Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance

David R. Cheriton, Timothy P. Mann
Computer Science Department
Stanford University

April 3, 1997

Abstract

Naming is an important aspect of distributed system design. A naming system allows users and programs to assign character-string names to objects and subsequently use the names to refer to those objects. With the interconnection of clusters of computers by wide-area networks and internetworks, the domain over which naming systems must function is growing to encompass the entire world.

In this paper, we address the problem of a global naming system, proposing a three-level naming architecture that consists of global, administrational, and managerial naming mechanisms, each optimized to meet the performance, reliability, and security requirements at its own level. We focus in particular on a decentralized approach to the lower levels, in which naming is handled directly by the managers of the named objects. Client name caching and multicast are exploited to implement name mapping with almost optimum performance and fault tolerance. We also show how the naming system can be made secure. Our conclusions are bolstered by experience with an implementation in the V distributed operating system.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Distributed Systems; D.4.3 [**Operating Systems**]: File Systems Management—*directory structures, distributed file systems*; D.4.7 [**Operating Systems**]: Organization and Design.

General Terms: Design, experimentation, measurement, performance, reliability.

Additional Key Words and Phrases: Naming, distributed system, fault tolerance, cache, multicast.

Authors' present addresses: D. R. Cheriton, Computer Science Department, Stanford University, Stanford, CA 94305; T. P. Mann, Digital Equipment Corporation–Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

This work was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431, by the Digital Equipment Corporation, and by the IBM Corporation under a Graduate Fellowship.

1 Introduction

Naming is an important aspect of distributed system design. A naming system allows users and programs to assign character-string names to objects and subsequently use the names to refer to those objects. Named objects commonly include hosts, electronic mailboxes, and files, as well as (less commonly) programs in execution, display windows, and network connections. With the interconnection of clusters of computers by wide-area networks and internetworks, the domain over which naming systems must function has grown to encompass the entire world. For example, it is desirable to be able to name any mailbox anywhere in the world, and quite unacceptable to be unable to name a mailbox with which one could otherwise communicate.

Large scale naming systems are subject to challenging reliability, security, and administrative requirements. A large system must continue to function reliably in spite of the failure of individual hardware

components. Moreover, a naming system must provide trustworthy service even though parts of the system may belong to many autonomous and mutually-suspicious groups—different departments, corporations, and even countries. Some progress on this large-scale design problem has been reported [2, 13, 16, 19, 22]; however, the performance of these systems appears inadequate for file naming—at best, it is adequate for naming hosts, mailboxes, and other relatively infrequently accessed objects.

Name lookup operations are a significant factor in system performance. For instance, Leffler et al. [17] attribute 40 percent of the system call overhead in UNIX to file name resolution. Also, Mogul's measurements of UNIX system call frequency [21] indicate that name mapping operations (`open`, `stat`, `lstat`) constitute over 50 percent of the file system calls. The frequent use of these functions results from accessing many small files (so there are typically few read or write operations per file open), as well as the frequent need to access file property information, such as the time of last modification.

In this paper, we describe a three-level naming architecture that consists of global, administrational, and managerial directory systems, each optimized to meet the performance, reliability, and security requirements at its own level. We focus in particular on a decentralized approach to the lower levels, in which naming is handled directly by the managers of the named objects; at the uppermost (global) level, we use familiar design ideas developed by others [16, 13]. Client name caching and multicast are exploited to implement name mapping with almost optimal performance and fault tolerance, recognizing that most named objects reside at the lower levels of a hierarchical name space. We also show how the naming system can be made secure. Based on our analysis of this design and our experience with its implementation in the V distributed operating system [7], we see the design as a sound basis for the implementation of a truly global naming system for distributed systems.

Our naming system is intended to serve a fully-connected internetwork that includes a multicast facility. Although the internetwork may be made up of many smaller component networks under separate administration, we assume that the (inter)network service can deliver a datagram from any connected host to any other, and that the address of the destination is independent of the location of the sender. We also assume that any set of hosts can form a *group* with a single address, to which any client can send (multicast) datagrams. Neither the hosts that send to the group nor the members themselves are required to possess a complete list of members; they need only know the group address. Datagram delivery to each group member succeeds or fails independently of delivery to the others, and failures are not necessarily reported to the sender. Multicast communication of this sort is available at the interprocess communication level in the V system [10], experimentally at the IP datagram level in the DARPA Internet [8, 11, 12], and at the data link level in the Ethernet [14].[1]

The next section describes the naming system design in some detail. Section 3 evaluates the system's performance, presenting an analytical model and validating the model by comparing it with measurements on the V implementation. Section 4 analyzes the fault tolerance of the system, showing that (for nearby objects) the system achieves optimum resiliency, in the sense that whenever an object is accessible at all, it is accessible *by name.* Section 5 outlines the problem of making the naming system secure and describes a solution. Section 6 compares our approach with related work. Section 7 presents our conclusions, notes some open issues, and proposes directions for future work.

2 The Naming System Design

Our naming system provides the ability to give objects conventional, hierarchically structured character string names and subsequently refer to the objects by these names. We use a name syntax in which the `%` character designates the root and the `/` character separates name components.[2] For example, the name `%edu/stanford/dsg/bin/listdir` is an absolute name with 5 name components.

The naming system provides three general classes of operations:

Binding – binding a name to an object, removing such a binding, or altering a binding.

Query – checking whether a given name is bound, listing the bound names in a given directory, etc.

[1] Our naming system can also be adapted to work with more restricted multicast facilities, such as one that provides multicast only within individual component networks of the internetwork.

[2] Our present implementation uses `[` in place of `%` for compatibility with an older V naming system [9].

Mapping – looking up the binding of a given name and delivering an operation request to the bound object's manager. A familiar example is the file open operation, where the requesting client specifies a file by name, and the naming system directs the request to the appropriate file server.

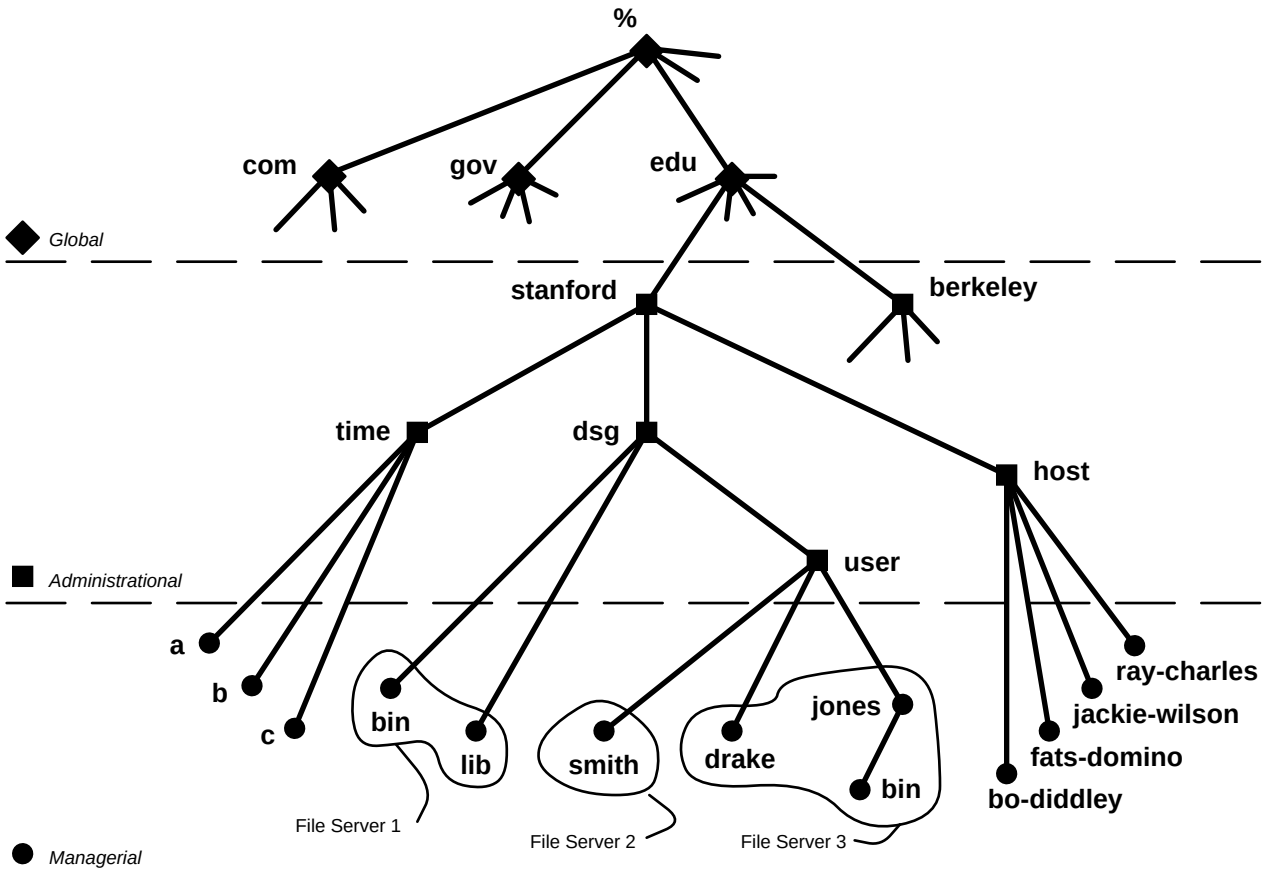

Figure 1: Three Levels of the Naming System

The directories in our naming design fall into three classes, or *levels*—*global, administrational* and *managerial*—as suggested by the small example in Figure 1. The levels are distinguished by their different performance, reliability, security, and administrative requirements.

Directories at the highest level—the *global* level—contain entries that represent the organizations and groups of organizations participating in the naming system. In Figure 1, for example, `%edu` is the directory of educational institutions, whose entries include `%edu/stanford`, designating the organization Stanford University, and `%edu/berkeley`, designating the organization U. C. Berkeley. There is generally no administrative superstructure covering the organizations named in a global directory; they are independent and (to some degree) mutually distrustful. Thus, security is an important consideration for these directories; for example, it is desirable to enforce the convention that Stanford does not use names starting with `%edu/berkeley` and Berkeley does not use names starting with `%edu/stanford`. High availability is also important because the failure of a global directory would make a large subtree of the naming system inaccessible; fortunately, the slow rate of change of naming information at this level makes a high degree of replication practical. Performance is less critical because clients keep cached copies of the information in these directories.

Directories at the second level—the *administrational* level—are owned and administered by individual, unified organizations. The entries in an administrational directory represent lower-level directories and services belonging to the same administration. For example, `%edu/stanford` is Stanford University's

highest-level administrational directory, while `%edu/stanford/dsg` is the root directory for a subordinate administration, the Distributed Systems Group. The administrational and global levels differ in three significant ways. First, trust is hierarchical at the administrational level, simplifying the security issues. That is, subordinate administrations may distrust one another, but they accept the authority of the parent administration in resolving name conflicts and other disagreements. Second, the administrational level in each administrative domain is critical to the functioning of that domain, while the global directory system is merely a connection to the outside world. Thus, each administrational directory should continue to be available to local clients in spite of failure of, or disconnection from, the global directory service. Finally, the entries of an administrational directory represent more localized and dynamic information than those of the global level, making it feasible, if not necessary, to implement these directories using techniques such as multicast that would not be appropriate at the global level. As with the global level, performance is a secondary consideration because of client name caching.

Each directory at the lowest level, the *managerial* level, is stored by a single object manager; its entries name objects and directories implemented by that manager.[3] For example (in Figure 1), `%edu/stanford/dsg/user/jones` is a managerial directory whose entries are files and directories belonging to a single user and stored on a single file server. The name `%edu/stanford/dsg/jones/bin` is Jones's directory of binary program images, also a managerial directory managed by the same file server. Although files are the prime example, any kind of object can be named using a directory implemented by its object manager. In V, for example, display windows, programs in execution, and network connections are all named in this way. The requirements on the managerial level differ from the higher levels in several ways. First, high performance for both lookup and update are important because managerial directories are accessed and updated frequently. Caching has less performance benefit at this level because of the rapid update rate. Second, the required availability of managerial directories varies, depending on the objects they name. A highly available object manager, such as a replicated file system, must have equally highly available managerial directories; moreover, each object manager should be able to function in the absence of the administrational and global directory levels. But a managerial directory need not be available when the object manager itself is unavailable. For example unreplicated files in the directory `%edu/stanford/dsg/user/jones` should remain accessible within Stanford as long as DSG file server 3 is up and accessible from the campus network, even if the global and administrational directory levels providing `%edu/stanford/dsg` are temporarily unavailable. However, if the file server is down or otherwise inaccessible, Jones's directory can be unavailable as well. Third, the need for security in managerial directories also varies depending on the objects they name. A highly secure object manager should be able to carry the same security over to its directories. Conversely, an unprotected manager should not be forced to accept an unwanted and inappropriate security overhead on its directories.

