Snowflake: Spanning administrative domains

Jon Howell*
David Kotz

Technical Report PCS-TR98-343†

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
jonh@cs.dartmouth.edu

December 9, 1998

Abstract

Many distributed systems provide a "single-system image" to their users, so the user has the illusion that they are using a single system when in fact they are using many distributed resources. It is a powerful abstraction that helps users to manage the complexity of using distributed resources. The goal of the Snowflake project is to discover how single-system images can be made to span administrative domains. Our current prototype organizes resources in namespaces and distributes them using Java Remote Method Invocation. Challenging issues include how much flexibility should be built into the namespace interface, and how transparent the network and persistent storage should be. We outline future work on making Snowflake administrator-friendly.

1 Introduction

When networks of microcomputers were novel, cluster software made the individual machines work together to give users the appearance of a *single-system image*; that is, users saw the cluster of processors as a single mainframe, which was a convenient conceptual model.

As computing systems become more ubiquitous, more and more users find themselves accessing multiple computer systems, often in different *administrative domains*, that is, in different departments or at different organizations. Cluster software requires central administration, so each cluster can be no bigger than a single administrative domain. Thus users find themselves using multiple single-system images, which defeats the purpose of the single-system image abstraction.

The goal of Snowflake is to let users build personalized single-system images that span administrative domains. Notice we said that *users* do the building. That is because each user's single-system image will vary according to the set of administrative domains that user accesses. Therefore, the most natural way to construct the image is per-user. To that end, Snowflake generalizes previous work in single-system-image clusters, and applies a philosophy of giving end users control over how resources are distributed.

*Supported by a research grant from the USENIX Association.
†Submitted to the 1999 Usenix Technical Conference.

In Section 2, we describe the design of Snowflake. We describe the current prototype in Section 3. In Section 4, we discuss some lessons we have learned. Section 5 discusses performance characteristics of Snowflake. In Section 6, we outline our future work. Section 7 describes related work that set the stage for this project, and Section 8 summarizes the presentation.

2 Design characteristics

The Snowflake architecture is designed around modularized applications capable of communicating transparently across the network. Modules (objects) discover one another using names in a ubiquitous namespace. In Snowflake, the user has complete control over the namespace seen by the applications she runs. Modular applications retrieve configuration information and acquire external resources by resolving names in a namespace. Therefore, by remapping names to point to private configuration information or distributed resources, the user can arrange for applications and data to be transparently shared between administrative domains.

Naming is the organizing element of a distributed system. Hence, a *namespace interface* is central to Snowflake's design. In UNIX, the file system's namespace unifies naming of many system resources, including storage, devices, sockets, and in some cases, processes and other abstractions. Like such conventional systems, but to a greater degree, Snowflake unifies resource naming with a single namespace; unlike conventional systems, Snowflake decouples naming from storage. The naming graph can be cyclical, so there is no requirement that a subtree of names reside on common storage (such as a UNIX mount). There are many possible implementations of the namespace interface: trivial namespaces simply hash names to object references; proxy namespaces can hide distribution by returning remote proxy objects; emulation namespaces can provide a Snowflake view on an external service such as a database or a UNIX file system.

In a conventional system, the name given to a resource determines where the resource is stored or implemented. Because Snowflake names are decoupled from storage, we also define a *container interface* that exposes storage decisions to the user. Container interfaces can be invoked to create new resources (of a type appropriate to the container), or to copy resources from some other container. The advantage of a decoupled container interface is that a user can uniformly exploit all of her storage resources in multiple administrative domains, while ignoring the storage locations of her resources in her day-to-day work. For example, an object can be moved from a distant container to a nearby container for better performance, without changing its name. Similarly, a namespace can be used to map names into nearby replicas, or perhaps distant replicas if local replicas are unavailable.

Snowflake allows system administrators to retain desired control over local resources, to enforce local security and resource-allocation policies. Sometimes administrative requirements will conflict with the realization of transparent single-system images. The goal of the Snowflake architecture, then, is to enable transparency wherever possible. The administrative domain boundaries of existing systems artificially destroy transparency; Snowflake is designed to avoid that problem.

3 Current implementation

In this section, we present the interfaces and implementations of namespaces and containers. Then we describe how Snowflake interoperates with existing conventional systems. Finally, we discuss some applications available inside the prototype environment.

