we looked at these lists, we immediately decided that the Spring system should have a strong and explicit architecture: one that would pay attention to the interfaces between software components, which is really how a system's structure is expressed. Our architectural goal for Spring then became

- Spring's components should be defined by *strong interfaces* and it should be *open, flexible and extensible*

By a strong interface we mean one that specifies *what* some software component does while saying very little about *how* it is implemented

This way of stating our purpose led us to develop the idea of an Interface Definition Language (IDL) [15] so that we could define software interfaces without having to tie ourselves to a single programming language, which would have made the system less open. We also believed that the best way to get many of the system properties we wanted was to use an object-oriented approach.

The marriage of strong interfaces and object-orientation has been a natural and powerful one. It helps achieve our goals of openness, extensibility, easy distributed computing, and security. In particular, it has made the operation of invoking an operation on an object one that is *type safe*, *secure* (if desired), and *uniform* whether the object and its client are collocated in a single address space or machine or are remote from one another.

We have used a microkernel approach in concert with IDL interfaces. The Spring Nucleus (part of the microkernel) directly supports secure objects with high speed object invocation between address spaces (and by a system extension, between networked machines). Almost all of the system is implemented as a suite of *object managers* (e.g., the file system, which provides file objects) running in non-kernel mode, often in separate address spaces, to protect themselves from applications (and from one another). Consequently, it is as easy to add new system functionality as it is to write an application in Spring, and all such functionality is inherently part of a distributed system.

Object managers are themselves objects: for example, a file system is an object manager that supports an operation for opening files by name. The file objects that it returns from *open* operations are generally implemented as part of the same object manager because it is convenient and natural to do so. Because of the similarity of an object manager and the traditional notion of a *server* (e.g., a file server), we use the two terms interchangeably in this paper.

The remainder of this paper will discuss

- IDL,
- the model and implementation of objects in Spring,
- the overall structure of the Spring system,
- the Spring Nucleus,
- the implementations of distributed object invocation, security, virtual memory, file systems, UNIX compatibility, and unified naming.

We will finish by drawing some conclusions from our experience designing and implementing the system.

2 Interface Definition Language (IDL)

The Interface Definition Language developed by the Spring project is substantially the same as the IDL that has been adopted by the Object Management Group as a standard for defining distributed, object-oriented software components. As such, IDL "compilers" are or have been implemented by a number of companies.

What does an IDL compiler do? After all, interfaces are not supposed to be implementations, so what is there to compile? Typically, an IDL compiler is used to produce three pieces of source code in some chosen target implementation language, e.g., C, C++, Smalltalk, etc.:

1. *A language specific form of the IDL interface*: For C and C++ this is a header file with C or C++ definitions for whatever methods, constants, types, etc. were defined in the IDL interface. We will give an example below.
2. *Client side stub code*: Code meant to be dynamically linked into a client's program to access an object that is implemented in another address space or on another machine.
3. *Server side stub code*: Code to be linked into an object manager to translate incoming remote object invocations into the run-time environment of the object's implementation.

These three outputs from an IDL compiler enable clients and implementations in a particular language, e.g., C++, to treat IDL-defined objects as if they were just objects in C++. Thus, a programmer writing in C++ would use an IDL-to-C++ compiler to get C++ header files and stub code to define objects as if they were implemented in C++. At the same time, the object's implementation might be written in C and would, therefore, have used an IDL-to-C compiler to generate the server side stub code to transform incoming calls into corresponding C procedure invocations on the C "objects" corresponding to the IDL objects.

2.1 An example

To give the flavor of IDL, Figure 1 shows an example of IDL use to define the Spring IO interface. For the purposes of this overview, details have been elided, but the example is derived from an actual use of IDL.

```
interface io {
    raw_data read(in long size) raises (access_denied, alerted,
                                        failure, end_of_data)

    void write(in raw_data data) raises (access_denied, alerted,
                              incomplete_write, failure, end_of_data)
};
```

FIGURE 1. IO Interface in IDL

The interface defines objects of type *IO*. In this example, any *IO* object has two operations defined on it, *read* and *write*. The *read* operation takes a parameter, *size*, of type *long*, and returns an object of type *raw_data*. The *write* method returns nothing (*void*) and takes a single argument, *data*, whose type is *raw_data.*

As noted above, instead of returning normally, a method may raise one of a number of defined exceptions.

