Decentralized Naming in Distributed Computer Systems

Timothy Paul Mann

Abstract

Designing a global character-string naming facility is an important and difficult problem in distributed systems. Providing global names—names that have the same meaning on any participating machine—is a vital step in welding a collection of individual computers into a single, coherent system. But the nature of large distributed systems makes it difficult to implement global naming with acceptable efficiency, fault tolerance, and security: network communication is costly, system components can fail independently, and parts of the system may belong to many autonomous and mutually-suspicious groups. Existing name service designs do not solve the problem in full; even the best current designs do not have the efficiency or capacity to name every object in a large system—they generally name only hosts or mailboxes, not files.

This thesis introduces a new paradigm for name service called *decentralized naming.* Directories at different levels of the global naming hierarchy are implemented using different techniques. The uppermost (global) level employs conventional distributed name servers for scalability, while at lower (regional and local) levels, naming is handled directly by the managers of the named objects. The name mapping protocol uses multicast for fault tolerance and a specialized caching technique for efficiency. A capability system provides security against counterfeit replies to name lookup requests.

The multicast name mapping technique is shown to have optimum resiliency, in the sense that whenever an object is accessible at all, it is accessible *by name.* An analytical model of cache performance is presented, is validated by comparison with measurements on a prototype implementation, and is used to set a limit on how large directories can grow before they must be treated as global rather than regional. The capability scheme is also analyzed: although it reduces both the efficiency and resiliency of name lookup, its impact can be made as small as desired by limiting the frequency with which security policy is allowed to change.

This technical report reproduces the author's Ph.D. thesis.

Acknowledgements

This thesis is dedicated to my parents, Roland and Ruth Mann. I regret that my father did not live to see its completion.

I would like to thank my advisor, David Cheriton, for believing in my ability to finish this thesis, urging me to keep at it, and providing many valuable suggestions and technical contributions. Mark Linton was most encouraging, especially during the months of self-doubt preceding my orals. Jeff Ullman applied his high standards to the proofs in Chapters 4 and 5, prompting me to make them much more solid.

I am grateful to the Distributed Systems Group at Stanford for their patience in serving as a user community for the V system naming implementation. Several group members contributed further by commenting on my ideas or helping to implement object managers that participate in the naming facility. Lance Berc, Peter Brundrett, Ross Finlayson, Keith Lantz, Joe Pallas, Michael Stumm, and Marvin Theimer were of particular assistance. Kenneth Brooks, Bruce Hitson, Rob Nagler, Bill Nowicki, and Paul Roy helped implement an earlier design [10].

My officemate Anil Gangolli deserves thanks for helping me review Poisson distributions as well as some other math I had forgotten. Joe Pallas acted as my TEX consultant, and Marvin Theimer passed on some additional TEX trickery from Howard Trickey. Mark Baushke and Cary Gray read drafts of Chapter 1 and made helpful comments.

Finally, I would like to thank my roommate Paul Veers for his firm Christian friendship and his inspiring example of hard work and strong faith.

This work was supported by the IBM Corporation under a Graduate Fellowship and by DARPA under contracts MDA903-80-C-0102 and N00039-83-K-0431.

Contents

1 Introduction **1**
- 1.1 The Problem . . . 1
- 1.2 Decentralized Naming . . . 2
- 1.3 Research Contributions . . . 10
- 1.4 What is Not Included . . . 11
- 1.5 Thesis Plan . . . 11

2 Related Work **12**
- 2.1 Remote File Access . . . 13
- 2.2 Distributed File Service . . . 14
- 2.3 Distributed Name Servers . . . 15
- 2.4 Other Related Work . . . 16
- 2.5 Chapter Summary . . . 17

3 Efficiency **18**
- 3.1 Cost Per Operation . . . 19
- 3.2 Cache Performance Model . . . 25
- 3.3 Measurements . . . 29
- 3.4 Limits to Growth . . . 33
- 3.5 Extension to Global Systems . . . 36
- 3.6 Chapter Summary . . . 37

4 Fault Tolerance **38**
- 4.1 System Model . . . 39
- 4.2 Name Mapping . . . 41
- 4.3 Binding Check . . . 44
- 4.4 Directory Listing . . . 47
- 4.5 Name Binding . . . 48
- 4.6 Replicating Global Directories . . . 50
- 4.7 Chapter Summary . . . 51

5 Security **53**
- 5.1 Mandatory and Discretionary Security . . . 53
- 5.2 Counterfeit Security Model . . . 55
- 5.3 Capabilities . . . 58
- 5.4 The Cost of Capabilities . . . 61

5.5 Can We Do Better? 64
5.6 Other Security Considerations 66
5.7 Chapter Summary 68

6 Concluding Remarks **69**
6.1 Summary 69
6.2 Future Work 72

List of Tables

3.1 Overall Statistics. 29
3.2 Statistics for Peak Half Hour. 30
3.3 CPU Cost Measurements. 31
3.4 Elapsed Time For Name Mapping. 32

4.1 Directory Types Along a Sample Pathname 51

5.1 Capability Field Lengths 63

Chapter 1

Introduction

1.1 The Problem

Designing a global character-string naming facility is an important and difficult problem in distributed systems.

The problem is important because, without global names—names that have the same meaning on any participating machine—a collection of computers can scarcely be viewed as a single, coherent system. Chaos and confusion are the rule when hosts do not share a common naming facility for globally available objects: users find that objects they called by one name when using one host are unavailable or called by another name when they move to another host; distributed or migrating programs find that the names they have been referencing on one host are invalid or have different meanings on the next.

The problem is difficult because of the characteristics of a large distributed system. Such a system can include a large and growing set of heterogeneous objects and hosts, with individual hosts and parts of the network subject to independent and intermittent failure. Parts of the system may be owned and controlled by many different autonomous and mutually-suspicious groups—different individuals, different departments within a university or corporation, different corporations, or even different countries. And of course, even with today's high-speed networks, communication between hosts is relatively costly compared to local computation.

These characteristics impose several challenging requirements on a naming facility. It must gracefully accommodate growth in the number and types of objects (and hosts) supported. It must be fault-tolerant—failures at one point in the system must have little or no effect elsewhere. Ideally, in fact, no matter how many failures occur, any set of hosts that remain up and interconnected should be able to continue interoperating as usual. It must support secure operation—in particular, it must solve the *counterfeit problem:* ensuring that when a client program sends out an operation request specifying a target object by name, it is not fooled by false (counterfeit) responses from servers unauthorized to bind that name. Yet the naming facility must be efficient, minimizing communication cost and avoiding bottlenecks, particularly when clients reference nearby or frequently-used objects.

Several designs for large-scale distributed naming facilities have been published [3,27,32], but these efforts have generally focused on the naming of relatively "large" objects, such as hosts or mailboxes. Naming to the level of individual files has not been included, apparently because these approaches do not offer high enough performance to be used every time a file is opened. Also, the level of availability they provide (through replication) is not needed when referring to individual, unreplicated files: when the contents of a file are unavailable, there is little utility in continuing to be able to look up its name.

A closer look at the naming problem suggests that no single implementation strat-

egy will be appropriate for every directory in a large hierarchical name space, because directories near the root of the tree can be expected to have substantially different usage characteristics from those near the leaves. Extrapolating from the behavior seen in smaller hierarchical naming systems, such as the UNIX [34] file system and the DARPA Internet Domain Naming service [30], I expect the directories in a large-scale system to fall into several broad usage classes, with the following characteristics.

Directories near the root are of *global* interest; that is, the entries they contain are accessed by nearly all client hosts. They must therefore be very highly available—at least for name lookup. These directories are also modified on occasion, but not nearly as frequently as they are read. When changes are made, it may be acceptable for them to propagate slowly through the system rather than appearing atomic, if this technique increases availability. Finally, a global directory is likely to contain entries for objects that are under multiple different administrations, so the authority to make changes must be carefully regulated.

Directories in the middle range of the tree are of *regional* interest; that is, each is accessed mostly by a group of client hosts that are geographically or administratively close to it—perhaps belonging to the same corporation or division. High availability remains important, but it is acceptable for the directory to become unavailable if the region it serves fails entirely. The rate of change is moderate. Entries in a regional directory refer to objects stored on multiple hosts, but generally all under the same administration.

Directories near the leaves of the tree are primarily of *local* interest; that is, each is accessed mostly by a small group of closely-associated client hosts—perhaps belonging to the same department or work group. Each directory at this level holds a set of related objects—for instance, the set of files containing the source code for a single large program—typically all stored on the same host. These directories are frequently accessed, and also change rapidly. In aggregate, most of the information stored by a large naming facility resides in the local directories.

Given this view of the problem, it is logical to look for solutions that use different implementation techniques at different levels of the naming hierarchy. One such solution, called *decentralized naming,* is introduced and evaluated in this thesis. It is shown to have attractive fault tolerance, efficiency, and security properties, and its practicality is demonstrated by a substantial prototype implementation for the V distributed operating system [7].

1.2 Decentralized Naming

In preparation for Section 1.3's summary of research contributions, this section gives an overview of decentralized naming and defines some necessary terminology.

Decentralized naming uses three directory implementation techniques—*global, regional,* and *local*—corresponding roughly to the three usage classes described above. *Global* directories are stored by specialized *directory servers;* for fault tolerance, they are fully replicated. *Regional* directories are partially replicated, with entries distributed across the object managers that implement objects named relative to them. Each *local* directory is stored exclusively by one object manager.

This naming technique is called *decentralized* because each object manager handles the naming for its own objects. In fact, each object manager knows the full absolute pathname for every object it implements; it is a *participant* in the implementation of each directory along the path. (A participant in a directory is a server that holds an authoritative record of one or more entries in that directory.) For example, in Figure 1.1 below, file server 1 implements a local directory named `[edu/stanford/dsg/bin`, so it is also a participant in the directories `[`, `[edu`, `[edu/stanford`, and `[edu/stanford/dsg`, holding the entry

for `edu` in the directory `[`, the entry for `stanford` in the directory `[edu`, and so forth.[1] An object manager ordinarily records only those directory entries required to define the absolute names of the objects it manages. For example, in `[edu/stanford/dsg/user`, file server 3 records only the names `jones` and `mann`, not `smith`. Directory servers also participate in some directories; a directory server holds a complete list of entries for each directory it participates in.

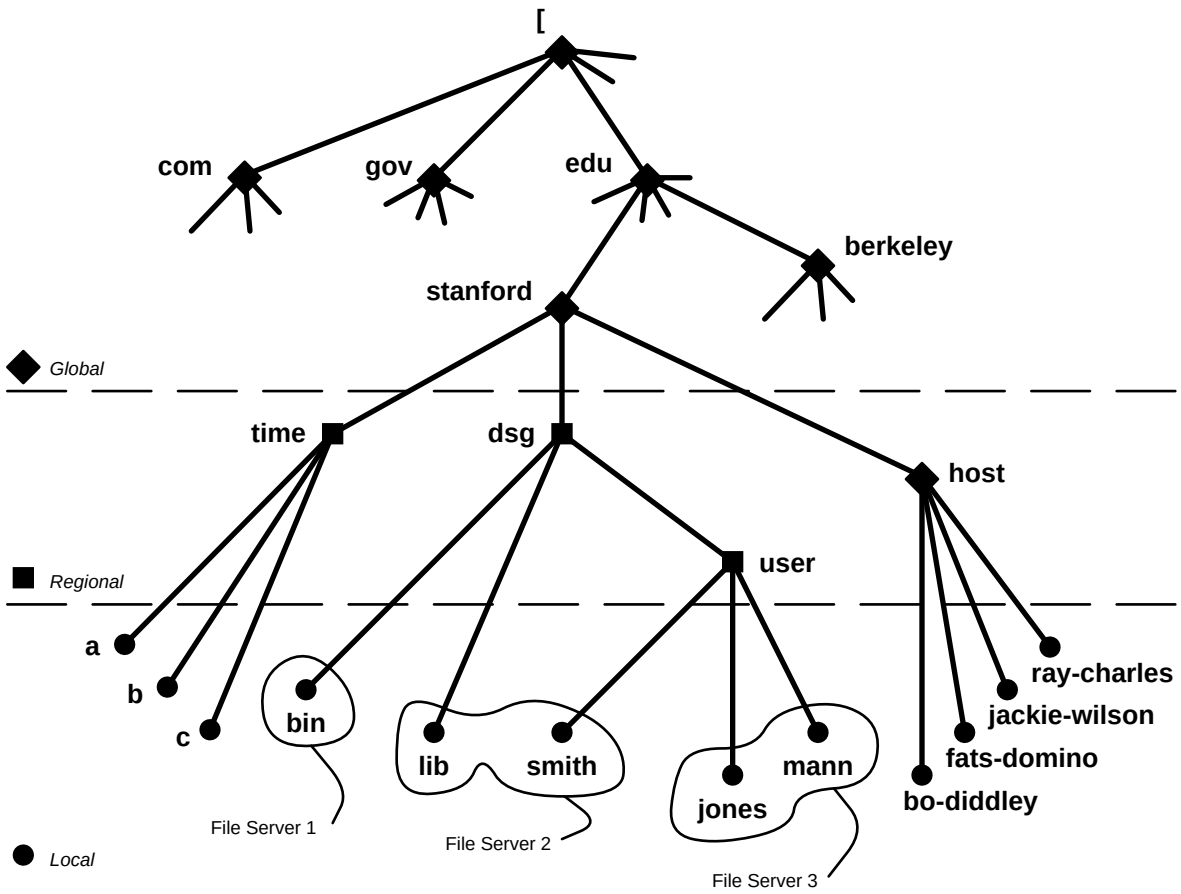

Figure 1.1: A Small Example

Local, regional, and global directories are distinguished by the number and type of servers that participate and the roles that the various participants play in name mapping (lookup). Each is detailed below.

Local Directories

A *local* directory has exactly one participant. One object manager stores all entries in the directory, handles all name mapping in the directory, and manages all the objects named relative to it. The manager knows that it is the only participant, so it *covers* all pathnames that pass through the directory.[2] For example, in Figure 1.1, every object

[1] In these examples, pathname components are delimited by "/" characters, and absolute names are flagged by a leading "[" character. This convention is used in the V naming implementation.

[2] An entity is said to *cover* a name if it authoritatively knows either what the name is bound to, or that the name is not bound.

whose name begins with the prefix `[edu/stanford/dsg/user/smith` is a file or directory whose contents and name binding are stored by file server 2. Every descendant of a local directory is local as well and has the same manager; for example, `[edu/stanford/dsg/user/smith/source/emacs` would also be a local directory stored by the same manager as `[edu/stanford/dsg/user/smith`.

Regional Directories

A *regional* directory has multiple participants. Each participating object manager stores a subset of the entries in the directory—specifically, each manager holds the entries for exactly those child directories in which it participates. For example, as shown in Figure 1.2, file server 2 holds the entry for `smith` in the `user` directory, while file server 3 holds the entries for `jones` and `mann`. The entry for a regional child directory is thus replicated at each participant in the child, while that of a local child is stored only by the child's manager. In the `dsg` directory, for instance, file server 1 binds `bin` and file server 2 binds `lib`, while *both* file servers 2 and 3 bind `user`. Every descendant of a regional directory is either regional or local.

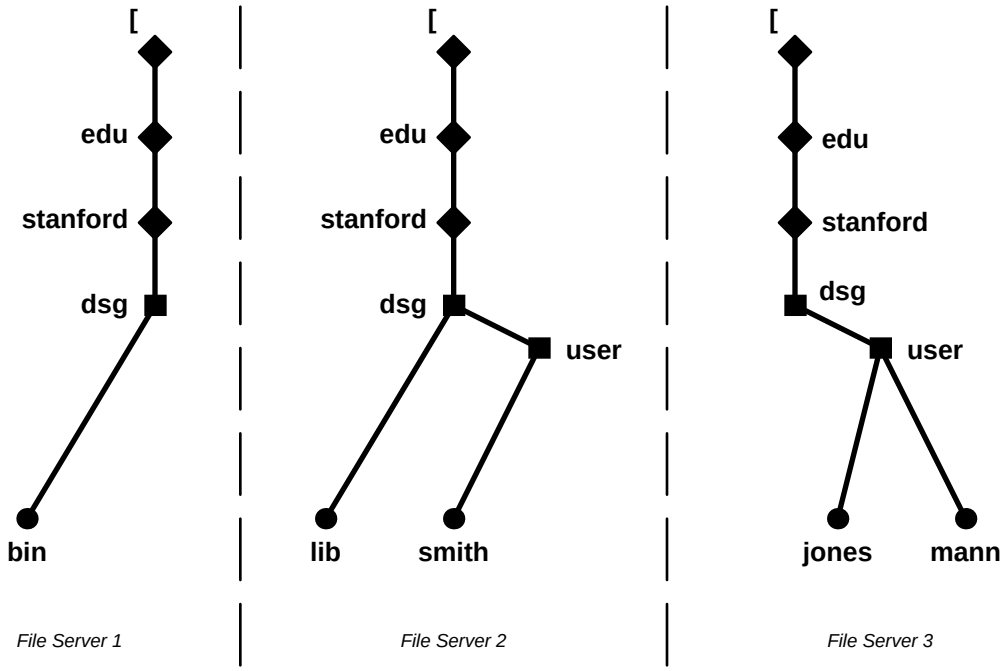

Figure 1.2: Information Held by Each Manager

Name mapping in a regional directory is performed by sending a request to every participant in the directory; only those participants that cover the given name respond. To make such operations more efficient, the participants in each regional directory form a *participant group* to which multicasts can be directed; name mapping requests are then multicast to the group and processed in parallel by its members. The underlying network is assumed to provide multicast with the following semantics: Any set of hosts (or processes) can form a *group* with a single address. Neither the clients 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. When a message is sent to the group, delivery to each member is assumed to succeed or fail independently of delivery to the others, and failures are not

necessarily reported to the sender.[3]

One or more participants in a regional directory may hold a *name list* for the directory—a complete list of single-component names for which directory entries exist. A name list does not include the directory entries themselves, only the names—that is, it does not record *what* the listed names are bound to, only that each is bound to *something.* (For example, the name list for `[edu/stanford/time` in Figure 1.1 would simply be `(a, b, c)`. It would not indicate that the named objects are time servers, give their network addresses, or the like.) Name lists are used primarily to prevent duplicates from arising when new names are defined in a directory; a new name must be added to the list before it can be bound. They have other uses as well—if a client attempts to map an unbound name in a regional directory, a name list holder that receives the request can generate an error response because it covers all the unbound names: it knows that any name not on its list is unbound. There are three useful approaches to storing the name list—off line, at managers, or at directory servers.

Off-line name lists are used in directories where new names are only defined manually by a system administrator. For example, new host names are only chosen when new machines are acquired, so at an institution with only a few hosts (called, say, Marquette), the directory `[edu/marquette/host` could be made regional, with the name list simply written on a piece of paper and kept in a system administrator's desk drawer. This approach is simple and symmetrical, but has the drawback that no on-line participant covers the directory's unbound names. In the example, a client that references an unbound Marquette host name receives no reply—its name mapping request times out, leaving the client uncertain whether the name is unbound or is bound to a host that has crashed.

At the opposite extreme, all (or most) managers participating in a regional directory may keep a complete name list. In Figure 1.1, for example, both file server 2 and file server 3 might well keep a complete name list for `[edu/stanford/dsg/user` in stable storage. Either one is then prepared to give an error response when a client attempts to map an unbound name in the directory; however, cooperation is required to keep both replicas up to date when either file server adds or deletes a directory entry.

An intermediate approach is to include one or more directory servers as participants in the directory and give them the responsibility of maintaining the name list. If more than one server is included, they coordinate with each other as necessary to keep the copies identical, as with replicated global directories. This approach keeps the object managers simple, yet retains the benefits of having the name list available on line. It does, however, put an additional burden on the directory servers. In the figure, the `[edu/stanford/host` directory would become a candidate for this implementation strategy if many additional hosts were added.

Global Directories

A *global* directory has many participants, some object managers and some directory servers. Each participating directory server holds a complete copy of the directory—all entries, with the binding for each name given. Most operations on the directory (including name mapping) are handled by the directory servers, so the participating object managers need not form a multicast group. The object managers do, however, continue to store entries for the subdirectories in which they participate.

Most global directories are replicated on several directory servers to improve the fault tolerance and efficiency of read operations (such as name mapping). In an internetwork, the root directory would typically be replicated at least once on every subnetwork, and

[3]Group communication of this sort is available in the V system using process groups [11], at the Ethernet data link level using multicast addressing [18], and experimentally in the DARPA Internet using host groups [9,13,14].

lower-level global directories would be replicated on those subnetworks where they are heavily used. Update algorithms such as those developed for use in Grapevine [3] and its descendants are well suited for use in global directories. Updates are infrequent in the highest-level directories of a large hierarchical name space, and so the slow update propagation rate and "eventual consistency" property of these algorithms should be acceptable.

Whether a particular directory is implemented as global or regional is a matter of administrative choice. Name mapping in a regional directory is more fault-tolerant, but it is more efficient in a global directory, and the efficiency advantage increases as the directory grows to include more participants. This issue is discussed further in Chapter 3, which derives a practical limit on the size of regional directories, based on performance considerations.

1.2.1 Name Mapping

Name mapping is an extension of name lookup. It accepts a name and a message as its arguments, looks up the name's binding, and delivers the name and message to the manager of the bound object, if any. It returns either a response from the object manager or (if the name was unbound or the manager could not be contacted) a failure indication. This definition of name mapping reflects the view that the usual reason for looking up a name is as a preliminary step in performing an operation on the named object. For example, if a client program wants to open a file `foo`, it calls the `OpenFile` routine with the target file specified by name. In a decentralized naming system, the `OpenFile` request is "piggybacked" on the name mapping request for `foo` by including it in the message that is delivered to `foo`'s manager. This technique saves network traffic by allowing both the name lookup and object operation to be requested in a single packet.

With the directories stored as described above, name mapping can be performed using a simple (but inefficient) protocol involving the directory servers and multicast.[4] The client begins by submitting its operation request to a directory server for the root directory, which looks up the first name component. If the name maps to another global directory, the server forwards the request to a server for that directory (perhaps itself), and this process is repeated until a regional or local directory is reached. If a regional directory is reached, the last directory server forwards the request as a multicast to the participant group for the directory. Each participant then examines the remaining name suffix to determine whether it covers the name; those that do not cover the name ignore the request. A participant that does cover the name is either the manager of the named object or knows that the name is unbound. In the former case, it performs the requested operation and returns the results; in the latter case it returns a *not found* error indication. If a local directory is reached, the directory server forwards the request on to the directory's manager, which of course covers the name, and it handles the request.

An example of the basic name mapping technique is shown in Figure 1.3. A client program needs to open a file named `[edu/stanford/dsg/user/mann/phonebook`. It first submits its request to a directory server for the root directory "`[`", which looks up `edu` in its copy of the directory. The request next passes through directory servers for `edu` and `stanford`. Determining that the name `dsg` is bound to a regional directory, the server for `stanford` passes the request on to the directory's participants—file servers 1, 2, and 3—by multicasting it to the participant group. File server 1 does not hold a binding for the name's fourth component, `user`, so it ignores the request, assuming another participant in `[edu/stanford/dsg` will take care of it. File server 2 holds a binding for `user`, but does not hold a binding for `mann` in the regional directory `[edu/stanford/dsg/user`, so it too ignores the request. File server 3 does know the binding status of the given name, because it implements `[edu/stanford/dsg/user/mann` as a local directory. It therefore completes

[4] The next section explains how caching is used to improve efficiency.