The following sections describe the realization of each level in our design as motivated by the above characteristics, proceeding from the managerial level to the global level.

2.1 Managerial Directories

A *managerial directory* is implemented by the object manager that implements the objects named in the directory. The directories implemented by one object manager represent one or more complete subtrees of the name space, *covering* that portion of the name space.[4] The object manager also stores the absolute name of the root of each subtree. For example (in Figure 1), the subtrees rooted at the directories `%edu/stanford/dsg/bin` and `%edu/stanford/dsg/lib` are both implemented by DSG file server 1, which thus covers all the names with prefixes `%edu/stanford/dsg/bin` and `%edu/stanford/dsg/lib`. Accordingly, file server 1 stores all the files and directories under these two subtrees.

Each object manager directly implements all operations on the names it covers. On receiving a naming operation request, the manager looks among the subtrees it implements to find the one whose name is a prefix of the name supplied in the request. It then completes the operation using the directories in that

[3] A single object that is logically replicated or partitioned among several object managers is treated by the naming system as a collection of *subobjects,* all bound to the same name. The naming system supports locating at least one of the subobject managers. Object-specific protocols are required to access the other managers, such as is required for an update operation.

[4] We say that an entity *covers* a name if it authoritatively knows either the definition of the name or that the name is undefined.

subtree. (We assume for the moment that the requesting client knows which manager covers the name and thus sends the request directly to the right one.)

Integrating naming with object management in this way has several advantages. First, when a client requests an operation on an object specified by name, both the name lookup and the object operation can be completed in a single message exchange. Further, the reliability of the naming matches that of the object implementation: an object's name can be looked up whenever the object itself is available—whenever its manager is up and accessible. This property of our design contrasts with designs in which a file server can be up and connected to the network, but unavailable for use because the (separate) name server is down. Moreover, the replication of an object manager for added reliability results in the replication of the name bindings of its objects as well. The close coupling of the name and object implementations also facilitates maintaining consistency between objects, object names, and object properties, because all the information for an object is maintained in one server. The naming implementation can be customized and optimized for each type of object manager, where this is of benefit.[5] Finally, the security mechanism for communicating with the manager and controlling access to information can automatically be applied to the naming operations as well.

Taken together, the managerial directories record every name-to-object binding in the system, but several additional pieces are needed to construct a complete naming service. First, clients need a way to find out—efficiently, reliably, and securely—*which* manager covers any given name, so they can send their requests to the right manager. Next, clients need a practical way to tell when a name is *unbound.* With the mechanism described so far, some unbound names are not covered by any object manager, and clients have no way of distinguishing such names from names bound by a manager that is temporarily down or unavailable. In addition, some mechanism is needed to implement operations on directories above the managerial level—for example, a means to list the entries in `%edu/stanford/dsg`, even though each entry is a managerial directory that is implemented by a different server machine.

The following subsections describe the parts of the design that implement these additional services. Efficient, fault-tolerant manager location is provided by name prefix caches and multicast, while coverage of unbound names and management of higher-level directories are provided by administrational and global directory managers.

2.2 Name Prefix Caches and Multicast

Each client of the naming system maintains a *name prefix cache.* The cache is a set of *entries,* each associating a name prefix[6] with a *directory identifier.* A directory identifier consists of two fields: manager identifier and specific directory identifier, where the specific directory identifier is assigned to the directory by the manager. When a client program invokes a name mapping operation, a runtime library routine in the client's address space searches the client's cache for the longest entry that is a prefix of the given name, and uses the result to decide where to send the operation request. Binding and query operations also use the name cache to locate the right manager.

A cache search is considered a *hit* when it returns a cache entry containing the name of a managerial directory. In this case, the client sends its request directly to the server machine on which the manager is running, using the manager identifier in the cache entry to address it. When the manager receives the request, it maps the given specific directory identifier to a directory descriptor, verifies that the name of that directory matches the name prefix specified in the request, and maps the rest of the name starting from the identified directory, thereby saving the work of looking up the entire name. It then carries out the requested operation if possible and returns the results to the client.

When the cache does not hit on a managerial directory, but does return some information, we call the result a *near miss.* A near miss can either return a cache entry corresponding to a local administrational directory, or an entry corresponding to a directory outside the local administration.

If the cache returns a local administrational entry, the client multicasts a *probe* request on the given name to the group of managers specified by the cache entry, adds the information in the response to its cache,

[5] The V implementation also provides a standard library of manager naming routines, for use when customization is not needed.

[6] A pathname n_1 is considered a prefix of n_2 if the components of n_1 respectively match the initial components of n_2; so `%a` is a prefix of `%a/b`, but not of `%ab`.

and then proceeds as in the cache hit case. In general, a probe request takes a name as its argument, and returns a cache entry associating a prefix of the given name with a directory identifier. The prefix returned is the shortest one that names a managerial directory, if any does, otherwise the whole name; the response comes from the manager (or managers) that cover the prefix. For example (referring to Figure 1), if the name being mapped is `%edu/stanford/dsg/user/smith/mail`, and the client's cache contains an entry for `%edu/stanford/dsg/user`, then the cache returns a directory identifier (m, s) for this entry. The embedded manager identifier m addresses the group of object managers that collectively cover `%edu/stanford/dsg/user`—the *participants* in that directory—in this case including DSG file servers 2 and 3.[7] The probe operation to the group m receives a response from file server 2 containing the directory identifier (m', s') for `%edu/stanford/dsg/user/smith`; this information is added to the client cache; and the client proceeds by sending its request to file server 2 using the address m'.

If the name corresponds to a directory outside of the local administration, the cache returns a directory identifier that specifies a *liason server,* a local server that serves as a cache and front-end to the global directory system. The client then sends a probe request on the given name to the liason server. As above, the probe request returns a name prefix and directory identifier for the first managerial directory in the given pathname. (If the pathname is not long enough to reach a managerial directory, the probe operation returns a directory identifier specifying the liason server itself as the manager.)

The client ordinarily never experiences a complete miss, because the cache is primed with a cache entry associating the root directory name ("`%`") with a local liason server. However, if this entry becomes stale (due to a liason server crash, for example), as a last resort the client multicasts a probe request to all *nearby* object managers and liason servers (say, all those within one hop on the internetwork). If any of those managers covers the name, it responds to the probe, just as in the local administrational near miss case. In addition, if a liason server receives the probe, it responds with a new cache entry for the root directory. This mechanism can be implemented using *scoped multicast*—the ability to restrict a multicast to those members of the addressed group that are within a specified distance from the sender [8].

Cache consistency is maintained by detecting and discarding stale cache entries *on use.* A cache entry becomes *stale* when its directory identifier is out of date; either the specified manager no longer covers the name associated with the cache entry or the specific directory identifier no longer identifies a directory by that name. Thus, when a client tries to use a stale cache entry, it ends up sending its name request or probe to the wrong manager (or group), or providing an incorrect specific directory identifier. If the manager no longer exists, the client receives no reply and times out. If the manager exists but no longer covers the given name, or the specific directory identifier is incorrect, the manager reports that fact back to the client. In either case, the client recognizes that its cache entry is (or may be) stale, deletes the entry, and issues a probe for up-to-date cache data. If the client originally received the cached information from a liason server, it asks the liason server to reverify the correctness of its information. This client feedback effectively causes the liason server to detect stale data in its own cache on use (by clients) as well.

The name caching mechanism we have just described has three major advantages.

First, matching on a prefix rather than on the full name allows a small number of cache entries to cover a large number of names, resulting in high cache hit ratios and good performance, as detailed in Section 3. For example, if `%edu/stanford/dsg/lib` is implemented on one manager, a single cache entry allows every naming operation on files in this subtree to go directly to the correct manager, without the overhead of a multicast or global directory lookup. The subtree rooted at `lib` can reasonably contain thousands of files.

Further, even when a client's cache does not hit on a managerial directory, a near miss can still provide information that reduces the amount of work required of the shared naming system. In particular, a near miss at the administrational level typically reduces the scope of multicast for the resulting probe to a small subset of the administration's object managers, because the probe is only multicast to the participants in the directory returned by the cache search. The longer the prefix returned by a near miss, the more work is saved. For instance, the participant group of `%edu/stanford` would include all object managers at Stanford—hundreds, if not thousands of computers—making a multicast to this group very expensive. But the participant group of `%edu/stanford/dsg` is much smaller, and that of `%edu/stanford/dsg/user` is smaller yet.

[7] As discussed in Section 2.4 below, an administrational directory manager is also included among the participants; it covers the unbound names in the `user` directory.

Finally, with on-use cache consistency checking, there is no need to inform all clients when a cache datum they are storing becomes invalid, yet stale data never causes a name to be mapped to the wrong object. This consistency protocol exploits the fact that cache data is used only by clients to decide *where* to send messages, not to allow them to perform operations locally. Thus, the correctness of the cached information can be (and is) automatically checked on each use.

2.3 Refinements

Several important refinements on this basic caching scheme are incorporated into our V implementation.

First, directory identifiers are temporary, volatile identifiers. Their binding to directories need not be preserved across server reboots, and they can be reused. In fact, a manager can freely invalidate a directory identifier at any time, the only penalty being poorer performance of name requests that subsequently use that identifier—cache entries that contain the identifier become stale and must be refreshed the next time they are used. On-use cache consistency checking ensures that invalidation and reuse of directory identifiers never causes names to be mapped incorrectly. For example, suppose a client adds to its cache an entry that associates the name `%edu/stanford/foo` with the identifier $(142, 857)$, but $(142, 857)$ is later invalidated by the object manager and reused to identify `%edu/stanford/bar`. Then suppose the client attempts to open a file called `%edu/stanford/foo/output`. Although the client does find the stale entry in its cache, prompting it to send object manager 142 a request citing specific directory identifier 857, the manager does *not* mistakenly open `%edu/stanford/bar/output` in place of the requested file. Instead, after looking up directory 857, the manager recognizes that the directory's name does not match the name provided in the client's request, and returns an error indication to the client. The client then discards its stale cache entry and probes for a new one.

It is useful for both manager identifiers and specific directory identifiers to be volatile. On the DARPA Internet, for example, one might construct a manager's identifier from the Internet address of the host currently running it, together with a socket number or other local identifier. With this representation, a manager's identifier must change whenever it migrates to another host, and it may change whenever the manager program is restarted, depending on how socket numbers are allocated. Within each manager, one might allocate specific directory identifiers as hash indices into the manager's internal memory data structures, pointing to the associated directories. Thus, management of directory identifiers can be implemented entirely with in-memory data structures; they need no stable storage. We have found in-memory implementation to be particularly convenient for object managers running at the guest level in existing operating systems, such as the one we have implemented on UNIX to provide file access from V.

A further extension of the cache mechanism provides a simple and efficient implementation of *current working directory,* as in UNIX [23]. In V, the runtime system in each client's address space stores the absolute name and directory identifier for the client's current working directory. When the runtime system syntactically recognizes a relative name, it prefixes it with the working directory's absolute name, then (unless a longer prefix match exists in the cache), sends off the request using the associated directory identifier. The client program is spared the inconvenience of supplying the absolute name on each request, and the object manager is spared the work of looking up the entire name. If the working directory identifier becomes invalid, the runtime system discards it and requests a new one using the stored absolute name, just as is done with stale cache entries.

Finally, client caches can be preloaded with a number of commonly used entries on initialization to reduce the initial miss cost on startup. Cache preloading is particularly important in the V implementation, where every program has a separate cache in its own address space. In this implementation, a program's initial cache contents are inherited from its parent command interpreter or "shell." Placing a name cache in the address space of every program makes the caches efficient to access, simplifies the implementation of on-use consistency checking, and facilitates transparent program migration [31].

