A Uniform Name Service

for Spring's UNIX Environment

*Michael N. Nelson** *Sanjay R. Radia*

Sun Microsystems, Inc.

Mountain View, CA 94043 USA

Abstract

The Spring operating system provides a uniform name service that can be used to associate any name with any object independent of the type of object, and allows arbitrary name spaces to be created and used as first-class objects. We have used this name service to unify the many UNIX® name spaces. Objects that on UNIX systems are typically stored in separate name spaces are all accessible via a single uniform name service in Spring. In addition, it is easy to add new Spring objects that are not currently available in UNIX systems without modifying the underlying name service.

1. Introduction

Typical UNIX systems have various kinds of nameable objects, such as files, printers, and services, and have several name services, each tailored for a specific kind of object. Such *type-specific* name services are usually built into the subsystem implementing the specific type of object. Examples of the many type-specific name services in typical UNIX systems include:

- A UNIX file system which provides its own mechanism for naming file objects, including operations for binding names to files and accessing files by name. The file system is also responsible for storing and managing the database of name-to-object bindings.
- The name space for printers which is stored in */etc/printcap* and is used by the various printing commands.
- Light weight name services such as environment variables which have their own name space with library implementations.
- Distributed versions of UNIX systems usually have one or more higher level name services called *directory* services such as Sun's NIS which are used for resource location, mail addressing, and authentication. Directory services bind names to data values.

Each of these name services has its own syntax for names, and its own interface and implementation for performing operations on the name space. The client is burdened with the requirement of dealing with different names and name services depending on what objects are to be accessed. Disjoint parts of the name space must be used for different types of objects since, for example, it is not possible to bind a file as an environment variable. Another problem is that the non-uniform name spaces generally cannot be composed together. For example, the name spaces for files cannot be attached to the name spaces for environment variables. Furthermore, defining new objects or services that are accessed by name is difficult, since the new objects must be made to "fit in" somewhere or a new name space must be defined and implemented.

* Authors current affiliation: Silicon Graphics, Inc. (mnelson@sgi.com)

In systems such as the UNIX system that do not have a unifying notion of objects, a popular technique has been to make each object provide the file interface and for each object implementation to implement the file system naming interface (the file being the most versatile and well defined entity in the system). Plan 9 [1 , 2], has used this approach extensively. Although it is possible to make many objects look like files, it would be better if this constraint were not necessary since the file interface is limited in capturing the semantics of all objects. Furthermore, it is difficult to make every object behave like a file; some object types and name spaces cannot be easily fit into the file name space.

This paper describes the use of a uniform object-oriented naming system to unify many UNIX name spaces. We compare and contrast our approach with that of Plan 9; although there are similarities, there are many subtle and important differences. The naming system described in this paper was developed as part of the Spring operating system project. This naming system works together with the Spring UNIX subsystem [3] to provide users and normal UNIX programs uniform naming access to most types of UNIX objects.

2. Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects.* A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of the object and its operations are specified in an *interface definition language* (IDL). IDL supports both notions of *single* and *multiple* interface inheritance.

A Spring *domain* is an *address space* with a collection of *threads.* A given domain may act as the server (implementor) of some objects and the clients of other objects. The server and the client can be in the same domain or in different domains. In the latter case, the representation of the object includes an unforgeable nucleus *door identifier* that identifies the server domain [4].

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management [4, 5]. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server.

A typical Spring node runs several servers besides the kernel. These include a name server, file servers, a linker domain that manages and caches dynamically linked libraries, a network proxy that handles remote invocations, a device server that provides basic terminal handling as well as frame-buffer and mouse support, and a UNIX server that provides support for running UNIX binaries on Spring [3].

3. Spring Naming

The Spring name service [6] allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique [7, 8]. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time.

Resolving a name is an operation on a context to obtain the object denoted by the name; binding a name is an operation to associate a name with a particular object. These operations return or take as a parameter the object itself, not a lower-level identifier for the object as some systems do [9 , 10].