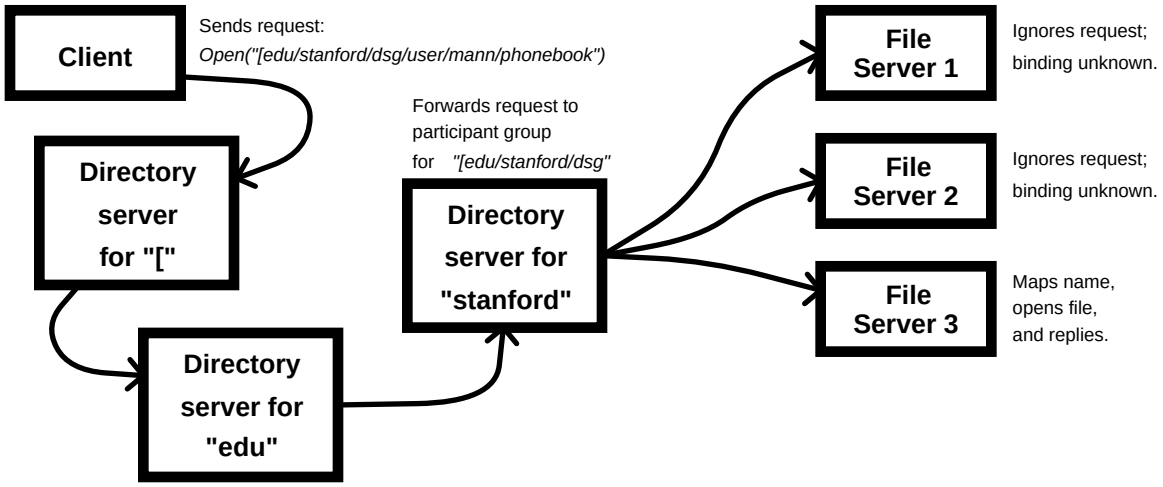

Figure 1.3: Name Mapping Using Directory Servers and Multicast

the name mapping to find the requested file, opens it, and returns a handle on the open file to the original client, as shown.

There are three possible outcomes to this name mapping procedure: it either succeeds, fails with an error reply, or fails with no reply.

Name mapping succeeds, as in the example above, when the given name is bound and the client is able to communicate with the bound object's manager.

Name mapping fails with an error reply when the given name is not bound, but is covered by some object manager with which the client is able to communicate. For instance, if the client in the previous example had invoked the `OpenFile` operation with the misspelled name `[edu/stanford/dsg/user/mann/phnoebook`, file server 3 would have recognized the name as unbound and returned an error indication.

Name mapping fails with no reply when the client is not able to communicate with any entity that covers the given name—either a hardware fault has made it inaccessible, or there is no such entity. Generally, several retransmissions and a time delay are required before the client concludes that no reply is likely to be forthcoming. In the example, if file server 3 crashes or is partitioned away from the rest of the network, a client presenting the name `[edu/stanford/dsg/user/mann/phonebook` would receive no reply. To illustrate the second possibility, suppose that the name list for `[edu/stanford/dsg/user` is not kept on line, and a client presents the name `[edu/stanford/dsg/user/amnn/phonebook`. In this case, all three file managers ignore the request and again, the client receives no reply. This example shows why (as remarked above) it is useful to keep the name list of a regional directory on line: it would be preferable for the client to receive a prompt reply stating that the name is definitely *not* bound, instead of waiting through a timeout period and then remaining uncertain as to the reason for the failure.

The basic technique just described is rather inefficient, but it is not the last word in decentralized name mapping; the next subsection describes a more efficient and sophisticated technique that takes advantage of naming information cached by clients.

1.2.2 Name Caching

Each client of the naming facility keeps a *prefix cache* in its local memory. The cache records bindings between name prefixes (that is, directory names) and directory managers or groups. These caches are used to reduce the average cost of performing naming opera-

tions; they do so by reducing the number of multicasts and the number of requests sent to directory servers. Whenever a client is attempting to map the name n, it begins by looking in its cache to find the longest matching prefix of n.[5] If such a prefix match is found, the client sends the request to the manager or group indicated by the cache. If no match is found, the client falls back on the basic name mapping procedure, sending the request to a directory server for the root. A cache lookup is considered a *hit* if the prefix match reaches a local directory, a *near miss* if it reaches a regional or global directory below the root, or a *miss* if there is no prefix match.

Cache entries are normally created *on demand.* Whenever a cache miss or near miss requires the client to use multicast or go through a directory server, the server that responds also provides information for the client's cache. For instance, suppose the client in our running example attempts to map the name `[edu/stanford/dsg/user/jones/mbox`, but finds only the prefix `[edu/stanford` in its cache. It will receive a response from file server 3 indicating that it manages `[edu/stanford/dsg/user/jones` as a local directory.

Entries can also be *preloaded* into a cache to reduce the impact of startup misses. For example, in the V implementation, all bindings in the root directory are preloaded into each client's cache.

Cache consistency is maintained by detecting and discarding stale cache entries *on use.* A cache entry becomes *stale* when the name prefix it contains is no longer covered by the manager or group it indicates. When a stale cache entry is used, the result is that the name request is sent to the wrong manager (or group). If the manager no longer exists, the request fails with no reply. If the manager exists but no longer covers the given name, it reports that fact back to the client. In either case, the client recognizes that its cache entry is (or may be) stale, and retries its request without using that entry. In no case can the name be mapped to the wrong object.

1.2.3 Nearby Groups

One important question remains to be answered: when a client's cache misses entirely, how does it *find* a root directory server to fall back on? *Nearby groups* provide a solution to this problem and also improve the fault tolerance of name mapping.

The nearby group mechanism works as follows. Conceptually, for each host H's there is a multicast group b_H consisting of all object managers and directory servers that are near to H—say, within one or two hops through the network—called the nearby group for H. Normally, a client's cache includes an entry mapping from the root directory name ("`[`") to the nearest root directory server, which matches any name that is not matched by any other cache entry. If the root entry is missing or proves stale, however, H multicasts its next name mapping request to b_H. If one of the nearby managers binds the given name, it will respond to the request; if a nearby root directory server is up and accessible, it will respond with a new root cache entry pointing to itself. This technique makes access to nearby objects independent of failures in the global directory system.[6]

This mechanism can be implemented without actually creating a large number of different groups. Instead, all servers for the root directory, together with all participants in every top-level regional directory, join a single *global group.* When a client needs to send to its nearby group, it multicasts its request to the nearby members of the global group—say, to all members that can be reached in one or two hops through the internetwork. (Scoped multicast of this sort is available in the IP host group facility [9].)

[5] 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`.

[6] One can also make access to distant objects resilient against the failure of nearby directory servers by introducing more retries to larger groups.

1.2.4 Generic and Group Naming

In a distributed system, objects that are logically one unit are frequently either replicated or split into fragments, with each copy or fragment maintained by a different object manager on a different host. If clients are to see a replicated or fragmented object as a single entity, it is important that the naming system be able to bind all its parts (or *subobjects*) to a single name. Mapping this name should return *all* subobjects if they are fragments of the main object; it is sufficient to return *any one* subobject if they are copies. Names bound in these ways are called *group* and *generic* names, respectively, as opposed to *specific* names, which are bound to exactly one object.

A decentralized naming facility implements generic names by relaxing the restriction that only one participant in a regional (or global) directory can bind a given name to an object it manages. Then when a client's name mapping request is sent to the directory's host group, several participants can respond. The client software selects one response and caches the identity of the responder. From then on it sends requests only to the subobject it first selected, until that manager no longer binds the name or is no longer accessible. Such events are treated as cache misses, and cause the client to fall back on multicast name mapping to select a new referent for the name.

Generic naming has proven particularly useful in the V system. For example, the Distributed Systems Group at Stanford keeps a complete tree of standard system files (such as executable binaries for common commands) on each of several file servers in the Computer Science building. Client programs reference these files using a well-known generic name, so that if one file server crashes, the others transparently take over its load. Note that the naming system treats each tree root as an entirely separate local directory; it is not responsible for keeping the tree copies identical. Thus the problem of file replication has been factored out of the naming system and treated separately.

Because a group-named object is fragmented among multiple managers, not replicated, name mapping requests on it should be transmitted to all its managers. Therefore, a group-named object has a multicast group associated with it, whose members are the managers of its subobjects. A client whose cache misses on the group name is given the group address to put into its cache; it then multicasts subsequent name mapping requests to the group. (The naming system does not take responsibility for ensuring that such multicasts reliably reach all subobject managers; in cases where reliable multicast is required, the participating managers must implement it themselves.) Regional directories can be viewed as group-named objects—they are fragmented across multiple participants, with some entries held by each.

1.2.5 Other Naming Operations

Besides name mapping, decentralized naming provides three other "read" operations on name bindings—directory listing, binding check, and inverse name mapping; and two "write" operations—name binding and name unbinding.

The *directory listing* operation returns a list of all bound names in a specified directory, optionally including a type-dependent descriptor for each bound object. The difficult case here is listing a regional directory with no on-line name list holders. The V implementation uses an unreliable, "best-efforts" protocol to provide such listings: the client repeatedly multicasts a request to the participant group for the directory, each time appending a list of participants that have already responded and therefore should not respond again, until the request has been transmitted several consecutive times with no response. It then collates all the received replies to produce a listing.

The *binding check* operation accepts a name and reports whether it is bound. It differs from name mapping in that its definition does not require it to send a message to the

manager of the bound object (if any).

Inverse name mapping operations accept a manager-specific low-level identifier for an object and return the object's absolute name. The `pwd` command of UNIX, for example, performs an inverse name mapping on the user's working directory. With decentralized naming, such operations are particularly cheap and easy to implement because (as mentioned above) each object's absolute name is known locally by its manager.

A *name binding* operation creates a new binding between a given name and a given object. If the specified object's manager already covers the given name, binding is straightforward and no more costly than name mapping. Acquiring coverage of new names is more problematical; it requires reliable communication with the server that previously covered the name.

Name unbinding deletes the binding between a given name and object. If the specified name is still to be covered by the same object manager, unbinding is straightforward. Giving up coverage requires some additional care.

This completes our overview of decentralized naming. Further details are introduced as needed in subsequent chapters.

1.3 Research Contributions

This thesis investigates the properties of decentralized naming. It presents results in the areas of efficiency, fault tolerance, and security.

One group of results characterizes the *efficiency* of decentralized naming. First, the average cost of performing each of the five major naming operations is derived in terms of basic system parameters, including the cache hit ratio, and is compared with the theoretical optimum. Next, an analytical model of cache performance is presented, and it is validated by comparing its predictions with measurements taken on the V naming implementation. Finally, it is shown that the maximum practical number of object managers that can participate in a regional directory is limited by the fact that both the cost of mapping a name and the average name mapping load *per participant* contain terms that are proportional to the number of participants, and an estimate of this maximum for real systems is derived.

A second group of results characterizes the *fault tolerance* of decentralized naming. Name mapping for nearby objects is shown to have the optimum possible fault tolerance; whenever an object is accessible at all, it is accessible *by name.* All faults are tolerated except failure of the named object's manager or network failures that prevent communication with it—either of which would prevent operations on the named object from succeeding, even if its name could still be mapped. Optimum fault tolerance is achieved, however, only because name mapping is defined to require communication with the named object's manager, and is not required to distinguish failures caused by unbound names from those caused by inaccessible object managers. Decentralized binding check, which is not defined in this way, is shown to have weaker fault tolerance properties. It is also shown that name binding in a distributed system cannot be made resilient against as large a set of faults as can name mapping, regardless of whether decentralized naming or some other technique is used. A special case of decentralized name binding can and does achieve the same resiliency as name mapping, however.

A third group of results lies in the area of *security.* The use of decentralized naming does not complicate the problem of providing mandatory security in a distributed system, but a solution to the *counterfeit problem* of discretionary security (mentioned above) is required. This thesis presents such a solution, based on capabilities, evaluates the solution's impact on the efficiency and resiliency of naming, and argues that no better solutions are available. It is shown that, in general terms, one can approximate the efficiency and resiliency of unsecure decentralized naming more and more closely as the detailed security

policy is allowed to change less and less frequently.

Finally, this thesis demonstrates the practicality of decentralized naming, by describing a substantial prototype implementation that is in daily use in the V distributed operating system.

1.4 What is Not Included

This thesis concentrates on the implementation of regional (and local) directories because it is the most interesting and novel part of the decentralized approach. A detailed discussion of mechanisms for implementing replicated global directories is omitted because the required technology has already been rather well explored by other workers [3,40,27,32,42]. Also, the prototype V implementation does not include any global directories: even the root directory is implemented as a regional directory (with some optimizations). In a positive sense, the success of this implementation indicates that a system confined to a single local or campus-wide net is small enough that global directories and their complex replication mechanisms are not needed.

The discussion of security in Chapter 5 concentrates on the counterfeit problem, because it is the most interesting security problem that arises in a decentralized naming facility. Most security issues that have been considered by other authors are either not directly related to naming, or can be solved in the same way in a decentralized naming system as in any other.

1.5 Thesis Plan

The next chapter surveys related work in naming. Chapter 3 evaluates the performance of decentralized naming, discussing the caching mechanism used to achieve high performance, and estimates the limits on the size of a regional directory that are imposed by performance considerations. Chapter 4 evaluates the fault tolerance of decentralized naming. Chapter 5 is concerned with the security of decentralized naming, describing and evaluating a solution to the counterfeit problem. Chapter 6 summarizes the research and suggests directions for future work.

Chapter 2

Related Work

Naming has been recognized as a fundamental issue in computer systems for many years. Many sorts of objects need names, spanning a wide range in granularity, from individual memory cells to large networks of computers within an internetwork. Many types of names are used, from numeric identifiers chosen for the convenience of hardware devices to character strings or even pictures (icons) chosen for the convenience of human users. Common issues recur throughout this large problem space, but a single thesis can treat only a small part of it; the survey below therefore concentrates on a few naming systems that are closely related to the main topic of this thesis, merely hinting at the breadth of existing work in naming,

One of the most basic components of a von Neumann computer is the addressable memory—which is essentially a naming mechanism: memory addresses serve as low-level names for data and instructions within a program. Stored-program computers derive much of their flexibility from this mechanism, because it allows the binding between addresses and values to change from one program run to the next, and even during program execution. While the earliest and simplest computers maintained a fixed binding between addresses and physical memory cells, later designs incorporated such innovations as base registers, paging, and segmentation to support multiprogramming, virtual memory, and data sharing [20,15,24].

Because computers are programmed and used by people, it is also important to provide meaningful, mnemonic names for the objects they manipulate. Symbolic assembly languages and higher-level languages allow the programmer to assign such high-level names to the objects manipulated by his program. (Knuth and Trabb Pardo give an interesting account of the early development of these languages [25].) Most programming languages only provide names that are local to a single computation, however.

The task of providing global names for shared objects (particularly files) in a single-machine system has traditionally been assigned to the operating system. There is little published research on file naming in single-machine operating systems; Saltzer [37] gives an excellent overview of the state of the art in this area. Early systems provided a flat space of file names within each storage device, while the next step was to introduce a separate directory for each user account. Due to its greater flexibility, the hierarchical directory system of MULTICS [12] has strongly influenced more recent system designs, beginning with UNIX [34].

Naming in distributed systems presents a whole new class of problems. As was noted in the introduction, the nature of a large distributed system makes it difficult to construct a naming facility that is at once acceptably fault-tolerant, efficient, and secure. The remainder of this chapter discusses several existing distributed naming facilities, each representing an important contemporary style of naming architecture, and contrasts them with the decentralized approach.

2.1 Remote File Access

One simple type of distributed naming facility results when a single-machine operating system is extended to provide transparent access to remote files across a network. Some recent examples of such remote file access systems include Sun Microsystems' *Network File System* [39] and the *Newcastle Connection* [5]; an older example is *Cocanet UNIX* [36]. Each of the cited systems links together a network of UNIX [34] hosts by allowing hosts to "mount" foreign file systems as subtrees of their own root file systems. As an example, host Laurel might mount host Hardy's root file system as `/hardy`, allowing it to access Hardy's `/usr/spool/news` directory under the name `/hardy/usr/spool/news`.

Remote file access systems can be quite useful, but they differ fundamentally from decentralized naming in that the naming they provide is not global. That is, because there is no mechanism for keeping the multiple root file systems identical, the same object can have different "absolute" names when viewed from different hosts. This lack of uniform naming can cause difficulties for distributed application programs, because processes running on different hosts are in different naming domains. For example, an application using several hosts to process a data file would run into trouble if the file name were specified as `/usr/mann/data`; rather than opening the same file, participating processes on hosts Laurel and Hardy would respectively open `/laurel/usr/mann/data` and `/hardy/usr/mann/data`. Even if the user were to explicitly specify `/hardy/usr/mann/data`, the program would fail if Laurel did not have Hardy's file system mounted, or worse, had some other file system mounted under the name `/hardy`.

The efficiency and fault tolerance of these systems is inherently good, at least when each host has a local disk. Name mapping is efficient because the initial components of each name are mapped locally, the remainder by the host storing the file, and mapping is piggybacked on the `Open` operation; thus a remote file can be opened with a single packet exchange. Name mapping is fault-tolerant in that, once the initial `Mount` operation has located the remote host, client programs can continue to access files on it as long as it, the local host, and the network remain up; there are no other servers whose failure can prevent things from working. Both the efficiency and the fault tolerance are reduced when the client does not have its own disk, however, because name mapping then requires access both to the server that holds the actual file and to the one that holds the client's root file system. (The client can, of course, regain some efficiency by caching directory information from its root file system in local memory.)

Remote file access systems do not scale well, because if every host mounts every other, the number of mount points in the system is proportional to n^2. Because of this problem, each host in a large installation typically mounts only those file systems its users expect to need, so users who move from one host to another see a different set of available files on each.

Also, as installations grow, the `Mount` operation is performed more and more frequently, so its fault tolerance, efficiency, and detailed semantics become important issues. How are remote hosts and file systems named in the arguments provided to a `Mount` call, and how are those names mapped? Whatever mechanism is used for that purpose is effectively a part of the file naming system as well.

In conclusion, although remote file access systems do not provide a global name space, they are useful in some applications, and they tend to be highly fault-tolerant and efficient. They have also proven relatively easy to implement, even as extensions to existing UNIX systems. They are at their best when a cluster of hosts running single-machine operating systems need access to a common set of files, but do not need to run distributed programs.

2.2 Distributed File Service

Locus [44] represents another class of naming architecture, in which a single file system is extended across a network to form the backbone of a tightly-integrated distributed system. The Locus naming facility provides a uniform, global name space for objects (primarily files) stored by multiple servers. Its implementation techniques differ markedly from those of decentralized naming, however, giving it different efficiency and fault tolerance properties.

The Locus naming hierarchy is implemented as a set of nonoverlapping subtrees called *file groups,* analogous to *file systems* in UNIX. Each file group can be stored at any site, or replicated (fully or partially) at several sites. Both the files and the directories in a replicated file group are kept consistent by performing updates as multi-site atomic transactions. As in UNIX, the complete global tree is built up by first designating the root of one file group as the global root, then repeatedly using the `Mount` operation to attach new groups at the leaves of the exiting tree. Knowledge of where each file group is mounted is replicated at every site.

The basic name mapping technique used in Locus is quite inefficient, but acceptable efficiency is achieved by the addition of replication and caching. Mapping a single pathname can in principle require traversing several file groups at different storage sites before the site of the target file is reached. In practice, however, the root file group is replicated at every site and (presumably, as in ordinary UNIX installations) most mounted file groups are placed directly under the root, so that as in the remote access systems of the previous section, name mapping begins at the local host and proceeds directly to the host holding the named object. Thus, Locus name mapping could easily achieve the same efficiency as do remote access systems; however, a peculiarity in the implementation sacrifices much of that efficiency—the site that originates a name request also performs the lookup for every component in the pathname, reading directories over the network when they are not replicated locally. Sheltzer's thesis [41] discusses the addition of directory caching to solve this performance problem. His technique keeps the caches consistent 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 an added bookkeeping burden on the storage site and requires it to send notification messages that are often unnecessary (because the caching site will not use the updated page before the next change); on the positive side, however, the fact that cached information is always correct allows the holder to use the cache to perform directory listing as a purely local operation, with no need to contact the directory's storage site.

Directory replication is used to improve the fault tolerance of Locus name mapping as well as its efficiency. Replication is quite important; without it, failure of the site holding the root file group would bring down the entire system by making all files inaccessible. Of course, replication tends to *reduce* the resiliency of directory update operations, because an update must reach every copy to assure consistency. Locus deals with this problem by allowing directory updates to proceed even if some copies are unreachable—in fact, even if the system is partitioned, updates can proceed in each partition. An automatic merge procedure reconciles the partitions after they are rejoined, detecting any name clashes that have arisen and notifying the owners of the affected files by electronic mail. A polling protocol is used to detect network failures and establish a consensus on the membership of each partition; it is not clear how well this protocol will perform in large installations.

In conclusion, the Locus naming facility has a number of interesting features, but is not a full solution to the problem considered in this thesis. Its efficiency problems have already been discussed. Further, due to the replication of the root directories and the mount table on every host, its ability to scale up to large installations (of hundreds or thousands of hosts) is questionable.

2.3 Distributed Name Servers

In recent years, it has become popular to use *distributed name servers* to name hosts, mailboxes, and other objects of similar granularity within large internetworks. In this approach to naming, the global directory tree is distributed across multiple name servers scattered throughout the internetwork, with each directory typically replicated at several server sites. Terry's thesis surveys work in this area [42].

Perhaps the most advanced example of such a naming service is a design described by Lampson in a recent paper [27]. Unlike its predecessors Grapevine [3,40], and the Clearinghouse [32], this design supports an unlimited-depth naming hierarchy, and it is targeted for even larger installations—potentially incorporating every networked computer in the world into a single name space. It is similar to its predecessors in that each directory is replicated in full at an administratively selected set of sites, and update is performed non-atomically. Lookup in a directory is defined nondeterministically: it may return any name binding that was established since (or current at) the most recent "sweep" of the directory. Sweeps occur periodically for each directory, bringing all copies into an identical state. Conflicting updates are reconciled by timestamps—the latest update wins.[1]

Lampson's paper concentrates on naming objects of relatively large granularity (such as hosts or mailboxes), but mentions that "the name service can be used to name a file system." His colleagues have extended the design to provide global file naming in that way, forming a file's absolute pathname by concatenating a global file system name (provided by the name service) with a local file name (provided by the file server) [2]. To look up an absolute file name, one first submits it to the global naming service, which maps a prefix of the name to locate the server storing the file; the request is then passed on to that server to complete the name mapping. This technique is similar, but not identical, to the basic name mapping protocol of decentralized naming.

One major difference between decentralized naming and the extended Lampson design is that the latter does not make any use of multicast: it does not include either regional directories or the nearby-group feature of decentralized naming. In a sense, it is rather like an installation of decentralized naming that has been configured with no regional directories—all directories with entries on more than one host are managed by the global name service. For example, suppose a client host H with an empty name cache attempts to open a file called `[edu/stanford/dsg/fs1/george/calendar` that is stored on a file server FS_1. Under the extended Lampson design, the global name service must implement enough of the name to map to a particular server; for instance, if the entire tree rooted at `fs1` is implemented by FS_1, the global name service maps the prefix `[edu/stanford/dsg/fs1` to locate that server, then passes the request on to it. Under decentralized naming, on the other hand, if `dsg` is the first regional directory in the pathname, the global directory service maps the prefix `[edu/stanford/dsg` to find a multicast address for the participants in `dsg`, then forwards the request to that address. (Or if the global directory service cannot be reached and FS_1 is nearby to H, H's multicast to nearby servers reaches FS_1.) FS_1 then maps the remainder of the name and responds to the request. The request succeeds as long as FS_1 is up and the global directories `[`, `edu`, and `stanford` are available, or even without the global directories if FS_1 is nearby to H. To approach this resiliency under the Lampson design would require the `dsg` directory to be replicated at FS_1 (and at each host named by an entry in the directory—there is nothing special about FS_1 in this example). Multicast name mapping cannot simply be tacked on as an added feature in the Lampson design, because its philosophy is that objects do not necessarily know their own names—the name service does not inform an object when its name is changed.

The second major difference between decentralized naming and Lampson's design is in

[1] The design includes an authentication service as well, which is described in a companion paper [19].

their name cache consistency mechanisms.[2] In the Lampson design, the cached result of a name lookup carries an expiration time assigned by the service. The data is guaranteed to be valid until that time, but must be discarded thereafter. Lampson does not address the question of how to choose expiration times—clearly, if expiration times are too short, cache entries will not live long enough to give a useful cache hit rate, but if the times are too long, they restrict the frequency with which name bindings can be changed. Decentralized naming, on the other hand, employs on-use cache consistency checking. Again, this technique cannot simply be tacked on to the Lampson design because objects do not necessarily know their own names.