2.4 Administrational Directories

Administrational directories are implemented in a decentralized fashion using object managers and *administrational directory managers;* collectively the managers that cooperate in implementing an administrational

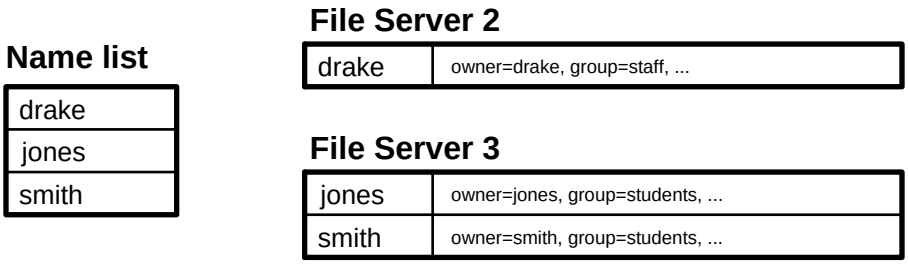

Figure 2: Distribution of Administrational Directory Information

directory are called its *participants,* and they form a multicast group called its *participant group.* An administrational directory manager covers the unbound names in each directory it manages, while the bound names are covered by the object managers that implement objects named relative to the directory. Figure 2 illustrates how information is distributed in the directory `%edu/stanford/dsg/user` of Figure 1. In the example, we say that file server 2 covers the name `drake`, because it knows what that name is bound to, while file server 3 covers the names `jones` and `smith`. The administrational directory manager holds a list of bound names, but it is not considered to cover these names, because it does not know what objects they are bound to.[8] The directory manager does, however, cover all the unbound names in the directory, because it knows that any name not on its list of bound names is unbound.

As another example (again referring to Figure 1), the participants in `%edu/stanford/dsg` are file servers 1, 2, and 3, plus a directory manager. File server 1 covers the names `bin` and `lib`, while both file servers 2 and 3 covers `user`. The directory manager covers the remaining unbound names.

Each participant in an administrational directory responds to probes on the names it covers. For example, file server 3 would respond to a probe on the name `%edu/stanford/dsg/user/jones`, while the directory manager would respond (with an error indication) to a probe on the unbound name `%edu/stanford/dsg/user/robinson`. Every name is covered by at least one participant, so if a client probes a name and receives no response, it can infer that the manager that covers the name is down or inaccessible. To get more information about the problem, the client can then attempt to query the directory manager directly. For example, if a client receives no response to its probe on `%edu/stanford/dsg/lib`, it can then query the directory manager for `%edu/stanford/dsg`. If the directory manager responds, it confirms that `lib` is bound and return any information it has about `lib`'s manager.

Administrational directory listing is coordinated by the directory's manager. A client that needs only a list of names simply obtains it from the directory manager; if it needs more information about the named objects, it contacts their managers. Note that if all administrational directory managers fail, local clients can still obtain a "best efforts" partial directory listing by multicasting a request to the group of participants and collating the replies that come in. There is, however, no way to know if such a listing is complete—some names will be missing if one of the participating managers is down or inaccessible over the network.

An administrational directory's manager can serves to coordinate access to the directory by clients located outside the local administration. A client accesses a remote administrational directory through the local liason server and the global directory system; these servers direct the client to the proper administrational directory manager. Thus, a client can list an administrational directory and probe names in the directory without having to multicast outside of its own administrational domain. (There is, in fact, no need in this design to multicast over a domain larger than a single administration.)

The main advantage of this technique for implementing administrational directories is that it insulates each object manager participating in an administrational directory from faults in the other participants—even faults in the administrational directory manager itself. For example, even if file server 3 and the directory manager for `%edu/stanford/dsg/user` are both down, file server 2 can still respond to name mapping requests and probes on `%edu/stanford/dsg/user/smith`. The price one pays for this insulation

[8] The directory manager may know which object manager covers each name on its list, but such information is not necessary for name mapping; it is primarily useful in fault diagnosis and recovery.

is that probes are multicast to all participants, imposing a load on all of them. Probes, however, are only generated when a client's cache misses, and as we show in Section 3, it is easy to achieve high enough cache hit ratios to make the multicast load insignificant.

Although name mapping in an administrational directory can continue in the absence of its directory manager, name binding and unbinding operations cannot (they require access to the list of bound names), so it is still useful to replicate the manager, or at least to replicate the information it stores. We believe it is most practical to implement an administrational directory manager as a simple, stateless program that can run on any host, accessing data kept in an underlying replicated storage system—perhaps in files on replicated file servers, or even in directories stored by the global directory servers. It seems convenient to include the functions of the liason server in the same program as well, particularly if administrational directory data is stored in the global directory service.

2.5 Global Directories

Unlike the administrational and managerial levels, directories at the global level store information that is cooperatively managed and widely distributed among many administrations. Lampson [16] presents a credible design for a truly global (worldwide) directory system of this type, with a high degree of replication (using gradual propagation of updates) to support a high degree of availability. Demers et al. [13] also present some algorithms for relaxed update of replicated information. We believe these approaches are sound bases for the construction of a global directory system, and simply assume the existence of such a system with the required scale and availability. Thus, we focus on interfacing to such a global directory system, not on its design.

The global directory system is accessed through the liason servers, acting as *front ends.* They act as intermediaries for all client operations at the global level, translating if necessary between the client protocol (used at the managerial and administrative directory levels) and the global directory system client interface. Thus, our design can be used with existing global directory systems and can easily be modified to work with new directory systems. The caching performed by the liason servers improves the expected response time for global level queries and reduces the load on the global directory system. An alternative approach would be to ask clients to deal with the global directory system directly. This approach requires that either the client know the specific protocol used by the global directory system, or that the global system use the same protocol as the managerial and administrational directory managers. It also eliminates the level of caching provided in the directory managers.

If the global directory system becomes unavailable within some administration, that administration continues to function autonomously but loses its ability to reference remote objects (outside the administration). Given the degree of replication used within the global directory system designs cited above, we expect this scenario to arise only when the administration is disconnected from the rest of the world, in which case remote objects would be inaccessible even if their names could be looked up.

This section has given a general description of our naming system design, partitioned into managerial, administrational, and global directory levels. The following sections examine the performance, reliability, and security aspects of this design.

3 Performance

Decentralized naming relies heavily on prefix caching for performance; without caching, its name mapping protocol would not be efficient enough for use in large systems. The inefficiency arises because each multicast to an administrational directory's participant group imposes a load on every participant. With a high enough cache hit ratio, however, multicast is avoided on most requests, dramatically improving the average efficiency. The hit ratio also plays a large role in determining where the boundary between global and administrational directories should go. As it increases, multicasts become less frequent, so larger directories can be handled satisfactorily with administrational techniques. Our discussion of performance therefore focuses on the effectiveness of caching.

To simplify the exposition, we initially discuss systems configured with no global directories—systems where even the root directory is implemented using administrational techniques—then extend the results to

global configurations.

3.1 Load Per Operation

We evaluate the processing load imposed by naming operations by counting *packet events.* A packet event is the transmission or reception of a network packet. Thus, a unicast message costs *two* packet events—one at the sender and one at the recipient. A multicast with g recipients costs a total of $g + 1$ packet events—one at the sender, and one at each recipient. Packet events are a good metric here because the bulk of the processing overhead that naming operations impose is in the generation and reception of network packets. Our cost analysis assumes that no packets are dropped by the network and that responses are not delayed long enough to trigger retransmissions by the requestor. We evaluate the cost of name mapping only; name binding and the other naming operations are comparable [18].

Equation 1 is a conservative estimate for C_{map}, the average number of packet events required to map a name; its derivation is given below.

$$C_{\text{map}} = 4h + (r + m + 7)(1 - h) \tag{1}$$

In this equation, h is the cache hit ratio, r is the number of retransmissions required to determine a host is down, and m is the number of object managers in the system. Both client and server packet events are counted. The equation is valid for names that are covered by exactly one manager (the normal case).

The derivation of Equation 1 uses a simple "hit or miss" model of cache behavior, in which a cache lookup is considered to be a hit only if (1) the data it returns is still valid (not stale), and (2) the matched prefix refers to a managerial directory. All other outcomes are considered misses, and the worst-case miss cost is charged for each, yielding a simplified, conservative formula for C_{map}.[9]

When there is a cache hit, name mapping costs four packet events. The client unicasts its operation request message directly to the correct object manager, and the manager's unicasts the operation result in response. Thus the client sends one packet and receives one packet, and so does the manager, for a total of four packet events.

When there is a cache miss, as many as $r + m + 7$ packet events may be needed. This worst-case cost is incurred when the cache returns stale data referring to a host that is no longer up, and after the stale data is discarded, there is no information about the given name left in the cache. In this case, the client first sends off a request to the address given in the stale cache entry. The client detects that the addressed host is down by retransmitting its request r times and receiving no response (r packet events). At this point the client discards its stale cache data, and is left (we have assumed) with no cached information about the given name—not even a shorter prefix that narrows down the lookup to a administrational directory below the root. Thus, the client next multicasts a probe request to all m object managers participating in the root directory ($m + 1$ packet events), and receives a unicast response from the object's manager (2 packet events) containing a corrected cache entry. Finally, the client unicasts its request to the correct manager and receives a unicast response (4 packet events). Summing these values, the total cost for this case is $r + m + 7$.

Combining the two cases yields Equation 1 above.

It is clear from Equation 1 that C_{map} is close to the optimum value 4 if the miss ratio $1 - h$ is small compared to $1/(r + m + 7)$, as illustrated in Figure 3 below.[10] For example, C_{map} is about 4.17 in an installation with 50 object managers, $r = 4$, and $h = 99.7\%$.

Because it includes the cache hit ratio as a parameter, Equation 1 says nothing in itself about the practical usefulness of decentralized naming. Therefore we go on to consider what hit ratios can be expected in real systems, and what those hit ratios imply about the practicality and scalability of decentralized naming techniques.

[9] Such an estimate is quite accurate when misses are infrequent [18].

[10] The optimum is 4 because of the definition of name mapping: at least one message from client to manager is required to carry the operation request, and one return message is required to acknowledge the request and carry the results, for a total of four packet events.

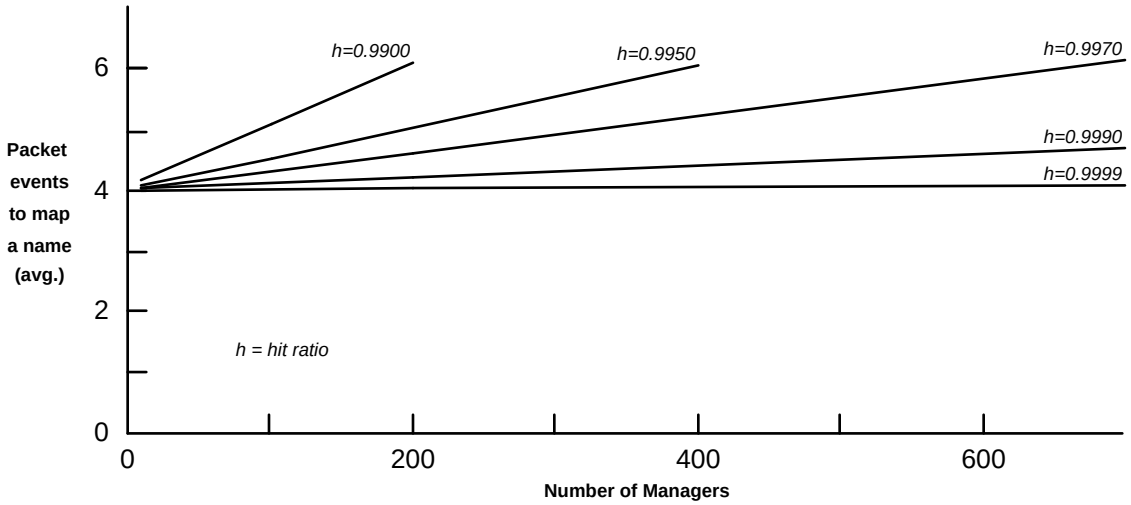

Figure 3: Average Cost of Mapping as a Function of Number of Managers.