Since a context is like any other object, it can also be bound to a name in some context. By binding contexts we can create a *naming graph*. The UNIX file system is a naming graph that is frequently restricted to a tree. Given a context in some naming graph, a sequence of names can be used to refer to an object relative to that context. Such a sequence of names is called a *compound name*. A name with a single component is called a *simple name*. Informally, we refer to the naming graph spanned by a context as a name space, which includes all bindings of names and objects that are accessible directly or indirectly through that context.

Spring contexts provide support for the Spring security model. When an object is bound, an ACL can be given that specifies which principals are allowed which rights for the object. When an object is resolved, a set of desired *modes* is specified. Modes are a superset of rights. For example, read and write modes correspond directly to read and write access rights; however, append mode implies write access but also indicates the “mode” with which the object should be accessed when writes occur. When a mode is specified to a resolve operation, an object with the desired modes is returned if the client doing the resolve is allowed the corresponding rights.

Contexts and name spaces are first class entities in Spring: a context, and the name space it spans, can be manipulated directly. A context can be passed around like any other object. For example, two applications can exchange and share a private name space. In the UNIX system, 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. Name spaces can be composed by binding a context to a name in some name space (this is a generalization of the mount operation). The first class nature of contexts also makes it very simple to support ordered merges as described below.

3.1. Names

A name consists of a sequence of one or more components. Each component is an unordered set of elements. The identifier element of a component is an arbitrary UNICODE string that is never parsed (only compared for equality) by the name service. Other elements of a component encode version numbers (allowing references to the latest version) and an object “kind” (analogous to file name extensions). The presentation and parsing of names is relegated to user interface software. The current syntax that is used for Spring names is identical to the syntax used for UNIX path names.

3.2. Naming Operations

The primary interface between a client and the name service (the context interface) is simple. The following are the common operations on contexts:

- named_object = context.resolve (name, mode)

  Returns the object denoted by the name. The mode argument indicates intended use of the object.

- context.bind (name, binding_type, named_object, acl)

  Establishes a binding in the context (or in a context reachable from it, if the name is a compound name). The binding type is used to distinguish bindings that the name service has to process during resolution. Symbolic links specify *symbolic_binding*, name spaces are grafted by specifying *context_binding*, and normal objects specify *normal_binding*. The *acl* is the binding’s access control list.

- new_context = context.bind_new_context(name, acl)

  Creates a new context with a particular name. It is also possible to create contexts that are (as yet) unnamed, although that is an operation on the name server interface, not the context interface.

- iterator = context.get_all_bindings(name)

  Returns the binding information in the context.

- context.unbind(name)

  Deletes a binding

3.3. Ordered Merges

Given a set of contexts one can construct a new context using that set. An ordered merge context in Spring (like union mounts in Plan 9) is a context whose implementation contains several contexts. The result is that the name spaces of all of the involved contexts are merged. We use ordered merges for constructing the per-domain name space in Spring and also for search paths.

Ordered merges in other systems are usually restricted to special situations such as command search paths or unions at mount time (as in Plan 9). In Spring, since contexts objects can be manipulated directly, we did not clutter the context interface with a notion of merges. Instead, we provided it as a separate context implemented in a library. We give several examples below of our use of ordered merges in constructing name spaces.

3.4. The Spring Naming Environment

The Spring name service imposes no polices such as the notion of a global root. Such notions are built as policies on the top of the name service. The policies that we use in Spring are based on a private per-domain (per-process) view of naming coupled with shared name spaces. Below we first describe the shared machine and village name spaces; these name spaces are used to construct part of the per-domain private name spaces. Our per-domain name spaces are based on many of the ideas in [11] and are similar to the per-process name spaces in Plan 9.

3.4.1. Machine and Village Name Spaces

Each machine has a per-machine name space. This name space is implemented by a per-machine name server and contains the domain name space, all machine-specific services, and the devices exported from the machine.

A collection of machines is called a village. Each village has a per-village name space. This name space is implemented by one of the machines in the village and contains information global to the village such as:

- the name spaces for all of the machines. Each machine name server binds a context that represents the machine's name space into the village.
- globally accessible services such as NIS.