In conclusion, although the extended Lampson design is similar in some ways to decentralized naming—both use replicated directories at the uppermost levels of the naming hierarchy and local directories at the lowest—they differ in important respects. Decentralized naming explores the ideas of multicast for fault tolerance and caching with on-use consistency for efficiency, not considered in the Lampson design.

2.4 Other Related Work

2.4.1 Domain Naming

The Domain Name service recently adopted in the DARPA Internet [30,31] is a simpler system in the same class as Grapevine and the Lampson design. The design is simplified by assuming that updates are infrequent enough to be handled manually by human administrators—the name service interface does not define any way for a client program to request the addition or deletion of a name binding. Placement and update of directory replicas are also handled manually (though some implementations may offer automated assistance). A serious drawback of these simplifications is that they put a heavy burden on system administrators, offering many opportunities for human error to disrupt the system.

2.4.2 Prefix Tables

Welch and Ousterhout [46] describe an extension of the UNIX file system to distributed operation, using *prefix tables* to locate file servers. Prefix tables are quite similar to the prefix caches discussed in this thesis and provide similar efficiency benefits. As implemented, however, they are less flexible: each prefix table is statically loaded with a set of prefixes at boot time. The referent for a prefix can change during operation, but new prefixes cannot be added to the table, nor can old ones be deleted. The authors describe the design of a mechanism for adding new prefixes dynamically, but do not describe any way of detecting when old prefixes should be removed entirely. Their scheme also appears to be vulnerable to the consistency problem discussed in Section 4.2.5 of this thesis.

The directory implementation underlying Welch and Ousterhout's prefix table mechanism is entirely different from that employed in decentralized naming. There are no regional or global directories; instead, every directory is managed by exactly one server. File servers near the root of the tree delegate authority for some of their subdirectories using *remote links,* yielding a structure similar to that of Locus. One difference from the Locus approach is that a remote link does not indicate which server implements the subdirectory in question; instead, the client must broadcast to find it.

[2] As with decentralized naming, caching is important in the Lampson design, because its basic name lookup procedure is often costly—looking up a single pathname can entail contacting several name servers, some distant from the client. Because of this cost, Lampson rates caching as "very desirable," even when his name service is not applied to file naming—and he states that a file directory system is required to be "much faster" than a service that names only hosts, mailboxes, and the like [27], making caching even more important when the system is extended to name files.

2.4.3 Early V System Work

Decentralized naming is an extension of a design described in an earlier conference paper [10]. The earlier design also distributed the responsibility for object naming among the system's object managers and used a similar name mapping protocol, but it did not provide a uniform global name space. Instead, each workstation was provided with a small, independent name server to store local aliases and the top level of the naming hierarchy. A set of conventions outside the naming system proper ensured that most workstations had similar views of the name space. Decentralized naming replaces these with the multicast name mapping mechanism and per-client name caches described in this thesis.

2.5 Chapter Summary

The ultimate global name service has not yet been constructed—existing systems leave room for improvement in both fault tolerance and efficiency. Decentralized naming attacks these problems using a new combination of techniques, including multicast name mapping and prefix caching with on-use consistency.

Chapter 3

Efficiency

It is important for a distributed naming facility to be efficient, because name mapping operations are performed frequently—every time a file is opened, for example. Small files (1 kilobyte or less) are prevalent in modern program development environments [29,21], and name mapping can easily make up a substantial fraction of the total cost of opening and reading such files.

Decentralized naming relies heavily on prefix caching for efficiency; without caching, its name mapping protocol would not be efficient enough for use in large systems. The inefficiency arises because each multicast to a regional 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 regional directories should go; as it increases, multicasts become less frequent, so larger directories can be handled satisfactorily with regional techniques. This chapter therefore focuses on evaluating the effectiveness of caching.

The primary results presented are as follows:

- The average cost of name mapping (and several other naming operations) is given in terms of the cache hit ratio and other system parameters.
- An analytical model of cache performance is presented, and is validated by comparison with measurements taken on the V naming implementation. The V measurements show a hit ratio of 99.7%, and the model predicts similar hit ratios (99.00–99.98%) in most applications of decentralized naming.
- Performance considerations are shown to limit the number of object managers that can practically participate in a regional directory to a few thousand.

To simplify the exposition, the initial sections of this chapter discuss systems configured with no global directories—systems where even the root directory is implemented using regional techniques. A later section then extends the results to configurations that include global directories.

Section 3.1 evaluates the average cost per use for several important naming operations. These costs depend on the name cache hit ratio, which is derived analytically in Section 3.2. Section 3.3 presents and discusses measurements of the actual cache hit ratio and other parameters of the V naming implementation. The cost functions derived in the first three sections include a term that varies linearly with the total number of object managers in the system; Section 3.4 shows that this property limits the size of a system with a regional directory at its root, and Section 3.5 extends the argument to establish a size limit for the regional directories in a system with a global directory at its root. The chapter closes with a summary.

3.1 Cost Per Operation

This section evaluates the cost of naming operations in terms of *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 cost metric here because naming operations generally do not take much processing time; their cost is dominated by the cost of communication. This section's 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. The root directory is assumed to be regional (and hence no directories are global).

3.1.1 Name Mapping

Determining the average cost of name mapping is a complex problem because of the large number of cases involved. There are many possible levels of "near miss" between the extreme possibilities of a hit that leads to a local directory and a miss that returns no cache information at all. It is not difficult, however, to develop a conservative cost estimate based on a simplified model of cache behavior that considers all misses together and charges the worst-case cost for each; such estimates are acceptably accurate when misses are infrequent. This section states and derives such an estimate, then goes on to illustrate how inordinately complex the estimate would become if it were extended to consider all miss cases separately.

Equation 3.1 is a conservative estimate for C_{map}, the average number of packet events required to map a name; its derivation is given below.

$$C_{\text{map}} = 4h + (r + m + 3)(1 - h) \tag{3.1}$$

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 analysis leading to Equation 3.1 is based on a simple "hit or miss" model of cache behavior. Under this model, 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 local 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}.

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 $r + m + 3$ 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 regional directory below the root. Thus, the client next retransmits its request as a multicast to all m object managers participating in the root directory ($m + 1$ packet events). Finally, the client receives a single unicast response from the object's manager (2 packet events), containing the operation result and a corrected cache entry. Summing these values, the total cost for this case is $r + m + 3$.

Combining the two cases yields Equation 3.1 above.

It is clear from Equation 3.1 that C_{map} will be close to the optimum value 4 if the miss ratio $1 - h$ is small compared to $1/(r + m + 3)$, as illustrated in Figure 3.1 below.[1] For example, C_{map} will be less than 4.16 for an installation with 50 object managers, $r = 4$, and $h = 99.7\%$.

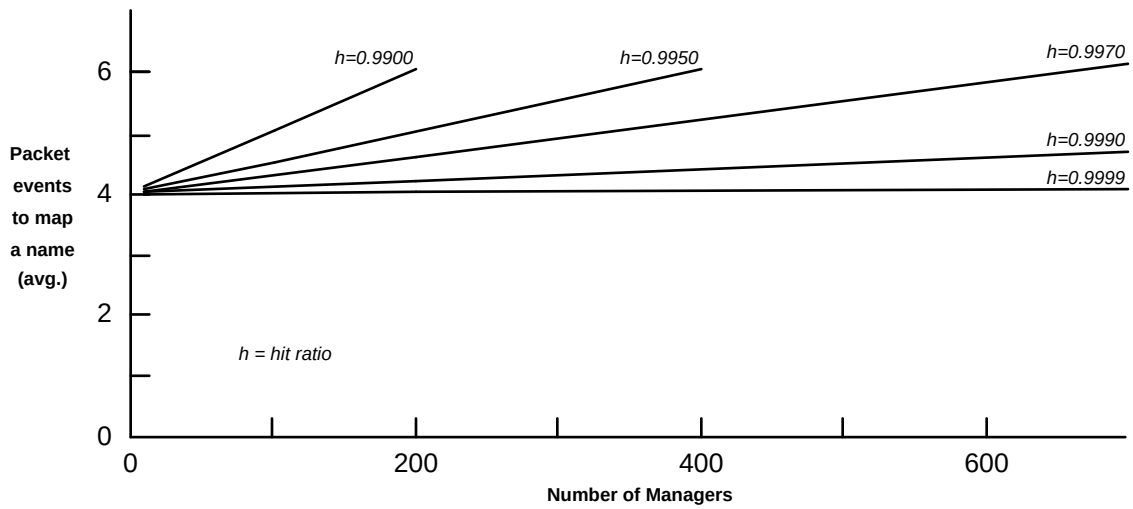

Figure 3.1: Average Cost of Mapping *vs.* Number of Managers.

It is somewhat more costly to map a generic name than a specific name. Although the cost is the same when there is a cache hit (4 packet events), when there is a cache miss *each* manager that binds the name responds to the client's multicast. Thus if g managers bind the name, the worst-case cost becomes $r + m + 2g + 1$ instead of $r + m + 3$, making the average-case cost $C_{\text{map-generic}} = 4h + (r + m + 2g + 1)(1 - h)$.

It is still more costly to map a group name (or the name of a regional directory). In this case, each manager that binds the name responds regardless of whether the cache hits or misses. In the case of a cache hit, the client multicasts to precisely the g managers that bind the name and receives g responses, for a total cost of $3g + 1$ packet events. In the case of a miss, there are g responses to the final multicast, so the worst-case cost is again $r + m + 2g + 1$ instead of $r + m + 3$, making the average-case cost $C_{\text{map-group}} = (3g + 1)h + (r + m + 2g + 1)(1 - h)$.

Details of the Cache Miss Case

The remainder of this section sketches in the details that were omitted from the "hit or miss" model of cache behavior given above. A flowchart (Figure 3.2) summarizes the possible outcomes of a cache lookup and name mapping attempt, and gives the cost of each. The cost of attempting to map an uncovered name is also given. These details are provided to illustrate how excessively complex it would be to extend Equation 3.1 to consider all cases individually.

The two cases considered in Equation 3.1 correspond to the paths (1, 3, 7) and (1, 3, 6, 9, 2, 12) in Figure 3.2. The best-case (cache hit) path traverses blocks 1, 3, and 7, for a

[1] In all cases, $C_{\text{map}} \leq 4$, because the definition of name mapping requires at least one unicast message from client to manager carrying the operation request, and one return message acknowledging the request and carrying the results.

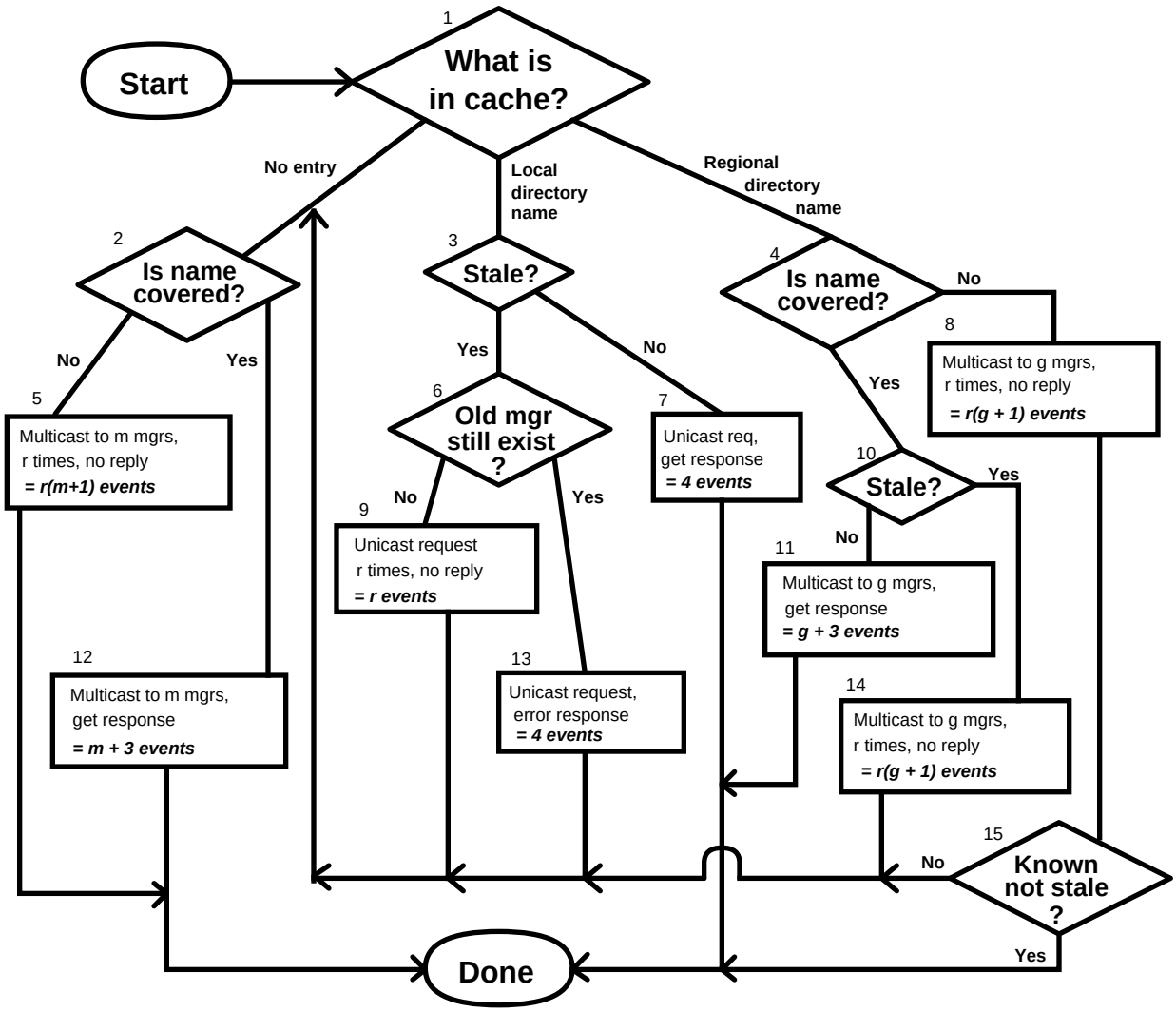

Figure 3.2: Number of Packet Events Required for Name Mapping.

total cost of four packet events, as noted above. The worst-case path for a covered name traverses (1, 3, 6, 9, 2, 12), for a total cost of $r + m + 3$, also as noted previously.

The figure also shows several cache-miss cases that are less costly than the worst case. For example, the cost of a miss is less than the worst case when the cache lookup returns stale data, but the manager referenced in the stale cache entry still exists—path (1, 3, 6, 13, 2, 12). In this case, the initial, misdirected request is transmitted only once and receives an error reply, rather than being retransmitted several times with no reply. As another example, when the initial lookup returns no data, rather than stale data, path (1, 2, 12) is followed. In the latter case, there is no initial, misdirected request; the client multicasts immediately.

Paths beginning with block 4 illustrate the cost savings that are gained through inclusion in the cache of prefixes that map to regional directories. Paths through this block are taken when the cache lookup returns a regional directory name as the longest prefix match (termed a "near miss"). If the entry is valid, the near miss reduces the cost of name lookup as compared with a total miss, because it allows the client to multicast its request to $g < m$ managers rather than all m. If the entry is stale, however, path (1, 4, 10, 14,

2, 12) is taken. This path may appear to be more costly than (1, 3, 6, 9, 2, 12), which we have been considering the worst case, but in fact, g should always equal 0 in block 14, making the costs the same. The reason g is expected to be zero is that, when a regional directory with participant group G is deleted, the group is disbanded (i.e., its membership g is reduced to 0), and the identifier G is not reused for a new group until it is very unlikely that any client still has a binding to G in its cache.

The figure also illustrates the cost of attempting to map an *uncovered* name, which is considerably higher than the worst-case cost of mapping a covered name. Possible paths through the flowchart include (1, 2, 5), (1, 3, 6, 9, 2, 5), (1, 4, 8, 15), or worst of all, (1, 4, 8, 15, 2, 5).

Block 15 requires some explanation at this point: it represents an optimization that can be applied if some regional directory names are statically defined, so that their bindings to participant group identifiers can never become stale. If the client knows that the cache entry it used cannot be stale, it can take the *yes* path out of block 15, thereby avoiding an extra multicast to all managers (block 5), and retaining the cache entry for later use rather than discarding it.

Evaluating the cost of mapping generic or group names requires some extensions to Figure 3.2. The cost of mapping a generic name is generally the same as that of a specific name, except when there is a successful multicast (blocks 11 and 12), in which case several replies are sent instead of just one. In the case of regional directory names and group names, the path beginning with block 3 is never taken, and again, several replies are sent in blocks 11 and 12.

Finally, two small differences between the above discussion and the current V naming implementation should be noted. First, the V implementation differs slightly in its handling of cache misses. Figure 3.2 assumes that whenever the cache returns data that appears to be stale, the client software retries the name mapping operation as a multicast to all managers; i.e., it ignores the cache completely. The V implementation, on the other hand, retries the operation using what remains in the cache after the apparently stale entry is removed. For example, if the cached prefixes `[storage` and `[storage/pescadero` both match the name `[storage/pescadero/user/fred`, but the send using the longer prefix fails, the retry makes use of the prefix `[storage`—referring to the figure, the stale entry is first removed, then the retry begins at block 1 rather than 2. If the `[storage` entry is not stale at this point, the lookup cost is reduced by this policy, since the entry is taken advantage of. If both entries are stale, however, the cost is increased, because the retry will also fail and a second retry will be needed to map the name. We do not yet have enough data to determine which policy gives better average performance.

Another small difference is that, under V, cache data and operation results are not both returned in a single message. Instead, client software handles a cache miss by multicasting a request for new cache data (a `QueryName` operation), then transmitting the actual name mapping operation in a separate message, sent to the address that was returned in the `QueryName` response. Thus two additional unicast messages (4 packet events) are required in the cache-miss case, increasing the approximate average cost given in Equation 3.1 to

$$C_{\text{vmap}} = 4h + (r + m + 7)(1 - h) \tag{3.2}$$

This change, of course, has little effect on C_{map} when $h \approx 1$. In Figure 3.2, the effect is to add two more unicast messages to blocks 11 and 12.

3.1.2 Name Binding

It is not difficult to evaluate the average cost of name binding, but there are several cases to consider. The primary division is between cases in which the client knows which manager

is to bind the given name and those in which it requests that the name be bound by whatever manager already covers it.

Implicit Manager Specification

Some operations use *implicit* (or *by-name*) manager specification—the manager that already covers the given name is requested to bind it. The client need not know that manager's identity when it issues such a request. An example is *object creation by name,* which accepts a name and object type as its arguments, creates a new object of the specified type, and binds the name to it. The name is required to have been unbound and covered by exactly one server; the new object is managed by that server. For instance, if `[edu/stanford/dsg/user/mann` is a local directory managed by a file server at Stanford, the operation `CreateFile([edu/stanford/dsg/user/mann/newfile)` creates a new file on that file server with the given name.

With implicit manager specification, the average cost of name binding is the same as that of name mapping. The client issues an operation request identical in format to a name mapping request, requesting that the given name be bound to an object on the manager that covers the name. When there is a cache hit, the request is unicast directly to the manager and the response unicast back, at a total cost of 4 packet events. When there is a cache miss, the worst-case cost is $r + m + 3$, as was derived in Section 3.1.1 above.[2]

Explicit Manager Specification

Other operations use *explicit* manager specification, where the client knows beforehand what manager is to bind the name and sends the name binding request directly to it. An example is "mounting" a new file server's directory tree into the global name space; both the new name and the identity of the file server must be given in the operation request.

With explicit manager specification, the cost of name binding depends on where the given name was covered before the operation. Assuming the client already knows a unicast address for the manager that is to bind the name, there are three subcases: (1) the name was already covered by the selected manager, (2) it was covered by a different manager, or (3) it was not covered.

In subcase (1), the cost is 4 packet events. The client unicasts its request to the manager; the manager in turn carries out the binding request and unicasts its reply.

In subcase (2), the cost is 4 packet events plus the cost of the protocol to transfer coverage to the new manager. Again, the request and final reply are unicast. Coverage transfer involves name mapping to find the current coverage holder, plus an extra packet to complete the three-way handshake (described in Section 4.5.3).

In subcase (3), the cost is 4 packet events plus the cost to determine that the name appears uncovered. Once more, the request and final reply are unicast. The cost of attempting to obtain coverage of a globally uncovered name is the same as that of attempting to map it (given above). Note that the operation fails after incurring this cost.

The cost of binding g objects to a generic or group name is roughly g times the cost of binding a single object to a specific name (using explicit manager specification). The name is bound to each object, one at a time, and the cost of establishing each binding is essentially the same as that of binding a specific name. There is a small difference in that if one or more of the managers needs to request permission to cover the new name (case

[2]Note that with implicit manager specification, it is impossible to bind a (previously) uncovered name, and that attempting to do so costs the same as attempting to map an uncovered name.

(2) above), it may receive replies from each of the several managers that already cover the name, rather than just one.

The cost of name unbinding is similar to that of name binding. Its evaluation is left to the reader.

3.1.3 Directory Listing

Another important naming operation is *directory listing.* Its cost can be shown to depend on the class of directory (local, regional, or global), and whether the listing includes only the bound names, or the names plus a descriptor for each bound object. For local (or global) directories, with or without attributes, the cost essentially varies linearly with the size of the directory, just as it does in most approaches to naming. The cost is similar for regional directories if the name list is available and only the names are to be listed. If the attributes are also to be listed, the cost includes a term proportional to the number of managers participating, because the client must request the attributes for each name, resulting in contacting every manager that binds at least one name in the directory. The best-efforts protocol used for directory listing in the absence of an on-line name list is even more costly. It and the local directory case are examined in more detail below.

The cost of listing a local directory is equal to the cost of mapping its name, plus enough additional packet events to return all the directory entries to the client:

$$C_{\text{sm-list}} = 4h + (r + m + 3)(1 - h) + \lambda s \tag{3.3}$$

Here, s is the size of the directory (number of entries), and λ is a constant that depends on how many entries fit into a packet. In the V implementation, directory entries are transmitted one to a packet, each in response to a separate request packet, and there is an additional pair of packets exchanged to "close" the directory after the last entry is read, so λs in Equation 3.3 is replaced by $4s + 4$.[3]

The cost of listing a regional directory with no on-line name list depends on the number of entries and the number of times the entries are replicated. Specifically, the cost in the cache-hit case is $(r + 1)(g + 1) + \lambda s'$, while the cost in the cache-miss case is $m + 1 + r(g + 1) + \lambda s'$. Here s' is the total number of entry *replicas:* if two different managers have a copy of the same entry, it is counted twice in s'. If no entries are replicated, $s' = s$. These costs arise as follows: a client lists a directory of this type by repeatedly multicasting a request to the participant group for the directory, each time appending a list of members that have already responded and therefore should not respond again, until the request has been transmitted r consecutive times with no response. The initial request is multicast to r managers if the cache hits or m if it misses, resulting in $r + 1$ or $m + 1$ packet events respectively. The directory entries are then returned in $\lambda s'$ packets, along with cache information if the initial request missed in the cache. Finally, the request and membership list (which are assumed to fit into a single packet) are retransmitted r times as a multicast to the g group members.

As in the local-directory case, the directory listing protocol actually used in V is somewhat less efficient than the idealized version described above; it requires $(r+7)(g+1)+4s-6$ packet events in the cache-hit case, and $m + (r + 6)(g + 1) + 4s - 5$ in the cache-miss case. First, $g + 1$ events are required to multicast the initial request to all participants in the directory (or $m + 1$ if the cache misses). Each manager then sends a response, resulting in $2g$ more events. Next, the client retransmits its request r times ($g + 1$ events per try), and receives no responses. It now requests and receives each directory entry in a separate

[3]One could, of course, reduce the cost of directory listing by caching directory entries in the client. Such caching is not considered in this thesis because it introduces additional cache consistency problems, and it benefits only the performance of directory listing, not of name mapping.

packet exchange, for a total of $4s$ additional events. Finally, an additional unicast packet is sent to each manager to inform it that the client is done reading directory entries from it, and these packets are acknowledged, resulting in $4g$ events. Summing these costs yields the total given above.

