Snowflake: Breaking The Administrative Boundary

Jon Howell*

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

May 1, 1998

1 Introduction

Single-system images are an abstraction over networked hardware that gives users the convenience and simplicity of a single central computer, while exploiting the economies and scale of a network of nodes. Many research systems have created a single-system-image user interface over a network of back-end computation and storage hardware, usually to good effect. Examples include Plan 9, Amoeba, and Spring.[PPD+95, TvRvS+90, MvRT+90, MGH+94, NR94]

However, although these systems typically exploit the economies of networked hardware, they fail to capitalize on network scalability. In existing systems, a system administrator aggregates networked resources into a single-system-image cluster, and then users sit down at terminals and use the cluster as though it were one machine.

The problem is that today's users commonly require the services of resources in multiple administrative domains. Traditional clusters would either require that either the user use each cluster as a separate system image, or that the administrators of the clusters conspire to unite their services into a larger single-system image. The first approach defeats the purpose of a single-system image. The second clearly cannot scale: by extension of the Kevin Bacon Hypothesis, the set of all users is transitively closed, and therefore every computer in the world must belong to the same administrative domain.

Our solution to this problem is to introduce and implement a distributed system design based around the philosophy that all resource aggregation should be performed in "user space."[1]

2 Philosophies

First, a Snowflake implementation must supply a unified namespace interface that can be both accessed and implemented by ordinary users. The ability to implement parts of the namespace allows this one mechanism to subsume the many ad-hoc namespaces appearing in traditional UNIX-like systems. This is a feature that Plan 9 and Spring have shown to be not only feasible, but a simplifying concept.

*Supported by a research grant from the USENIX Association.

[1]When we refer user space, we mean the space of programs not requiring administrator privileges; we are not referring to the distinction between the user and kernel execution modes of a processor.

Second, once Snowflake objects have located one another via the unified namespace, they communicate using object-oriented messages over explicitly-typed interfaces. Typed interfaces provide two benefits. One advantage is that tools can discover whether or not it makes sense to operate on an object. UNIX and Plan 9, which use a least-common denominator byte-stream interface, appear to make all tools interoperable. However, in reality, byte streams contain some internal structure, which amounts to an implicit interface. The other advantage of explicit interfaces is that an object can export more than one interface. For example, a "directory" object, that exports a namespace interface, can also export another interface (e.g. a database query interface) if it is appropriate for the kind of data it contains.

Finally, Snowflake avoids declaring any distinguished namespaces. That is, unlike UNIX, there are no per-process signal actions. Unlike Plan 9, there is no per-process mount table. In UNIX, programs rely on being loaded into virtual memory at constant locations, so virtual addresses are a distinguished namespace; in Snowflake, for languages that can see virtual addresses, we will use position-independent code. When protection domains are used, there is always some per-domain index into a capability table for "extraprocess" communication: UNIX' file descriptor table, Mach ports, or a capability list. Snowflake programs depend on a common software layer to abstract away that distinguished namespace, so that two independent components may share a protection domain (and its associated capability table) transparently, without interfering with one another.

3 System Construction

With the system substrate and program components built according to the philosophies outlined above, we have carefully preserved the freedom of the user to control the aggregation of resources.

Because a component never expects to find resources at locations distinguished by the operating system substrate, the user has control over all aspects of its operation. Specifically, the users controls the namespace seen by the component. This handily frees users from programs that load configuration data from a system administrator-owned file. The user can arrange for namespace queries to return proxies to remote objects, implementing distribution transparent to the program.

This Snowflake philosophy discourages designs that limit a user's control over resource aggregation. The administrator only controls those resources that need to be limited for protection or resource allocation reasons, and (by convention) imposes no more abstraction than is needed. Each user aggregates resources acquired from multiple administrative domains into a personal single-system image. One is always free to draw resources from a new administrative domain into one's personal single-system image, never requiring the administrators to conjoin their domains.

4 Prototype Status

We are prototyping a system that conforms to the Snowflake philosophy in Java, exploiting Java's Remote Method Invocation to provide explicitly-typed interfaces. We have designed and implemented a namespace interface as Java interfaces. We are building a checkpointer for the Java Virtual Machine that gives us a persistent container in which to begin accumulating system objects.[How98] We have begun building applications that exploit the distributed nature of the Snowflake design.

Acknowledgements

Thanks to my advisor David Kotz for guidance in the project and his always thoughtful comments on writing. Sun Microsystems provided software used in this project.

References

[How98] Jon Howell. Straightforward Java persistence through checkpointing. Technical Report PCS-TR98-330, Dartmouth College, Computer Science, Hanover, NH, April 1998. Available at: `ftp://ftp.cs.dartmouth.edu/TR/TR98-330.ps.Z`.

[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.

[NR94] M.N. Nelson and S.R. Radia. A uniform name service for Spring's Unix environment. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 201–209, January 1994.

[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.

[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.