3.4.2 Per-Domain Name Space

Each domain has its own private name space that can be customized to access arbitrary name spaces such as the machine and village name spaces. A per-domain name space gives great flexibility in sharing objects and name spaces in a distributed environment. It has also allowed us to incorporate private name spaces such as the environment variables name space. Ordered merge constructions are used for further flexibility in tailoring name spaces. The result is a powerful naming environment. A domains's private name space typically contains (see Figure 1):

- Private name bindings

  The domain's name space has bindings for environment variables and program input/output objects. (A special mechanism to pass standard IO objects as implicit parameters is not needed.)
- Shared name spaces

  Several shared name spaces are attached to the domain's name space using well-known names: *user* (the name spaces of different users), ~ (the home name space of the user owning the domain), and *dev* (devices). If there were, for example, a worldwide global name space, we would attach it to the name space of a domain using a well-known name.
- Generic name spaces containing standard system objects

  A domain's name space has generic name spaces that contain system objects: *sys* (executables under *sys/bin* and libraries under *sys/lib*) and *services* (such as *services/authentication*). The name spaces of *sys* and *services* have parallel structures in both the machine and village. These name spaces contain copies of such system objects for local use[1].

  The *sys* and *services* name spaces in a domain's private name space are typically an ordered merge of the corresponding name spaces of the machine and village. The merge arranges for the local instance to be visible first. A user can also keep similar structures in his home contexts which can be used for further tailoring of these name spaces. During remote execution, the merges are automatically reevaluated to take advantage of the name spaces of the remote site.

---

1. The local copies are in addition to what is kept automatically by our caching scheme. Local copies allow a machine to maintain local autonomy, especially during disconnected operation.

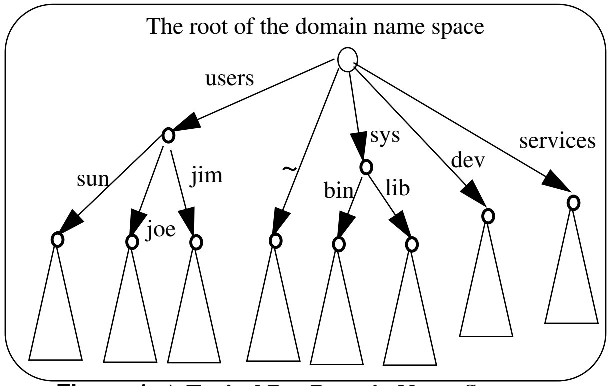

Figure 1. A Typical Per-Domain Name Space

Each parent domain passes to its child a context defining the child's private name spaces. This is usually a copy of the parent's context. Since most domains make few changes to their name spaces, we can make the child's context be a copy-on-write copy of the parent's. The init domain hand-crafts the name space given to its children. When a user logs in, the name space is further modified according to the user's profile.

4. Implementing UNIX Name Spaces

We have used the Spring naming system to unify many traditional UNIX name spaces. In this section we will describe our implementation of the various UNIX name spaces.

4.1. The UNIX Name Space on Spring

In order to allow UNIX applications to run on Spring we provide the UNIX naming environment via the per-domain name space. The UNIX root becomes the per-domain context of the domain that implements the UNIX process. The current working directory becomes a symbolic link called ".cwd". We add bindings for /bin, /lib, /etc, /usr, and other generic UNIX contexts. The per-domain context often contains additional Spring entries such as Spring commands and libraries to allow UNIX users on Spring to access Spring resources. This is accomplished by an ordered merge; for example /bin is an ordered merge of the context containing UNIX commands and the context containing Spring commands.

4.2. Generic Naming Implementation

The Spring naming system is comprised of several name servers. Most of these name servers run generic name server code that can store objects of any type. This code is available in a library and can be used to build name servers on native Spring and under Spring emulation on the SunOS system. The only name spaces that are not implemented by the generic name server library code are the file system and NIS.

4.3. File System

The Spring file system supports file servers that have an integrated name service such as that in the UNIX system. In addition we have extended typical UNIX file system naming in two ways:

- Files can be bound in other non-file-system parts of Spring name spaces.
- Non-file Spring objects can be bound into file system name spaces.

We also allow access to file systems on UNIX machines on the network via gateway name servers that translate names between Spring and the UNIX system.

The UNIX file system name space maps directly into the Spring name service. UNIX files are mapped to Spring files and UNIX directories are mapped to Spring contexts.