The cost estimates derived in the above sections (3.1.1–3.1.3) say nothing in themselves about the practical usefulness of decentralized naming, because every formula includes the cache hit ratio as a parameter. The next sections, therefore, go on to consider what hit ratios can be expected in real systems and what they imply about the practicality and scalability of decentralized naming techniques.

3.2 Cache Performance Model

This section develops 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 shows that hit ratios of well over 99% can be expected under realistic assumptions about those parameters. The parameters in question 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. In the subsections below, we first obtain 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 references at all. 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.

This section derives the following formula for $\overline{h}$, the systemwide average steady-state cache hit ratio:

$$\overline{h} = 1 - \sum_j \sum_k \frac{\beta}{\beta_{j,k} + v_k} \tag{3.4}$$

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 local subtree k is denoted as $\beta_{j,k}$.[4] 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.[5] 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 3.4 is derived as follows.

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

$$\overline{h}_{j,k} = 1 - \frac{\beta_{j,k}}{\beta_{j,k} + v_k} \tag{3.5}$$

[4] A *local subtree* is a complete subtree of the global naming hierarchy, whose root is a local directory that has a regional (or global) directory as its parent.

[5] Thus, of course, the hit ratio can vary with time.

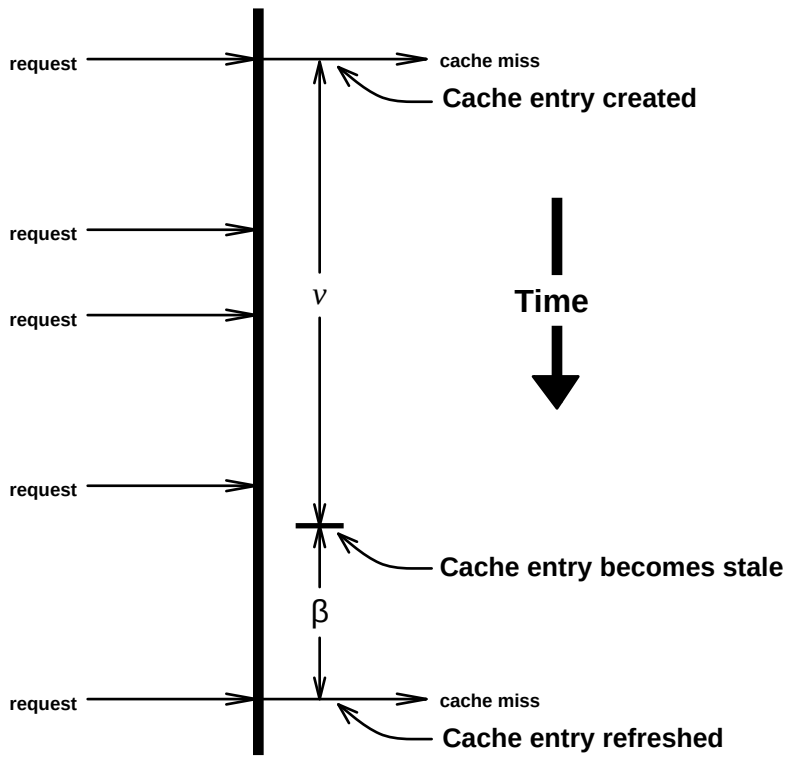

Figure 3.3: Average Intermiss Time Equals $v + \beta$.

because the average time between misses is $\beta_{j,k} + v_k$, as illustrated in Figure 3.3. Whenever a miss occurs, the client acquires a new cache entry that will be valid for a time v'. The next miss will occur 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 3.5 above.

Equation 3.4 is then derived 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

This section argues that it is reasonable to expect values of $\overline{h}$ in the range 99.00–99.98% for typical systems using decentralized naming. The argument proceeds by showing that values in this range are to be expected for individual client/subtree pairs with high traffic, and contending that such pairs should dominate the global average due to locality of reference.

The graph in Figure 3.4 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 $v_k = 5000$, $\overline{h}_{j,k} = 0.9998$.

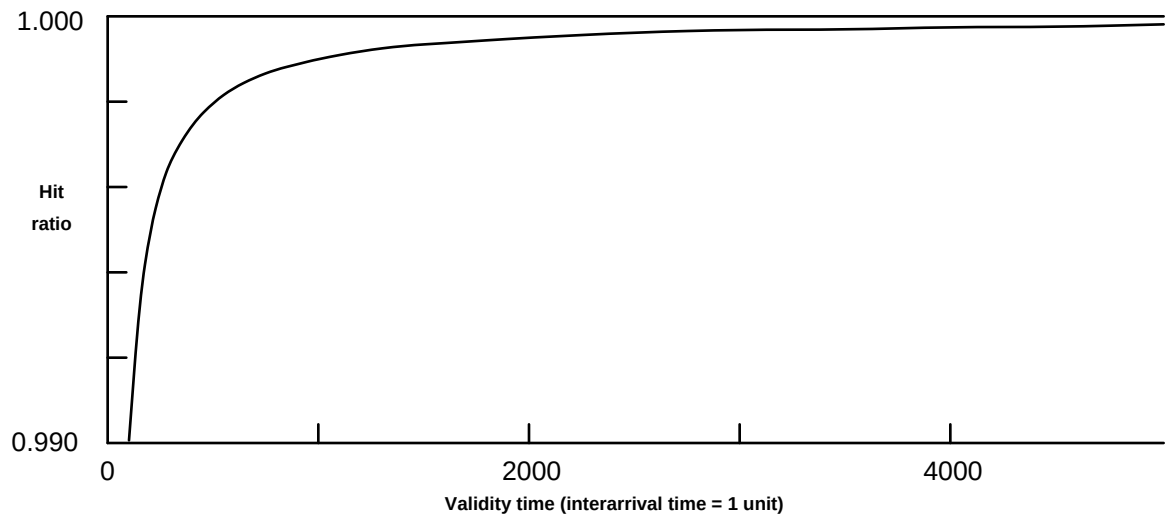

Figure 3.4: Hit Ratio *vs.* Validity Time.

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 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 local 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 will be to this small subset of the total available. 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 will be 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 that each of a server's regular clients will access it more than 1000 times between successive crashes. It is also reasonable to expect that a subtree newly assigned to a

particular server will (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 will, in general, be 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. 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 3.4 to reflect the initial misses that occur after a client cache is created can be shown to yield Equation 3.6:

$$h = 1 - \sum_j \sum_k \frac{\beta}{\beta_{j,k} + v_k} \cdot \max\left(0,\ 1 - \frac{\beta_{j,k}}{l_j}\right) \tag{3.6}$$

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 was created as an empty cache and the time it will be 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 3.6 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 3.6 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 3.6 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, $h_{j,k}$ will equal $0.999 \cdot \overline{h}_{j,k}$—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, $h_{j,k}$ will be 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 its caches a long lifetime. This technique gives each client program a separate name cache in its own address space, rather than using a single cache per client machine, to avoid the overhead of interprocess communication on each cache reference.[6] If each such cache were to start out empty, startup misses would have a severe impact, because many programs make only a few name references during their lifetimes. The V implementation avoids this problem by starting each new program with a copy of its parent program's cache, thus achieving a startup miss ratio near that of a per-machine cache, as shown by the measurements in the next section.

[6] Shared memory between separate programs is not available under V.

3.3 Measurements

This section presents some measurements taken on the V implementation of decentralized naming. Including such measurements in this thesis serves several purposes.

- To show that a real system can in fact achieve the cache hit ratios that were claimed to be typical in Section 3.2.
- To show how the cost figures of Section 3.1, given in terms of packet events, translate into CPU consumption on client and server machines.
- To give the reader a concrete idea of the elapsed time needed to perform naming operations, and of the space consumed by cache data and naming code in clients and servers.
- To show that naming operations are performed frequently enough that it is important to implement them efficiently.

The measurements were taken on the V installation at Stanford's Computer Science Department. Our installation at the time 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, the CSD 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 3.1 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 query), for a hit ratio of 99.7%. Note this measurement counts references to uncovered names (resulting in a failing multicast query) as cache misses, resulting in a conservative estimate of hit ratio.[7]

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 queries:	780 (0.20%)
Failing multicast queries:	380 (0.10%)
No query required:	385466 (99.70%)

Table 3.1: Overall Statistics.

[7] The current V implementation leaves many names uncovered because the name lists for its regional directories are always kept off line.

Table 3.2 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%.

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

Table 3.2: Statistics for Peak Half Hour.

I obtained the data in Tables 3.1 and 3.2 by instrumenting the naming routines in V's client library. With the modified library in place, each program collects statistics on its own name mapping behavior, totals them, and reports them to a system statistician process just before exiting. Each workstation runs such a statistician process. Periodically, a master statistician program multicasts a request for statistics to the workstation statisticians, which respond with their current totals, then clear them. The master statistician records the systemwide totals in a log file.

A rough computation based on the model of Section 3.2 shows reasonable agreement with these measurements. The computation assumes 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. It 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 the CSD V installation, and they are each rebooted twice a week after dumps are taken, so it is reasonable to assume v_k is equal to 3.5 days for each. Workstations are rebooted more frequently, often more than once a day, so we can take l_j to be 18 hours for each workstation. From the data in Tables 3.1 and 3.2 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 3.6 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

The measurements reported in this section 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. It is of

particular interest that, even during a peak activity period, less than 0.00361% of each server's available CPU time was consumed in discarding multicast requests for names it did not cover, because (as discussed in Section 3.4 below) the cost of such multicasts is the major obstacle limiting the size of regional directories in large systems. This measurement shows that the CSD V installation is still far from that limit.

Table 3.3 reports the results of an experiment performed to measure the CPU cost of 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—path (1, 3, 7) in the flowchart of Figure 3.2. In the *miss/covered* case, the given name missed in the cache but was covered by some object manager—path (1, 2, 12) in the flowchart. In the *miss/uncovered* case, the given name name was not covered by any object manager—path (1, 2, 5) in the flowchart.[8] 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 3.3: 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 the CSD V installation spend a very small fraction of their available CPU time in bystander processing. Assuming there are enough servers that most servers are bystanders even on successful queries, 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.

[8]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.

3.3.3 Elapsed Time

It is important to measure the elapsed time taken by naming operations, as well as CPU consumption, because the two are not directly related. On the one hand, all servers receiving a multicast request process it in parallel, resulting in some savings in elapsed time. On the other hand, the elapsed time for each operation includes the time for one or more packets to cross the network, and for some operations, it includes a timeout period during which the client is waiting for a reply that will not arrive. Examples of the latter include attempting to map an uncovered name or listing a regional directory with no on-line name list. For brevity, this section presents elapsed time measurements for name mapping only.

Table 3.4 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.

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

Table 3.4: Elapsed Time For Name Mapping.

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 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 encountered fairly rarely (0.10% of all names mapped); however, the 5-second delay can be annoying to the user who inadvertently types in such a name. In such cases the user will typically notice his mistake after a second or two of delay and interrupt execution of the program from the keyboard.

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—enough so that there has been no need to put a size limit on the cache, and it appears that no such limit will be needed even for 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, as the file server uses all available memory for disk buffers—three to eight megabytes

on our Sun- and VAX-based file servers. (The server naming library itself allocates little space at run time—at most a few kilobytes.)

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 local 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 will almost certainly 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 a regional directory at its root. For example, although the V implementation works well on the Stanford CSD network, it would be quite impractical to extend it to a nationwide or worldwide internetwork without adding a global directory level. This section takes a detailed look at the limits to growth in decentralized naming systems without global directories (*regional* systems). The next section applies these observations to systems that include global directories (*global* systems), where they set a limit on how large a directory can grow before it must be made global instead of regional.

3.4.1 Availability of Multicast

The availability of multicast is currently a technological limit on the size of network in which regional name mapping can be used, 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 a regional decentralized naming system. However, techniques for internetwork multicast are currently under investigation [9], and of course techniques for internetwork *broadcast* have long been known [4,45]. Thus, it makes sense to assume the technological limits will be overcome, and to ask what other limits are encountered as systems are increased well beyond 1000 hosts.

3.4.2 Cost Per Operation

Another limit to the growth of a regional system arises from the linear increase of name mapping cost with system size. The graph in Figure 3.1 (Section 3.1.1) illustrates the problem: if the number of managers in the system is increased while the hit ratio remains constant, the average cost of mapping a name increases linearly, with the slope of the cost function equal to the miss ratio $1 - h$. At some size, C_{map} will become unacceptably large. Increasing the hit ratio raises this limit but does not eliminate it.

In a system using replicated global name servers, on the other hand, the number of packet events required to map a name in the cache miss case is proportional to the number

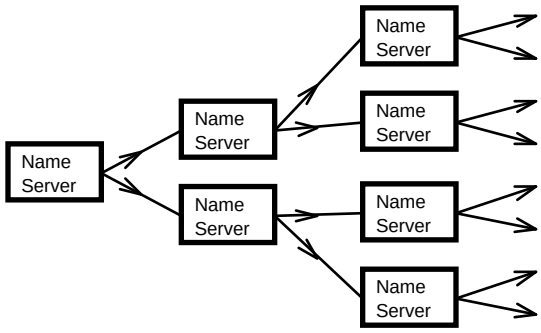

Figure 3.5: Name Mapping with Distributed Name Servers.

of servers 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 logarithm of the system size, assuming name servers at each level have about the same fanout (number of links to servers at the next level), as suggested by Figure 3.5. (The figure shows a fanout of two for clarity, but 10–100 would probably be more typical in a real system.) This growth property suggests that distributed name servers should be used for the uppermost levels of extremely large hierarchical naming systems, as is done in global decentralized naming.

3.4.3 Load Per Manager

A further difficulty in scaling up a regional 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, as explained in the following paragraphs.

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.

$$L = c \cdot a \cdot \left(1 - h + \frac{3-h}{m}\right) \tag{3.7}$$

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

Equation 3.7 is derived as follows. A cache hit costs 2 packet events at the manager; a worst-case cache miss costs a total of $m+3$ packet events at managers, since in the worst case, a multicast to all m managers is required.[9] Thus the average cost of mapping a single name is $[2h + (m+3)(1-h)]m^{-1}$ packet events per manager. Multiplying this expression by the client activity level and number of clients gives Equation 3.7.

One way of looking at Equation 3.7, illustrated in Figure 3.6, 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

[9] When only manager packet events are counted, the worst case is path (1, 3, 6, 13, 2, 12) in Figure 3.2.

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 a also remains constant; in this figure, $\kappa = 10$ clients per server and $a = 1$ request per time unit.

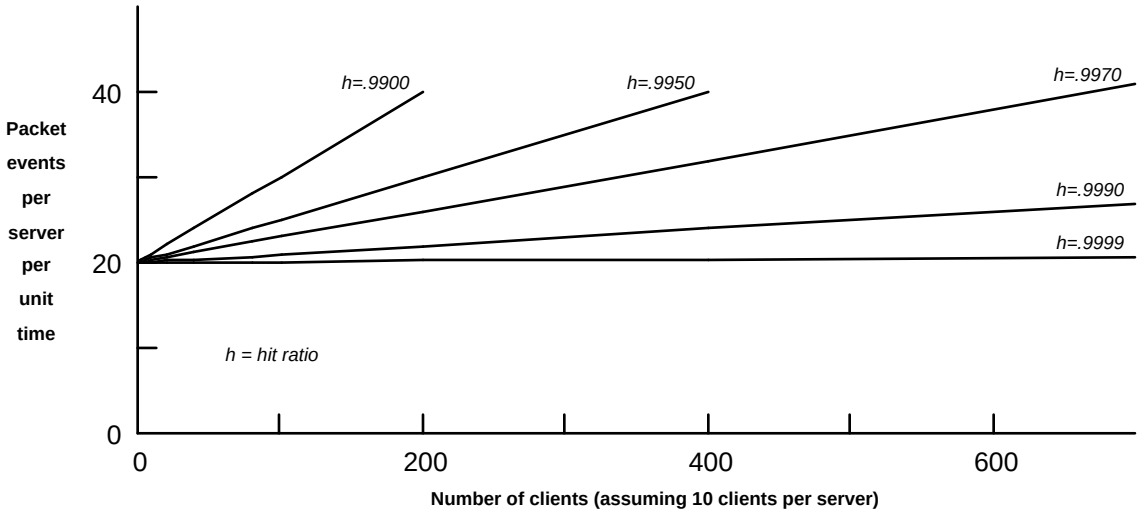

Figure 3.6: Load Per Server *vs.* System Size (With Constant κ)

As the system continues to grow, eventually the servers will become saturated by the increased naming load, and it will be 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 3.7. The graph assumes that each server has a fixed load-handling capacity L of 30 naming packet events per unit time, and that 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. For example, with a hit ratio of 0.997, the number of clients per server decreases from 15 to 9 as the total number of clients grows to 4000.

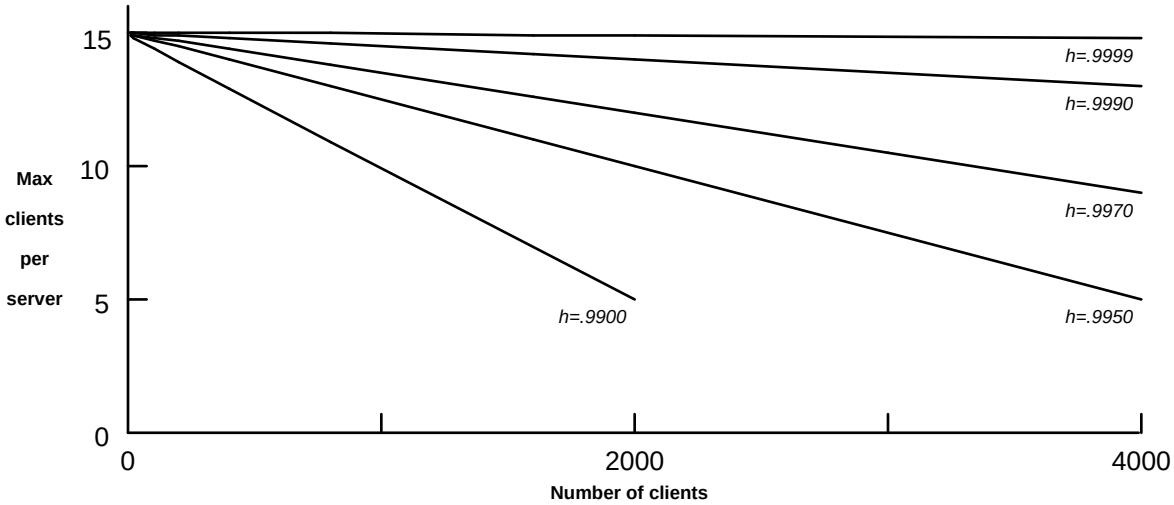

Figure 3.7: κ *vs.* System Size (Constant Naming Load Per Server).

It is important to note that Figure 3.7 actually overestimates the problem. In reality, servers do not have a fixed limit on the number of *naming* packet events they can handle; instead, there is a limit on the *total* packet events. There are many sources of packet events that do not increase with system growth, as long as the number of servers grows linearly with the number of clients—for instance, reading from an open file. Thus, for example, if 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 each server can handle 150 total packet events per unit time, κ decreases much more slowly, as shown in Figure 3.8. Again, κ is plotted on the y-axis against c on the x-axis.

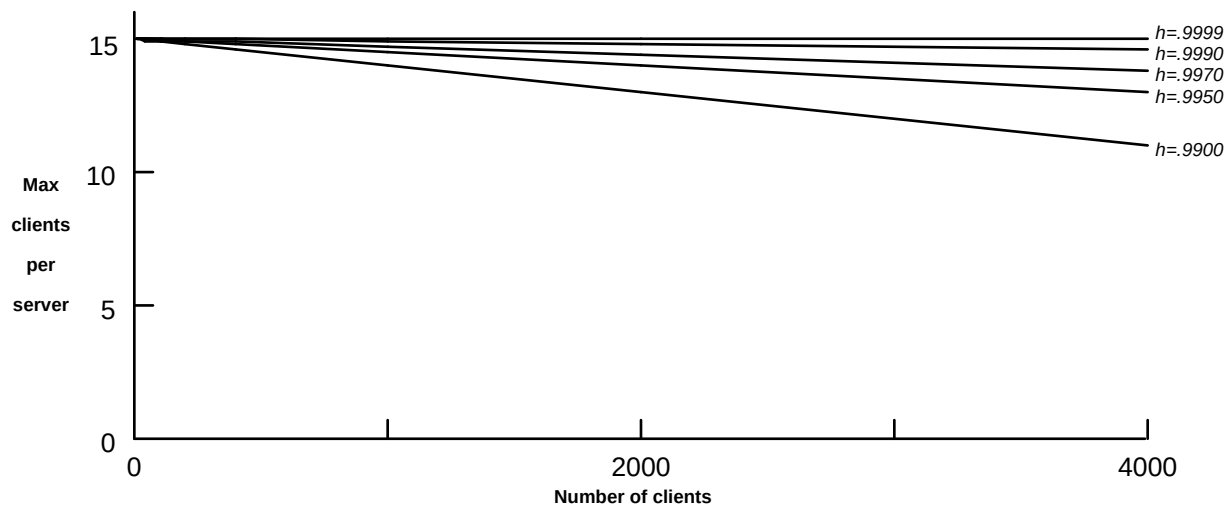

Figure 3.8: κ *vs.* System Size (Constant Total Load Per Server).

In light of the results of this and the previous section, it is clear that regional 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.

3.5 Extension to Global Systems

This section argues that the above results for regional systems can be used to establish a limit on how high in a global naming hierarchy the boundary between regional 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 regional subtrees hanging from the common global directory mechanism.[10] Each subtree can then be analyzed as an independent system—the global directory servers 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 a regional system applies almost without change to a regional 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).[11] The only difference is that a worst-case miss costs $r + d + m + 3$ instead of $r + m + 3$, where d is the number of

[10] A *regional subtree* is a complete subtree of the global naming hierarchy, whose root is a regional directory that has a global directory as its parent.

[11] 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.

packet events incurred in going through the global directory service 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 server, requiring one unicast packet from each directory's server 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 regional systems discussed earlier, any directory with more than a few thousand participants should be considered global rather than regional. The exact cutover point depends on the relative values that are placed on efficiency and resilency. Efficiency is improved by switching to global techniques in directories with fewer participants, but as shown in the next chapter, these techniques give poorer resiliency. On the other hand, resiliency is improved by using regional techniques, but as was shown above, these techniques give poorer efficiency.

3.6 Chapter Summary

This chapter has evaluated the efficiency of decentralized naming. Both the absolute performance and the scalability of regional name mapping techniques have been shown to depend critically on the cache hit ratio. The chapter has described a model of cache performance that predicts high hit ratios in typical decentralized naming installations—ranging from 99% to 99.98% and higher—and has validated the model using measurements taken on the V implementation. Using these figures, estimates of the maximum practical size for a regional directory have been derived. These estimates indicate that any directory with more than a few thousand participants should be treated as global rather than regional.

Chapter 4

Fault Tolerance

To be practically useful, a large distributed system must include some degree of fault tolerance. As a system grows to include more and more components, it becomes less and less likely that all components will be functioning at any given moment—hosts crash; networks drop packets or become partitioned. Because such faults are common, they should at worst cause temporary and localized failures near where they occur. Ideally, no matter how many faults occur, any set of hosts that remain up and interconnected should continue interoperating as usual; in particular, if hosts A and B remain connected, each should continue to be able to access all objects stored on the other *by name*.

This thesis defines two criteria for fault tolerance, called *reliability* and *resiliency*. Informally, a system is *reliable* if it meets its specification in spite of the occurrence of faults; it is *resilient* if faults do not prevent it from performing its intended service. These criteria are distinct because fault-tolerant systems typically specify two possible correct outcomes for each operation request: the operation may *succeed,* performing the requested action and returning results to the invoker, or it may *fail,* returning an error message. Any other outcome—such as returning incorrect results with no error message—violates the specification. Thus an operation's implementation is reliable if faults do not cause it to violate its specification; it is resilient if faults do not cause it to fail.[1]

This chapter evaluates the fault tolerance of decentralized naming. It considers only omission and crash faults, not Byzantine faults. Reliability is not difficult to achieve under this fault model, so the chapter concentrates on the more interesting problem of achieving resiliency. The main results presented are as follows:

- As the global level of the name service is made more resilient, the resiliency of decentralized name mapping approaches the optimum achievable in any distributed naming system; it would achieve optimum resiliency if the global level could be made perfectly resilient. Moreover, decentralized name mapping does achieve optimum resiliency for the names of objects with *nearby* managers—managers that are within range of the multicast sent out when a client's cache misses entirely.
- Decentralized binding check has a suboptimal resiliency, which varies depending on the replication level of regional name lists. It is argued that the achieved resiliency is "good enough" in a practical sense.
- Name binding cannot be made as resilient as can name mapping, no matter whether decentralized naming or another distributed technique is used. A common special case, however—creating an object and simultaneously giving it a name that was already covered by its manager—has the same resiliency properties as name mapping.

