Strange Bedfellows: Issues in Object Naming Under Unix

Douglas B. Orr, Robert W. Mecklenburg and Ravindra Kuramkote

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

E-mail: dbo@cs.utah.edu, mecklen@cs.utah.edu, kuramkot@cs.utah.edu

Abstract

Naming plays a key role in the design of any system that exports services or resources. Object systems may export many different categories of names: instances, components of records, types, etc. Operating systems export the names of files, devices, and services. Integrating an object base with existing operating system facilities can improve accessibility of the object base resources. We consider the benefits and pitfalls of integrating an object base namespace with the Unix namespace.[1]

1 Introduction

Naming plays an important role in modern systems design. Uniform naming of files and devices, long recognized as an important innovation, was first made widely available by the Unix[2][RT78] operating system. Newer systems such as Plan 9[PPT+93] extend uniform naming to most system resources, including network connections, processes, etc. Object bases export names for a wide variety of smaller grain entities, such as instances, components of types, types, etc. Object bases share the same need as operating systems for contextual naming (e.g., "current directory") and hierarchical or relational organization of the entities they export.

[1]This research was sponsored by Hewlett-Packard's Research Grants Program and by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014–91–J–4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government, or Hewlett-Packard.

[2]Unix is a trademark of Unix System Laboratories.

The Mach Shared Objects project (MSO) is part of an ARPA initiative focusing on persistent object bases (POB)[LK92]. In the MSO project, we find ourselves faced with the questions not only of how to manage names within this object base, but how these names relate to the "real world" of the operating system within which the MSO Object Manager exists.

The marriage of an object base namespace with a traditional Unix namespace produces benefits and problems. The Unix naming framework has evolved over time to support a particular semantics. The framework includes facilities for hierarchical organization and protection, and supports a (fixed) set of operations that can be invoked on the objects found within the namespace. While this model is too limited to allow seamless incorporation of an object base into the Unix world, there are many advantages gained from using a single, uniform namespace. At the same time, integration within the Unix namespace places burdens on objects intended to be viewable as first-class Unix entities.

In this paper we discuss the issues involved with such a marriage. We begin by exploring the pros and cons of mandating a single namespace. We review the naming facilities required by a modern object base. We discuss features and restrictions of the framework for objects provided by Unix. Finally, we describe a strategy for merging the two namespaces and the power and limitations of the resulting combined facilities.

2 One Namespace or Many?

Early in the MSO project we encountered a ubiquity of names. The instances of objects maintained within our POB are externally accessible, and thus require names. Object instances are uniquely named by object identifiers (OIDs). String names (*symbolic labels*) are mapped onto objects represented by OIDs. Those objects are instances of classes, described by named *dossier* objects. Dossiers describe object instance layout and specify, by name, a class implementation. Class implementations reside within *modules*[BL92], which group and provide a naming scope for methods and static data elements. Finally, the MSO Object Manager is implemented under Unix, which maintains a namespace of files and devices of which the Object Manager implementation makes use.

Within this paper we concern ourselves primarily with the resolution and use of symbolic names. Given that so many of these namespaces exist, there exists the opportunity for gross duplication of effort in those areas common to all. Browsing, collection, protection, and simple I/O are facilities common to most or all objects contained within these namespaces. Implementation of a separate, similar facility for each different namespace would be a waste of effort, would tend to have inconsistent and non-uniform implementations, and is contrary to the standard OO doctrine concerning abstraction.

An analogous situation exists with a number of Internet namespaces, such as that defined by most implementations of the File Transfer Protocol (ftp). Objects in this namespace must be transferred to the local host via a manual mechanism before applications may use them. When Sun Microsystems added the Yellow Pages (later NIS) distributed database[Wei85] to complement their network file system, it was not integrated into the namespace. Sun was forced to provide alternate naming facilities (e.g., `yp_bind`, `yp_match`) and reimplement familiar utilities to work specifically within the domain of the Yellow Pages (e.g., `ypcat`, `ypmatch`). The Object Store system incorporates the Unix namespace into its namespace and provides its own mechanisms to access operating system facilities[LLOW91].

Clearly, it is desirable, to the greatest extent possible, to avoid this sort of duplication of effort. While working within a single namespace largely achieves that goal. Using a single namespace is not without its limitations, of course. Choosing to adopt a single convention means applying a single style of organization to all names, a single protection paradigm, etc. These restrictions may not be acceptable to some name providers but do not pose excessive problems within the MSO context. Therefore the MSO project has chosen to consolidate all its namespaces into one.