3.2 Cache Performance Model

In this subsection, we develop a statistical model from which the expected cache hit ratio for a given decentralized naming installation can be computed in terms of other system parameters, and show that hit ratios of well over 99% can be expected under realistic assumptions about those parameters. The input parameters are (1) the number of name mapping requests issued per unit time, (2) the average length of time a name cache entry is valid, (3) the average length of time a client cache remains in use before it is discarded, and (4) the "locality of reference" observed in name usage. We begin by obtaining a formula for the steady-state hit ratio, then evaluate the ratio for some typical parameter values, and finally discuss startup misses, which can make the observed hit ratio less than the steady-state hit ratio.

3.2.1 Steady State Hit Ratio

The *steady-state* hit ratio is the hit ratio for client caches that have been in existence long enough to have gathered a (possibly stale) entry for every manager the client ever references. Section 3.2.3 below shows that the hit ratio for an initially empty cache rapidly approaches the steady-state ratio after a few startup misses.

We derive the following formula for $\overline{h}$, the steady-state cache hit ratio averaged across all clients:

$$\overline{h} = 1 - \sum_j \sum_k \frac{\beta}{\beta_{j,k} + v_k} \tag{2}$$

The generation of name mapping requests is assumed to be a Poisson process, and the average interarrival time for requests generated by client j that reference a name in managerial subtree k is denoted as $\beta_{j,k}$.[11] The symbol v_k represents the expected validity time for a cache entry that identifies which manager implements names in subtree k; that is, the average interval from the time such a cache entry is acquired to the time it becomes invalid. The summation is taken over all clients and all subtrees that exist at the moment for which the hit ratio is being evaluated. Finally, β represents the global average interarrival time for name mapping requests; it is equal to $\left(\sum_j \sum_k \beta_{j,k}^{-1}\right)^{-1}$. Equation 2 is derived as follows.

First, observe that the steady-state hit ratio for a single pair (j,k) is given by

$$\overline{h}_{j,k} = 1 - \frac{\beta_{j,k}}{\beta_{j,k} + v_k} \tag{3}$$

[11] A *managerial subtree* of the global naming hierarchy is a complete subtree whose root is a managerial directory, and whose root's parent is a administrational (or global) directory.

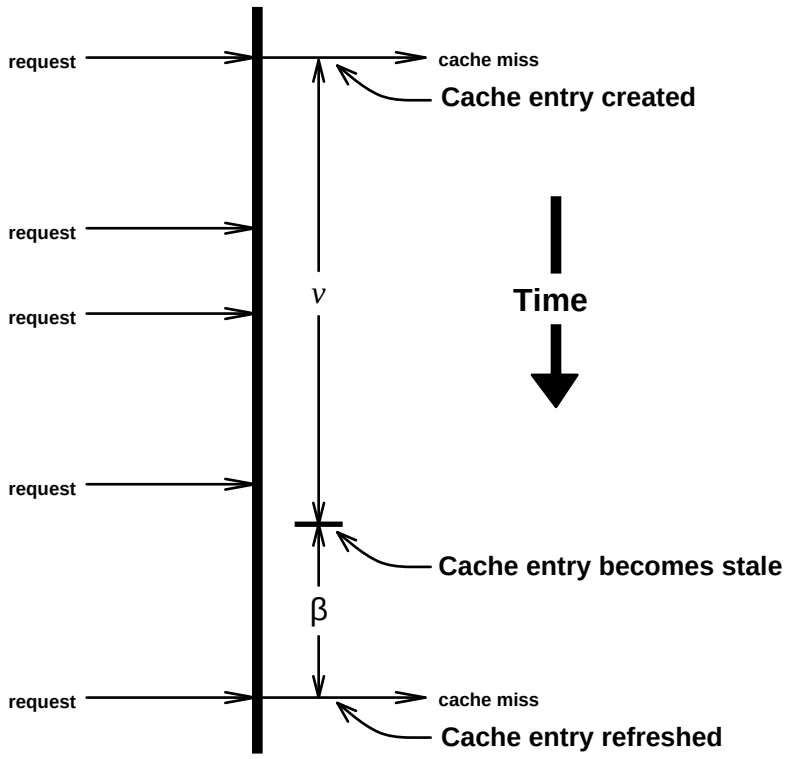

Figure 4: Average Intermiss Time Equals $v + \beta$.

because the average time between misses is $\beta_{j,k} + v_k$, as illustrated in Figure 4. Whenever a miss occurs, the client acquires a new cache entry that remains valid for a time v'. The next miss occurs on the first request that arrives after the entry becomes invalid—that is, at time $v' + \beta'$ for some $\beta' \geq 0$. Now, we know that the average value of v' is v_k, and because we have assumed that the generation of requests is a Poisson process, we also know that the average time from the end of v' to the next request (i.e., the expected value of β') is equal to the Poisson parameter $\beta_{j,k}$. Therefore, the average time between misses is $\beta_{j,k} + v_k$. The miss ratio can now be computed as the average number of misses per unit time divided by the average number of requests per unit time, and the hit ratio as 1 minus the miss ratio, yielding Equation 3 above.

Equation 2 is then obtained by taking the average steady-state hit ratio across all client/subtree pairs, weighted by the frequency with which requests are generated involving that pair. The average is formed by multiplying each pairwise miss ratio by the corresponding request rate $\beta_{j,k}^{-1}$, summing these terms, dividing the result by the global request rate β^{-1}, and simplifying.

3.2.2 Typical Values

We argue that it is reasonable to expect values of $\overline{h}$ in the range 99.00–99.98% for typical systems using decentralized naming. We first show that values in this range can be expected for individual client/subtree pairs with high traffic, and then contend that such pairs should dominate the global average due to locality of reference.

The graph in Figure 5 illustrates how the steady-state hit ratio for a given client/subtree pair varies with the average validity time of cache data. In the graph, the average time between requests $\beta_{j,k}$ is normalized to 1 unit, and the average validity time v_k (plotted on the x-axis) varies from 100 to 5000. The steady-state hit ratio $\overline{h}_{j,k}$ is plotted on the y-axis. At $v_k = 100$, $\overline{h}_{j,k} = 0.9901$, while at $v_k = 5000$, $\overline{h}_{j,k} = 0.9998$.

One expects a strong locality of reference property to hold in applications of naming to large distributed systems. For example, in a distributed system containing a mixture of personal workstations and shared file

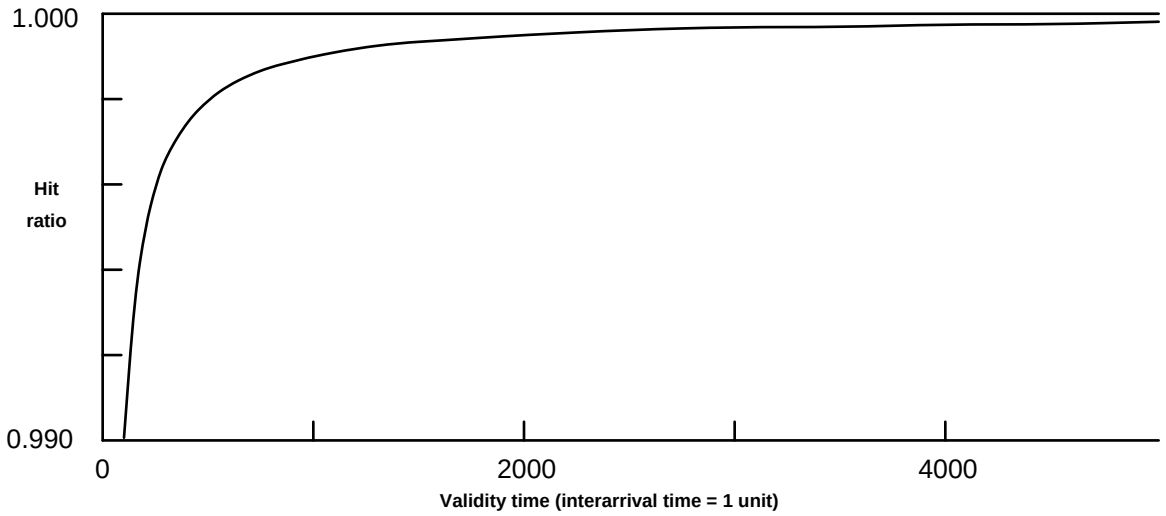

Figure 5: Hit Ratio as a Function of Validity Time.

servers, it is reasonable to expect a given user's workstation to use two or three file servers almost exclusively during the course of a day, even if hundreds of servers are available. The user probably keeps all his personal files on one file server, all in the same managerial subtree, perhaps loads standard system programs (text editor, compiler, etc.) from a subtree implemented by a second file server, and perhaps references a third server to access shared files belonging to his work group. There may be a few references to other servers, but most are to this small subset of the total available. Let us call (j, k) an *active* client/subtree pair if subtree k is a member of the subset that client j is using frequently.

When this locality property holds, the vast majority of all name references involve active client/subtree pairs, so their pairwise hit ratios $\overline{h}_{j,k}$ dominate the global average hit ratio $\overline{h}$. For example, suppose that a given client j accesses subtrees 1, 2, and 3 frequently (once per unit time); subtrees 4, 5, and 6 infrequently (once per 100 time units); and subtrees 7, 8, and 9 very rarely (once per 10000 time units). If $v_k = 1000$ for all nine subtrees, j's overall average hit ratio is 99.8%, quite close to its hit ratio with respect to 1, 2, or 3, which is 99.9%. The hit ratio with respect to 7, 8, or 9 is only 9.1%, but these misses have little effect on the overall average since the subtrees are accessed so infrequently.

Finally, it seems quite reasonable to expect the ratio of v_k to $\beta_{j,k}$ to be 1000 or more for active client/subtree pairs, putting the global average hit ratio into the desired range. Basically, only two types of event can cause a cache entry to become invalid: (1) a server may crash and be restarted with a new low-level identifier, or (2) the assignment of subtrees to servers may change. Both these events should be rare compared to name mapping requests. In a production system, crashes should be infrequent, so that it is quite reasonable to expect each of a server's regular clients to access it more than 1000 times between successive crashes. It is also reasonable to expect a subtree newly assigned to a particular server to (on average) be referenced more than 1000 times by each of its regular clients before it (or a part of it) is reassigned to a new server. For example, one does not frequently move trees of files from one server to another, because this typically involves copying a substantial amount of data from one disk to another or physically moving disk packs.

3.2.3 Startup Misses

The true hit ratio h for a decentralized naming installation is less than the steady-state hit ratio $\overline{h}$, because the latter does not count the initial misses that occur when a new, empty cache is created. Let us call such misses *startup misses.* Startup misses have little effect on h if client caches have long lifetimes compared to $\beta_{j,k}$, but can reduce h substantially if the caches have short lifetimes. This effect is quantified below.

Modifying Equation 2 to reflect the initial misses that occur after a client cache is created can be shown

to yield Equation 4:

$$h = 1 - \sum_j \sum_k \frac{\beta}{\beta_{j,k} + v_k} \cdot \max\left(0,\, 1 - \frac{\beta_{j,k}}{l_j}\right) \tag{4}$$

In this equation, the symbol l_j represents the *lifetime* of client cache j; that is, the number of time units between the time it is created as an empty cache and the time it is discarded. Each term of the original summation has been multiplied by $\max(0,\, 1 - \beta_{j,k}/l_j)$.

The basic insight leading to Equation 4 is that for each client/subtree pair (j, k), j's first name reference to k following the creation of its cache is always a miss, while the remainder are hits with probability $\overline{h}_{j,k}$. Thus the probability of a reference from (j, k) being a startup miss is $\min(1,\, \beta_{j,k}/l_j)$. Equation 4 is then obtained by writing an expression for the probability that a given reference is neither a startup miss nor a steady-state miss (i.e., that it is a hit), then computing the weighted average over all client/subtree pairs. Note that, as with $\overline{h}$, one can expect the global average h to be dominated by the pairwise hit ratios of active client/subtree pairs.