[1] This concept of failure is similar to the notion of *exception* in programming languages or *abort* in database systems. Failure is an unusual event that may be undesirable, but is not catastrophic, because the system reports it and remains in a consistent state.

The next section describes the system model used, while the following four sections discuss the most important naming operations. Section 4.2 evaluates the resiliency of decentralized name mapping, Section 4.3 that of binding check, Section 4.4 directory listing, and Section 4.5 name binding. The initial sections take the resiliency of the global directory servers as a parameter; Section 4.6 estimates the resiliency that can be expected of them. Section 4.7 summarizes the chapter.

4.1 System Model

The arguments to be presented in this chapter require a more precise model of naming and distributed systems than has been given so far. Such a model is outlined in this section.

4.1.1 Faults

For our purposes, a *distributed computer system* consists of a set of host computers interconnected by a multicast network. A *multicast network* allows any host to transmit a message and have it delivered to one or many destination hosts in a single operation. Multicasts are addressed to *host groups;* the membership of a host group g is the set of hosts that have taken the necessary (implementation-dependent) action to receive messages sent to address g.[2]

The system is assumed to be subject to *crash* and *omission* faults (only). Hosts are subject to crash faults; when a host crashes, it immediately ceases sending or receiving messages. The network is subject to omission faults; that is, dropped packets. On a multicast, omission faults can occur independently for each group member.

For simplicity, this chapter generally considers crash and omission faults together, as *access faults.* An access fault on a given host B is said to have occurred when either (1) B crashes, (2) an omission fault prevents a message addressed to B (or to a group including B) from reaching B, or (3) an omission fault occurs on a message sent by B.

Each object in the system is *managed* by some host. An object's manager stores the object's representation, implements all operations on the object, and accepts operation requests from other hosts.[3] (Objects that logically have multiple managers are viewed as consisting of multiple subobjects, each with a single manager, all bound to the same name.) The host at which an operation request originates is called the *requestor* or *client host.*

Most operations are specified to have two possible correct outcomes: the operation may *succeed,* performing a specified action and returning a success indication, or it may *fail,* returning an error indication. Any other outcome violates the specification; it is *incorrect.* In the failure case, there is no guarantee that the specified operation was *not* performed; in particular, cases where the requestor does not receive any reply across the network are treated as failures, even though the object's manager may have received the request and carried out the action.

The *resiliency* of an operation's p's implementation $I(p)$ is characterized by its *failure set* $F_{I(p)}$: the set of all minimal fault combinations that can cause $I(p)$ to fail. A combination (set) f of faults is said to be capable of causing $I(p)$ to fail if there is some set of initial conditions and parameters to p for which p's specification permits it to succeed, but the implementation $I(p)$ can fail when all the faults in f occur together. For example, if a

[2]For example, on an Ethernet or other bus network, a host joins group g by instructing its network interface to accept packets addressed to g. On an internetwork, joining a group may involve sending a message to a gateway or some other agent [9].

[3]See Jones [23] for definitions of *object* and *operation.*

file is implemented using read-any/write-all replication, with copies at hosts A and B, the failure set $F_{\texttt{Read}}$ of the `Read` operation is $\{\{A,B\}\}$, while $F_{\texttt{Write}} = \{\{A\},\{B\}\}$.[4]

Similarly, the *reliability* of an operation's implementation is characterized by its *incorrectness set:* the set of all minimal fault combinations that can cause it to violate its specification. Implementations are normally expected to be *all-reliable* against omission and crash faults; that is, their incorrectness sets should be empty.

The resiliency of two implementations can be compared by comparing their failure sets. Failure sets are partially ordered: $F \succ F'$ if and only if every element of F' is a subset of some element of F and $F \neq F'$. If $F_a \succ F_b$ for two implementations a and b, a is said to be *more resilient* than b; that is, a "greater" failure set is defined to be one that gives greater resiliency. An implementation is *all-resilient* if its failure set is empty. Thus, continuing the previous example, `Read` on the replicated database becomes more resilient if the number of copies is increased to three (at hosts A, B, and C), because its failure set becomes $\{\{A,B,C\}\}$. Also, `Read` is more resilient than `Write`. Note that failure sets under $\succ$ form a lattice, whose greatest element is the empty set $\emptyset$ (all-resiliency) and whose least element is the set $\{\emptyset\}$, if the greatest lower bound of F and F' is defined to be $\{f : f \in F \cup F' \wedge \neg(\exists f' \in F \cup F', f' \subset f)\}$. Intuitively, the greatest lower bound of two failure sets F and F' represents the best resiliency that is not greater than either F or F'. If an operation a is implemented by performing operations b and c, both of which must succeed for a to succeed, then F_a is the greatest lower bound of F_b and F_c.

4.1.2 Naming

A distributed naming system stores a *binding relation,* a relation between names and objects, and provides operations to examine and modify the relation. A name is said to be *bound* if it is related to some object; *unbound* if not. A *specific* name is related to at most one object, as opposed to *generic* or *group* names, which may be related to more than one object. In this chapter, all names are assumed to be specific. The binding relation is stored in a distributed fashion: each host holds a set of assertions about the relation. This representation is *consistent* if no contradictions arise when all the assertions are taken together; it is *complete* if the entire binding relation can be deduced from them.[5]

A *read quorum* for a name n is a minimal set of hosts q such that pooling all assertions held by hosts in q gives sufficient information to determine either that (1) n is bound to an object O_n, or that (2) n is unbound. The symbol Q_n represents the set of all read quorums for n. (For example, if any two of the three hosts a, b, c make up a read quorum for n, $Q_n = \{\{a,b\},\{b,c\},\{a,c\}\}$.) A manager M *covers* a name n if $\{M\}$ is a read quorum for n. It *exclusively* covers n if it is the only read quorum for n (and is aware that it is the only quorum).

A *write quorum* for a name n is a minimal set of hosts w for which one can change what n is related to (preserving consistency) by changing only assertions held by hosts in w. Note that every write quorum for a given name n must intersect every read quorum for n.

A characteristic feature of the decentralized approach to naming is its use of *decentralized binding storage.* Binding storage is decentralized if and only if, for any name n bound to an object O_n that is managed by host $M(O_n)$, $Q_n = \{\{M(O_n)\}\}$. That is, each object manager is by itself a read quorum for the names of its objects (and no other set of hosts

[4]These sets are written in terms of access faults, with an access fault on a given host denoted by the host's name. It is assumed here and throughout the chapter that requestors do not crash while waiting for operations to complete.

[5]It is assumed that changes to these assertions are (or can be) totally ordered in time by a system of Lamport clocks [26], so that "the set of all assertions held at time t" is well-defined.

is a read quorum). This knowledge is what makes multicast name mapping and on-use cache consistency checking work.

4.2 Name Mapping

How reliable and resilient is name mapping in a decentralized naming system? To answer that question, this section gives a specification for name mapping and a model for the decentralized name mapping protocol, then considers how well the model meets the specification in the presence of faults.

4.2.1 Specification

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 pair (n, m) to the object's manager $M(O_n)$ and returns a reply, or else fails. If n is unbound, the operation always fails.

This specification takes the view that the main purpose of name mapping is as the first step in performing other operations whose target objects are specified by name. The name mapping step locates the target object and sends its manager a request message; the manager in turn carries out the requested operation and sends back its results in a reply message.

4.2.2 Protocol

This chapter uses the following (simplified) model of the decentralized name mapping protocol.

Binding storage is assumed to be decentralized. When an object manager M receives a mapping request (n, m), it examines its local assertions about n and proceeds as follows. If it knows n to be bound to an object O_n that it manages, it replies "success." If it knows n to be unbound, it replies "failure: name unbound." Otherwise, it does not reply.

The global directory service is modeled as another operation *Glob* that is called as a subroutine by the name mapping protocol. When *Glob* is invoked by a client host H with parameters (n, m), it either causes a name mapping request with parameters (n, m) to be sent to a set of object managers S that includes every read quorum for n, or else fails. (The request is marked as having come from H, so any replies from members of S are directed to H.)

Each client host H maintains a *cache* C_H. A cache is a finite set of *entries* of the form (N, a), where N is a set of names and a is a manager or group address. Each client also holds the address b_H of a group of object managers; the members of b_H are said to be *nearby* to H.[6]

A client host H attempting to issue a name mapping request (n, m) runs the following algorithm:

1. Select an entry (N, a) from C_H, such that $n \in N$. If no such entry exists, go to step 5.
2. Send (n, m) to address a and wait until either (1) a reply arrives, or (2) a timeout period t expires.

[6] In the implementation, b_H corresponds to the use of scoped multicast to send to all object managers that are near the client—say, within a small number of hops on the network.

3. If a reply arrives, return it. Done.
4. If no reply arrives, delete (N, a) from the cache and go to step 1.
5. Send (n, m) to address b_H and wait until either a reply arrives or the timeout period t expires.
6. If a reply arrives, return it. Done.
7. If no reply arrives, invoke $Glob(n, m)$ and wait until either a reply arrives or the timeout period t expires.
8. If a reply arrives, return it. Done.
9. If no reply arrives, return failure. Done.

4.2.3 Reliability

The first question to be answered about this protocol is whether it is reliable—does it meet its specification in spite of omission and crash faults? It is straightforward to show that it does.

Theorem 4.1. Decentralized name mapping is all-reliable.

Proof: We first show that the protocol always terminates, then show that it meets its specification at all exit points.

Except for the loop in steps 1–4 of the algorithm run by H, the protocol consists entirely of straight-line code, and all steps that wait for a message from another host are protected by timeouts. The loop always terminates because C_H is of finite size, and step 4 deletes one element each time around the loop, so at worst, step 1 must jump to step 5 when C_H becomes empty.

All exits from H's algorithm either return failure due to no reply, or return a reply (which may itself read "failure"). A failure return always meets the specification. If a non-failure reply is returned, it must be correct: the protocol permits a non-failure reply to be sent to H only by $M(O_n)$, and that only after $M(O_n)$ has received (n, m), in which case the specification has been met. ■

4.2.4 Resiliency

The resiliency of this protocol is the next question of interest. This section shows that it has optimum resiliency for names bound to objects with nearby managers, and that its resiliency for other names is limited only by the resiliency of the global directory servers—that is, its resiliency is the greatest lower bound of the optimum resiliency and the resiliency of *Glob*.

Definition 4.2. An operation implementation is said to be *ABMA-resilient* if its failure set is $\{\{M(O)\}\}$, where O is the object being operated on and $M(O)$ is its manager. That is, the only fault combinations that can cause such an implementation to fail are those that include the object's manager. (*ABMA* stands for "all but manager access.")

Name mapping with parameters (n, m) is considered to be an operation on the object O_n bound to n if n is bound.

Lemma 4.3. Decentralized name mapping with arguments (n, m), n bound, is ABMA-resilient if $M(O_n)$ is nearby to the client host H; otherwise its failure set F_{nm} is the greatest lower bound of $\{\{M(O_n)\}\}$ and the failure set of *Glob*. (If n is unbound, F_{nm} is empty.)

Proof: First, suppose that H's algorithm exits at step 3, 6, or 8. These steps can return failure only if the received message indicated "failure: name unbound." But in that case, the name was in fact unbound, in which case the specification requires failure to be returned; so any faults that may have occurred during execution of the protocol were not the cause of the failure.

Otherwise, H's algorithm must exit at step 9. Case 1: Suppose n is bound and $M(O_n)$ is nearby to H, that is, $M(O_n) \in b_H$. If $M(O_n)$ receives (n, m), the protocol requires it to reply: because binding storage is decentralized, $M(O_n)$ knows locally that n is bound to O_n. Now in step 5 (n, m) was sent to a group including $M(O_n)$, so if no reply is received, either the request message was not delivered to $M(O_n)$ (omission fault), $M(O_n)$ crashed, or $M(O_n)$'s reply was not delivered (omission fault), and all these cases are access faults on $M(O_n)$. Therefore the operation fails only if there is an access fault on $M(O_n)$—it is ABMA-resilient.

Case 2: In step 7, $Glob(n, m)$ was invoked. There are then two possibilities: (i) *Glob* sent a name mapping request to a group including a read quorum for n, or (ii) *Glob* failed. In subcase (i), if n is bound, *Glob* sent the request to a group including every read quorum for n. By an argument similar to that of case 1, if no reply arrived, there must have been a manager access fault. (And as before, if n is unbound, the specification requires a failure return, so any faults that may have occurred were not the cause of the failure.) In subcase (ii), *Glob* failed. Therefore, in case 2, the only faults that can cause name mapping to fail are manager access faults or a combination of faults that causes *Glob* to fail. So the failure set in this subcase is the greatest lower bound of $\{\{M(O_n)\}\}$ and the failure set of *Glob*. ■

Lemma 4.4. ABMA-resiliency is the optimum (i.e., greatest achievable) resiliency for name mapping.

Proof: Because name mapping is specified to succeed only when it sends a message to $M(O_n)$ and receives a reply, it cannot succeed in the presense of an access fault on $M(O_n)$. Therefore a failure set equal to $\{\{M(O_n)\}\}$, i.e., ABMA-resiliency, is an upper bound on the resiliency of any implementation. And ABMA-resiliency is achievable—in particular, decentralized name mapping achieves it if the system is configured with every object manager host nearby to every client host (by Lemma 4.3). ■

Finally, the main result of this subsection follows immediately from the above lemmas.

Theorem 4.5. Decentralized name mapping with arguments (n, m), n bound, has the optimum resiliency achievable for name mapping if $M(O_n)$ is nearby to the client host H; otherwise its failure set F_{nm} is the greatest lower bound of the optimum failure set and the failure set of *Glob*. (If n is unbound, F_{nm} is empty.)

Proof: By Lemma 4.4, the phrase "ABMA-resiliency" in Lemma 4.3 can be replaced by "optimum resiliency," and the set $\{\{M(O_n)\},\}$ by "the optimum failure set." ■

4.2.5 Reusable Directory Identifiers

Although decentralized name mapping as modeled above is all-reliable, an important practical performance optimization involving reusable directory identifiers can compromise reliability if it is not implemented carefully. This subsection describes the optimization, discusses its importance, and outlines a way to maintain all-reliability when it is used.

The basic idea of the optimization is to avoid having object managers redo name processing that has already been done by the client. It operates as follows. Cache entries are extended to consist of a name prefix p (representing the set N of names that begin

with prefix p), a manager or group address a, and a directory identifier d. The directory identifier is a compact numeric identifier for the directory named by p, assigned by the manager or group a. If a client's cache lookup returns entry (P, a, d), the client sends the triple (d, n', m) to the indicated address a, where n' is the suffix remaining after p is stripped from n, in place of of sending the full name n and message m to a. Each manager that receives this request begins running its internal name lookup routine at directory d, thereby avoiding lookups in all the directories on the path from the root to d—the client has effectively done that work as part of its cache search routine.

A reliability problem arises if the identifiers a and d are restricted to be numbers chosen from a finite set. As new object managers and directories are created and old ones destroyed, eventually the system will run out of unused pairs (a, d) and must begin to assign new meanings to previously used pairs. If at the time (a, d) is assigned to a new directory named p', an old, stale entry (P, a, d) remains in the cache of some client H, it can result in incorrect name mapping. For example, say p is `[edu/stanford/dsg/user/john` and p' is `[edu/stanford/dsg/user/mary`. Then if H attempts to open the file `[edu/stanford/dsg/user/john/profile`, it will get `[edu/stanford/dsg/user/mary/profile` instead.

One way to solve this problem is to treat directory identifiers and manager (group) addresses as T-stable identifiers (in Cheriton's terminology [8]). An identifier is T-stable if it is not reused for at least T seconds after becoming invalid, for some specified value of T. In this application, each manager or group a must avoid reusing any directory identifier d it has issued for at least T_a seconds after its previous assignment becomes invalid, and the system must avoid reusing any address a for at least T_{sys} seconds after its old assignment becomes invalid. The reliability problem is then avoided if clients discard the directory identifier d from any cache entry (P, a, d) that was last successfully used (or acquired) more than $\max(T_a, T_{\text{sys}})$ seconds ago. When a client finds such an entry in its cache, it is still able to reduce network traffic by sending only to a rather than its nearby group b_H, but the optimization of sending only (d, n', m) no longer applies; the client must submit the full name n.

4.3 Binding Check

There is a serious practical problem with the name mapping operation as specified above: when it fails, it is not required to indicate whether it failed because the given name was unbound or because of a fault. The decentralized name mapping protocol does sometimes return "failure: name unbound" for unbound names, but at other times it returns "failure: no reply," in which case the client does not know for certain whether the name was bound. To focus on this problem and evaluate its difficulty, this section defines a *binding check* operation and discusses its resiliency.

4.3.1 Specification

The binding check operation accepts a name n and returns the name's binding status (*bound* or *unbound*), or else fails, returning an error message.

4.3.2 Achievable Resiliency

This section considers what resiliency is achievable in any sort of distributed naming system, not necessarily decentralized. It is shown that all-resiliency can be achieved if binding

check is optimized in isolation, but there is a tradeoff: name binding and unbinding become less resilient as binding check is made more resilient.

Let a *binding check quorum* for a name n be a minimal set of hosts whose pooled knowledge suffices to determine whether n is bound, and let BCQ_n represent the set of all such quorums for n. Note that a binding check quorum need not be a read quorum, because the hosts are not required to have enough knowledge to determine what n is bound to, only whether it is bound; however, every read quorum is also a binding check quorum. Clearly, a correct implementation of binding check can return success only if the requestor communicates (perhaps indirectly) with every member of some binding check quorum for the name n—if it is unable to do that, it cannot have gathered enough information to determine with certainty whether n is bound.

Under these definitions, one can in principle implement binding check with all-resiliency by configuring the binding check quorums appropriately. In particular, suppose that every host is made a binding check quorum for every possible name, by including the binding status of every name among every host's locally-held assertions about the binding relation. Then binding check can be performed as a local operation at any requesting host, so network failures or crashes at other hosts cannot cause it to fail. This is, however, the only implementation that achieves all-resiliency: if some host H is not a binding check quorum for some name n, and H requests a binding check on n, a sufficient number of omission faults on the network can prevent H from communicating with any binding check quorum for n, thereby causing the request to fail. In practice, of course, such an implementation is unworkable, because changing the binding status of any name would require updating the local knowledge of every host—giving name binding and unbinding disastrously poor resiliency and efficiency.

More generally, as one increases the resiliency of binding check, the achievable resiliency for name binding is reduced. This tradeoff arises because, first, if a given name n is unbound, before a correct implementation of the name binding operation can succeed when invoked on n, it must communicate (perhaps indirectly) with at least one member of every binding check quorum for n—if it missed some quorum q entirely, the pooled assertions of q's members would continue to identify n as unbound. So a combination of access faults on every member of any binding check quorum for an unbound name n must cause name binding on n to fail, implying that each binding check quorum is a member of (or a superset of a member of) the failure set NBF_n for name binding on n—i.e., that BCQ_n is an upper bound on the failure set of name binding. Therefore, as one improves the resiliency of binding check by increasing the number (or reducing the size) of binding check quorums, the upper bound on the resiliency of name binding is reduced. A similar argument holds for unbinding bound names.

Given this tradeoff, neither the resiliency of binding check nor that of binding can be the sole criterion for evaluating the "goodness" of a naming system; they must be weighed according to their relative importance in the intended application. The next section describes where decentralized naming in general, and the V implementation in particular, fall along the scale of possible resiliency choices.

4.3.3 Resiliency When Decentralized

The simple model of decentralized naming presented in this chapter imposes just one constraint on the resiliency tradeoff between binding check and binding: because decentralized binding storage requires $\{M(O_n)\}$ to be a read quorum for n if n if bound, it must also be a binding check quorum for n. The model says nothing about the composition of quorums for unbound names.

More constraints are imposed by an actual implementation using global, regional, and local directories (as discussed informally in earlier chapters), but some flexibility remains; in particular, the resiliency of binding check can be increased or decreased by adding or

deleting on-line copies of the name list in regional directories. The remainder of this section informally evaluates the achieved resiliency, then gives some justification for the choices made.

Consider the *name bound* case first. For a bound name n, the set of binding check quorums BCQ_n is almost the same as the set of read quorums Q_n. The only difference is that each name list holder in a regional directory is a binding check quorum for every name directly under the directory,[7] even though it is not a read quorum. Binding check accordingly uses a slightly modified version of the name mapping protocol; the differences are as follows:

- There is no request message m.
- The reply message "failure: name unbound" is replaced by "unbound."
- Other reply messages are replaced by "bound."
- An object manager host M (including a name list holder) replies "bound" whenever it knows that the given name n is bound, not only when it is bound to an object managed by M.
- *Glob* is replaced by *Glob'*, which relays binding check requests to every binding check quorum, not only to every read quorum.

The resiliency of this protocol is close to ABMA, but somewhat better, because for a name n that is directly under a regional directory d, failure can only be caused by a combination of access faults on $M(O_n)$ and all holders of the name list for d. To illustrate the difference, suppose `[edu/stanford/mailbox` is a regional directory of mailboxes stored on various hosts at Stanford. When a user tries to send mail to `[edu/stanford/mailbox/horace.jones`, he would like to find out promptly and reliably whether that mailbox actually exists, even if the mail cannot be delivered immediately. If there is a name list holder for the directory on line, it can report promptly whether the mailbox exists, even if the host it is stored on is down; without a name list holder, there is no way to distinguish between the cases of "host down" and "no such mailbox."

The *name unbound* case remains to be considered. Let the *bound prefix* $B(n)$ of a pathname n be the longest prefix of n that is bound. The failure set F_{bc} for binding check on an unbound name n varies depending on the implementation style—global, regional, or local—used for the directory named by $B(n)$.

1. If $B(n)$ is a global directory, F_{bc} is equal to $F_{Glob'}$, the failure set for the global directory service.
2. If $B(n)$ is a regional directory, F_{bc} is the greatest lower bound of the set of name list holders for $B(n)$ and the set $F_{Glob'}$, or just the set of name list holders if they are all nearby to the requestor. If there are no on-line name list holders for $B(n)$, binding check always fails when n is unbound.
3. If $B(n)$ is a local directory managed by M, F_{bc} is the greatest lower bound of $\{\{M\}\}$ and $F_{Glob'}$.

Verification of these results is left to the reader; the arguments are similar to those used in Section 4.2.4 above.

This level of resiliency is arguably a reasonable choice for practical implementations of decentralized naming. For file names, it is similar to that provided by other naming services. For example, in Lampson's design [27], 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

[7] A pathname n is *directly under* a directory d if deleting the last component of n leaves the name of d.

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, except in cases where the file server's root is defined in a regional directory with no on-line name list holders. In such a case, when the server is down, binding check fails even on its root directory—in a sense, the naming system cannot tell whether the server exists when it is down. This failure mode is certainly undesirable in some applications (if users are uncertain what file servers exist or what their names are, for example), but it only arises when the system administration chooses to configure a regional directory without on-line name list copies, so it can always be avoided when unacceptable.

4.4 Directory Listing

Another operation commonly provided in naming systems is *directory listing;* this section briefly (and informally) discusses its resiliency. The directory listing operation provides a complete list of the single-component names that are bound directly under a given directory. The operation fails if it is unable to provide a complete list. For simplicity, this definition does not require any information about the named objects to be returned—only their names.