Snowflake is designed around modularized application components that communicate across the network. Many researchers have worked on architectures for modular applications, and in

Snowflake, we take the presence of such an architecture as a given. For our current prototype, then, we use Java objects and Remote Method Invocation as a substrate for distributed applications [WRW96, Fla97].

3.1 Namespaces and containers

We have defined a Java Namespace interface, which is simple as one might expect:

Object	`lookupName(`*String* `name)`
Object	`lookupPath(`*String* `path)`
String`[]`	`listAllNames()`
void	`bind(`*String* `name, `*Object* `objRef)`

There are many possible implementations of the namespace interface. Simple namespaces are implemented using hash tables. Fancier namespaces translate requests into lookups in existing services (such as a UNIX directory, or a web-accessible database). A "proxy" namespace provides a symbolic link by passing all requests through to another Snowflake namespace.

In any system, names resolve to some underlying addressing scheme. Design decisions include whether that addressing scheme should implement features such as network transparency or object replication. Snowflake names currently resolve to Java objects, which may be RMI remote stubs. The result is that the underlying address layer can include network transparency and object replication.

The container interface lets users allocate new objects of a given type, or copy an existing object into a container's storage:

Object	`allocate(`*String* `name, `*String* `className)`
Object	`store(`*Serializable* `name, `*Object* `objRef)`

Classes are passed by name to `allocate`, because Class objects cannot be `Serializable` or `Remote`. The `store` method copies the referenced object into the container using object serialization, and returns a reference to the new object. Containers are always namespaces, so namespace operations apply to containers as well. The namespace operation `bind(`*name*`, null)`, for example, is used to free a resource from a container. The `listAllNames()` operation can be used to discover how the storage in a container is being used. The resulting list names each object in the container as it is known to the container; the objects may also be referenced by more symbolic names from other namespaces.

We provide persistent storage by running a simple implementation of the Container interface inside a Java virtual machine that has been made orthogonally persistent using our *Icee* process checkpointer [How98]. The checkpointer snapshots the entire Java process, so that if the machine crashes, the JVM can be completely restored from a checkpoint file. Icee provides hooks so that a Container can involve itself in the checkpointing and recovery process, if desired.

3.2 Communicating with conventional systems

To exploit the body of applications and resources available from conventional operating systems, we have implemented some emulation services. First, UNIX files can be accessed from the Snowflake world by `UnixContainer`, an implementation of Container that resolves lookup requests in a UNIX directory. Looking up the name of a file returns an object that has `getInputStream()` and `getOutputStream()` methods. Looking up the name of a directory returns another `UnixContainer`.

Going in the opposite direction, we can access Snowflake objects from UNIX programs as if they were UNIX files. We can, for example, use `vi` to edit a Snowflake object that has file input and output methods. We used the technique designed by Alexandrov et al. for their UFO user-level file-system extension [AISS97]. We invoke `vi` from a monitor program that interposes on `vi`'s system calls using the `/proc` debugging interface. File-system directory operations are transformed into Namespace lookups and Container operations, and file accesses are rerouted to the resulting objects inside the Snowflake environment.

We created a third emulation technique to reuse existing Java code. Although Java has a nice object model, its class libraries still reek of conventional file-system architecture, and map well only to conventional file systems, with their notion of current working directory and assumption that a given system has only one root directory, rather than a namespace configured by each user. We wished to reuse Sun's `javac` Java compiler inside Snowflake, reading code from arbitrary Namespaces, and writing class files to arbitrary Containers. The first challenge was just to replace the functionality of library classes such as `java.io.File` and `java.io.InputStream`. If Java's class library used the Factory design pattern to instantiate file and file stream objects, we could simply alter the factory to return objects of our own devising. Unfortunately, Java's design and implementation are not so modular. Worse yet, if we simply replace the FileInputStream class (for example), we lose access to the UNIX file space provided by the original implementation. If we rename the class to preserve its functionality, sockets still break because the native methods of the `java.io.Socket` class depend on the internal representation of FileInputStream. Our solution was to modify client classes to instantiate our replacement class. We wrote a tool that filters class files, replacing references to FileInputStream with references to Snowflake's SFFileInputStream. We so modified all of the classes relevant to `javac`. The technique is unsavory, but it is a consequence of poor modularity in the Java class libraries.