A complete description of IDL is given in [15].

3 Objects in Spring

Although all Spring interfaces are defined in IDL, IDL says nothing about how operations on an object are implemented, or even how an operation request should be conveyed to an object.

The users of an object merely invoke operations defined in its interface. How and where the operation is actually performed is the responsibility of the object run-time and of the object implementation. Sometimes the operation will be performed in the same address space as the client, sometimes in another address space on the same machine, sometimes on another machine.

We will often use the phrase "invoke an object" as a short form for "invoke an operation on an object".

3.1 Server-based objects

Many objects in Spring are implemented in servers that are in different address spaces from their clients. We provide special support for these kinds of objects by automatically generating *stubs* (see Section 2) which take the arguments for these calls and marshal them for transmission to the server and which unmarshal any results and return these to the client application. These stubs use our subcontract mechanism (see Section 3.3) to communicate with the remote server.

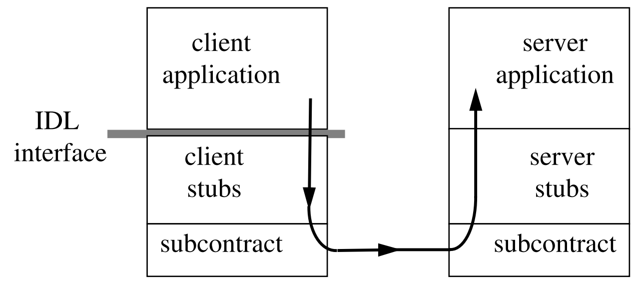

FIGURE 2. A call on a server-based object

Typically, server-based objects use the Spring *doors* communication mechanism (see section 5.1) to communicate between the client and the server. Most subcontracts optimize the case when the client and the server happen to be in the same address space by simply performing a local call, rather than calling through the kernel.

3.2 Serverless objects

Spring also supports serverless objects, where the entire state of the object is always in the client's address space. This implementation mechanism is suitable for lightweight objects such as names or raw_data. When a serverless object is passed between address spaces, the object's state is copied to the new address space. Thus passing a serverless object is more like passing a struct, while passing a server-based object is more like passing a pointer to its remote state.

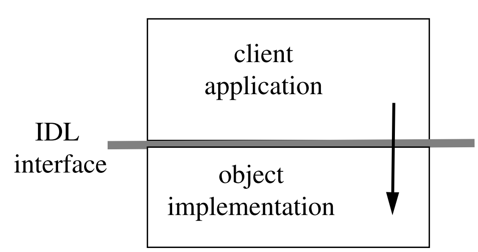

FIGURE 3. A call on a serverless object

3.3 Subcontract

Spring provides a flexible mechanism for plugging in different kinds of object runtime machinery. This mechanism, known as *subcontract,* allows control over how object invocation is implemented, over how object references are transmitted between address spaces, how object references are released, and similar object runtime operations [2].

For example, the widely used *singleton* subcontract provides simple access to objects in other address spaces. When a client invokes a singleton object, the subcontract implements the object invocation by transmitting the request to the address space where the object's implementation lives.

We have also implemented subcontracts that support replication. These subcontracts implement object invocation by transmitting a request to one or more of a set of servers that are conspiring to support a replicated object.

In addition we have used subcontract to implement a number of different object runtime mechanisms, including support for cheap objects, for caching, and for crash recovery.

4 Overall system structure

Spring is organized as a microkernel system. Running in kernel mode are the *nucleus*, which manages processes and inter-process communication, and the *virtual memory manager*, which controls the memory management hardware. The nucleus is entered by a trap mechanism. The virtual memory manager responds to page faults but also provides objects that interact with external pagers (see Section 8.2) and, in this guise, looks like any other object server.