Given that one namespace is to be created, and that the existing Unix namespace is one component, the question arises as to whether it is better to adapt the Unix namespace to support that of the object base, or vice versa. The existence of a large number of useful utilities that manipulate the Unix namespace leads us to incorporate the MSO namespaces into the Unix namespace.

3 Naming and Invocation Facilities

In a complete POB, names are used in at least two contexts: naming persistent data and naming persistent class implementations. Each context imposes requirements on naming facilities. Persistent data requires that users be allowed to retrieve instances by name. The storage and retrieval of instances requires dossiers, as described earlier. Our view of persistent code is based on the Object/Meta-Object Server (OMOS)[OM92]. This server constructs class implementations from meta-object descriptions. These implementations and meta-objects must also be named. We discuss requirements for each of these contexts in turn.

In our POB persistent objects are considered to be rooted graphs of objects where each object may be provided with an OID. Any object with an OID can be bound to an externally visible name. The object namespace is inherently hierarchical. Users can explicitly create hierarchies of names for organizational purposes (e.g., directories). Object graphs form hierarchies of names when more than one object in a single graph is named. There is a syntactically seamless two-way mapping between the organizational namespace and the object name space.

Each object graph is actually the most recent view on a versioning hierarchy which records the modification history of the graph. This versioning is controlled by a set of library function calls which manage concurrency. In general, the version of an object is not considered part of its name, but as an attribute of the persistent entity. Thus, a simple `ls` of an object directory will reveal an entry for each unique named object, where a named object may have an arbitrary number of versions which are not immediately visible. Access to these versions must be specifiable by a specific version number or date.

It is highly desireable that location information not be encoded within the symbolic name. It should be possible to reconfigure an existing database without affecting the names of persistent objects. Protection must be incorporated into the naming scheme. Finally, use of this naming scheme should not impose undue burdens on application programs wishing to manipulate named objects. In the extreme case, it would be advantageous if applications which did not interpret the contents of files (e.g., `ls`, `cat`) could be used unchanged to manipulate named objects.

As noted above, persistent objects must include not only the data members of the object but also the implementations of function members. In MSO, this aspect of persistence is handled by the OMOS server. Executable objects are described in OMOS by meta-objects which contain a blueprint for constructing the method implementations from modules. The meta-object must be named, as must the component parts. To support type evolution, the names found within OMOS require a notion of "version" as well.

Through the names found in these two contexts, the MSO POB provides a rich facility for management of classes, access to arbitrary instances of objects, and the ability to invoke a wide variety of methods on those instances. These are the object base facilities we wish to access via the Unix namespace; any implementation must support the indicated requirements.

4 Unix Object Semantics

Merging namespaces implies the potential of uniform operations on objects found in the namespaces being merged. Since we have chosen to work within the Unix context, we need to examine the implications of this decision with respect to an object base composed of arbitrary object instances.

Fitting within the Unix framework implies a restriction of functionality. The two mechanisms provided for integrating a client name provider into the Unix framework, the VFS file system switch[Kle86] and the NFS protocol[ea85], support a fixed set of operations. The VFS switch operations are linked into the kernel and accessed through a virtual function table. The NFS operations (largely a subset of the VFS operations) are defined by a network protocol.

The Unix `open` operation resolves a string name, returning a handle to the object represented by that name in the form of a Unix file descriptor. The Unix file system operations defined by its provider are accessible via this handle. File systems providing a directory service support an acyclic graph structure accessed by name, through operations such as `lookup`, `remove`, `rename`, etc. Various kinds of attributes are set and reported by the `attr` operations. The Unix permission model is supported by `chown` and `chmod` operations. Operations on non-directory objects are mostly limited to simple byte stream and page I/O.

In addition to restrictions on available functionality within the Unix framework, Unix applications require that certain methods be implemented by an object if it is to be usable in an expected fashion. For example, `make` will only work properly if an update time is maintained for an object. As a result, if objects exported to the Unix namespace are to be treated as first-class Unix entities, it may not be sufficient to only implement a small number of I/O primitives to meaningfully export objects into Unix.

5 The Unholy Union

Interoperation with Unix applications and objects is highly desirable, but far from the full functionality desired for an object base. The methods available to objects within the POB cannot be limited to the small, fixed set available under Unix. As noted in the previous section, Unix provides us with a mechanism to resolve an object name to a object handle (the file descriptor) in a protected fashion. From this, we must develop mechanisms to invoke arbitrary operations on the underlying object.

Since the OID represents an absolute name for an object instance, independent of heterogeneous namespaces, it is a convenient and appropriate handle with which an application can perform operations on the instance. Use of this handle requires some additional form of authentication for security.