It is clear from Equation 4 that the observed hit ratio h depends strongly on the lifetimes of client caches. If a typical client cache lives long enough for the client to make 1000 name references to each of the subtrees it is actively using, $h_{j,k}$ equals $0.999 \cdot \overline{h}_{j,k}$—only a small reduction. On the other hand, if a typical client cache only lives long enough for the client to make one name reference to each subtree, $h_{j,k}$ is reduced to nearly zero. Thus, it is clearly important for an implementation of decentralized naming to preserve client cache information as long as possible.

The V implementation uses *cache inheritance* to give cached data a long lifetime. Although each client program has a separate cache in its own address space, it inherits its initial cache contents from its parent program (usually the V command interpreter or "shell"). Our measurements indicate that this technique gives nearly as high an overall hit ratio as a per-machine cache would.

3.3 Measurements

To validate our cache model and to give a concrete example of how decentralized naming performs, we now present some measurements taken on the V implementation.

At the time the measurements were taken, our installation at Stanford consisted of about 35 Sun and MicroVAX II workstations, three file servers running the V kernel, and five VAX/UNIX systems providing additional file service, all interconnected by Ethernet. During the measurement period, the workstations were being used in their normal fashion to support day-to-day tasks including software development, word processing, and remote access to other hosts on the DARPA Internet.

3.3.1 Hit Ratio

The measured hit ratios were excellent, and in good agreement with the analytical model of Section 3.2. Over about 24 days of 24-hour operation, our V installation showed an average cache hit ratio of 99.70%. During the half hour for which the arrival rate of name requests was highest, the average hit ratio was 99.97%. Based on measurements of the request arrival rate, and estimates of the rate of client and server reboots, the model predicts hit ratios of approximately 99.71% and 99.997% for these two periods.

Table 1 summarizes the statistics from which the 24-day average hit ratio was computed. Statistics were reported for a total of $6.033 \cdot 10^7$ seconds of workstation running time, with an average of 25.15 workstations reporting each half hour. During this time, 386626 name mapping requests were issued, of which 385466 were cache hits (i.e., they were carried out with no need for a multicast probe), for a hit ratio of 99.7%. Note that this measurement counts references to uncovered names (resulting in a failing multicast probe) as cache misses, resulting in a conservative estimate of hit ratio.[12]

Table 2 summarizes the statistics for the peak half hour of the measurement period. During this period, 30300 names were mapped—fully 7.8% of the 24-day total, and more than in any other half hour slice of the measurement period. There were only 9 cache misses, for a hit ratio of 99.97%.

[12] The reason so many uncovered names were mapped is that, at the time these measurements were taken, the V implementation did not include on-line administrational directory managers, so every undefined name in an administrational directory appeared uncovered.

Experimental period:	Oct 17–Nov 9, 1985
Workstation-seconds:	$6.033 \cdot 10^7$
Average workstations reporting:	25.15
Total names mapped:	386626
Successful multicast probes:	780 (0.20%)
Failing multicast probes:	380 (0.10%)
No probe required:	385466 (99.70%)

Table 1: Overall Statistics.

Experimental period:	11:41–12:11, Nov 4, 1985
Workstation-seconds:	52383
Workstations reporting:	27
Total names mapped:	30300
Successful multicast probes:	8 (0.026%)
Failing multicast probes:	1 (0.0033%)
No probe required:	30291 (99.97%)

Table 2: Statistics for Peak Half Hour.

A rough computation based on the model of Section 3.2 shows reasonable agreement with these measurements. Let us assume that each client made about the same number of name mapping requests during the experiment, and that the global hit ratio was dominated by their interaction with our most frequently used file servers. The computation also assumes that name caches are per-workstation to avoid the complication of modeling V's per-program caches with inheritance. Currently, two servers provide the bulk of all file service to our V installation, and they are each rebooted twice a week after dumps are taken, so let us assume that v_k is equal to 3.5 days for each. Workstations are rebooted more frequently, often more than once a day, so let us take l_j to be 18 hours for each workstation. From the data in Tables 1 and 2 we can compute $\beta_{j,k}$ to be 156.04 for the 24-day experiment, and 1.7288 for the peak half hour. Plugging these figures into Equation 4 yields hit ratio estimates of 99.708% and 99.9968% respectively.

Several factors could account for the difference between the measured and predicted hit ratios. The discrepancy in the 24-day value is small, and could easily be accounted for by slightly inaccurate estimates of v_k and l_j, by the fact that V uses per-program caches with inheritance rather than per-machine caches, or the other shortcuts taken in computing the prediction. The predicted hit ratio for the peak half hour is, however, quite a bit higher than the observed value. This difference could be due to unusual behavior during that particular half hour; for example, several references to little-used servers, or several workstation reboots.

These figures also indicate that name mapping is a common enough operation that it is important to optimize its performance. During the peak half hour, for example, there were 0.578 name mapping operations performed per workstation per second, for a total of 15.6 operations per second over all 27 workstations. In a larger installation, of course, the overall total would be proportionately higher.

3.3.2 CPU Cost

Table 3 reports the results of an experiment performed to measure the CPU cost of decentralized name mapping. The experiment measured the time required to perform a trivial operation (`GetContextId`) on an object referenced by name, for each of three cases of interest. In the *hit* case, a cache hit allowed the operation to be completed in a single unicast message transaction. In the *miss/covered* case, the given name missed in the cache but was covered by some object manager, which responded to a multicast probe. In the *miss/uncovered* case, the given name name was not covered by any object manager—the client multicasted

and received no response.[13] CPU time measurements were taken on the client workstation, on the server covering the specified name, and on another server participating in the naming system but not covering the specified name (a "bystander").

Case	Client (ms)	Server (ms)	Bystander (ms)
Hit	3.38 ± 0.13	3.89 ± 0.082	0
Miss/covered	26.7 ± 5.5	11.6 ± 0.30	6.42 ± 0.21
Miss/uncovered	16.0 ± 1.1	—	9.29 ± 0.75

Table 3: CPU Cost Measurements.

The experiment was structured as follows. A test program, linked with the standard client naming library, ran in a loop, repeatedly trying to map the same name. (For the *miss/covered* case, the program cleared the name cache before each trial.) CPU usage measurements were taken on the test program, running on one workstation, and on instances of a server program running on two other workstations. The server was the V in-memory file server ("RAM disk"). The tests were run on Sun-2/50 workstations with 10 MHz MC68010 processors and Ethernet interfaces based on the Intel 82586 chip. A *test run* measured the total time for 100 to 10000 trials; the average time per trial was obtained by dividing this total by the number of trials. The table gives the means and sample standard deviations of the times obtained on four test runs.

These figures, together with the statistics of Section 3.3.1, show that servers in our V installation spend only a small fraction of their available CPU time in bystander processing. Assuming there are enough servers that most servers are bystanders even on successful probes, we can compute an average of 0.0221 ms per naming operation consumed on each server in processing operations in which it is a bystander. During the experimental period, there were 386626 name mapping operations observed in $6.033 \cdot 10^7$ workstation-seconds, for an average rate of $6.4 \cdot 10^{-3}$ operations per workstation per second—or taking the average number of workstations to be 25, 0.16 operations per second. Thus on the average 0.000355% of each server's time was consumed in bystander processing over a 24-hour period—a negligible amount. The peak load observed over any half hour of the experimental period was 16.5 operations per second (with 27 workstations reporting). During this period the cache miss ratio was only 0.025% and the uncovered ratio only 0.00625%, both much lower than the daily average. Repeating the above computation with these peak load figures, it appears that 0.00361% of each server's time was consumed in bystander processing during the peak period—still negligible.

These measurements provide support for the practicality of decentralized naming by showing that, in our installation, only a small fraction of the available client and server CPU time is consumed in processing name mapping requests. The small amount of time spent in bystander processing is of particular interest, because (as discussed in Section 3.4 below) the cost of such processing is the major obstacle limiting the size of administrational directories in large systems.

3.3.3 Elapsed Time

Table 4 lists the *elapsed* times required for name mapping in the same three cases measured in Section 3.3.2. The experiment was performed using the same test program and the same hardware described in that section.

Although the elapsed times for the *hit* and *miss/covered* cases are comparable to the sums of the client and server CPU times, the time for the *miss/uncovered* case is quite long (over 5 seconds), because it includes a timeout by the client. In general, such a timeout requires $r \cdot t_r$ seconds, with r (the number of retransmissions, counting the initial transmission) determined by the required resiliency of name mapping as compared with the frequency of omission faults in the communication medium, and t_r (the time interval between retransmissions) determined by the expected time to receive a response. In our Ethernet-based V installation, both the retransmission interval and the number of retransmissions could be reduced significantly were it

[13] The cost of detecting stale cache data was not measured. Detecting and replacing a stale cache entry that maps to an existing server adds to the miss case approximately the time for mapping a name in the *hit* case; an entry that maps to a nonexistent server adds approximately the time for the *miss/uncovered* case.

Case	Elapsed Time (ms)
Hit	9.23 ± 0.24
Miss/covered	47.7 ± 9.2
Miss/uncovered	5379 ± 92

Table 4: Elapsed Time For Name Mapping.

not for the need to communicate with a guest-level implementation of the V interkernel protocol running on our UNIX systems (outside the UNIX kernel). Fortunately, uncovered names are rarely encountered—even in the preliminary implementation we measured, with no directory servers to cover the unbound names in administrational directories, only 0.10% of all names mapped were uncovered. In a full implementation, a name appears uncovered only when the server that covers it is down or inaccessible over the network.

3.3.4 Space Cost

One might expect decentralized naming to have a substantial space cost, because it places some global naming information in every server, a name cache in every client, and some naming code in both clients and servers. Experience with the V implementation, however, has shown that the cost is low—low enough that there has been no need to put a size limit on the cache, and there apparently will be no need to do so even in much larger installations.

In servers, the space cost for naming support is not large relative to the overall size of the servers. For example, in the case of the V disk file server, the server naming library (which compiles to 12408 bytes of code and static data on the MC68000) represents only 14% of the total static size of the server, and is an insignificant fraction of its run-time size, which consists mostly of disk buffers.

The static space cost in client programs is also small in comparison with their total size. The client naming library for V occupies 4936 bytes on the MC68000 if all of its routines are used (not normally the case). This space cost is comparable to that imposed by other standard library routines—for comparison, `doprnt` (the main module that implements the C formatted printing routine `printf`) alone compiles to 1276 bytes on the MC68000.

The run-time space cost in client programs is due mostly to the name cache, which never grows very large. Recall that a client's cache contains at most one entry for each managerial subtree that the client has referenced. Because of locality, a given client is quite likely to reference only a small fraction of the available subtrees during its lifetime, and to be actively using less than 5–10 at any given moment. In the V implementation, each name cache entry occupies 22 bytes of memory plus the length of the name prefix it refers to, which is typically less than 32 bytes. Thus a name cache of 10 entries occupies less than 540 bytes of memory.

3.4 Limits to Growth

There are some practical limits to how large a system can be built with an administrational directory at its root. For example, although the V implementation works well on our network at Stanford, it would be quite impractical to extend it to a nationwide or worldwide internetwork without adding a global directory level. This subsection takes a detailed look at the limits to growth in decentralized naming systems without global directories (*administrational* systems). The following subsection applies these observations to systems that include global directories (*global* systems), where they set a practical limit on how large a directory can grow before it must be implemented using global rather than administrational techniques.

3.4.1 Availability of Multicast

The availability of multicast is currently a technological limit on the size of network that an administrational directory can span, but this limit may not exist for long. Today's network technology provides multicast only within a local-area network, such as a single Ethernet cable, not across long-haul networks or even across

multiple Ethernets connected by gateways. This problem would seem to set a practical limit of around 1000 hosts on the maximum size of an administrational decentralized naming system. However, techniques for internetwork multicast are currently under investigation [8], and of course techniques for internetwork broadcast have long been known [3, 33]. Thus, it makes sense to assume the technological limits will be overcome, and to ask what other limits are encountered as systems are expanded well beyond 1000 hosts.

3.4.2 Load Per Operation

Another limit to the growth of a administrational system arises from the linear increase of name mapping cost with system size. The graph in Figure 3 (page 11) illustrates the problem: if the number of managers in the system is increased while the hit ratio remains constant, the average load imposed by mapping a name increases linearly, with the slope of the cost function equal to the miss ratio $1 - h$. At some size, C_{map} becomes unacceptably large. Increasing the hit ratio raises this limit but does not eliminate it.