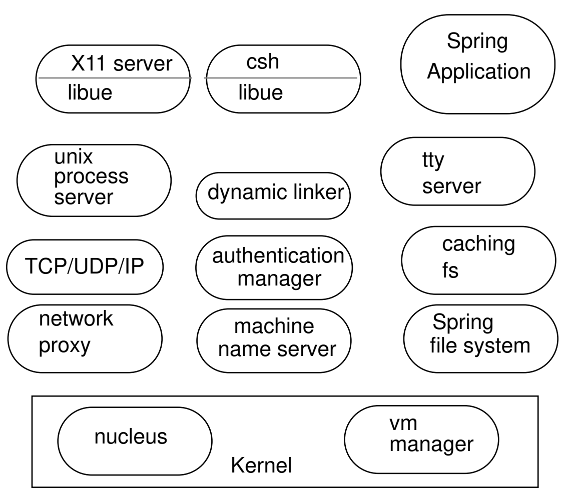

FIGURE 4. Major system components of a Spring node

All other system services, including naming, paging, network IO, filesystems, keyboard management, etc., are implemented as user-level servers. These servers provide object-oriented interfaces to the resources they manage and clients communicate with system servers by invoking these objects.

Spring is inherently distributed. All the services and objects available on one node are also available on other nodes in the same distributed system.

5 The nucleus

The nucleus is Spring's microkernel. It supports three basic abstractions: *domains*, *threads*, and *doors* [1].

Domains are analogous to processes in Unix or to tasks in Mach. They provide an address space for applications to run in and act as a container for various kinds of application resources such as threads and doors.

Threads execute within domains. Typically each Spring domain is multi-threaded, with separate threads performing different parts of an application.

Doors support object-oriented calls between domains. A door describes a particular entry point to a domain, represented by both a PC and a unique value nominated by the domain. This unique value is typically used by the object server to identify the state of the object; e.g., if the implementation is written in C++ it might be a pointer to a C++ object.

5.1 Doors

Doors are pieces of protected nucleus state. Each domain has a table of the doors to which it has access. A domain refers to doors using *door identifiers*, which are mapped through the domain's door table into actual doors. A given door may be referenced by several different door identifiers in several different domains.

Possession of a valid door gives the possessor the right to send an invocation request to the given door.

A valid door can only be obtained with the consent of the target domain or with the consent of someone who already has a door identifier for the same door.

As far as the target domain is concerned, all invocations on a given door are equivalent. It is only aware that the invoker has somehow acquired an appropriate door identifier. It does not know who the invoker is or which door identifier they have used.

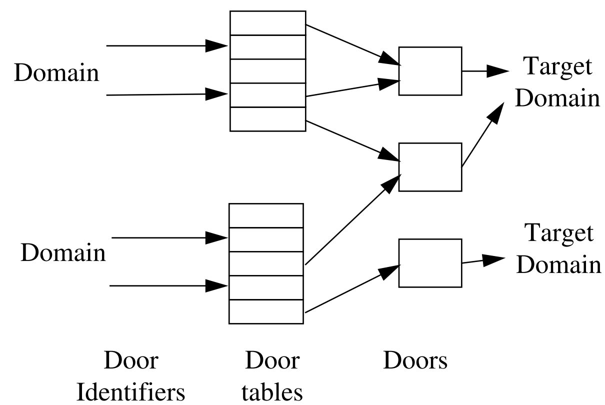

FIGURE 5. Domains, doors, and door tables

5.2 Object Invocation Via Doors

Using doors, Spring provides a highly efficient mechanism for cross-address-space object invocation. A thread in one address space can issue a door invocation for an object in another address space. The nucleus allocates a server thread in the target address space and quickly transfers control to that thread, passing it information associated with the door plus the argument data passed by the calling thread.

When the called thread wishes to return, the nucleus deactivates the calling thread and reactivates the caller thread, passing it any return data specified by the called thread.