The resiliency of directory listing is closely related to that of binding check, because either operation can be defined in terms of the other. On the one hand, one can implement binding check on a pathname n by listing each directory whose name is a prefix of n (proceeding from left to right), and checking whether the next component of n is in the returned listing. On the other hand, asking for a listing of directory d amounts to asking which names in the directory[8] are bound; it returns the set of all names in the directory for which binding check would return "bound," or fails if binding check would fail for any name in the directory. Thus any set of hosts q that includes a binding check quorum for every name of the form d/c is (or includes) a directory listing quorum for d—that is, the pooled information of the hosts in q is sufficient to perform the directory listing operation on q. Therefore, one would expect the failure set of a reasonable implementation of directory listing on d to be the greatest lower bound taken over the failure sets for binding check for every name of the form d/c. It is straightforward to achieve this resiliency in a decentralized naming system—the name of the directory is mapped in the usual way to locate a local manager, regional name list holder, or global server for the directory; that agent then returns the directory listing.

Under the above definition, unfortunately, directory listing cannot be usefully applied to regional directories that have no on-line name list holder. On any such directory with at least one unbound name, a reliable implementation of listing must always fail, because binding check on unbound names always fails in such directories. Even if an implementation of listing could find all the bound names, it would have no way to be certain that its list was complete—that is, that all of the omitted names were unbound—and so it could not safely return success.

To work around this problem, the V system defines and implements a different directory listing operation for regional directories without on-line name lists, called *best-efforts directory listing.* The operation is specified to return a subset of the names bound in a given directory (or fail). There is no guarantee about how many names are returned, but the implementation makes its best effort to return all names bound to currently accessible objects. Best-efforts listing is substantially weaker than ordinary directory listing, but is useful in cases where inaccessible objects are not of interest—for example, listing the hosts that are currently available for remote execution. In the V implementation of best-efforts listing, a client multicasts its listing request to a host group that includes all participants

[8] Names of the form d/c, where c is a single-component name

in the target directory (perhaps using the global directory service to relay the request to the appropriate group), and collates all the replies that come in. It then retransmits the request several times, each time including a list of hosts that have already responded and therefore should not reply again, until no further replies are received. The resulting list clearly includes the names of all accessible objects—any missing name could only have been omitted because of an access fault on the manager of the bound object.

4.5 Name Binding

The final major operation whose resiliency has not yet been discussed is name binding. This section shows that the general case of name binding cannot be made ABMA-resilient in any distributed naming system (decentralized or not). Nevertheless, an important special case—object creation by name—has the same resiliency properties as name mapping.

4.5.1 Limitations on Resiliency

Even if optimized in isolation from other operations, there is a limit to the resiliency that can be achieved in a distributed implementation of name binding: the following theorem shows that it cannot be made all-resilient, or even ABMA-resilient. Intuitively, this limitation arises because the network can partition, and when it does, there must be some way of preventing inconsistent bindings from being established in two partitions.

Theorem 4.6. No all-reliable implementation of name binding in a distributed system can be ABMA-resilient. (The system is assumed to include at least two object manager hosts, but its naming is *not* assumed to be decentralized. Its representation of the binding relation is assumed to be complete, so that every name has at least one read quorum.)

Proof: Let n be a name, and let Q_n be the set of all read quorums for n. Let U be the set of all hosts. Case 1: If $Q_n \neq \{U\}$, choose a quorum $q \in Q_n$ such that $q \neq U$, and choose a host $b \in U$ such that $b \notin q$. Then suppose that b issues a request to bind the name n to some object O that b itself manages, and that n is currently unbound, but access faults prevent b from sending a message to any host in q, even indirectly. So b's request cannot have caused the assertions held by any host in q to have changed. Now if the binding request succeeds, we have a contradiction: q is a read quorum for n, so the pooled assertions of all hosts in q suffice to tell whether n is bound, but n's binding status is claimed to have changed without change to the assertions held by any member of q. If the request fails, ABMA-resiliency is violated. Case 2: If $Q_n = \{U\}$, assume some host $f \in U$ crashes, and choose a host $b \in U$ different from f. Now suppose that b issues a request to bind the specific name n to some object O that b manages. In this case, the hosts that remain up do not have enough information to determine whether n is already bound, so the binding request cannot succeed without risk of a consistency violation. If the request fails, however, ABMA-resiliency is again violated. ■

There is also a practical limit on the resiliency of name binding, imposed by its interaction with binding check and name mapping. Carrying out a name binding request requires contacting every member of some write quorum, so increasing the resiliency of name binding entails increasing the number of write quorums (or reducing their size). But because every read quorum must intersect every write quorum to assure consistency, such a change implies an increase in the size of the read quorums (or a reduction in their number). Making read quorums larger reduces the efficiency of name mapping and binding check—dramatically if the quorums were initially small (as with decentralized naming: one manager, or one manager plus a few global directory servers). Making the quorums larger or reducing their number also reduces resiliency by increasing the number or reducing the

size of elements in the corresponding failure sets. Therefore, because name mapping is the more common operation, it seems appropriate for a naming design to maximize the resiliency and efficiency of mapping, not binding, as decentralized naming does.[9]

4.5.2 Object Creation by Name

Despite its limited resiliency in the general case, there are special cases of name binding that have more attractive resiliency properties—in particular, *object creation by name* is similar to name mapping in both its optimum resiliency and its achieved resiliency in a decentralized implementation.

Object creation by name accepts a name and object type as its arguments, creates a new object of the specified type, and binds the name to it. The new object is managed by the server that previously covered the given name. The operation fails if the name was already bound, if it was not exclusively covered by a single manager, or if the covering manager could not be accessed. Creation by name is one of the most common ways of binding names in centralized computer systems, and there seems to be no reason it should not be equally prevalent in distributed systems.[10]

ABMA-resiliency is the optimum resiliency for object creation by name: The operation certainly requires access to the new object's manager, so its resiliency can be no better than ABMA. And as with name mapping, ABMA-resiliency is achieved in a decentralized implementation if the system is configured with every object manager nearby to every client.

A decentralized implementation of object creation by name achieves the same resiliency as does decentralized name mapping; that is, it is ABMA-resilient in the absence of global-level failures. It is easy to see why: the operation can be performed using basically the same protocol as name mapping. The only difference is that the covering object manager, in place of responding "failure: name unbound" to the client's request, creates the requested object, binds the given name to it, and returns a success indication. (If the name is already bound, of course, the covering manager responds "failure: name bound.")

A similar result holds for a special case of name unbinding—object deletion by name from a local directory. Here again, contact with the object's manager is necessary and sufficient to carry out the operation.

4.5.3 Coverage Transfer

For the general case of name binding, a decentralized implementation may have to transfer coverage of the given name from one object manager to another. How that is done, and what resiliency is achieved, are sketched below.

The general case of decentralized name binding is carried out in two steps: acquiring exclusive coverage followed by locally creating the binding. Suppose a client host H is trying to bind a name n to an object O_n managed by $M(O_n)$. H sends its name binding request to $M(O_n)$, which takes responsibility for carrying out both steps. (This convention makes sense because the second step would requires communication with $M(O_n)$ in any case.) $M(O_n)$ requests exclusive coverage from whatever entity currently holds coverage,

[9] Another alternative, not considered here, is to sacrifice all-reliability in favor of greater write resiliency. For example, the *available copies* replication technique [22] allows a write operation to return success after modifying all copies that can be contacted. Holders of copies that fall out of touch with the rest eventually notice the trouble and discard their data; in the meantime, however, they can return out-of-date results to read requests.

[10] In UNIX, for example, the `creat()` system call is a creation by name operation—it takes a file name as its argument and creates a file by that name, stored on the same disk partition as other files in the same directory.

locating it using basically the same protocol as for binding check, and receives a reply stating exactly what coverage was granted. If n is currently bound, the coverage request is refused. Otherwise, there are three cases: (1) If n is currently covered by the global directory service, its first component n_1 is added to the (possibly replicated) global directory, and $M(O_n)$ is given coverage of all names of the form d/n_1. (2) Similarly, if n is currently covered by the name list for a regional directory d, its first component n_1 is added to the name list, and $M(O_n)$ is given coverage of all names of the form d/n_1. (3) If n is currently covered in a local directory d, the local directory must be converted to regional before the request can succeed; if its manager is not willing to allow the conversion, it refuses the request.

Some care must be taken to transfer and maintain coverage reliably, to avoid having some names covered inconsistently or not covered at all. For example, each manager must record its coverage in stable storage, so that if the manager crashes, its coverage is not lost when it comes back up. One must also take care that only one instance of a given manager comes up and tries to use the recorded coverage.[11] Finally, one must take care that coverage transfer is performed atomically—that coverage is not lost or duplicated if network failures occur while a transfer is being carried out. The familiar three-way handshake suffices for this purpose: If host A is trying to obtain coverage from B, it first sends a request to B. If B decides to grant the request, it records that fact in stable storage and sends a "success" response to A. A then acknowledges the response, allowing B to delete its record of the transfer. If faults interrupt the handshake at any point, A and B retain enough information to abort the transfer or complete it later.

The resiliency achieved by coverage transfer is as follows. If a client host H is trying to bind a name n to an object O_n managed by $M(O_n)$, the operation fails if there is an access fault on $M(O_n)$ or a global directory service failure that prevents H from contacting it, or (in case 1) a write failure in the global directory service, (in case 2) an access fault on any copy of the required name list (assuming read-any/write-all replication), or (in case 3) access to the manager of the local directory d.

The general case of name unbinding is quite similar to that of name binding. In this case, coverage transfer may be necessary to maintain the convention that unbound names in a regional directory are covered by the name list. For example, suppose manager A binds the name `[edu/stanford/dsg/user/jones` to a local directory, and the parent directory `user` is regional. If A is then asked to delete the local directory, it must also remove `jones` from the name list for `user`. The reliability and resiliency properties of releasing coverage in this way are similar to those of acquiring coverage.

4.6 Replicating Global Directories

The global directory service represents a possible point of failure for most decentralized naming operations, so it is important to make it resilient. One way of doing so is to replicate the global directories. This section briefly examines the resiliency impact of such replication on the naming system as a whole.

Replicating global directories improves the resiliency of most naming operations. In particular, we have seen that the failure set for (non-nearby) name mapping is the greatest lower bound of (1) a manager access fault and (2) the failure set for a read on the global directory service. Assuming that the global directory service is structured so that any single copy of a given directory constitutes a read quorum, increasing the number of copies improves the resiliency of reads on that directory. For example, suppose that client host H

[11] For this reason, most object managers in the V naming implementation re-request coverage for each of their names during their initialization phase, as a "sanity check." If there is no duplication of coverage, no response is received.

is trying to map the name `[edu/stanford/dsg/V/source/lib/naming/cache.c`, that the directory types are as shown in Table 4.1 below, and that the named object's manager is not nearby to H. By inspection, the failure set for this name is $\{\{DSG_2\},\{B,D\},\{A,B,C\}\}$. Increasing the number of copies of either `[` or `edu` would improve the mapping resiliency for this name.

`[`	Global, replicas at servers A, B, C
`edu`	Global, replicas at servers B, D
`stanford`	Regional, participants throughout Stanford
`dsg`	Regional, participants in Computer Science building
`V`	Regional, with DSG file servers 1 and 2 participating
`lib` and below	Local, at DSG file server 2

Table 4.1: Directory Types Along a Sample Pathname

The impact of replication on most other naming operations is similar to its effect on mapping, because most operations can require reading global directories, but do not need to modify them.

There is one negative effect of replication: increasing the replication level of a global directory *reduces* the resiliency (and efficiency) of name binding in that directory. That is, in the above example, adding a new directory `[edu/berkeley` would become more costly and less likely to succeed if the number of copies of `edu` were increased. This difficulty arises because name binding in a global directory requires access to a write quorum for the replicated information, and since any single directory replica constitutes a read quorum, a write quorum must include every replica—potentially a large number.[12]

Updates to global directories are expected to be infrequent, however, so it may be acceptable to improve their resiliency by performing them non-atomically. That is, any server holding a copy of the directory will accept an update request, carry it out locally, and return success, then take responsibility for propagating the update to the other copies. While an update is propagating, clients may see either the old or new state, nondeterministically. If no permanent faults occur, every update eventually reaches all directory copies. Conflicting updates are possible, but occur rarely and are eventually detected. Protocols of this nature have been developed and used with some measure of success in Grapevine, Clearinghouse, and the Lampson naming design.

4.7 Chapter Summary

This chapter has evaluated the fault tolerance of decentralized naming, showing in particular that decentralized name mapping approaches ABMA-resiliency (which is the optimum) as the global directory mechanism is made more resilient, and achieves ABMA-resiliency for nearby objects.

Binding check and directory listing can in principle be made arbitrarily resilient, but it is costly to do so. Decentralized naming makes each slightly more resilient than name mapping.

Name binding cannot in general be made ABMA-resilient, but decentralized name binding has the same resiliency as name mapping in a common case (object creation by name), and provides a reasonable level of resiliency in the general case.

[12] Even using available-copies replication, one must enforce a lower bound on the number of copies written (typically half) to prevent multiple conflicting updates from being accepted during periods when the network is partitioned.

The resiliency of name mapping (and most other operations) can be improved by keeping more replicas of each global directory; however, doing so makes name binding in those directories less efficient and (unless non-atomic updates are allowed) less resilient.

Chapter 5

Security

This chapter discusses how to adapt decentralized naming to function in an environment where the hosts do not all trust one another. The major question to be answered is, when a client host multicasts a request for naming information, how does it know which replies to believe? I assume that the system has some well-defined security policy that specifies which managers are authorized to respond to queries about any given name. A response from an unauthorized manager is termed a *counterfeit*. A *counterfeit-secure* naming system is one that includes a reliable mechanism for preventing counterfeit responses from being accepted as valid. The *counterfeit problem* is the problem of providing such a mechanism.

The primary results of this chapter are (1) a solution to the counterfeit problem in decentralized naming systems, and (2) an evaluation of the cost of this solution, in terms of its impact on the efficiency and fault tolerance of naming. I also argue that no solution with significantly lower cost is likely to exist.

The next section explains how this chapter is related to existing work in security, while the following section gives the security model to be used. Sections 5.3 presents a technique for detecting and rejecting counterfeits. Its cost is evaluated in Section 5.4, and Section 5.5 considers whether better solutions are possible. Section 5.6 discusses a few additional security issues that arise in decentralized naming systems, and Section 5.7 summarizes the chapter.

5.1 Mandatory and Discretionary Security

Counterfeit rejection is a problem in discretionary security, not mandatory security. In this section I briefly discuss how mandatory security can be provided in a distributed system, then discuss the relationship between counterfeit rejection and other problems in discretionary security.

Mandatory security models typically define a lattice of security levels, assign a security level to each process or information container in the system under consideration, and require the system to prevent information flow from A to B unless either $level(A) \leq level(B)$, or A is a *trusted subject*—a process that can be relied upon to "sanitize" the information it passes to B by eliminating material that B is not cleared to possess [1,28]. For example, a process at the *confidential* level must not be allowed to read a *secret* file, or (if it is not a trusted subject) to write an *unclassified* file.[1]

It is certainly possible to enforce mandatory security in a distributed system. A simple (but draconian) technique is to require that all hosts in the system operate exclusively at

[1]These examples correspond respectively to the *simple security property* and **-property* of Bell and LaPadula [1].

a single security level. It is also possible to enforce security in a multi-level system by providing each host with a security kernel that regulates all access to the network. The kernels can prevent impermissible information flow by tagging each outgoing message with its security level and refusing to deliver an incoming message to a process at a lower level than the message. (In fact, in a multi-level system, it is *necessary* to regulate network access to enforce mandatory security. If a process P below the highest security level has unrestricted read access to the network, it is impossible to ensure that it does not read information it is not cleared for: even if the kernels force all information above the lowest level to be encrypted, a malicious process P' could transmit information to P using any of a number of covert channels; for example, modulating the length of its messages.)

In view of these facts, the remainder of this chapter discusses only the counterfeit problem, an aspect of discretionary security. For simplicity, the discussion is phrased in terms of a single-level system in which all processes have unrestricted access to the network, with each client process responsible for counterfeit checking on responses to its own requests. However, the results can be applied directly to a multi-level system in which the security kernel does not implement discretionary security. It would also be feasible to implement the counterfeit rejection mechanism within a security kernel.

Discretionary security mechanisms give their users the ability to place further restrictions on the dissemination of information they possess, beyond those imposed by mandatory security. For example, a user may want to keep certain of his files private, even from other users with the same security clearance. Discretionary security models commonly formalize the notion of "user" under the term *principal* [38]. Each process in the system is associated with a principal, corresponding to the person or organization that takes responsibility for its actions and from whom it derives its authorization. An *access matrix* records the system's detailed security policy; its rows corresponds to objects, its columns to principals, and each entry gives the list of operations that the corresponding principal is authorized to perform on the corresponding object.

One important problem in implementing discretionary security is how to do *authentication*—how to determine whether a client process is entitled to be treated as the principal it claims to be. Centralized operating systems typically provide a simple user login procedure for authentication. In distributed systems, authentication is somewhat more complex, but known cryptographic protocols can be used to authenticate clients to servers across a network [43].

An equally important (but less well studied) problem in discretionary security is *authenticating the system to the user;* the counterfeit problem considered in this chapter is one aspect of it. When a user logs into a single-machine operating system, he needs some assurance that he is communicating with the real system, not an imposter that might violate the security policy; i.e., the system must authenticate itself to him. A more complex version of the same problem arises in large distributed systems: when a client requests an operation on some object, it needs some assurance that it is communicating with the object's true manager, not an imposter. Because large systems include object managers that represent many different, mutually distrustful principals (different companies, departments, etc.), one must assume that the legitimate manager of one object may act as an imposter if queried about other objects, falsely claiming to be their manager as well. When clients use decentralized name mapping to find object managers, authenticating the managers that respond requires a solution to the counterfeit problem. For example, if DEC employee Smith attempts to open and write into a private file `[com/dec/wrl/user/smith/secrets` on a DEC file server, but another file server operated by IBM sends out a counterfeit response to its the name mapping request and captures the data, Smith's discretionary security has been breached.

The next section defines the counterfeit problem more precisely, while subsequent sections present and evaluate a solution.

5.2 Counterfeit Security Model

A *security policy* divides possible events within a system into two classes, *allowed* and *disallowed.* The occurrence of a disallowed event is termed a *security violation.* A *security model* is a formal or semi-formal framework for describing particular security policies. This section presents a security model tailored to solving the counterfeit problem in decentralized hierarchical naming systems. The policies described by the model define what a counterfeit name response is, and require that whenever such a response is generated, any client that receives has enough information to detect that it is counterfeit and reject it.

5.2.1 Definition

This section begins by defining a general model GEN suitable for any naming system, then particularizes the model to decentralized hierarchical naming.

For each name n in the global name space, each principal P is *authorized* to make zero or more assertions about n's binding status. A *naming authorization function* describes what assertions are permitted; it is a time-varying function that gives, for each principal/name pair (P, n), the set of assertions P is authorized to make about n. The detailed security policy of each system modeled by GEN defines what statements are considered to be "assertions about n" for each n. *Making* an assertion means presenting it (as the truth) in a message to some other principal; an unauthorized assertion made in this way is said to be *counterfeit.* Authorization functions have the following *closure* property: if an assertion a is implied by other assertions P is authorized to make, then P is authorized to make the assertion a as well. For example, if P is authorized to assert that n is bound to a directory, it is authorized to assert that n is bound.

The security policies modeled by GEN disallow clients from accepting counterfeit assertions.[2] The policies do not disallow the *sending* of counterfeit assertions, because under our assumption that all processes have unrestricted network access, there is no way to construct a mechanism to enforce policies that make such restrictions. A *counterfeit-secure* system implementation is one that includes a reliable mechanism in clients for rejecting counterfeits; informally, the *counterfeit problem* is the problem of providing such a mechanism in a way that does not prevent the system from performing its function.[3]

The authorization function itself is stored by principals called *security agents,* one of which is designated the *security chief.* A GEN security policy allows the chief to state the value of the authorization function for any arguments, or to *delegate* portions of its authority to other security agents, allowing them to state the function's value for certain sets of arguments designated by the chief. Security agents are included in the model to reflect the fact that the detailed security policy of a real system can change, and that such changes must be communicated from the principals that make them to the principals that are affected by them. Agents are simply principals that have authority to change the authorization function; the model does not dictate their implementation, which can itself be distributed.

This completes the definition of GEN. We turn now to HIER, a more specific model within the class GEN, tailored specifically for decentralized hierarchical naming.

The model HIER begins by assuming the global name space is hierarchical; that is, it is structured as a rooted tree, and global names are pathnames—they describe a path through the tree beginning at the root. Each non-leaf node of the global tree is a directory, and

[2] A client receiving an assertion in a message may either *accept* it (i.e., add it to its store of knowledge, act on it as the truth, etc.) or *reject* it (ignore it, consider it possibly false).

[3] In particular, it is not acceptable to implement counterfeit security by rejecting every incoming message!

a directory listing operation is provided that enumerates the branches extending outward from a directory.

The naming authorization function grants or denies permission to make assertions of the following forms about each name n:

1. An assertion that n is bound, optionally stating what it is bound to.
2. An assertion that n is unbound.
3. An assertion that n is bound to a directory, optionally giving a (partial or full) directory listing.
4. Any assertion restricting what managers may hold a binding for n. (For example, an assertion that some prefix of n is bound to a directory with a specified participant group address.)

For simplicity, HIER restricts the range of the authorization function to three values: *strongly authorized, weakly authorized,* or *unauthorized.*

A principal P that is *strongly* authorized for a name n is authorized to make any of the four types of assertion about n listed above, including giving a full listing if n is bound to a directory. Further, P is strongly authorized for every name with n as prefix, and is at least weakly authorized for every prefix of n.

A principal P that is *weakly* (and not strongly) authorized for a name n has the following more restricted permissions. First, P is authorized to state that n is bound to a directory, but not authorized to state that it is not bound (or is not bound to a directory). Next, P is authorized to give a partial listing of the directory bound to n, restricted as follows:

- P may state that a name component `c` is in the directory, if and only if it is also authorized for n/`c` (either strongly or weakly).
- P may state that a name component `c` is *not* in the directory, if and only if P is also strongly authorized for n/`c`.

Finally, P is weakly authorized for every prefix of n.

A principal P that is *unauthorized* for a name n is not permitted to make any assertions about it.

Unless otherwise noted, HIER is used as the security model throughout the rest of this chapter.

5.2.2 Why This Model?

HIER is a fairly simple model, but is flexible enough for our purposes. It allows the security agents to specify which principals are authorized to respond to which names, by dividing the name space into subtrees, each of which is strongly authorized to a different set of principals. For example, a principal P_1 could be strongly authorized for names with the prefix `[a/b` but unauthorized for `[a/c`, while another principal P_2 is strongly authorized for `[a/c` but unauthorized for `[a/b`. As a result of this assignment, both principals acquire weak authorization for `[` and `[a`, but as a practical matter, weak authorization does not grant enough power to allow them to do any harm.

It might seem attractive to simplify HIER by eliminating weak authorization; such a model is, unfortunately, inadequate. Let SIMP be a model that restricts the range of the naming authorization function to two values: *authorized* or *unauthorized.* A principal that is authorized for a name n has permission to make any of the assertions about n enumerated above. A principal that is unauthorized for n does not have permission to make any assertions about n. It is shown below that this model is inadequate, because

it would require each principal that is authorized for any name to be authorized for all names.

The reader who is not interested in the proof that SIMP is inadequate may wish to skip to the next section at this point.

Lemma 5.1. Under the SIMP model, authorization for a prefix of any given name implies authorization for the entire name.

Proof: First, note that in a hierarchical naming system, asserting that a name n is not bound (or is not bound to a directory) also implies that no name of the form n/m is bound. Now suppose that principal P is authorized for a particular name n but not for n/m. So P is authorized to assert that n is unbound, and because this statement implies that n/m is unbound, by the closure property of authorization functions P is also authorized to assert that n/m is unbound. But by assumption, P is unauthorized for n/m, so it is not authorized to assert that n/m is unbound. Contradiction. Therefore if P is authorized for n, it must also be authorized for all names of the form n/m. ■

Lemma 5.2. Under the SIMP model, authorization for a name implies authorization for every prefix of that name.