In a system using global directory managers, on the other hand, the number of packet events required to map a name in the cache miss case is proportional to the number of directory managers in the path from the global root name server to an object, not to the total number of object managers. It therefore increases only as the log of the system size, assuming directory managers at each level have about the same fanout (number of links to managers at the next level). This growth property suggests that global directory managers should be used for the uppermost levels of large hierarchical naming systems.

3.4.3 Load Per Manager

A further difficulty in scaling up a administrational decentralized naming system arises because the average naming load *per object manager* contains a term that is proportional to the number of clients, but not inversely proportional to the number of managers. That is, as the number of clients increases, there is a component of the load on each server that increases proportionately and *cannot* be reduced by increasing the number of object managers. ("Load" here is measured in packet events per unit time.) This load component arises directly from the use of multicast to handle cache misses.

A computation similar to those of Section 3.1 yields the following expression for L, the average naming load per manager, in a system with c clients and m object managers.

$$L = c \cdot \alpha \cdot \left(1 - h + \frac{5 - 3h}{m}\right) \tag{5}$$

Here α is the average *activity* level of each client; that is, each client, on the average, generates α name mapping requests per unit time. In the notation we have been using, $\alpha = c^{-1} \sum_j \sum_k \beta_{j,k}^{-1}$. As before, h is the cache hit ratio.

One way of interpreting Equation 5, illustrated in Figure 6, is that it implies a linear increase in the naming load on each server as a system increases in size, with the slope of the increase depending on the cache hit ratio. The graph plots the number of clients on the x-axis and the number of name mapping packet events per server per unit time on the y-axis. It assumes that the ratio of client hosts to server hosts remains constant as the system grows (that is, $c = \kappa m$ for some constant κ), and that α also remains constant; in this figure, $\kappa = 10$ clients per server and $\alpha = 1$ request per time unit.

As the system continues to grow, the servers eventually become saturated by the increased naming load, and it becomes necessary to reduce the number of clients per server to compensate. This observation leads to another way of looking at the growth problem, illustrated in Figure 7. The graph assumes that (1) each server has a fixed load-handling capacity L of 150 packet events per unit time, (2) there are an average of 8 non-naming packet events generated for every client name mapping request (so that naming represents 20% of the packet events when there are no cache misses), and (3) the number of clients per server κ is set just low enough to keep the servers within that capacity. It plots κ on the y-axis against c on the x-axis. Under these assumptions, the number of clients that can be handled per server decreases linearly, but slowly, as the total number of clients grows.

In light of the results of this and the previous section, it is clear that administrational decentralized naming systems cannot be scaled up indefinitely; however, it appears that systems including thousands of hosts can be quite practical, at least from a performance standpoint.

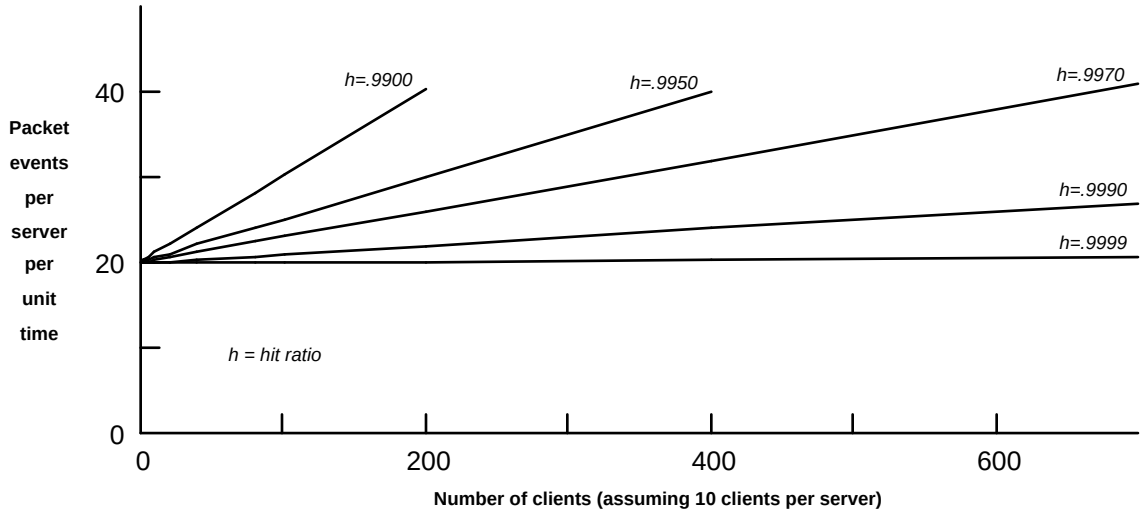

Figure 6: Load Per Server as a Function of System Size (With Constant κ)

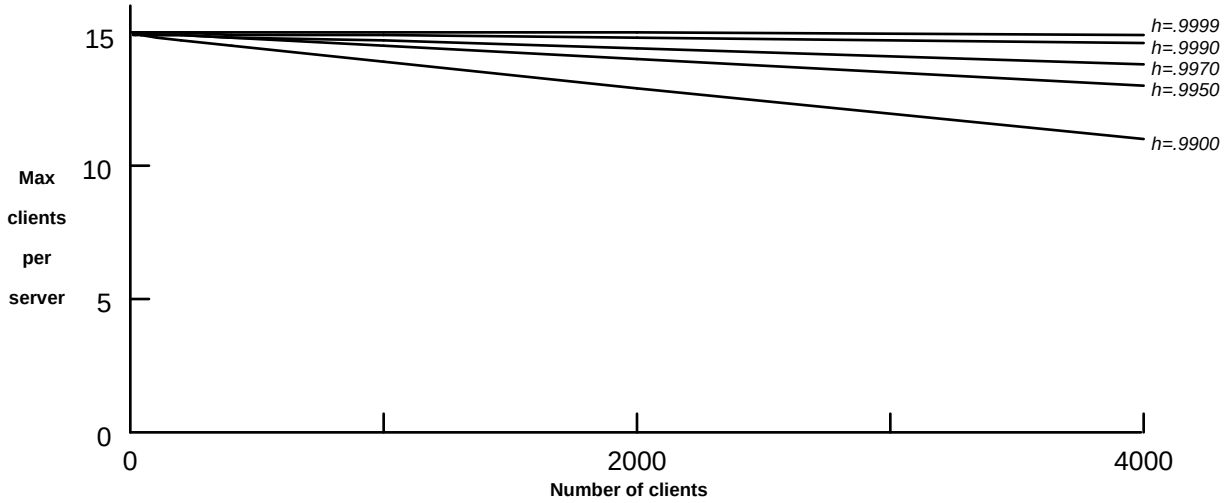

Figure 7: κ as a Function of System Size (With Constant Total Load Per Server).

3.5 Extension to Global Systems

We argue that the above results for administrational systems can be used to establish a limit on how high in a global naming hierarchy the boundary between administrational and global directories can be drawn. That is, they determine which directories *must* be made global.

A global decentralized naming system can be viewed as a set of administrational subtrees hanging from the common global directory mechanism.[14] Each subtree can then be analyzed as an independent system—the global directory managers direct each client name request to exactly one subtree, so each one receives some fraction of the total mass of requests.

The above analysis of name mapping in an administrational system applies almost without change to

[14] An *administrational subtree* is a complete subtree of the global naming hierarchy, whose root is an administrational directory that has a global directory as its parent.

an administrational subtree S in a global system, with the total number of managers (m) replaced by the total number of participants in the root of the subtree (m_S).[15] The only difference is that a worst-case miss costs $r + d + m + 7$ instead of $r + m + 7$, where d is the number of packet events incurred in going through the global directory managers to find the participant group for S. The term d is at most equal to twice the path length from the global root to the root of S (because each global directory could be kept at a different directory manager, requiring one unicast packet from each directory's manager to the next). The path length is roughly proportional to the log of the total number of global directories in the system; thus it is small enough compared to m that it can be treated as a constant. It therefore has no more effect on the analysis or results than would a change in the value of r.

Therefore, in a global system with similar parameters to the administrational systems discussed earlier, any directory with more than a few thousand participants should be made global rather than administrational. The exact cutover point depends on the relative values that are placed on performance and resiliency. Performance is improved by switching to global techniques in directories with fewer participants, but as shown in the next section, these techniques give poorer resiliency. On the other hand, resiliency is improved by using administrational techniques, but as was shown above, these techniques give poorer performance.

4 Fault Tolerance

A distributed naming system of the size we are interested in must be prepared to deal with faults in the object manager hosts, client hosts, and communication network. In fact, some fraction of the total system resources can be expected to be faulty at all times. We show that our decentralized naming design is both highly *reliable* and highly *resilient* in the presence of faults. We say an operation is *reliable* with respect to some class of faults F if, in spite of the occurrence of faults in F, it either *succeeds,* performing the requested action and returning correct results to the invoker, or *fails,* returning an error message, as laid out in its specification.[16] An operation is *resilient* with respect to some class of faults F if a fault in this class cannot cause the operation to fail. (A fault is said to *cause* an operation to fail if there is some set of arguments and initial conditions under which the operation's specification permits it to succeed, but in the presence of the fault, the operation sometimes fails.)

We focus primarily on network omission (packet loss) and server crash faults. Client crash faults are not considered, but present no additional problems for our algorithms. We concentrate on the problem of achieving resiliency, given that reliability is relatively straightforward to achieve under this fault model. Some remarks on tolerating Byzantine faults are included at the end of this section. We argue informally throughout; for a more formal and complete treatment of this material, the reader is referred to Mann's thesis [18].

The next three subsections discuss the fault tolerance of the three classes of decentralized naming operations—mapping, query, and binding. In the first subsection, we show that decentralized name mapping achieves optimum resiliency for names bound to objects with *nearby* managers—managers that are within range of the multicast sent out when a client's cache misses. For other names, the resiliency of name mapping is dependent on the resiliency of the global directory system. The next subsection shows that query operations would require full replication to achieve optimum resiliency, yet in practice provide acceptable resiliency if the global directory system is acceptably resilient. The third subsection discusses the binding and unbinding operations, focusing on the problem that increased replication can make these operations less resilient and more costly. A final subsection discusses Byzantine faults.

4.1 Name Mapping

The *name mapping* operation accepts a name n and a message m as its arguments. If n is bound to an object O_n, the operation sends the name and message to the object's manager $M(O_n)$ and returns a reply,

[15] Recall that a directory's participant set includes the union of the participant sets of all its descendants, so every manager that names anything in a subtree participates in the subtree's root.

[16] This concept of failure is similar to the notion of *exception* in programming languages or *abort* in transaction systems. A failure is undesirable, but not catastrophic, because the system reports it and is prepared to deal with it.

or else fails, returning an error indication. If n is unbound, the operation always fails.[17]

The optimum achievable resiliency for any implementation of name mapping is *ABMA-resiliency. ABMA* stands for "all but manager access": an operation's implementation is said to be ABMA-resilient if the only fault combinations that can cause it to fail are *manager access faults*—faults that prevent communication with the specified object's manager, such as a crash fault on the object manager or a network omission fault that prevents communication with it. No implementation of name mapping can be more than ABMA-resilient, because the definition of name mapping requires round-trip communication with the named object's manager. But ABMA-resiliency is achievable (at least in theory); for example, one could achieve it by multicasting all name mapping requests to a group including every object manager.

Decentralized name mapping does not achieve ABMA-resiliency for all objects, but it does achieve such resiliency for a limited class of objects—those with nearby managers. As described in Section 2, a client attempting to map a name n repeatedly examines its cache, discards (apparently) stale data, and issues probes for more cache data. It does not stop until it has either (1) received a reply to its name mapping request, (2) been informed that n is unbound, or (3) timed out on a multicast probe to the group of nearby object managers. If n is bound to a nearby object, case (1) means success, case (2) cannot occur, and case (3) can occur only if an access fault prevents the multicast from reaching the object's manager or prevents the manager's reply from reaching the client. Thus, name mapping for nearby objects is resilient against all but manager access faults.