For a call with minimal arguments, Spring can execute a low-level cross-address-space door call in 11 µs on a SPARCstation 2, which is significantly faster than using more general purpose inter-process communication mechanisms [1].

Doors can be passed as arguments or results of calls. The nucleus will create appropriate door table entries for the given doors in the recipient's door table and give the recipient door identifiers for them.

6 Network Proxies

To provide object invocation across the network, the nucleus invocation mechanism is extended by *network proxies* that connect up the nuclei of different machines in a transparent way. These proxies are normal user-mode server domains and receive no special support from the nucleus. One Spring machine might include several proxy domains that speak different network protocols.

Proxies transparently forward door invocations between domains on different machines. In Figure 6, when a client on machine B invokes door Y, this door invocation goes to network proxy B; B forwards the call over the net to its buddy, proxy A; proxy A does a door invocation; and the door invocation then arrives in the server domain.

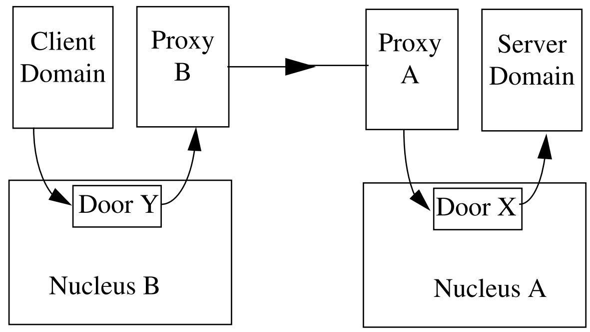

FIGURE 6. Using proxies to forward a call between machines

Notice that neither the client nor the server need be aware that the proxies exist. The client just performs a normal door invocation, the server just sees a normal incoming door invocation.

Door identifiers are mapped into *network handles* when they are transmitted over the network, and are mapped back into doors when they are received from the network.

A network handle contains a network address for the creating proxy, and a set of bits to identify a particular door that is exported by this proxy. In theory the set of bits is large enough to make it hard for a malicious user to guess the value of a network handle, thereby providing protection against users forging network handles.

7 Spring's security model

One of Spring's goals is to provide secure access to objects, so that object implementations can control access to particular data or services. To provide security we support two basic mechanisms, Access Control Lists and software capabilities.

Any object can support an Access Control List (ACL) that defines which users of groups of users are allowed access to that object. These Access Control Lists can be checked at runtime to determine whether a given client is really allowed to access a given object.

When a given client proves that it is allowed to access a given object, the object's server creates an *object reference* that acts as a software capability. This object reference uses a nucleus door as part of its representation so that it cannot be forged by a malicious user. This door points to a *front object* inside the server. A front object is *not* a Spring object, but rather whatever the server's language of implementation defines an object to be.

A front object encapsulates information identifying the principal (e.g., a user) to which the software capability was issued and the access rights granted to that principal.

A given server may create many different front objects, encapsulating different access rights, all pointing to the same piece of underlying state. Later, when the client issues an object invocation on the object reference, the invocation request is transmitted securely through the nucleus door and delivered to the front object. The front object then checks that the request is permissible based on the encapsulated access rights, and if so, forwards the request into the server. For example, if the client issued an update request, the front object would check that the encapsulated access included write access.

When a client is given an object reference that is acting as a capability they can pass that object reference on to other clients. These other clients can then use the object reference freely and will receive all the access that was granted to the original client.

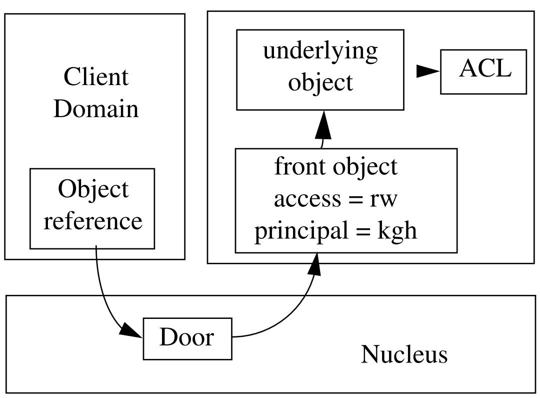

FIGURE 7. A client accessing a secure object

For example, say that user X has a file object foo, which has a restricted access control list specifying that only X is allowed to read the file. However X would like to print the file on a printserver P. P is not on the ACL for foo, so it would not normally have access to foo's data. However, X can obtain an object reference that will act as a software capability, encapsulating the read access that X is allowed to foo. X can then pass that object reference on to the printserver P and P will be able to read the file.