Proof: First, note that in a hierarchical naming system, asserting that a name of the form n/m is bound also implies that n is bound (to a directory). Now suppose that principal P is authorized for a particular name n/m, but not for n. So P is authorized to assert that n/m is bound, and because this statement implies that n is bound, by the closure property of authorization functions P is also authorized to assert that n is bound. But by assumption, P is unauthorized for n, so it is not authorized to assert that n is bound. Contradiction. Therefore if P is authorized for any name of the form n/m, it must also be authorized for n. ■

Theorem 5.3. Under the SIMP model, authorization for any name implies authorization for every name.

Proof: In the above lemmas, substitute the root of the naming hierarchy for n. Now Lemma 5.2 implies that every principal that is authorized for any absolute name is authorized for the root, while Lemma 5.1 implies that every principal that is authorized for the root is authorized for every absolute name. Therefore a principal that is authorized for at least one name is authorized for every name. ■

It is possible to work around this problem with SIMP, by statically defining certain directories and making them well-known to all clients. Specifically, suppose that a given directory is statically bound to its name n, and that this binding and the directory's contents (list of immediate descendants) are known to all clients. In that case it is sensible to weaken the closure requirement slightly, allowing a principal P to be authorized for assertions that imply things about n even when P is not authorized for n itself, because every client has independent knowledge of n's binding status against which it can check such assertions. A directory that is made static in this way can have descendants that are authorized to disjoint sets of principals. For example, the root directory `[` of a company-wide system might be statically defined to contain only the four entries `management`, `marketing`, `production`, and `development`, with the list of principals authorized for each prefix well-known to all clients. A file server S belonging to the marketing department could then be authorized for the `[marketing` prefix without requiring authorization for `[`, `[management`, etc.; yet S would have no difficulty in mapping the names it binds (e.g., `[marketing/projections`), even when no other servers are up. That is, $\{S\}$ appears to be a read quorum for `[marketing/projections`.[4]

[4]Strictly speaking, $\{S\}$ is not a read quorum, because some of the client's local knowledge is taken into account in mapping the name. The read quorums are actually sets of the form $\{c, S\}$, where c is a client.

This solution is simple, and may be adequate for some applications, but is rather inflexible. It amounts to giving every client a complete copy of the naming authorization function, which is practical only if the function has a compact representation (few static directories) and does not change while the system is running. Therefore, the remainder of this chapter uses the more flexible model HIER, for which Theorem 5.3 does not hold and which therefore does not force any directories to be statically defined.[5]

5.3 Capabilities

This section describes a solution to the counterfeit problem using *capabilities.* 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 $a(K)$. Under the GEN model, $s(K)$ would be a security agent. Whenever $p(K)$ is claiming the right to perform action $a(K)$, it simply presents the capability, plus its own signature to verify that it is authorized to use the capability.

When capabilities are provided, a client does not have to maintain any knowledge of the global security policy. Instead, it simply rejects any assertion that is not accompanied by a capability to validate it.

To use capabilities, all participating principals must agree on how to identify principals and verify their signatures, which principals are authorized to sign capabilities, and the language in which capabilities are written. These three points are discussed in the following three subsections.

5.3.1 Principal Identifiers and Signatures

When considering a solution to the counterfeit naming problem that involves rejecting some messages based on which principal sent them, one must take care to avoid circular reasoning, because principals themselves are known by names. A principal that claims its name is P is itself making an assertion that could be counterfeit. Thus it might seem that a counterfeit-secure principal naming service is required before one can implement a counterfeit-secure decentralized naming service, thereby requiring a centralized naming service (such as a key distribution center) at the lowest level to avoid an infinite regress of decentralized naming services.

Fortunately, however, there is a way around this problem. Suppose we have a public-key encryption system [17] that also provides digital signatures (for example, RSA [35]). We can then *identify* each principal by its public key. That is, a principal's public key is considered to be the lowest-level name for that principal, its *principal identifier.*[6] No trusted name service or key distribution center is needed to authenticate the sender of a digitally signed message as being a particular principal *referred to by its identifier,* because, given only the signed message and the public key, one can verify with high probability that only the holder of the corresponding private key could have sent that message. (And conveniently, the principal identifier is also a key that can be used to send private messages or conversation keys to the principal it identifies.)

[5] Arguments similar to those in Lemmas 5.1 and 5.2 do apply to this model, but rather than revealing problems with it, they simply establish its inheritance properties (for prefixes and suffixes of names authorized to a given principal) as theorems.

[6] Chaum terms a public key used in this way a *digital pseudonym* [6]. He advocates this approach as a way of allowing people to provide credentials to computer systems without giving up their privacy; it allows a person to store his credentials under one or many pseudonyms that have no visible connection with one another or with his real name.

In more detail, the approach works as follows. Each principal randomly chooses (or is assigned) a public key that defines a unique encryption function E_p, and a matching secret key that defines a decryption function D_p.[7] For convenience, assume that $E_p(D_p(x)) = x$ as well as $D_p(E_p(x)) = x$ for all messages x, and that the cipher is equally strong when the roles of the encryption and decryption functions are exchanged in this way. Now a message x for principal E_p is sent privately by first encrypting it with E_p, while a message from principal E_p is digitally signed by encrypting it with D_p. We then *define* the principal identifier E_p to be bound to that entity (or set of entities) that possess D_p.

As a sidelight, this mechanism leads to a simple, operational definition of *principal:* any entity that can generate a key pair and hold the decryption key secret can act as a principal, and by extension, so can any set of entities that can distribute a decryption key among themselves and prevent it from leaking further. For example, a person with a pocket calculator can be a principal, and so can a single computer system, or a single process within a computer system (assuming in the latter case that the operating system can be trusted not to steal the process's secret key). A group of people or processes can act as a single principal if they have some sufficiently secure way to distribute the initial secret D_p among themselves.

5.3.2 Authority to Sign Capabilities

The simplest way of regulating the authority to issue capabilities is to view the system security chief as a single principal with a well-known principal identifier E_{chief}, and agree that any capability signed by the chief is valid. Under this approach, a client can begin secure operation knowing only the capability language, the cryptosystem, and one public key, E_{chief}. It is, of course, undesirable for D_{chief} to be widely known, because it is in effect a master key to the entire naming system.

One can avoid the need to have a process on line that knows D_{chief} by introducing *delegation.* A delegation capability K_d, issued to a principal $p(K_d)$, states that $p(K_d)$ is authorized to sign capabilities for the rights $a(K_d)$, or any subset of those rights. Capabilities for such rights, signed by $p(K_d)$, are then accepted if accompanied by the delegation capability K_d. Given this mechanism, the security chief P_0 can delegate subsets of its rights to security agents P_1, P_2, etc., by creating delegation capabilities for them. Thereafter, P_0 need not be available on line until a delegation capability expires, or some authority is needed that P_0 has not delegated.

This mechanism minimizes the amount of knowledge a client needs to begin secure operation, but may not fit the higher-level policies of some organizations. For example, an organization with a committee at the highest level might find it more appropriate to require the signatures of several principals on top-level capabilities instead of appointing a single chief. This and other extensions to the capability mechanism can be implemented as part of the initial agreement on what principals are authorized to issue capabilities, or as part of the capability language.

5.3.3 Capability Language

A simple capability language is sufficient to implement the naming security model HIER. The $a(K)$ portion of each capability includes a single name, for which the capability grants strong authorization. $p(K)$ is the principal identifier of the principal to which the capability grants rights. The $t(K)$ portion is given in some agreed-upon time units—say, milliseconds

[7] Random choice should give an acceptably high probability of uniqueness, because principal P choosing the same key pair as principal Q amounts to P guessing how to decrypt the messages sent by Q, and a strong system must make the latter event extremely unlikely.

since the beginning of 1970, GMT. The triple $(p(K), a(K), t(K))$ is signed by $s(K)$, the principal granting the capability, and $s(K)$'s principal identifier is appended to the result to yield the complete capability.

Note that, in accordance with the definitions of strong and weak authorization, a capability for name n implies strong authorization for any name of which n is a prefix, and (at least) weak authorization for any prefix of n. There is no real need for capabilities that grant weak authorization directly, because as noted above, managers are generally granted weak authorization for a name n only as a byproduct of having been granted strong authorization for some name n/m of which n is a prefix. A manager that needs to prove it has weak authorization for n can use its capability for n/m to do so.

The action field $a(K)$ of a delegation capability includes a *delegation bit,* indicating that $p(K)$ is permitted to delegate the authority granted by K to other principals. Delegation works as follows: an unexpired capability L with $s(L) = p(K) \neq E_{\text{chief}}$ is accepted as valid if accompanied by a valid capability K, where $a(K)$ has the delegation bit set and $a(L)$ is a right implied by $a(K)$. That is, $a(K)$ may grant authorization for the same name as $a(L)$, or some prefix, and $a(L)$ itself may or may not have the delegation bit set.

5.3.4 Application to Decentralized Naming

This capability scheme is powerful enough to provide reliable counterfeit security in a decentralized naming system. In an installation that uses it, every naming response is signed with the principal identifier of the responding object manager, and clients reject any response that does not include valid capabilities sufficient to establish the signer's right to make the naming assertions in the response. In particular, the cache information returned in response to a multicast name mapping request on a name n contains not just a *claim* that a particular manager (or manager group) implements all names with a certain prefix, but also a capability K (or set of capabilities) demonstrating the responder's right to make that claim. The client caching such a response would also cache knowledge of the expiration time $t(K)$, and consider the cache entry invalid after that time. The exact nature of K depends on whether n is bound to a local, regional, or global directory.

If n is bound to a local directory, its manager provides a strong-authorization capability for n or some prefix of n to justify its claim to be n's manager.

If n is bound to a regional directory, on the other hand, a responding participant M that does not have strong authorization for n includes a special *participant-address* capability along with the cache information in its response. A participant-address capability for n grants M the right to state that n's participant group address has a particular value G_n. Such a capability can be signed by any principal with strong authorization for n; the capability type and the value G_n appear in the $a(K)$ field. Participant-address capabilities are needed because weak authorization alone is not sufficient to permit M to return n's participant group address—giving the address implicitly asserts a restriction on what managers can implement names within the directory n (only members of the group), an assertion that only managers with strong authorization for n are permitted to make.[8]

Finally, if n is bound to a global directory, the directory server that responds to the request returns a capability demonstrating strong authorization for n. (To allow it to handle directory listing requests, a directory server is given strong authorization for each global directory it participates in.)

[8] If HIER were to permit managers with weak authorization to make such assertions, a manager with weak authorization for `[a` but no authorization for `[a/b` could interfere with attempts to map the name `[a/b` by giving out a false participant group address—one bound to a group not including the manager of `[a/b`.

5.4 The Cost of Capabilities

Capability-based security does not come for free. The following two subsections evaluate its cost—that is, its impact on the efficiency and fault tolerance of name mapping.

5.4.1 Impact on Efficiency

There are two kinds of efficiency costs to be considered: the additional messages needed to obtain new capabilities after old ones have expired, and the additional time needed to process ordinary messages that contain capabilities.

Additional Messages

Roughly speaking, the number of capability request messages generated per unit time is inversely proportional to the average time a capability is valid. This relationship is made more precise below.

We begin by deriving the average arrival rate of requests for new copies of a given capability. Let $v(K)$ be the total validity time for capability K; that is, if $i(K)$ is the time when K was issued, $v(K) = t(K) - i(K)$. Assume that the generation of client requests for action $a(K)$ is a Poisson process, with an average of $\alpha_{a(K)}$ requests per second. Assume also that a capability K is only requested and passed on to clients through the manager that performs the actions it authorizes, and that the manager only requests a new copy of K when a request for action $a(K)$ arrives after all previously issued capabilities for $a(K)$ have expired. Then, whenever a capability K is issued, there will be no further requests in the subsequent $v(K)$ seconds; after that, the next request is expected after an additional $1/\alpha_{a(K)}$ seconds. Thus requests for capability K arrive, on average, every $v(K) + 1/\alpha_{a(K)}$ seconds, so their average arrival rate is $(v(K) + 1/\alpha_{a(K)})^{-1}$ requests per second.

Now we can write an expression for the *total* system load imposed by issuing capabilities, that is, for the overall arrival rate α_{cap} of requests for new capabilities. Let the set of all outstanding capabilities be $\{K_1, K_2, \ldots, K_k\}$, of size k. Then α_{cap} is the sum of the contributions by each capability, that is,

$$\alpha_{\text{cap}} = \sum_i [v(K_i) + 1/\alpha_{a(K_i)}]^{-1} \tag{5.1}$$

We can draw two conclusions from this formula.

First, as remarked above, the cost imposed by issuing capabilities varies inversely with their validity times—so long as the validity times are long enough to dominate the expected inter-usage times. Globally, $\alpha_{\text{cap}} \to 0$ as $\min_i v(K_i) \to \infty$. In fact, if $v(K_i) \gg 1/\alpha_{a(K_i)}$ for all i, and all the $v(K_i)$'s are multiplied by a factor ϕ, α_{cap} is inversely proportional to ϕ. It is reasonable to suppose that $v(K_i)$ is chosen large enough that $v(K_i) \gg 1/\alpha_{a(K_i)}$, at least for most i, because otherwise a large fraction of the requests for action $a(K_i)$ would result in a request for a new capability.

Figure 5.1 illustrates how α_{cap} decreases as the validity time of capabilities increases. Each curve assumes a constant number k of capabilities in the system, each with the same request arrival rate α_K. The validity time $v(K)$ (also assumed the same for each capability) is plotted on the x-axis, and the system load α_{cap} on the y-axis. Both axes are logarithmic. It is evident that, once $v(K)$ is made large enough, as it continues to increase, α_{cap} decreases in inverse proportion—that is, the graph approximates a straight line.

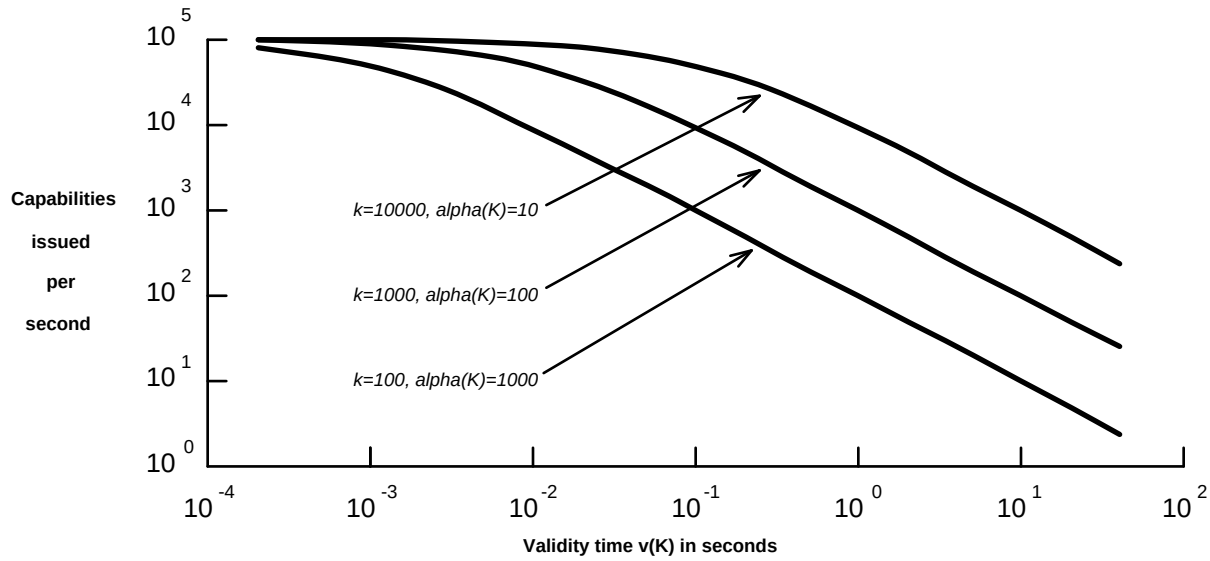

Figure 5.1: System Load *vs.* Capability Lifetime

Next, note that as the number of capabilities in the system is increased, we must increase their validity times proportionately if we wish to avoid increasing the overall load α_{cap}. To see this, assume an existing capability K expires and is reissued as two capabilities K_1 and K_2, with the set of actions authorized by K split between the two: $a(K_1) \cup a(K_2) = a(K)$ and $a(K_1) \cap a(K_2) = \emptyset$. Then the arrival rate $\alpha_{a(K)}$ is also split between the two: $\alpha_{a(K_1)} + \alpha_{a(K_2)} = \alpha_{a(K)}$. How should $v(K_1)$ and $v(K_2)$ be chosen? If $v(K_1) = v(K_2) = v(K) > 0$, the total system load α_{cap} is increased by the split—in the worst case, if $v \gg 1/\alpha$ for both the new capabilities, each one individually contributes the same cost as the old one did, doubling the total. One can, however, prevent the cost from increasing by choosing the new v's larger than the old one; in fact, choosing $v(K_1) = v(K_2) = 2v(K)$ is always sufficient. That is, it is always the case that

$$\frac{1}{v(K) + 1/\alpha_{a(K)}} \geq \frac{1}{2v(K) + 1/\alpha_{a(K_1)}} + \frac{1}{2v(K) + 1/\alpha_{a(K_2)}} \tag{5.2}$$

The proof of this statement is straightforward but tedious. By taking derivatives, one shows the right-hand side of Equation 5.2 achieves its maximum when $\alpha_{a(K_1)} = \alpha_{a(K_2)} = \alpha_{a(K)}/2$, in which case equality holds.

Per-message Overhead

Capability-based counterfeit security also imposes a per-message time and space overhead on the naming system. Some time is required to run the public-key encryption and decryption algorithms on messages, and messages are made longer by the inclusion of capabilities and principal identifiers. This section estimates the overhead quantitatively, assuming the RSA [35] cryptosystem is used. The cost appears small enough to be acceptable in applications where counterfeit security is needed.

Based on the assumptions in Table 5.1, capabilities can range up to about 320 bytes long. Principal identifiers are assumed to be 80 bytes because this length (equivalent to about 200 decimal digits) is typical of RSA keys. Six-byte time values give millisecond resolution across a range of several thousand years. The $a(K)$ field includes the pathname

Field	Description	Length
$a(K)$	Name	Varies, typically ≤ 64 bytes
$p(K)$	Principal granted to	80 bytes
$t(K)$	Expiration time	6 bytes
$s(K)$	Principal granted by	80 bytes

Table 5.1: Capability Field Lengths

prefix covered by the capability, the delegation bit, and for participant-address capabilities, a participant group address and directory identifier (which total 8 bytes in the V implementation). The $a(K)$, $p(K)$, and $t(K)$ fields are encrypted using $s(K)$, making about three 80-byte blocks, and the 80-byte identifier $s(K)$ is transmitted in the clear, for a total of 320 bytes.

In total, most name responses are lengthened by less than 750 bytes. This figure assumes that most capabilities are issued by direct delegates of the security chief, so that a response typically includes two capabilities, K_1 authorizing the response, and K_2 authorizing the signature $s(K_1)$ of K_1. The response must also include the 80-byte principal identifier of the responder (which signs the response by encrypting the remainder of it), and a 6-byte timestamp copied from the request,[9] for a total of 726 bytes. In the current V naming implementation, a `QueryName` response is short (less than 100 bytes on the Ethernet), so even if it were increased by 750 bytes, it would still fit in a single Ethernet packet (up to 1500 bytes).

The time cost to generate or check a capability is substantial, but not intolerable. Encryption and decryption in RSA are slow; the original paper on RSA estimated it would take "a few seconds" to encrypt or decrypt a 200-digit (80-byte) block on a general-purpose machine, and prototype hardware constructed by Rivest ran at only six kilobits per second with 100-digit blocks [16]. Although public-key systems that run faster than RSA are under development, nothing substantially better is available yet. Fortunately, capabilities do not need to be generated or checked frequently; once a client C has seen a capability authorizing M for a particular part of the name space until time t, C need not receive additional capabilities from M until t expires, as long as it is confident the responses it is receiving are from M. Such confidence can be maintained by using a conventional cryptosystem to communicate after the initial capability-checking step, using a conversation key supplied by M along with its capability.

In conclusion, although the use of capabilities does increase the average time and space cost of name responses, this overhead appears small enough to be acceptable in applications where counterfeit security is needed.

5.4.2 Impact on Fault Tolerance

The additional messages required by capability-based counterfeit security do not only reduce a naming system's efficiency; they also reduce its fault tolerance. Only resiliency is affected, not reliability. Failure to receive a requested capability can stop names from being mapped, by making it impossible for their manager to demonstrate its authorization to map them. Such failures can occur either due to failure of a security agent, or due to lost messages on the network. This section quantifies the impact of capability-based security on resiliency, then examines some ways in which increased resiliency can be obtained at the cost of reduced efficiency.

[9] The timestamp is used to match responses with requests, preventing responses from being recorded and replayed.

Under the assumptions of the previous section, the use of capabilities has a severe impact on resiliency: any fault that prevents a capability from being delivered also prevents a request from being satisfied. This is true because a manager never requests a new capability until it is immediately needed to satisfy a request. As α_{cap} increases, the fraction of all requests that are vulnerable to this sort of fault increases with it. If α_{name} is the overall system arrival rate of naming requests, the ratio r_{vul} of vulnerable requests to total requests is $\alpha_{\text{cap}}/\alpha_{\text{name}}$.

Note that if faults occur, α_{cap} will be higher than was calculated above; that is, efficiency is reduced as well. Whenever a capability K is requested but not acquired due to a fault, the next request that requires K will trigger another request for K. Assuming the faults are independent and occur with probability p_{fault} on each capability request, the long-term effect on α_{cap} is as though $v(K)$ were replaced by $(1 - p_{\text{fault}})v(K)$ in formula 5.1; that is,

$$\alpha_{\text{cap}} = \sum_i \left[(1 - p_{\text{fault}})v(K_i) + 1/\alpha_{a(K_i)}\right]^{-1} \tag{5.3}$$

One can make a capability-based secure naming system more resilient (in the sense of lowering the fault-vulnerable ratio r_{vul}), at the cost of poorer efficiency. One technique is to have each manager request a new copy of each of its capabilities K at some time $t'(K) < t(K)$, that is, before its old copy expires. If each manager does this, a fault on any single capability request is tolerated, because the old capability is still valid for a time. After such a fault, the next client request that comes in after the old capability has expired will trigger a new capability request, which should succeed or fail independently of the previous failure.[10] Using this technique, however, makes α_{cap} independent of the arrival rate of client requests, because a new capability is requested regardless of whether the client needs it or not. So, defining $v'(K)$ such that $v(K) - v'(K) = t(K) - t'(K)$, Equation 5.3 becomes

$$\alpha_{\text{cap}} = \sum_i \left[(1 - p_{\text{fault}})v'(K_i)\right]^{-1} \tag{5.4}$$

Thus if there are many capabilities that are infrequently used compared to their validity times, this technique increases α_{cap} substantially.

A compromise approach is to make a capability K *eligible* to be re-requested at time $t'(K)$, but to defer actually doing so until the next client request arrives. With this arrangement, nearly any single fault can be tolerated for frequently-used capabilities, because a new client request will nearly always arrive between times $t'(K)$ and $t(K)$ if $t'(K)$ is chosen to be sufficiently early. But the efficiency penalty for infrequently-used capabilities is not as large, because the arrival rate of client requests does not drop out of the formula for α_{cap}; in this case,

$$\alpha_{\text{cap}} = \sum_i \left[(1 - p_{\text{fault}})v'(K_i) + 1/\alpha_{a(K_i)}\right]^{-1} \tag{5.5}$$

Thus, if faults are rare, adopting this approach has the same efficiency impact as reducing the validity time of each capability K from $v(K)$ to $v'(K)$.

5.5 Can We Do Better?

This section argues that no solution to the counterfeit problem for decentralized naming can have significantly lower cost than the capability scheme described above. Specifically,

[10] Of course, in practice requests that occur close together in time do not fail independently, so $t(K) - t'(K)$ must be reasonably long compared to the mean time to recovery from a failure of the capability-granting mechanism.

it shows that the cost tradeoff exhibited by the capability scheme—between how frequently the naming authorization function is allowed to change, and how many additional messages are needed as compared with a non-secure naming system—is a necessary property of any counterfeit-secure naming system, and argues that, for any given limitation on authorization changes, the capability scheme comes close to minimizing the number of additional messages required.