For objects that are not nearby, a larger class of faults can cause name mapping to fail. First, of course, manager access faults can cause failure. In addition, faults that affect the global directory servers can cause failure by making necessary information inaccessible. For example, if a client at Berkeley attempts to map the name `%edu/stanford/dsg/smith`, but all servers holding copies of the `%edu` directory are inaccessible and the client does not have any information about `%edu/stanford` in its cache, the name mapping fails. Finally, if all copies of an administrational directory's manager fail, clients in remote administrations can no longer map names in that directory, because (as mentioned in Section 2.4), such clients are not able to multicast to the directory's participants—they send their requests to the directory manager instead. For example, if the administrational directory manager for `%edu/stanford` fails, clients at Stanford can continue to map names in this directory using multicast, but clients at Berkeley cannot. In practice, we expect a reasonable level of replication to make these additional failure modes rare.

No other classes of faults can cause name mapping to fail. For example, failure of a liason server cannot cause name mapping to fail, because liason servers are stateless—a new liason server can be started up any time an old one fails, on any host (assuming a copy of the server program is available). The loss of cached data in the liason server can only cause performance degradation, not name mapping failure.

We believe this level of resiliency is reasonable, even though it is not optimum. Optimum resiliency requires that each client be able to map any object's name, even if that client and object manager are the only hosts in the system that are up and communicating. This in turn is possible only if each client is able to multicast to every object manager (or at least, to every object manager for which the client does not have complete, correct name coverage information). As was shown in Section 3, multicast to all managers is too expensive to be practical in systems containing more than a few thousand manager hosts.

A name mapping request can fail with the return value "failure: no response," indicating that the client runtime system was unable to get a response at some step of the name mapping protocol. For example, the manager of the named object may have crashed. As another example, the name may be invalid at the administrational level, but the administrational directory manager is unavailable and thus unable to indicate the invalidity. In such cases, the name mapping fails without determining whether the name is bound or not, information the client might require. We refer to this as the *binding check* problem, discussed in the next subsection.

[17] Name mapping on a replicated or distributed object—one that consists of multiple subobjects with distinct managers—is considered to succeed if it communicates with at least one subobject manager. Any additional protocol required to communicate with other managers for consistent access or update is object-type specific; it is implemented by the subobject managers, not by the naming system. For example, in the UIO interface [6], each manager of a replicated object maintains a list of the other managers storing replicas.

4.2 Query

We discuss two query operations in this section: binding check and directory listing. The binding check operation takes a name n and returns the name's binding status (*bound* or *unbound*), or else fails, returning an error message. Logically, whenever a server is binding a new name, a binding check is required to determine whether the name is already defined, to prevent ambiguous names. Also, binding check can provide additional information when a client's name mapping request times out—the client may be able to determine that the name is bound even though the object manager that binds the name is down. The directory listing operation takes a name n and lists the names that are bound in the directory specified by n, or fails, returning an error message. We show that it is impractical to implement these operations with optimum resiliency, discuss the actual resiliency provided by a decentralized implementation, and argue that the latter resiliency is a reasonable compromise for practical systems.

For a system in which every name is covered, optimum resiliency for binding check and directory listing is achieved if (and only if) every host has complete, correct knowledge of which names are bound and which are unbound. In this case, binding check and directory listing are purely local operations at each client—no external servers or network communication are required—so they are resilient against all faults.[18] On the other hand, if some client host C does not have information about some name n, C's implementation of binding check on n (or directory listing on the directory containing n) cannot be resilient against all faults—in particular, it is not resilient against a set of faults that completely cuts C off from the network. Clearly, however, it is not practical to replicate knowledge of every name's binding status at every node in a large system where names are bound and unbound frequently.

As a practical compromise, decentralized naming provides a resiliency for binding check that is just slightly better than the resiliency of name mapping, using a protocol that is a slight variant of the name mapping protocol. When the name mapping protocol would call for a multicast probe, which can be answered only by a manager that covers the name in question, the binding check protocol multicasts a query that can be answered by any manager knowing the name's binding status. For example, if `%x/y` is an administrational directory and `%x/y/z` is a managerial directory on manager M, the administrational directory manager responds *bound* to a binding check query on `%x/y/z` even if its manager M is down. The protocols are otherwise identical. Thus for bound names, binding check succeeds whenever name mapping would succeed, plus the additional case just mentioned. For unbound names, binding check succeeds whenever a manager that covers the name is accessible—generally, the manager of directory corresponding to the name's longest bound prefix. For example, if `%x/y` is bound but `%x/y/w` is unbound, binding check on `%x/y/w` succeeds (returning *unbound*) if the manager of `%x/y` is accessible.

This level of resiliency is arguably a reasonable choice for practical implementations of naming. For file names, it is similar to that provided by other naming services. For example, in Lampson's design [16], the global name service records the binding of each file server's name, but not the names of individual files on the servers. So when an (unreplicated) file server is down, binding check on its own name—that is, on the name of its root directory—can still succeed, but on any file below its root, the operation fails. The same is true of decentralized naming.

In our design, each directory is assigned a manager (or replicated across several managers), so directory listing is as resilient as name mapping—a directory can be listed whenever its manager can be contacted, and a directory's manager is contacted by applying the name mapping protocol to the directory name. In addition, administrational directory listing can be made more resilient at some cost in reliability. That is, a client can multicast to an administrational directory's participants when the directory's manager is unavailable, obtaining a list of the names each participant binds. Collating these lists produces a directory listing. This protocol is unreliable, in the sense that it can produce an incomplete listing with no warning to the client: the client has no way of knowing whether all participating managers received and responded to its request. Nonetheless, information obtained in this way can be useful.

Although these query operations can be made more resilient, in general, increasing their resiliency requires increasing the replication of directory information; this in turn increases the overhead for binding operations, as described in the next subsection.

[18] Resiliency against all faults is possible for binding check and directory listing, though it is impossible for name mapping, because the specification of name mapping requires communication with the object manager that binds the name, while the specifications of binding check and directory listing do not.

4.3 Binding

Viewed abstractly, binding operations—operations that add, remove, or modify name bindings—are update operations on (possibly) replicated data. The degree of replication varies across the managerial, administrational, and global directory levels. Managerial directories are often not replicated; where they are, the choice of replication and crash-recovery mechanisms is manager-specific. An administrational directory's entries are partitioned across its participating managers; the list of bound names held by its directory manager may or may not be replicated, depending on the manager implementation. Similarly, the degree to which global directories are replicated is a design and configuration choice for that part of the system.

With conventional techniques for managing replicated data, such as weighted voting [15], a high degree of replication results in either a large write quorum, making updates expensive, or a large read quorum, making queries expensive, or both. The techniques of Lampson [16] and Demers et al. [13] appear to be effective ways to trade off strict correctness (allowing temporary inconsistency) to obtain better resiliency and performance for updates to global directories. It is less clear whether these techniques are necessary or acceptable at the administrational or managerial level.

4.4 Other Fault Classes

Our naming design makes no attempt to handle Byzantine faults in their full generality. Although Byzantine faults in distributed systems can in general be tolerated using replication coupled with Byzantine agreement protocols, we expect many simple, unreplicated servers to use our naming system, and we expect that nearly all clients will be unreplicated. Moreover, in many applications, the cost of running a multi-round Byzantine agreement protocol outweighs the benefits of tolerating Byzantine faults, particularly given the rarity of such faults.[19] Implementations of our design can, however, tolerate some fault classes beyond simple omission and crash faults.

First, our V implementation handles timing faults by using timeouts and sequence numbers to convert them to omission faults, a well-known technique. That is, a packet that is delivered too late is recognized as such and discarded by the recipient. Such packets appear to have been dropped by the network.

Further, a large class of "malicious" faults is handled by the security mechanism we discuss in the next section. Using this mechanism, authorization to cover any given name can be granted to some object managers and denied to others, preventing the unauthorized managers from compromising the reliability of operations on that name. For example, when a client multicasts a probe request, faulty or malicious object managers that receive the request could issue incorrect responses. The security mechanism provides a reliable way for the client to filter out incorrect responses from unauthorized managers.

5 Security

A naming system is secure if it ensures that (1) servers do not provide information to clients that are not authorized to have it, (2) servers do not accept unauthorized updates, and (3) clients do not accept information from servers that are not authorized to provide it. The latter requirement is of particular concern in a decentralized naming system, where it implies that clients must recognize and discard counterfeit responses to their multicast naming requests. (We define a *counterfeit* response to a naming request to be a response sent by a manager that is not authorized to cover the portion of the name space in question.) We assume that the system has some well-defined security policy that specifies (1) which clients are authorized to access each directory, and (2) which servers are authorized to cover each portion of the name space.

Providing secure directory access is straightforward; the access control mechanism for directories can be modeled after the mechanism used for file access control. That is, a manager checks each incoming naming request and rejects those for which the requesting client does not appear on the access lists for the directories involved.[20]

[19] Up to $t + 1$ rounds of messages among all replicas are required to tolerate t Byzantine faults [29].

[20] We assume an authentication mechanism (cryptographic signatures, for example) that allows the requesting client to prove its identity.

Counterfeit rejection is the reverse problem—it requires clients to check the authorization on information coming from managers. The counterfeit problem is similar to the classic problem of "authenticating the system to the user" when logging onto a centralized computer system, but is more complex—there is not just one system to authenticate, but many different object managers, owned by different administrations and authorized to cover different parts of the name space. Moreover, the client does not know in advance which manager it wants to hear from, only that it wants the authorized manager for the name it is presenting.

We favor an approach in which each manager caches an unforgeable *capability* describing the portion of the name space it is authorized to cover, which it returns in its response to each naming request. Conceptually, a capability K is a document stating that "principal $p(K)$ is authorized to perform action $a(K)$ until time $t(K)$," signed by some principal $s(K)$, where $s(K)$ is authorized to issue capabilities for $a(K)$. Whenever a manager $p(K)$ responds to a client C's naming request, it includes an appropriate capability K with its response, and applies its own signature to the whole package to certify that it is in fact coming from $p(K)$. These capabilities cannot be revoked, but they can be made to expire, allowing coverage authorization to be changed from time to time. Every client is initialized with enough information to be able to check incoming capabilities for validity. Thus, clients do not incur extra network traffic. A manager only incurs additional network traffic when it receives a client request for which all its capabilities have expired, forcing it to request a new capability before it can answer the request. Capabilities can be implemented using RSA [24] or any other cryptosystem that provides digital signatures. Further details are given in Mann's thesis [18].

Capability-based security does have some cost, both in performance and in resiliency. The security mechanism reduces resiliency because, if the authorization service fails or becomes inaccessible, servers can no longer operate once their capabilities expire. It also introduces some performance overhead because of the need for managers to include capabilities in their naming responses and for clients to verify them, and the need for managers to acquire new capabilities after old ones expire. A trade-off arises—between, on the one hand, performance and resiliency, and on the other hand, flexibility and strength in the security mechanism—for two reasons. First, capabilities can be made short-lived to make it possible to cancel authorization on short notice and reduce the danger of compromise—but doing so imposes a higher performance and resiliency cost on the managers. Similarly, one can improve security by strengthening the cryptosystem used to implement capabilities—but doing so will typically also increase their size and increase the processing time needed to generate and check them.

In general, reduced performance and resiliency are an unavoidable cost in any counterfeit-secure implementation of decentralized naming that allows coverage authorization to be granted and revoked dynamically. Whenever a client and manager communicate, one or the other must contact a security authority occasionally to verify that the server is (still) authorized to respond to the client's requests.

The above security mechanism has not been implemented to date; however, we do not see any significant difficulty in doing so. We have described it here to demonstrate the feasibility of making a decentralized naming implementation secure.

6 Related Work

We have concentrated in this paper on the administrational and managerial levels of our naming design, under the assumption that directories at the global level can be implemented using known name server technology. Perhaps the most advanced work of that kind to date has been Lampson's design [1, 16], an outgrowth of the work on Grapevine [2, 27] and the Clearinghouse [22]. The Domain Name service [19, 20] of the DARPA Internet is a more limited system in the same class.[21] Additional work in this area is surveyed in Terry's thesis [30]. These global name services are typically used to map from names to unique identifiers for hosts, mailboxes, or services. As mentioned previously, the reliability, security, and scalability requirements on these systems impose a significant performance cost on name lookup operations, so that the systems are only practically applicable to names that do not need to be looked up frequently—it is too expensive to invoke the global name service every time a file is opened.