3.3 Experimental services

Our current prototype has internal storage, a `cp` command for moving data into and out of the environment, and a version of the `javac` compiler that can read and write objects in the Snowflake namespace. Class files stored in a Snowflake container can be loaded into the Java Virtual Machine by a custom ClassLoader. The ClassLoader looks up the class file with the `lookupName()` operation, casts the resulting object to a stream, reads the stream into an array, and passes the array to Java for verification and loading.

We built a distributed calendar as an example of an application that benefits from seamlessly spanning administrative domains. In our calendar, individual event records are Remote, Serializable objects. A namespace stores a pool of events to be viewed by a user interface. Other namespaces can perform union, filter, and query operations on namespaces, presenting another namespace as output. A user can use a union namespace to join calendars from work, home, and school. In the prototype, he uses a shell[1] to create a Union namespace object and pass the Union a list of namespaces, after which the Union object acts like a namespace containing the union of the mappings in the given list. The user may next install a filter object to extract only those events that match his current interests. Finally, he views the resulting namespace with a user-interface tool that formats the event objects as a calendar.

The calendar represents the sort of service Snowflake provides. The user is free to arrange the distribution of his calendar's events by composing namespace objects and operations; the result

[1]In the prototype, the shell is textual. We envision graphical tools that manipulate namespace operators, and even tools that provide namespace operators tailored to specific applications. The important point is that the distribution mechanism is never hidden inside the application, out of the user's control.

is a single image of distributed event records. The mode of distribution is not defined by either administrators or the application programmer, but by the user. Intermediate tools may assist users in managing distribution, but distribution is decoupled from specific applications. In the same way that UNIX text programs naturally compose to produce useful pipelines, applications that follow Snowflake conventions naturally compose to produce user-configurable distributed applications.

4 Lessons learned and open questions

Our first prototype of the Snowflake design led to some lessons learned and suggested some questions about the nature of a user-configured distributed system.

- *Explicit interfaces are helpful.* Snowflake objects have explicit interfaces (an object can be queried by a meta-interface to discover what interfaces are available), and a given object can implement multiple interfaces.[2] This is valuable because it allows tools to discover the type(s) of objects, and it allows objects to be available to more tools by offering multiple interfaces. In the calendar example, a query object implements both a query interface, to specify the query that will select calendar events, and a namespace interface, to present the results of the query.

  UNIX files have only one interface, reading and writing a stream of bytes. Programmers circumvent this by encoding richer interfaces implicitly into file formats and stream protocols. The result is that a given resource can only have one interface, and in the absence of prior knowledge, a program can only guess at the format of a file. In contrast, Snowflake's explicit interfaces allow resources to expose multiple interfaces (to interoperate automatically with more tools), and avoid the guesswork of discovering what interface(s) a resource actually provides. Snowflake objects can always implement a stream interface when it is appropriate.

- *What context should Namespaces be coupled to?* In traditional systems, the namespace is coupled to the machine or the cluster: any process running in the cluster sees the same namespace root. In Plan 9, namespaces are per-process [PPD+95, PPT+93]. This allows per-user namespaces, in that each user's collection of processes can share the user's custom namespace. Currently in our prototype, objects must be passed a namespace that they store for later use. The `FileInputStream` replacement code uses one namespace for any callers in the virtual machine (it is stored in a static class variable). Perhaps the right model, emulating Plan 9, is to have one namespace per thread.

- *Are string names enough?* What if names could be arbitrary objects? Namespaces with such bindings would not necessarily be useful to browsing humans, but perhaps it would help the user organize his resources. For example, in the calendar application, a database of events could be represented by a namespace that, when given a special query object as a name to resolve, replies with another namespace containing matching events. However, allowing arbitrary objects as names may be overloading the namespace interface to provide functionality that should be left to other explicit interfaces. It may make caching namespaces even more difficult (See Section 5).

- *Can namespaces be arbitrary mappings?* Currently, namespaces are strictly functions by the definition of the Namespace interface: a name lookup operation can return only one object. What if namespaces could return duplicate keys? This would stray even farther from the

[2] This is a Snowflake specification, but the use of Java remote objects provides the implementation for free.

usual semantics associated with namespaces, but relaxing the unique key constraint would allow database schemas (such as our calendar example) without unique keys to map nicely into the namespace interface.