We propose a framework whereby string names are resolved to protected Unix handles. The protected Unix handle can then be used to derive an OID and a ticket with which authenticated access to the object can be verified. The OID and ticket can be used to invoke arbitrary methods on the underlying object. Applications wishing only to perform Unix operations (such as standard Unix programs) need not go through the process of dereferencing the protected Unix handle. Applications performing more sophisticated operations retain the benefits of a single namespace without losing the power of the wide range of methods available within an object base.

The interface between the POB and traditional Unix tools can be further enhanced by allowing "views" of objects to be obtained through extensions to the naming convention. We implement versioning as an extention to the standard Unix naming scheme, where `/<version>` is appended to the object name. This scheme has been implemented successfully in Apollo's Domain System Engineering Environment and [PPT+93]. Other extensions might include appending an operation name, `/<op>` (e.g.,

`/bytestream`). These operations could be mapped to member function operations, to allow an object to export arbitrary operations via the namespace.

6 Issues and Limitations

Adopting the Unix model is not without its problems. One problem involves basic object architecture. Objects typically contain state; a traditional role of object methods is to manipulate that state. Object access via a file descriptor is implicitly remote — the object state is typically maintained in a server and not in the application. Generalized method invocation not involving file descriptors, however, does not imply this restriction. As a result, we are left with the question of where to maintain the object state, and the problems of synchronization and consistency if the state resides in more than one place.

Access via file descriptor has the restriction that there is a system-imposed limit to the number of open file descriptors a process may hold at any given time. This limit tends to be small (20-256). As a result, file descriptors unacceptable for use in applications that use a large number of objects simultaneously. When used exclusively to establish a protected "backdoor" connection, however, the problem of restrictions on the number of open file descriptors is manageable.

Use of generalized objects via Unix is complicated by the expectations underlying many Unix applications. Standard utilities make assumptions that may no longer be valid in the face of the additional capabilities of generalized objects. For example, `cp` assumes that it can duplicate the structure of an object with the `open` operation and the contents using successive `read`s and `write`s. When using objects from a POB, the `open` operation may fail to reproduce the type information now implicit in their structure, and `read` and `write` may not be sufficient to reproduce their state. As a result, either standard Unix commands must be modified to work within the new, more sophisticated world, or their users must be made to understand the related changes in their operation.

Finally, Implementation of a name provider under Unix has its own set of challenges. The world of Unix namespaces is not the friendly and easily extensible place the existence of the VFS file system switch might lead one to believe. The VFS file system switch was introduced by Sun as an abstraction of the file system designed to allow constructing a single namespace from multiple file system types.

Unfortunately, the realities of VFS are less elegant than the concept. In reality, there are a number of VFS interfaces, since most manufacturers have chosen to make their own extensions from the base VFS developed by Sun. In addition, since the manufacturers are the primary clients of VFS, the VFS API and implementation tend to change subtly without notice[Web93]. And, since not all manufacturers support dynamic linking of kernel modules it is sometimes necessary to specially configure and statically link new file system support into the kernel.

An alternative option is extend the Unix namespace via servers implementing the NFS protocol. Using the NFS protocol has the distinct advantage that the NFS protocols are much more rigidly fixed and uniform than the specifications for VFS. An NFS framework has the disadvantage that the semantics of NFS are stateless, implying inherent difficulty when used in a stateful context. In addition, the set of methods supported under NFS is a subset of those supported under VFS, further restricting the possible derived Unix functionality.

7 Conclusion

Maintaining a single namespace for operating system and object base objects gives the advantage of uniformity and avoids duplication of functionality of common applications. While it is not possible to support arbitrary object operations within the existing Unix context, using it as the basis for naming provides enough functionality to be very useful. We have proposed a simple model to allow access to the additional functionality not provided for by Unix.

Acknowledgements

We would like to thank Gary Lindstrom and Charles Clark for much help in reviewing this document.

References

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20-23 1992. IEEE Computer Society. Also available as Technical Report UUCS-91-017.

[ea85] R. Sandberg et al. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, 1985. Usenix Association.

[Kle86] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference*, pages 238–247, Atlanta, Georgia, 1986. Usenix Association.

[LK92] Gary Lindstrom and Robert R. Kessler. Mach Shared Objects. In *Proceedings Software Technology Conference*, pages 279–280, Los Angeles, CA, April 1992. DARPA SISTO.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[OM92] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[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):72–75, April 1993.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1930, July/August 1978.

[Web93] Neil Webber. Operating system support for portable filesystem extensions. In *Proceedings of the Winter 1993 USENIX Conference*, pages 219–228, San Diego, California, January 25-29, 1993. Usenix Association.

[Wei85] P. Weiss. Yellow Pages protocol specification. Technical report, Sun Microsystems, Inc., 1985.