The use of software capabilities in Spring makes it easy for application programs to pass objects to servers in a way that allows the server to actually use the given object.

8 Virtual Memory

Spring implements an extensible, demand-paged virtual memory system that separates the functionality of caching pages from the tasks of storing and retrieving pages [7].

8.1 Overview

A per-machine virtual memory manager (VMM) handles mapping, sharing, protecting, transferring, and caching of local memory. The VMM depends on *external pagers* for accessing backing store and maintaining inter-machine coherency.

Most clients of the virtual memory system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of memory that can be mapped into address spaces. An example of a memory object is a file object (the file interface in Spring inherits from the memory object interface). Address space objects are implemented by the VMM.

A memory object has operations to set and query the length, and an operation to *bind* to the object (see Section 8.2). There are no page-in/out or read/write operations on memory objects. The Spring file interface provides file read/write operations (but not page-in/page-out operations). Separating the memory abstraction from the interface that provides the paging operations is a feature of the Spring virtual memory system that we found very useful in implementing our file system [13]. This separation enables the memory object server to be in a different

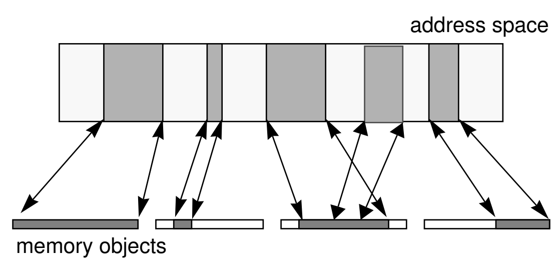

FIGURE 8. User's view of address spaces

An address space is a linear range of addresses with regions mapped to memory objects. Each region is mapped to a (part of) a memory object. Each page within a mapped region may be mapped with read/write/execute permissions and may be locked in memory.

machine than the *pager* object server which provides the contents of the memory object.

8.2 Cache and Pager Objects

In order to allow data to be coherently cached by more than one VMM, there needs to be a two-way connection between the VMM and an external pager (e.g., a file server). The VMM needs a connection to the external pager to allow the VMM to obtain and write out data, and the external pager needs a connection to the VMM to allow the provider to perform coherency actions (e.g., to invalidate data cached by the VMM). We represent this two-way connection as two objects.

The VMM obtains data by invoking a *pager* object implemented by an external pager, and an external pager performs coherency actions by invoking a *cache* object implemented by a VMM.

When a VMM is asked to map a memory object into an address space, the VMM must be able to obtain a pager object to allow it to manipulate the object's data. Associated with this pager object must be a cache object that the external pager can use for coherency.

A VMM wants to ensure that two equivalent memory objects (e.g., two memory objects that refer to the same file on disk), when mapped, will share the data cached by the VMM. To do this, the VMM invokes a *bind* operation on the memory object. The bind operation returns a *cache_rights* object, which is always implemented by the VMM itself. If two equivalent memory objects are mapped, then the same cache_rights object will be returned. The VMM uses the returned object to find a pager-cache object connection to use, and to find any pages cached for the memory object.

When a memory object receives a bind operation from a VMM, it must determine if there is already a pager-cache object connection for the memory object at the given VMM. If there is no connection, the external pager implementing the memory object contacts the VMM, and the VMM and the external pager exchange pager, cache, and cache_rights objects. Once the connection is set up, the memory object returns the appropriate cache_rights object to the VMM.

Typically, there are many pager-cache object channels between a given pager and a VMM (see Figure 9 for an example).