Files are opened by resolving them using a particular context. For example, in order to open the file “/etc/passwd” read-write relative to the per-domain context, the following call would be issued:

```
file f = context.resolve(“etc/passwd”, access_read_write);
```

4.4. NIS

The NIS name space is hierarchical in nature and as such can be mapped into the Spring name service. The NIS name server exports a context that contains the default NIS domain. The context for each domain contains a context for each map. The context for each map contains all of the keys. A resolution of a key returns a string that contains the value. For example to lookup mnn’s passwd entry the following call would be issued on a context that represented the root of the NIS name space:

```
string s = context.resolve(“Wildfire/passwd/mnn”, access_read);
```

4.5. Environment variables

Environment variables are bound into the per-domain context. For example, looking up the value of the HOME environment variable would require the following call:

```
string s = context.resolve(“HOME”);
```

4.6. Domains

On SVR4 systems, information about processes can be acquired via the /proc file system. Information can be found in the various files that are stored in the directory for a given process. In order to perform an operation on the process, the read, write, and ioctl interface must be used on the appropriate file.

In Spring we export information about domains via a domain object. The domain object of a domain and other objects that the domain wants published to the outside world are bound in the per-machine name space of the machine on which the domain executes. These objects can be manipulated directly by invoking operations rather than encoding the request into an ioctl or an ascii string to a write call. For example to stop all of the threads in domain 27 one would issue the following calls:

```
domain d = context.resolve(“domains/27”, access_read);
d->stop();
```

4.7. Devices

Device objects are accessed via the dev context as in the UNIX system. The dev context is a merge of the per-domain *dev* context (which contains stdio objects), the *dev* context on the machine (which contains devices on that machine), and the *dev* context of the village. The *dev* context of the village can be used to provide special devices available on particular machines that are meant to be made widely available. In addition one can also access a device on another machine by using a name of the form “village/machine/machine_name/dev/device_name”.

4.8. UNIX I/O Uniformity

One of the advantages of the Plan 9 approach of making everything look like a file is that it provides a uniform object interface to clients of the name service. In Plan 9, all objects are files so all objects can be accessed via read and write calls. Unfortunately, it is unknown what is appropriate to pass to a write call or expect from a read call. The semantics of the read or write call depend on the type of the object involved.

In Spring we still provide a common I/O interface for those objects for which I/O is appropriate. This is done by having each object that supports the I/O interface inherit from the *io* class. If a program gets an object that it wishes to treat as an io object, then it can do a type system operation to traverse the type hierarchy to the io class. If this operation succeeds, then the object can be treated as an io object meaning that read and write operations can be invoked on the object.

In Spring the semantics of read and write operations are well defined. We do not attempt to overload the read and write operations to provide functionality that is best provided by methods on the object itself.

5. Related Work

Other object oriented systems have either not provided a uniform name service or provided one with serious restrictions. Programming language based object oriented systems such as Smalltalk rely on the name space of variables provided by the programming language. Others like Choices [10] have the serious restriction that the object managers must be integrated and reside with the name service; this makes it difficult to add new object types. Furthermore, Choices provides two name services: one for persistent objects and one for transient objects.

DCE [12] has made an attempt to provide uniformity by composing name spaces such as the file system name space onto the higher level directory name space. This is done through special entities called junctions. For example a junction point in the directory level name space called *fs*, signals that one is entering the file system name space (it is like a specialized mount point). What this provides is network-wide access to files: there is a uniform way of naming files and file systems in other parts of the network. The client is still faced with multiple name spaces and their corresponding naming interfaces. Furthermore, it is not clear how easy it is too create new junctions when introducing new object types and their corresponding name spaces. Thus, DCE does not provide a uniform name service but instead provides a uniform way of allowing network-wide access to the various name spaces.

Providing a uniform name service has been difficult in the UNIX system because the entities being named have different representations. Type-specific name services provide tokens that must be used for a particular purpose, since the type is assumed; for example resolving a file name as part of the open operation returns an integer which is used for file operations. On the other hand, directory services provide values that must be resolved at another level to be useful; for example, looking up a host returns a byte string that is a network address.