Our work can be viewed as extending these global directory system designs in three important ways. The first, basic extension is that each object manager in our system knows the full global names of the objects

[21] The Domain Name service design assumes manual addition or deletion of name bindings as well as manual placement and update of directory replicas, simplifying the design over the other examples.

it maintains, making the other two extensions possible. Second, for improved performance, clients use a name prefix cache, with consistency maintained by on-use detection of stale entries. On-use detection is made possible by the first extension—the manager receiving a request detects the client's use of a stale entry by checking the client-supplied global name against the global names of the objects it implements. Third, for improved resiliency, clients use multicast to locate objects with nearby managers, thereby functioning independently of the global directory system for most cache misses. This technique is also made possible by the first extension, which enables an object's manager to recognize the object's global name without help from the remainder of the directory system. In addition to developing these extensions, this paper contributes a careful examination of the performance, reliability, and security properties of our design by analysis and measurement.

Welch and Ousterhout [34] describe an extension of the UNIX file system using *prefix tables.* Their prefix tables are similar to our name prefix caches, but are less flexible. In their implementation, each prefix table is statically loaded with a set of prefixes at boot time—although the referent for a prefix can change during operation, new prefixes cannot be added to the table, nor can old ones be deleted.[22] Apart from the prefix tables, their system is quite different from ours. In particular, their system includes no administrational or global directories; instead, file servers near the root of the directory tree use *remote links,* to mount file systems on other servers as subtrees. In general, the Welch-Ousterhout design seems to be targeted for a campus-sized environment; it does not address the global issues we have considered paramount in our design.

The Locus [32] naming facility provides a similar network transparent name space for objects stored by multiple servers, but its implementation differs markedly from ours. The Locus directory hierarchy is built up of disjoint subtrees called *file groups,* each stored by one or more server hosts. The file groups correspond to UNIX *file systems,* and as in UNIX, they are assembled into a single tree by designating one group as the root and mounting others below it. The root file group is replicated at every node; it thus serves as a sort of prefix cache for the file groups mounted below it—that is, when mapping a name, a client looks up the first few components in its local copy of the root file group, then references (possibly remote) directories in other file groups for the rest. The record of where each file group is mounted is also replicated at every node. The scalability of this design is limited by its replication of the root file system and the mount table at every node. It is also limited by the use of atomic update across all sites to maintain the consistency of the root file system (and other replicated file groups).[23] A performance problem also arises from the fact that clients read directories over the network to look up names in file groups that are not stored locally. Sheltzer's thesis [28] proposes a directory caching technique to solve this problem; however, his design maintains cache consistency by requiring a directory's storage site to notify each holder of a cached directory page whenever the page changes. As compared with on-use consistency, this technique places a significant bookkeeping and communication burden on each directory storage site, further limiting the system's scalability. Overall, we are skeptical about the applicability of Locus techniques to systems of the scale we are contemplating.

The remote "mount" mechanism is also used by simple network file systems such as Sun Microsystems' *NFS* [26], the *Newcastle Connection* [4] and *Cocanet UNIX* [25]. Each of the cited systems links together a network of UNIX [23] hosts by allowing hosts to mount foreign file systems as subtrees of their own local file systems. As an example, host Laurel might mount host Hardy's root file system as `/hardy`, allowing Laurel to access Hardy's `/usr/spool/news` directory under the name `/hardy/usr/spool/news`. This approach is simple and adequate in the limited scenario of a cluster of hosts accessing a set of shared files. It does not, however, provide a consistent global name space for all files—the only way to ensure that all hosts use the same name for the same file is by careful manual management of each host's mount table. Systems using this approach also do not scale well, because if every host mounts every other, the number of mount points in the system is proportional to n^2, producing a significant management and computing overhead.

In general, we see our work as bridging the gap between conventional name service and file directory system technology. We have capitalized on the scalability, reliability, and security provided by global name services, while maintaining the efficiency and resiliency of local file directory systems.

[22] Their paper does discuss a planned extension to allow adding new prefixes at runtime.

[23] Locus ameliorates this problem to some degree by allowing directory updates to proceed even if some copies are unreachable—in fact, even if the system is partitioned—and using an automatic merge procedure to reconcile the partitions after they are rejoined and detect any name clashes that have arisen.

7 Conclusions

The decentralized naming architecture presented in this paper extends earlier work on global host and mailbox naming systems, providing a fast and flexible naming facility for performance-critical objects in distributed systems, such as files, programs in execution, and windows. Both expected and observed performance are close to optimum in terms of message traffic and response time. Resiliency against crash and omission faults is also close to optimum—the optimum is achieved for nearby objects, while for more distant objects resiliency is limited only by the fault-tolerance of the global directory servers. The name service can also be made secure against unauthorized behavior among the naming servers. Further, the design allows existing name spaces (such as existing file systems) to be incorporated as subtrees in the global name space with no modification to the existing system. These properties stem from several noteworthy aspects of the design.

The efficiency of the design derives from the fact that name handling for each object is implemented in the manager that implements the object, enabling name mapping to be performed as part of each operation that references an object by name. In contrast, implementing name handling as a separate lookup operation increases network traffic, server processing time, and response time to the client. Client-based name prefix caching allows the client, in the common case, to send each of its name mapping requests directly to the manager that implements the named object, performing as though it had an oracle providing this information. Name references relative to a client's working directory take even greater advantage of the caching mechanism, shifting some of the processing load of name mapping from servers to clients. Finally, when a client's cache misses, multicast provides parallel name lookup across a set of managers that might cover the name.

Some of these same features in the design also support fault tolerance. Because name handling is decentralized among all the object managers and object managers can be located using multicast, a nearby object is accessible by name whenever the object manager implementing the object is accessible.

These aspects of the design also contribute to its extensibility and flexibility. The name handling implementation at each object manager is independent of the global directory service and other managers, allowing the incorporation of pre-existing services into the name space. Naming conventions and even name syntax within each manager's subtree can vary from one manager to another.

The design makes the global character-string name of an object its only unique identifier, thereby avoiding the implementation difficulties and cost associated with using low-level unique identifiers. The directory identifiers used as cached hints in our design can be invalidated and reused at any time.

Our experience has also shown that the naming system works well in conjunction with file caching on client machines, a service recently added to V. In the V implementation, aliases for selected name prefixes are placed in each client's name cache, thereby directing requests for names in those subtrees of the name space to the local file caching server. For example, aliasing `%bin` to `%local/filecache/edu/stanford/dsg/bin` causes files from the DSG directory of system binaries to be cached under the name `%bin`. Each open request on a filename with the `%bin` prefix is presented to the file caching server, which satisfies the request if it has the file cached, or else opens the file on the remote server and adds it to the cache. In this application, the name cache provides a convenient place to store the aliases that redirect client requests to the file caching server. The name cache also functions in its normal role, reducing the name lookup load on the file caching server and reducing the need for multicast or name server access to locate remote resources.

There are two aspects to the design that we would cite as potential disadvantages. First, some name handling code is duplicated in each object manager and in each client. Although this is a legitimate concern, the amount of code for both servers and clients is modest, especially considering the current and expected future cost of memory. Second, the design relies for its resiliency on a multicast facility. In applications where lower resiliency is acceptable, the design could be modified to eliminate the dependence on multicast, relying entirely on the global directory service to respond to cache misses, but we do not believe such a change is necessary or desirable. We regard multicast as an important facility for building robust, flexible distributed systems, and we are working to make it more widely available [5, 8].

The design we have presented is primarily a *protocol* between clients and servers. An important next step is to carefully specify this protocol and offer it as an candidate for widespread use. Further work is also needed on replication techniques for global directories. In addition, we would like to to implement and gain experience with the secure version of the design. Nevertheless, based on the analysis we have presented in

this paper and the experience we have gained with the V implementation, we feel confident that the design is a sound basis for large-scale, efficient, reliable, and secure naming.

Acknowledgements

We are grateful to the Distributed Systems Group at Stanford for serving as a user community for the V system naming implementation, for commenting on the ideas presented here, and for helping to implement object managers that participate in the naming facility. We would especially like to thank Lance Berc, Peter Brundrett, Cary Gray, Ross Finlayson, Keith Lantz, Joe Pallas, and Marvin Theimer.

References

[1] A. D. Birrell, B. W. Lampson, R. M. Needham, and M. D. Schroeder. A global authentication service without global trust. In *Proc. 1986 IEEE Symposium on Security and Privacy*, pages 223–230. IEEE Computer Society, April 1986.

[2] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[3] D. R. Boggs. Internet broadcasting. Tech. Report CSL-83-3, Xerox, October 1983.

[4] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection—or UNIXes of the world unite! *Software Practice and Experience*, 12(12):1147–1162, December 1982.

[5] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the SIGCOMM '86 Symposium: Communication Architectures and Protocols*, pages 406–415. ACM, August 1986. Also *SIGCOMM Computer Communications Review* 16(3).

[6] D. R. Cheriton. UIO: A uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[7] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):105–115, March 1988.

[8] D. R. Cheriton and S. E. Deering. Host groups: A multicast extension for datagram internetworks. In *Proceedings of the Ninth Data Communications Symposium*. ACM, September 1985. Published as *Computer Communication Review* 15(4).

[9] D. R. Cheriton and T. P. Mann. Uniform access to distributed name interpretation in the V-System. In *Proceedings of the Fourth International Conference on Distributed Computing Systems*, pages 290–297. IEEE, 1984.

[10] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2), May 1985.

[11] S. E. Deering. Host extensions for ip multicasting. Technical Report RFC 988, Network Information Center, SRI International, July 1986.

[12] S. E. Deering and D. R. Cheriton. Host groups: A multicast extension to the Internet Protocol. Technical Report RFC 966, Network Information Center, SRI International, December 1985.

[13] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual Symposium on Principles of Distributed Computing*, pages 00–00, August 1987.

[14] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. The Ethernet: A local area network—data link layer and physical layer specifications, version 1.0. September 1980.

[15] D. K. Gifford. *Information Storage in a Decentralized Computer System.* PhD thesis, Stanford University, June 1981. Also available as Xerox PARC Technical Report CSL-81-8.

[16] B. W. Lampson. Designing a global name service. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 1–10. ACM, August 1986.

[17] S. Leffler, M. Karels, and M. McKusick. Measuring and improving the performance of 4.2BSD. In *1984 USENIX Summer Conference Proceedings*, pages 237–252, June 1984.

[18] T. P. Mann. *Decentralized Naming in Distributed Computer Systems.* PhD thesis, Stanford University, May 1987. Available as report STAN-CS-87-1179.

[19] P. Mockapetris. Domain names: Concepts and facilities. Technical Report RFC 882, Network Information Center, SRI International, September 1983.

[20] P. Mockapetris. Domain names: Implementation and specification. Technical Report RFC 883, Network Information Center, SRI International, September 1983.

[21] J. C. Mogul. *Representing Information About Files.* PhD thesis, Stanford University, March 1986. Available as Computer Science Technical Report STAN-CS-86-1103.

[22] D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, July 1983.

[23] D. M. Ritchie and K. Thompson. The UNIX timesharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[24] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[25] L. A. Rowe and K. P. Birman. A local network based on the UNIX operating system. *IEEE Transactions on Software Engineering*, SE-8(2):137–146, March 1982.

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. Technical report, Sun Microsystems, Inc., 1985.

[27] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[28] A. B. Sheltzer. *Network Transparency in an Internetwork Environment.* PhD thesis, University of California, Los Angeles, 1985. Available as UCLA Technical Report CSD-850028.

[29] H. R. Strong and D. Dolev. Byzantine agreement. Research Report RJ 3714 (42930), IBM Research Division, December 1982.

[30] D. B. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments.* PhD thesis, University of California, Berkeley, 1985. Available as UCB/CSD Technical report 85/228, and as Xerox PARC Technical report CSL-85-1.

[31] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V System. In *Proceedings of the 10th Symposium on Operating System Principles.* ACM, 1985.

[32] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 49–70. ACM, October 1983. Published as *Operating Systems Review* 17(5).

[33] D. W. Wall. Mechanisms for broadcast and selective broadcast. Tech. Report 190, Computer Systems Laboratory, Stanford University, June 1980.

[34] B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. Technical report, Computer Science Division, EECS Department, University of California, Berkeley, October 1985.