8.3 Maintaining Data Coherency

The task of maintaining data coherency between different VMMs that are caching a memory object is the responsibility of the external pager implementing the memory object. The coherency protocol is not specified by the architecture—external pagers are free to implement whatever coherency protocol they wish. The cache and pager object interfaces provide basic building blocks for constructing the coherency protocol. Our current external pager implementations use a single-writer/multiple-reader per-block coherency protocol [12, 13].

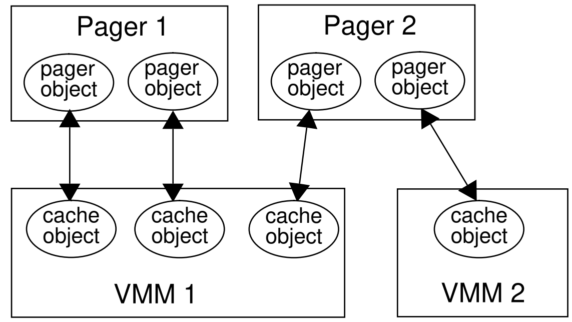

FIGURE 9. Pager-cache object example

A VMM and an external pager have a two-way pager-cache object connection. In this example, Pager 1 is the pager for two distinct memory objects cached by VMM 1, so there are two pager-cache object connections, one for each memory object. Pager 2 is the pager for a single memory object cached at both VMM 1 and VMM 2, so there is a pager-cache object connection between Pager 2 and each of the VMMs.

9 File System

The file system architecture defines *file* objects that are implemented by file servers. The file object interface inherits from the *memory object* and *io* interfaces. Therefore, file objects may be memory mapped (because they are also memory objects), and they can also be accessed using the read/write operations of the io interface.

Spring includes file systems giving access to files on local disks as well as over the network. Each file system uses the Spring security and naming architectures to provide access control and directory services.

A Spring file system typically consists of several layered file servers [5]. The pager-cache object paradigm is used by file systems as a general layering mechanism between the different file servers and virtual memory managers. Among other things, this has enabled us to provide per-machine caching of data and attributes to decrease the number of network accesses for remote files.

9.1 File Server Implementations

The Spring Storage File System (SFS) is implemented using two layers as shown in Figure 10.

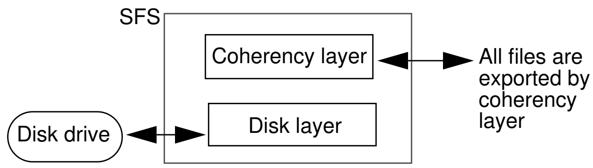

FIGURE 10. Spring SFS

The base disk layer implements an on-disk Unix compatible file system. It does not, however, implement a coherency algorithm. Instead, an instance of the coherency file server is stacked on the disk layer, and all files are exported to clients via the coherency layer.

The coherency layer implements a per-block multiple-reader/single-writer coherency protocol. Among other things, the implementation keeps track of the state of each file block (read-only *vs.* read-write) and of each cache object that holds the block at any point in time. Coherency actions are triggered depending on the state and the current request using a single-writer/multiple-reader per-block coherency algorithm. The coherency layer also caches file attributes.

The Caching File System (CFS) is an attribute-caching file system. Its main function is to interpose itself between remote files and local clients when they are passed to the local machine so as to increase the efficiency of many operations. Once interposed on, all calls to remote files end up being diverted to the local CFS.

An interesting aspect of CFS is the manner in which it *dynamically* interposes on individual remote DFS files. A *caching* subcontract is used to contact the local CFS in the process of unmarshalling file objects. When CFS is asked to interpose on a file, it becomes a cache manager for the remote file by invoking the bind operation on the file as described in Section 8.2.

10 Spring Naming

An operating system has various kinds of objects that need to be given names, such as users, files, printers, machines, services, etc. Most operating systems provide several name services, each tailored for a specific kind of object. Such *type specific* name services are usually built into the subsystem implementing those objects. For example, file systems typically implement their own naming service for naming files (directories).