The Plan 9 system has tried to provide uniform naming in a manner different than we have used. Plan 9 unifies the UNIX name spaces by making each nameable object provide the file interface and having each object implementation implement the file system naming interface. Given the lack of some unifying concept like an object, the Plan 9 approach is probably the best way of providing uniform naming in the UNIX system, or other non-object oriented systems. The main advantage of the Plan 9 naming model is that it fits directly into the UNIX way of doing things: UNIX users and standard UNIX tools already understand files and file systems so it is relatively easy for users and standard UNIX tools to use the Plan 9 system. However, with the trend towards object oriented systems, we believe that our approach is better than Plan 9's approach of attempting to make all objects look like files. In the rest of this section we will discuss why we believe that our object-oriented naming system is better than the Plan 9 file centric naming view. We will also discuss Plan 9's more flexible notion of a per-process mount table.

There are basically four major advantages of the Spring naming approach over the Plan 9 approach. The first advantage comes from Spring's object-oriented approach. In Spring, nameable objects have strong well-defined interfaces that are appropriate for the particular type of object. We do not attempt to make all objects obey the file interface and hide the object's true interface inside the *read*, *write*, and *ioctl* operations.

The second advantage of the Spring approach is that the effort required to make new object types nameable is low. In Spring, nameable objects do not have to made to obey the file interface and object managers do not have to implement the name service. For example, objects implemented by the Spring kernel such as domains (processes), VM objects, and kernel monitoring objects are routinely bound into the machine's name spaces; the kernel did not have to change the nature of these objects or implement a name service to make this possible. This means that we can easily add new nameable objects and new implementations of nameable objects to the system at any time. If someone comes up with a new object type, they do not have to modify the implementation of the name service to allow this object to be stored in a name space. Spring developers routinely create new objects and make them available via the name service.

In contrast to Spring, the Plan 9 approach requires that nameable non-file objects be made to look like files and implementations of the non-file objects must implement the file system directory operations. Although Plan 9 has been able to make many objects look like files, it does require a certain amount of effort which is not required in our system. Furthermore, we believe that the Plan 9 approach cannot succeed for all kinds of object. Indeed the network name space in Plan 9 (the name space that binds file servers) does not map very easily to a file system and hence is a separate non-file name service.

The third advantage of the Spring approach is that name spaces do not need to be segregated by the type of objects being bound in them. For example, we can store a file in our equivalent of the /proc file system and we can store a domain object in the file system. This functionality is not possible in Plan 9; the best one can do in Plan 9 is to use the union mount to attach special file systems behind the regular directory so as to allow regular files to be created “ahead” of the mounted special file system. Thus in spite of making most objects look like files, a lack of uniformity is evident since each file server can only bind objects of the same true type.

The last advantage of the Spring approach is the notion that contexts and name spaces are first class entities. In Spring, contexts can be manipulated directly and passed around like any other object. In the UNIX system, applications that need to exchange and share a private name space would have 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.

The first class nature of contexts has made it very simple for us to support context constructs like ordered merges. A client can construct an ordered merge context from a set of contexts directly; the resulting context can be used like any other context: it can be passed around, bound to a name, etc. We did not have to resort to special name resolution rules or mechanisms. If we were not able to manipulate contexts directly, we might have been forced to limit the use of ordered merges to “mount” time (as in Plan 9) and to make the notion of merges be a part of the naming interface. We also allow ordered merges to be specified as symbolic links in which case the meaning of the merge is evaluated when a name is resolved through that name binding. We believe such symbolic ordered merges could easily be added to Plan 9.

One place where Plan 9 is more flexible than Spring is Plan 9’s per-process mount table. In Spring mounting a naming tree on a shared sub-tree makes the mounted tree visible to all processes that share the sub-tree. In Plan 9 a naming tree would only be visible to the process doing the mount (this fits naturally with the UNIX system notion of a mount table). Plan 9’s per-process mount table is clearly more flexible. We decided not to pursue the per-process mount table for two reasons. First, we found that ordered merges allow a process sufficient flexibility in configuring its name space and have not found the need to add the notion of per-process mount tables. Second, when a private name space overlays a shared name space, resolving names in the shared name space becomes more complex in a distributed environment: one cannot blindly resolve pathnames in a remote shared name space as private local mounts may overlay parts of it.