- *How transparent can we make the network?* Another question is how transparent the network should be to object implementations. A single-address space (SASOS) implementation would allow the network communication to be totally transparent, but would bring with it the known scalability problems of SASOS designs, as well as the likely difficulty of scaling SASOSes across administrative domains (see Section 7). Barring a shared-memory system, we expect that no system can hide the difference in copy versus reference semantics. Java certainly does not; it requires classes to declare themselves Serializable (providing network copy semantics), Remote (providing network reference semantics), or both (providing network reference semantics by default). Opal's designers point out that while a SASOS design saves copies inside the cluster, structure-copying code must still be written for the times when users want to move data off the cluster [CLFL94]. Waldo et al. have argued that network transparency is not even desirable [WWWK94].

- *How transparent can we make persistence?* Our system provides orthogonal persistence, which saves great programmer effort in applications with pointer-rich data structures that would otherwise need to provide flattening and parsing I/O code. However, should the programmer want copy semantics for some subset of his pointer-rich data, he cannot evade the same issues: determining how deeply a copy operation dives, and deciding when referential integrity should be preserved versus when a duplicate object should be made.

5 Performance

The performance of Snowflake is dependent upon the object reference layer that it is built upon. In our prototype, that layer is Java RMI. The actual impact on system performance is mostly related to whether the Snowflake design conflicts with caching. User-configurable namespaces allow a user (or a program operating on her behalf) to insert caching proxies to reduce network latency; in this regard the Snowflake organization lends itself to caching. On the other hand, because Snowflake objects can have arbitrary types, each with arbitrary consistency semantics, no single caching strategy will be appropriate for all object types. The alternative is a conventional distributed file system that can cache aggressively and preserve consistency, but bears the limitation of a strict interface.

One component of Snowflake, our container mechanism Icee, can be directly measured. Figure 1 shows the performance of Icee checkpointing a trivial process (a class with an empty `main` method) and a process containing a Java compiler, with all of its attendant classes and data structures. In each case, we show the results of synchronously writing the process image to disk, and copying the image in memory. The maximum coefficient of variation was 0.08. The dominating factor is I/O latency, which is not surprising considering that even the trivial process has a five-megabyte image. Icee does not yet incorporate well-known optimizations, including incremental and asynchronous checkpointing.

Icee incurs run-time overhead while recording system calls that may need to be replayed during a checkpoint recovery. This overhead is currently about 7%, measured with a maximum coefficient of variation of 0.04. We could reduce the overhead by exploiting new features of Solaris 2.6 that make it possible to obtain the same information directly at checkpoint time, rather than by interposing on system calls during program execution.

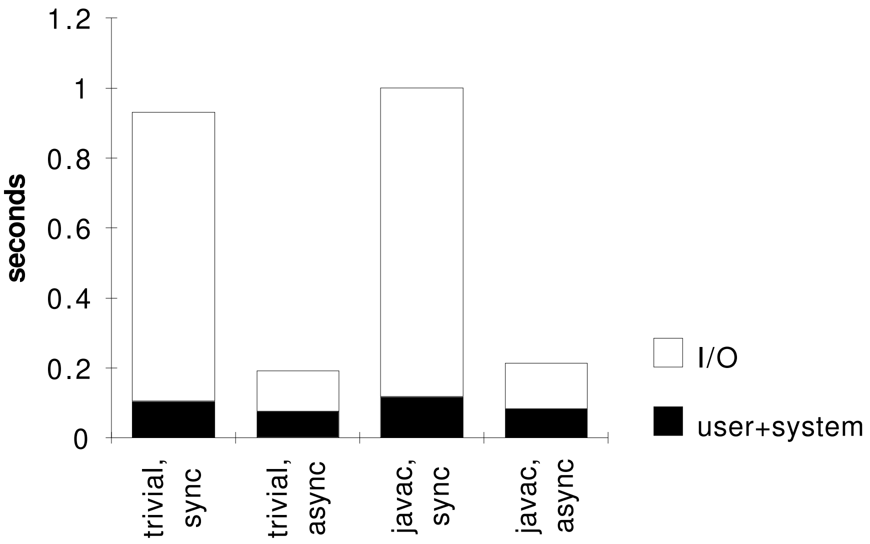

Figure 1: Checkpointer delay is reduced with memory copies.