In contrast, Spring provides a uniform name service [17]. In principle, any object can be bound to any name. This applies whether the object is local to a process, local to a machine, or resident elsewhere on the network, whether it is transient or persistent; whether it is a standard system object, a process environment object, or a user specific object. Name services and name spaces do not need to be segregated by object type. Different name spaces can be composed to create new name spaces.

By using a common name service, we avoid burdening clients with the requirement to use different names or different name services depending on what objects are being accessed. Similarly, we avoid burdening all object implementations with constructing name spaces—the name service provides critical support to integrate new kinds of objects and new implementations of existing objects into Spring. Object implementations maintain control over the representation and storage of their objects, who is allowed access to them, and other crucial details. Although Spring has a common name service and naming interface, the architecture allows different name servers with different implementation properties to be used as part of the name service.

The name service allows an object to be associated with a name in a *context*, an object that contains a set of name–to–object associations, or *name bindings*, and which is used by clients to perform all naming operations. An object may be bound to several different names in possibly several different contexts at the same time. Indeed, an object need not be bound to a name at all.

By binding contexts in other contexts we can create a *naming graph* (informally called a name space), a directed graph with nodes and labeled edges, where the nodes with outgoing edges are contexts.

Unlike naming in traditional systems, Spring contexts and name spaces are first class objects: they can be accessed and manipulated directly. For example, two applications can exchange and share a private name space. Traditionally, such applications would have had to build their own naming facility, or incorporate the private name space into a larger system–wide name space, and access it indirectly via the root or working context.

Since Spring objects are not persistent by default, naming is used to provide persistence [16]. It is expected that applications generally will (re)acquire objects from the name service. If the part of the name space in which the object is found is persistent, then the object will have been made persistent also.

A Spring name server managing a persistent part of a name space converts objects to and from their persistent form (much like the UNIX file system, which converts open files to and from their persistent form). However, since naming is a generic service for an open–ended collection of object types, a context cannot be expected to know how to make each object type persistent. Spring object managers have ultimate control of the (hidden) states of their objects. Therefore we provide a general interface between object managers and the name service that allows persistence to be integrated into the name service while allowing the implementation to control how its (hidden) objects' states are mapped to and from a persistent representation.

Because the name service is the most common mechanism for acquiring objects, it is a natural place for access control and authentication. Since the name service must provide these functions to protect the name space, it is reasonable to use the same mechanism to protect named objects. The naming architecture allows object managers to determine how much to trust a particular name server, and an object manager is permitted to forego the convenience and implement its own access control and authentication if it wishes. Similarly, name servers can choose to trust or not to trust other name servers.

The Spring name service does not prescribe particular naming policies; different policies can be built on the top. Our current policy is to provide a combination of system-supplied shared name spaces, per-user name spaces, and per-domain name spaces that can be customized by attaching name spaces from different parts of the distributed environment.

By default, at start-up each domain is passed from its parent a private domain name space, which incorporates the user and system name spaces. A domain can acquire other name spaces and contexts if it desires.

11 UNIX Emulation

Spring can run Solaris binaries using the UNIX emulation subsystem [6]. It is implemented entirely by user-level code, employs no actual UNIX code, and provides binary compatibility for a large set of Solaris programs. The subsystem uses services already provided by the underlying Spring system and only implements UNIX-specific features that have no counterpart in Spring (e.g., signals). No modifications to the base Spring system were necessary to implement Solaris emulation.

The implementation consists of two components: a shared library (*libue.so*) that is dynamically linked with each Solaris binary, and a set of UNIX-specific services exported via Spring objects implemented by a *UNIX process server* (in a separate domain). See Figure 4.

The UNIX process server implements functions that are not part of the Spring base system and which cannot reside in *libue.so* due to security reasons.

11.1 Libue

When a program is *exec*ed, *libue.so* is dynamically linked with the application image in place of *libc*, thus enabling the application to run unchanged.