6 Status

The Spring name service described in this paper has been implemented as part of the Spring operating system which is currently running on SPARCstation 1, 2, and 10 machines. A general name server implementation that is used for the machine, village, and per-domain name spaces is available as a library for Spring programs. This library is usable both by programs running on native Spring and programs running under a Spring emulation package on SunOS.

7 Conclusion

The Spring name service makes it possible to unify the many UNIX name spaces. Each of these name spaces fits seamlessly into the overall Spring name service. Our success at unifying the name spaces is evident from the fact that standard UNIX tools such as *ls* and *filemanager* can be used to browse any Spring name space without changing the tools. Thus users now can easily browse name spaces that once required special tools.

Spring provides more than just a simple unification of UNIX name spaces. Our use of object-orientation means that users not only see a uniform name space but they also get the advantages of strong interfaces and the ability to add new nameable objects and new implementations at any time without modifying the name service. The Spring name service is more powerful in allowing contexts and name spaces to be manipulated as first class entities. We believe that a powerful and uniform object-oriented name service like that used in Spring will provide the right infrastructure to allow the complex and diverse systems of the future to be usable by the average user.

8 References

[1] Pike R., Presotto D., Thompson, K., and Trickey, H., "Use of Name Spaces in Plan 9", Unpublished report, 1992.

[2] Pike R., Presotto D., Thompson, K., and Trickey, H., "Plan 9 from Bell Labs", *Proceedings of 1990 UKUUG Conference*, July, 1990, pp. 1-9.

[3] Khalidi, Y. A. and Nelson, M. N., "An Implementation of UNIX on an Object-oriented Operating System", *Proceedings of the 1993 Winter USENIX Conference*, January 1993, pp. 469-479.

[4] Hamilton, K.G. and Kougiouris, P., "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-160.

[5] Khalidi, Y.A. and Nelson, M.N., "The Spring Virtual Memory System," Sun Microsystems Laboratories, Technical Report SMLI-92-388, September 1992.

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

[7] Saltzer, J. H., "Naming and Binding Objects," in *Operating Systems: An Advanced Course*, volume 60, pages 99--208. Springer-Verlag, New York, 1978.

[8] Radia, S, "Names, Contexts, and Closure Mechanisms in Distributed Computing Environments," Ph.D. thesis, Univ. of Waterloo, Dept. of Comp. Sci., Waterloo, Ontario, Canada, 1989. Also report UW/ICR 90-01.

[9] Berabeu-Auban, J. M., et al., "The Architecture of Ra: A Kernel for Clouds", *Proceedings of the 22th Hawaii International Conference on System Sciences*, January 1989, pp. 936-945.

[10] Campbell, R.H., Islam, N., and Madany, P., "Choices, Frameworks, and Refinement," *USENIX Computing Systems*, 5, 3 (Summer 1992), pp. 217-257

[11] Radia, S. and Pachl, J., "The Per-Process View of Naming and Remote Execution", *IEEE Parallel and Distributed Technology,* Vol 1, No 3, August 1993, pp 71-80.

[12] Rosenberry W., Kenny D., Fisher G., "Understanding DCE", O'Reilly & Associates, Inc., California 1993.

Michael N. Nelson is currently a Member of the Technical Staff at Silicon Graphics. Before joining SGI he was one of the principal developers of the Spring Operating System at Sun Microsystems and was also a principal developer of the Sprite Operating System at UC Berkeley. His interests include distributed, object-oriented software, operating systems, and architecture. He has a Ph.D. in Computer Science from UC Berkeley. He can be reached at Silicon Graphics, 2011 N. Shoreline Blvd., Mountain View, CA 94043, USA, or via e-mail at mnelson@sgi.com.

Sanjay Radia is a senior staff engineer at Sun Microsystems, where is one of the principal developers of the Spring distributed operating system. His interests includes design and implementation of distributed software and operating systems, particularly using object-oriented technology. Previously, he has worked on several distributed system projects at I.N.I.R.A., France, and at the University of Waterloo. He has a Phd from University of Waterloo, Canada. He can be reached via email at srradia@sun.com.