6 Future directions

Our work to-date has focused on how to allow users to transparently access resources that span administrative domains. We have not addressed how the administrators that provide those resources will be able to control how those resources are used.

To export resources for use by Snowflake users, administrators will require varying degrees of control over the use of those resources. Some administrators will want to set local standards of authentication for users, to reduce the risk of their resources being stolen by imposters. Some administrators will want to be able to discourage their users from sharing resources with other outsiders. Some administrators will want detailed audit trails, to facilitate prevention and detection of unauthorized use.

Our future work on Snowflake will address how to deliver these features to administrators while minimizing the inconvenience seen by users. In some situations, administrative requirements will directly conflict with user convenience; in these instances Snowflake must side with the administrator, or the administrator will simply refuse to offer resources using the Snowflake model. But to the extent possible, our goal is to hide administrative boundaries from the users' experience.

7 Related work

Three categories of related work are especially relevant to our project. First, we examine single-system-image cluster software. Next, we look at worldwide systems. Finally, we discuss systems that give users greater configuration control than do conventional systems.

Single-system-image (SSI) cluster software allows administrators to unite a network of workstations into a single-system image. SSI clusters are centrally administered, and often do not scale well beyond local area networks. Important cluster systems based on kernel-to-kernel remote procedure call include MOS, Sprite, and Amoeba [BL85, OCD+88, TvRvS+90, MvRT+90]. Plan 9 uses a remote file protocol to distribute resources, with all sorts of resources mapped into the file (read/write) interface [PPD+95]. Spring is an object-oriented cluster system, with a sophisticated remote-method-invocation architecture that can hide the implementation of services such as replication in the method-invocation layer [MGH+94]. Solaris MC was an effort to merge the Spring cluster architecture with a commercial operating system [KBM+95]. The OSF Distributed Computing Environment (DCE) represents the standardization of several de facto distributed resource managers, also generally based on RPC [Joh91].

An important class of single-system-image clusters are those based on a Single Address Space Operating System (SASOS) architecture. In a SASOS, every process in the system shares a common address space, so that communication, replication and persistence can be hidden in the implementation of distributed shared memory. Significant examples of SASOS clusters include Opal, Angel, Mungi, and Brevix [CLFL94, MWSK94, HEV+98, FCH+93]. The SASOS architecture is a useful form of transparency, but to scale across arbitrary administrative domains a single address space would have to span the global Internet; that would introduce allocation and administrative problems [KC96].

Other systems aim to provide a world-wide distributed system. The worldwide approaches involve aggregating all resources into one large system, then partitioning that system among its users. Such a system would provide a single-system image across all administrative domains for all users. Examples include Legion, GLOBE, and Millenium/Borg [GWtL97, HST96, BDF+97, Ham98]. In Prospero, each user creates his own worldwide distributed file-system image. Prospero names refer to files of the conventional interface, and naming is coupled to storage [Neu92].

Another class of related systems provide users with extended control over resource configuration and distribution. Bershad and Pinkerton's watchdogs and 4.4BSD portals are kernel services that redirect file-system operations to user-mode processes [BP88, SP95]. UFO allows a user process to supply other processes with an alternate view of the file system namespace by interposing on those processes' system calls via the `/proc` debugging interface [AISS97].

In a Sprite cluster, every file and device is automatically remotely accessible, and user processes can implement pseudo-devices that look like files to clients of the file system [WO88]. Every Plan 9 process has its own mount table, allowing Plan 9 users to freely configure and distribute their resources around the cluster [PPT+93, PPD+95].

The SPACE "kernel-less system" and the Exokernel push large amounts of functionality out of the kernel into user-mode, where users can provide alternate abstractions [PB95, EK95, KEG+97]. In Fluke, the system is composed of recursive virtual machines. Therefore, rather than allocate resources from the administrator, users receive a complete virtual machine, and configure it as they please [FS96, FHL+96].

8 Summary

We have described the design and implementation of a distributed computing environment, called Snowflake, that allows users to construct and enjoy single-system images that span administrative domains. We discussed how Snowflake interacts with conventional systems, and described two applications currently available inside Snowflake: the Java compiler and a distributed calendar.