The *libue.so* library encapsulates some of the functionality that normally resides in a monolithic UNIX kernel. In particular, it delivers signals forwarded by the UNIX process server, and keeps track of the association between UNIX file descriptor numbers (fd's) and Spring file objects.

For each UNIX system call, we implemented a library stub. In general, there are three kinds of calls:

1. Calls that simply take as an argument an fd, parse any passed flags, and invoke a Spring service (e.g., *read*, *write*, and *mmap*). Most file system and virtual memory operations fall in this category.
2. Calls that eventually call a UNIX-specific service in the UNIX process server. Examples include *pipe* and *kill*.
3. Calls that change the local state without calling any other domain. *Dup*, parts of *fcntl*, and many signal handling calls fall into this category.

11.2 UNIX Process Server

The UNIX process server maintains the parent-child relationship among processes, keeps track of process and group ids, provides sockets and pipes, and forwards signals.

The UNIX process server is involved in forking and execing of new processes. It is also involved in forwarding (but not delivering signals). Since it keeps track of process and group ids, it enforces UNIX security semantics when servicing requests from client processes.

12 Conclusions

The Spring project chose to build a different operating system, one based on the notions of strong interfaces, openness and extensibility and designed to be distributed and suited to multiprocessors. Using object-oriented ideas and strong interfaces has been a natural fit, with a number of benefits:

- A standardized basis for open, distributed object systems via the Interface Definition Language and a simple client model for objects
- Easy distributed services and applications
- Readily extensible system facilities, such as file systems and name services
- Unity of architecture together with a wide range of implementation opportunities as in virtual memory management, naming, subcontract, and serverless objects
- Highly efficient inter-address space object invocation in support of a microkernel-based architecture.

Finally, designing in security mechanisms from the start has provided a system that can support a wide range of secure mechanisms in a networked environment, from the most relaxed to the most secure.

13 References

[1] Graham Hamilton and Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects," Proc. 1993 Summer USENIX Conference, pp. 147-160, June 1993.

[2] Graham Hamilton, Michael L. Powell, and James G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming," Proc. 14th ACM Symposium on Operating Systems Principles, pp. 69-79, December 1993.

[3] Graham Hamilton and Sanjay Radia, "Using Interface Inheritance to Address Problems in System Software Evolution," Proc. ACM Workshop on Interface Definition Languages, January 1994.

[4] Peter B. Kessler, "A Client-Side Stub Interpreter," Proc. ACM Workshop on Interface Definition Languages, January 1994.

[5] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," Proc. 14th ACM Symposium on Operating Systems Principles, pp. 1-14, December 1993.

[6] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," Proc. Winter 1993 USENIX Conference, pp. 469-479, January 1993.

[7] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-93-9, March 1993.

[8] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," Proc. 2nd Workshop on Microkernels and Other Kernel Architectures, September 1993.

[9] Michael N. Nelson and Graham Hamilton, "High Performance Dynamic Linking Through Caching," Proc. 1993 Summer USENIX Conference, pp. 253-266, June 1993.

[10] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "Caching in an Object-Oriented System," Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOOS III), pp. 95-106, December 1993.

[11] Michael N. Nelson and Yousef A. Khalidi, "Generic Support for Caching and Disconnected Operation," Proc. 4th Workshop on Workstation Operating Systems (WWOS-IV), pp. 61-65, October 1993.

[12] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," Proc. 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), September 1993.

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

[14] Michael N. Nelson and Sanjay R. Radia, "A Uniform Name Service for Spring's Unix Environment," Proc. Winter 1994 USENIX Conference, Jan. 1994.

[15] Object Management Group, "Common Object Request Broker Architecture and Specification," OMG Document 91.12.1, December 1991.

[16] Sanjay Radia, Peter Madany, and Michael L. Powell, "Persistence in the Spring System," Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOOS III), pp. 12-23, December 1993.

[17] Sanjay R. Radia, Michael N. Nelson, and Michael L. Powell, "The Spring Name Service," Sun Microsystems Laboratories Technical Report SMLI-93-16, October 1993.