Theorem 5.4. Under the following conditions, reliable counterfeit rejection requires (in the worst case) that at least one extra message be sent for each name response that is accepted by any client. Conditions: (1) The GEN model of counterfeit security is used. (2) The interconnecting network is subject to omission faults. (3) Neither the client issuing the name request nor any of the managers responding is a security agent. (4) The authorization function can change in any way at any moment; no prior notice need be given to principals that are not security agents. (5) Each naming request made by a client is completed (by accepting a response or rejecting all responses) before that client's next request is issued. (6) There is a time limit ω on how long clients wait for responses before giving up. (7) A response from manager M received at time t_2, corresponding to a request issued at time t_1, must be rejected as counterfeit unless M was authorized to give it *at some time in the interval* $[t_1, t_2]$.

Proof (informal): Suppose that client C issues a naming request Q at time t_1 and accepts a response R from manager M at time t_2. To be sure R is not counterfeit, C must know that at some time in the interval $[t_1, t_2]$, the authorization function permitted M to make the assertions contained in R; call this fact K. Because C is not a security agent, it cannot know fact K unless some security agent has asserted K in a message S_K (not necessarily sent directly to C); the sending of this message must be a different event from the sending of R because M is not a security agent. Now in the worst case, S_K is only sufficient to validate R, not any other name response, because (i) by condition (4), S_K need contain no information about the authorization function at times later than t, (ii) by condition (5), C's next name request will not be issued until some time $t_3 > t_2 \geq t$, so C cannot use S_K to validate another incoming response, and (iii) Q may be the only request received by manager M between times t_1 and t_2, so M cannot (in the worst case) use S_K to validate another, later outgoing response—M cannot delay sending a response to Q until it receives another request, because it might not receive another request before time $t_1 + \omega$. ■

Note that, as this theorem suggests, binding storage under GEN is no longer decentralized in the strict sense. That is, any read quorum for a bound name n must include at least one security agent, in addition to the manager of the object bound to n. Thus, it is not surprising that the counterfeit-secure version of "decentralized" naming described above is less efficient and less resilient than the non-secure version.

In the light of Theorem 5.4, the capability scheme of this chapter can be viewed as a way of reducing the cost of counterfeit-secure naming by limiting the frequency with which the authorization function can change. That is, it avoids requiring an extra message for every client request by modifying precondition (3) of the theorem. Specifically, the issuance of a capability K with expiration time $t(K)$ declares that principal $p(K)$ will *continue* to be authorized for action $a(K)$ at least until time $t(K)$.

The capability scheme seems to take the maximum possible advantage of this type of limitation on changes to the authorization function. For a capability that covers a single manager M, exactly one pair of extra messages is needed each time a request arrives at M after the most recent copy of the capability has expired—that is, each time the values of the authorization function it covers *could have* changed. One can hardly expect to do better in this case. For a capability that covers several managers $M_1, M_2, \ldots, M_m$, there are m pairs of messages. In this case one might be able to do better in favorable cases, by giving some managers their capabilities indirectly, piggybacked on other inter-manger message

traffic or by way of common clients; however, this technique seems rather impractical, and is not applicable in the worst case, where there are no direct inter-manager messages and no common clients.

5.6 Other Security Considerations

There are other security-related problems, beyond that of counterfeit rejection, that users might wish to see solved in a "secure" naming system. The following subsections discuss the feasibility of security models that include two of these considerations: privacy for requests, and consistency among responses.

5.6.1 Privacy For Requests

GEN does not model policies that require *request-privacy.* That is, when a client issues a naming request, there are no restrictions on which managers can hear the request, only on which responses can be accepted. In some applications, it can make sense to impose such restrictions. For example, even the existence of a document called `[gov/whitehouse/user/nixon/enemies/ralph-nader` might be considered sensitive information by its owner.

This section shows that request privacy can be implemented as cheaply as counterfeit rejection, if one is willing to place certain restrictions on the naming authorization function, but it is more expensive in the general case. (The cost measure here is number of messages.)

Consider the security model PRIV, defined like GEN, except that it is also a security violation if a naming request is received by any principal that is not authorized for the name.[11] It is assumed that principals that are authorized to receive a request will not relay it to unauthorized principals. It is also assumed that eavesdropping is possible on the network connecting clients and managers, so the only way to ensure that a message is private is to encrypt it.

In the case where all object managers are considered to act as one principal P_{sys}, it is no more expensive to implement a policy in the class PRIV than the corresponding policy in GEN. For example, if principal identification is implemented as described in Section 5.3.1 above, a client can guarantee its naming requests are private simply by encrypting each with E_{sys} before transmitting it. Only a legitimate manager, possessing D_{sys}, will be able to decrypt such requests.

Also, if the authorization function is static and well-known to all clients, and it authorizes each name to no more than one principal, policies in the class PRIV are again no more expensive than the corresponding policies in GEN. In this case, whenever a client issues a naming request specifying a name n that is authorized to principal P_n, it encrypts the request with E_n before transmitting it over the network.

In the general case, however, where the naming authorization function distinguishes among managers and is not known in advance by clients, PRIV is more expensive to implement than GEN. Under PRIV, a client cannot multicast its name mapping requests to many principals and check only the replies against the security policy. It must instead pre-evaluate which principal(s) are authorized to carry out each naming request it makes, and send the request only to them (i.e., encrypt it such that only they can receive it). Even when attempting to *learn* which principals are authorized for a given *(name, action)* pair, a client cannot send out its request except to a principal already known to be authorized for that pair, because otherwise its privacy could be violated. As a result, each time a client generates a naming request but does not know which managers are currently

[11] An encrypted request is not considered "received" if the receiver is unable to decipher it.

authorized to respond, it must first request that information from a known security agent, requiring an extra message pair beyond the request itself. This cost is greater than that of the capability scheme for GEN, where the expiration of a capability results only in extra messages from the affected managers, not from each of their clients.

5.6.2 Consistency Among Responses

GEN also does not allow one to include requirements for consistency between two or more responses as a direct part of the security policy. That is, all policies are required to be 1-checkable, in the following sense.

A constraint is *1-checkable* if any violation can be observed by looking at a single message. For example, the constraint that only disk drive managers can claim to bind names with the prefix `[device/disk` would be 1-checkable.

A constraint is *non-1-checkable* if violations can only be observed by comparing two or more messages. It is impossible for any single client to verify that a name response it receives is *not* part of a non-1-checkable consistency violation, because the client might have received only half of a pair of conflicting messages, with the other message going to a different client. For example, our definition of specific naming includes a *nonduplication* consistency constraint, stating that a given specific name may be bound to no more than one object at any time. It is impossible to detect a violation of this constraint without examining at least two purported name bindings.

Nevertheless, given a non-1-checkable constraint S, it is sometimes possible to find a 1-checkable constraint S' that is not unduly restrictive, yet guarantees, if it is not violated, that S is not violated. That is, if any two claims together violate S, one or the other must violate S'. For example, dividing the name space into nonoverlapping partitions, each covered by a different object manager, is an effective way of preventing duplicate name bindings from arising; it is effective precisely because it is 1-checkable. If any two managers bind the same name to different objects (violating S), one or the other must be violating the partitioning restriction S'.

In using GEN as our model, we basically adopt the view that if any non-1-checkable consistency constraint S is to be placed on a system, the most that the security policy can do to help enforce this restriction is to enforce a 1-checkable constraint S' that is strict enough to prevent any violations of S. If the security policy does not do this, it is implying that the managers are trusted to cooperate sufficiently to adhere to the constraint without checking on the part of clients. That is, in the case of nonduplication, if the naming authorization function gives two different managers strong authorization for the same name, they must simply be trusted to cooperate as necessary to ensure they hold only one binding between them.[12]

As an additional note, observe that if two managers are asked to cooperate to preserve a non-1-checkable consistency constraint, the security policy must permit them to exchange sufficient information to check that the constraint is not being violated. For example, if manager A is not permitted to know what names are bound by manager B, it is not safe for A to bind any name that B is also authorized to bind.

In summary, it is sensible to exclude non-1-checkable restrictions from our security policy model, for two reasons. First, it is impossible for a single client to verify that a name response it receives is not part of a non-1-checkable security violation. Second, the most practically important non-1-checkable restriction—nonduplication of specific name bindings—can easily be enforced by strictly partitioning the name space among managers that cannot be trusted to cooperate on their own.

[12] For that matter, any manager that is authorized to bind a specific name must be trusted to bind it to only one object at a time.

5.7 Chapter Summary

This chapter has defined the counterfeit problem for decentralized naming and presented a solution based on capabilities. The cost of this solution has been evaluated, in terms of its impact on the efficiency and fault tolerance of naming, and it has been argued that no better solutions are available. In general terms, one can approximate the efficiency and resiliency of non-secure decentralized naming more and more closely as the detailed security policy is allowed to change less and less frequently. The related issues of request privacy and non-1-checkable policies have also been discussed.

Chapter 6

Concluding Remarks

6.1 Summary

Designing a naming facility for large distributed systems is a difficult problem. Existing approaches have not yielded a single design that is at once acceptably efficient, fault-tolerant, and secure.

As a solution, this thesis has introduced and studied decentralized naming, a hierarchical naming architecture based on the concept that each object manager should handle the naming for the objects it manages. Keeping this knowledge at each manager enhances naming efficiency by supporting prefix caching with on-use cache consistency maintenance, and by allowing name mapping operations to be completed in a single packet exchange whenever the cache hits on a local directory. It enhances fault tolerance by supporting multicast name mapping in regional directories. In global directories, where there are too many participants for multicast to be practical, name mapping can be handled by replicated name servers built with known technology. The primary security problem of decentralized naming, the counterfeit problem, is solved using a capability scheme that can be implemented using known cryptographic technology.

The results of this research and their consequences are discussed further in the following subsections. The final section suggests some directions for future work.

6.1.1 Efficiency

Three characteristics of decentralized name mapping have been shown to account for much of its efficiency: its piggybacking of name lookup on other operation requests, the high hit ratio of its prefix caches, and its on-use cache consistency mechanism. These features depend strongly on one another for their effectiveness.

It is a simple but important characteristic of decentralized naming that name lookup is not treated as a separate operation to be implemented in isolation, but is instead treated as the first step in carrying out any operation request that specifies its target object by name. This piggybacking of name lookup, together with the decentralized storage of name bindings at object managers, means that lookup is free (in terms of communication cost) whenever the name cache hits. It is "free" because the client combines the lookup and operation requests into a single message sent directly to the named object's manager, which maps the name locally, performs the operation, and sends the results directly back to the client in a second message. Those two messages would have been required to perform the operation in any case.

The effectiveness of prefix caching is a second important characteristic of decentralized naming. Prefix caching is applicable and effective because of the hierarchical structure of

a decentralized name space, with directories becoming increasingly localized as one moves from the root towards the leaves. Matching a relatively short name prefix typically takes the client to a directory that is local to a single manager, so that the "free" case of name lookup is achieved. And each prefix matches a large number of names, tending to give the cache a high hit ratio. One of the main contributions of this thesis has been to make these observations precise in an analytical model of cache performance, and to validate the model by comparison with experimental results from the V implementation.

A third important efficiency feature of decentralized naming is its on-use cache consistency mechanism. With on-use consistency, having a stale entry in a cache costs nothing until the entry is *used,* at which point it is refreshed. Unlike mechanisms in which servers asynchronously notify clients when their cache entries become stale, on-use consistency is reliable; it is also less expensive. Asynchronous notification cannot be reliable, because the network can partition, separating client and server and preventing the notification from reaching the client, which then continues to use the stale data. It is also expensive because many unnecessary notifications are likely to be issued for entries that will never be used again. Further, unlike mechanisms in which cache entries time out, on-use consistency retains cache entries until they actually become stale, and it does not restrict the frequency with which data that might be cached is allowed to change. However, on-use consistency is inexpensive only because of the piggybacking of name lookup on other object operations: with piggybacking, each name lookup request is sent to the named object's manager in any case, so the manager is able to check cache consistency at the same time.

The main efficiency drawback of decentralized naming is that the cache does miss occasionally, resulting in a multicast that can be quite expensive. In the worst case, the multicast goes to every manager participating in the first regional directory in the given pathname, which of course includes all managers participating in its subdirectories, their subdirectories, and so forth. As discussed in Section 3.4, this effect places a limit on the growth of regional directories, and thus helps to establish where the boundary between global and regional directories must fall. Fortunately, however, the high cache hit ratios achieved in typical installations make it realistic to envision regional directories with over 1000 managers participating.

A minor efficiency drawback is that renaming a directory near the top of the naming hierarchy is relatively expensive, as compared with approaches in which an object's manager does not know its absolute name. Renaming a directory implies changing the absolute pathnames of all the objects that are named relative to it; that is, every node and leaf of the subtree rooted at it. With decentralized naming, making such a change require contacting the managers of all the objects involved to inform them of their new names. Fortunately, however, renaming at this level is not a frequent operation. As Lampson notes [27], changing the name of a top-level directory creates considerable confusion for users, so it is best to do it rarely.

6.1.2 Fault Tolerance

One of the major strengths of decentralized naming is the high resiliency it gives to name mapping. As was shown in Chapter 4, name mapping achieves optimum (ABMA) resiliency in installations that include only regional and local directories. This high resiliency stems directly from the fact that naming is decentralized—each object manager knows the names of its own objects—so multicast name mapping can always be used as a last resort, and will always succeed in mapping the name of any accessible object. In a system that includes global directories, the directory servers become an additional point of possible failure, but they can be replicated at moderate cost because they contain a relatively small and seldom-changing portion of the total information held by the naming system. And even if all servers for a given global directory are inaccessible, a client can still map the names of nearby objects using scoped multicast to nearby participants in the directory.

The resiliency of object creation and deletion by name is also a strength. Because these operations are performed locally by the manager of the named object, which is located using the name mapping protocol, they have the same resiliency as name mapping. Although this resiliency is not the optimum for the general case, it is optimum for these common special cases.

Decentralized naming is basically an application of a more general design principle for distributed systems, which we might call the *togetherness principle: Keep things together if they are most often used together, separately if they are most often used separately.* Following this principle yields advantages in both efficiency and fault tolerance. When two related objects are kept together on one host, the number of messages required to perform an operation that involves both of them is reduced, as well as the number of possible failure points. And when two unrelated objects are stored on separate hosts, of course, the amount of possible parallelism is increased, and the failure of either host makes only one of the objects inaccessible. Decentralized naming applies this principle by keeping names together with the objects they are bound to (in local directories), but separating names of unrelated objects that chance to be in the same regional directory, thus improving both the efficiency and the fault tolerance of name mapping.

The fault tolerance of decentralized naming is poorest for those operations that require access to regional name lists: regional directory listing, binding check, and coverage transfer. The togetherness principle does not provide much help here; although it suggests that copies of a directory's name list need not be kept by nonparticipants, it does not remove the need for a complete copy of the name list to perform these operations. These operations can therefore do no better than the classic tradeoff for replicated data: the operations that read the name list (listing and binding check) can be made more resilient by increasing the degree of replication, but doing this requires the write operations (coverage transfer) to update multiple copies, making them more costly and (if updates must be atomic) less resilient. Note that replicated directory systems have this same problem, but it is worsened by the fact that name mapping is a read operation on the directory, forcing a high degree of replication to achieve acceptable resiliency.

In sum, decentralized naming gives first priority to the resiliency (and efficiency) of the most common operation—name mapping. The other operations are then made as resilient and efficient as possible without compromising the performance of name mapping.

6.1.3 Security

This thesis has also examined the security aspects of decentralized naming. The use of decentralized naming does not make it more difficult to enforce mandatory security restrictions, and it does not complicate the discretionary checking of client authorization by managers, but it does make it more difficult for clients to be sure the purported object managers that respond to their requests are authorized to do so by the system's security policy; that is, to be sure the responses are not counterfeit.

Chapter 5 presented a solution to the counterfeit problem, using capabilities, and evaluated its impact on the efficiency and resiliency of name mapping. It was shown that, roughly speaking, the capability scheme approximates the performance of non-secure decentralized naming more and more closely as the detailed security policy is allowed to change less and less frequently. Different parts of the name space can be individually tuned by varying the validity times of individual capabilities.

One cannot expect to improve very much on this scheme. Any solution to the counterfeit problem can be expected to have some adverse performance impact on a decentralized naming facility, because, strictly speaking, the naming is no longer decentralized if the system includes security agents that can revoke an object manager's authorization to bind names—at least one security agent must be included in each read quorum. The capability scheme reduces the cost of contacting security agents by allowing the agents to "promise"

that the policy will not change for some set period of time, and it appears to take about as much advantage of that technique as possible—for a capability that covers a single manager M, exactly one pair of extra messages is needed each time a request arrives at M after the most recent copy of the capability has expired.

6.1.4 Other Results

This thesis has demonstrated the *practicality* of decentralized naming by describing a substantial prototype implementation that is in daily use in the V distributed operating system. Although the current installation includes only about eight file servers and fifty workstations, it has proven large enough to provide useful experience.

The V implementation also demonstrates the *extensibility* of decentralized naming. Most notably, its name space includes the file systems of several UNIX hosts as subtrees. There is also a server that can make the file system of any host on the DARPA Internet appear as a directory in the V name space, using the Internet's File Transfer Protocol [33] to access the remote files.

6.2 Future Work

Decentralized naming appears to be a promising area for continuing work. Most of the work that remains at this point is implementation and experimentation with some of the portions of the design that have not yet been incorporated into the V prototype. Two areas of particular interest are techniques for replicating name lists in large regional directories, and expansion of the existing regional implementation to include global directories and span an internetwork.

We do not yet have enough experience with large regional directories to set down guidelines for administering the replication of their name lists. In particular, it is not clear how to decide how many replicas are needed and what update mechanism should be used in a given situation. (The current V implementation does not address this question because it includes only directories with off-line name lists.) It is known that (roughly speaking) increasing the number of replicas makes reading more efficient and resilient, at the cost of making writing less efficient and resilient. To quantify this insight and draw useful conclusions from it, additional measurements are needed to determine the ratio of read to write operations, along with some careful modeling to determine the exact cost and resiliency as a function of the number of replicas. The cost and resiliency are also dependent on what algorithm is used to select and maintain agreement on the current set of active replicas. The simplest algorithm statically selects several replica sites, allows reading any replica, but forces write operations to fail unless all sites are up and accessible. More complex algorithms dynamically maintain a set of replicas that periodically contact one another, dropping or replacing any copy that cannot maintain communication with a majority of the other copies. Algorithms of the latter type incur an additional background communication cost on top of the basic cost of performing reads and writes, but achieve improved resiliency for writing. Additional study is needed to decide when the benefits of such algorithms are worth their cost.

It would also be useful to gain experience with an implementation that includes global directories and extends across an internetwork. We have begun to experiment with adding a global directory level to the V implementation, but our installation is not yet large enough to put any serious demands on the global level.[1] Experience with a really large

[1] In fact, V installations are currently unable to grow beyond a single local network because of limitations in the V kernel implementation. The naming protocols are layered on top of the message transaction protocol VMTP [8] provided by the V kernel, which is designed to work across internetworks; however, the current V kernel implementation does not.

installation would provide further guidance in deciding where to draw the line between the global and regional levels of the directory hierarchy. It would also reveal any unforeseen problems that might arise in interfacing the replicated global directory level to the regional and local levels.

Finally, of course, it would be pleasant to see the decentralized naming paradigm become widely used and adopted in a variety of future distributed system designs. It appears qualified to serve well.

Bibliography

[1] D. E. Bell and L. J. LaPadula.
Secure Computer System: Unified Exposition and MULTICS Interpretation.
Technical Report MTR-2997, The MITRE Corporation, January 1976.
Also available as U.S. Department of Commerce, National Technical Information Service, report AD A020 445, and as Air Force Systems Command, Electronic Systems Division, report ESD-TR-75-306.

[2] A. D. Birrell.
Private communication.
January 1987.
Digital Equipment Corporation, Systems Research Center.

[3] 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.

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

[5] 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.

[6] D. Chaum.
Security without identification: Transaction systems to make big brother obsolete.
Communications of the ACM, 28(10):1030–1044, October 1985.

[7] D. R. Cheriton.
The V kernel: A software base for distributed systems.
IEEE Software, 1(2):19–42, April 1984.

[8] 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).

[9] 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).

[10] 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.

[11] D. R. Cheriton and W. Zwaenepoel.

Distributed process groups in the V kernel.
ACM Transactions on Computer Systems, 3(2), May 1985.

[12] R. C. Daley and P. G. Neumann.
A general-purpose file system for secondary storage.
In *Proceedings of the Fall Joint Computer Conference*, pages 213–229, AFIPS, September 1965.

[13] S. E. Deering.
Host Extensions for IP Multicasting.
Technical Report RFC 988, Network Information Center, SRI International, July 1986.

[14] 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.

[15] J. B. Dennis.
Segmentation and the design of multiprogrammed computer systems.
Journal of the ACM, 12(4):589–602, October 1965.

[16] W. Diffie.
Conventional versus public key cryptosystems.
In G. J. Simmons, editor, *Secure Communications and Asymmetric Cryptosystems*, chapter 3, pages 41–72, Westview Press, Boulder, Colorado, 1982.
AAAS Selected Symposia Series.

[17] W. Diffie and M. E. Hellman.
New directions in cryptography.
IEEE Transactions on Information Theory, T-IT76:644–654, November 1976.

[18] 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.

[19] A. D. Birrell et al.
A global authentication service without global trust.
In *Proc. 1986 IEEE Symposium on Security and Privacy*, pages 223–230, IEEE Computer Society, April 1986.

[20] E. Codd et al.
Multiprogramming Stretch: Feasibility considerations.
Communications of the ACM, 2:13–17, November 1959.

[21] J. K. Ousterhout et al.
A Trace-Driven Analysis of the UNIX 4.2BSD File System.
Technical Report UCB/CSD 85/230, Computer Science Division, EECS Department, University of California, Berkeley, April 1985.

[22] N. Goodman et al.
A recovery algorithm for a distributed database system.
In *Proceedings, 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ACM, March 1983.

[23] A. K. Jones.
The object model: A conceptual tool for structuring software.
In *Operating Systems: An Advanced Course*, pages 7–16, Springer-Verlag, 1979.

[24] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner.
One-level storage system.
IRE Transactions on Electronic Computers, EC-11(2):223–235, April 1962.

Reprinted in Daniel P. Siewiorek, C. Gordon Bell and Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982.

[25] D. E. Knuth and L. Trabb Pardo.
The Early Development of Programming Languages.
Technical Report STAN-CS-76-562, Computer Science Department, Stanford University, August 1976.
Also in the *Encyclopedia of Computer Science and Technology*, ed. by Jack Belzer, Albert G. Holzman, and Allen Kent.

[26] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM, 21(7):558–564, July 1978.

[27] 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.

[28] C. E. Landwehr.
Formal models for computer security.
ACM Computing Surveys, 13(3):247–278, September 1981.

[29] E. D. Lazowska, J. Zahorjan, D. R. Cheriton, and W. Zwaenepoel.
File Access Performance of Diskless Workstations.
Technical Report 84-06-01, University of Washington, Department of Computer Science, June 1984.

[30] P. Mockapetris.
Domain Names: Concepts and Facilities.
Technical Report RFC 882, Network Information Center, SRI International, September 1983.

[31] P. Mockapetris.
Domain Names: Implementation and Specification.
Technical Report RFC 883, Network Information Center, SRI International, September 1983.

[32] 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.

[33] J. Postel and J. Reynolds.
File Transfer Protocol.
Technical Report RFC 959, Network Information Center, SRI International, October 1985.

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

[35] 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.

[36] 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.

[37] J. H. Saltzer.
Naming and binding of objects.
In *Operating Systems: An Advanced Course*, pages 99–208, Springer-Verlag, 1978.

[38] J. H. Saltzer and M. D. Schroeder.
The protection of information in computer systems.
Proceedings of the IEEE, 63(9):1278–1308, September 1975.

[39] 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.

[40] 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.

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

[42] 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.

[43] V. L. Voydock and S. T. Kent.
Security mechanisms in high-level network protocols.
ACM Computing Surveys, 15(2):135–171, June 1983.

[44] 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).

[45] D. W. Wall.
Mechanisms for Broadcast and Selective Broadcast.
Tech. Report 190, Computer Systems Laboratory, Stanford University, June 1980.

[46] 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.