We discussed some lessons learned and some open questions, and outlined our directions for future work in security and administrative issues. While many questions remain to be answered, we feel that the premise behind Snowflake is sound. Putting distribution under user control will allow users to construct convenient, personal single-system images that span administrative domains, a necessity as computing resources become ever more ubiquitous and distributed.

Acknowledgements

Jon Howell expresses his appreciation to the USENIX organization for funding his research on this topic. Thanks also to Sun Microsystems for providing software used in this project.

References

[AISS97] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris Scheiman. A novel way of extending the operating system at the user level: the UFO global file system. *;login: the USENIX Association Newsletter*, 22(2):40–41, April 1997.

[BDF+97] W.J. Bolosky, R.P. Draves, R.P. Fitzgerald, C.W. Fraser, M.B. Jones, T.B. Knoblock, and R. Rashid. Operating system directions for the next millennium. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 106–110, 1997.

[BL85] A. Barak and A. Litman. MOS: a multicomputer distributed operating system. *Software—Practice and Experience*, 15(8):725–737, August 1985.

[BP88] B.N. Bershad and C.B. Pinkerton. Watchdogs — extending the unix file system. *Computing Systems*, 1(2):169–188, Spring 1988.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, pages 271–307, November 1994.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 78–83, 1995.

[FCH+93] Martin Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes. Brevix design 1.01. Technical Report HPL-OSR-93-22, Operating Systems Research Department, Hewlett-Packard Laboratories, April 1993.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.

[Fla97] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, second edition, 1997.

[FS96] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 91–105, October 1996.

[GWtL97] Andrew S. Grimshaw, Wm. A. Wulf, and the Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[Ham98] Scott Hamilton. Inside Microsoft Research. *Computer*, 31(1):51–58, January 1998.

[HEV+98] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software— Practice and Experience*, 28(9):901–928, July 1998.

[How98] Jon Howell. Straightforward Java persistence through checkpointing. In *Proceedings of the Third International Workshop on Persistence and Java*, Tiburon, CA, September 1998. Available at: `http://www.sunlabs.com/research/forest/com.sun.labs.pjw3.main.html`.

[HST96] Philip Homburg, Maarten van Steen, and Andrew S. Tanenbaum. Communication in GLOBE: an object-based worldwide operating system. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 43–47, October 1996.

[Joh91] Brad Curtis Johnson. Distributed computing environment framework. Technical Report DEV-DCE-TP6-1, Open Software Foundation, June 1991.

[KBM+95] Yousef A. Khalidi, Jose M. Bérnabéu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi-computer OS. Technical Report TR-95-48, Sun Microsystems Laboratories, November 1995.

[KC96] David Kotz and Preston Crow. The expected lifetime of single-address-space operating systems. *Computing Systems*, 9(3):155–178, Summer 1996.

[KEG+97] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[MGH+94] J.G. Mitchell, J.J. Gibbons, G. Hamilton, P.B. Kessler, Y.A. Khalidi, P. Kougiouris, P.W. Madany, M.N. Nelson, M.L. Powell, and S.R. Radia. An overview of the Spring system. In *Proceedings of COMPCON '94*, pages 122–131, 1994.

[MvRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[MWSK94] K. Murray, T. Wilkinson, T. Stiemerling, and P. Kelly. Angel: resource unification in a 64-bit microkernel. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 106–115, January 1994.

[Neu92] B. Clifford Neuman. The Prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, Fall 1992.

[OCD+88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[PB95] Dave Probert and John Bruno. Implementing operating systems without kernels. Technical Report TRCS95-24, University of California at Santa Barbara, December 1995.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *ACM Operating Systems Review*, 27(2):72–76, April 1993.

[SP95] W. Richard Stevens and Jan-Simon Pendry. Portals in 4.4BSD. In *Proceedings of the 1995 USENIX Technical Conference*, pages 1–10, January 1995.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[WO88] B.B. Welch and J.K. Ousterhout. Pseudo devices: user-level extensions to the Sprite file system. In *Proceedings of the 1988 Summer USENIX Conference*, pages 37–49, 1988.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, SunLabs, November 1994.