A Decentralized Naming Facility

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

February 1, 1986

Abstract

A key component in distributed computer systems is the *naming* facility: the means by which high-level names are bound to objects and by which objects are located given only their names. We describe the design, implementation, and performance of a *decentralized* naming facility, in which the global name space and name mapping mechanism are implemented by a set of cooperating peers, with no central authority. Decentralization is shown to lend increased extensibility and reliability to the design. Efficiency in name mapping is achieved through specialized caching techniques.

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

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

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

Authors' address: Computer Science Department, Stanford University, Stanford, CA 94305

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

1 Introduction

A key component in distributed computer systems is the *naming* facility: the means by which *global, high-level* names are bound to objects and by which objects are located given only their names. *High-level names* are user-assigned character-string names, such as file names, user account names, mailbox names, host names, and service names. These names are distinguished from system-assigned low-level identifiers such as process identifiers and open file handles. A *global* naming facility provides names for objects in the system that can be passed between clients without change in interpretation, often referred to as *absolute* names.

In this paper, we present a decentralized approach to naming, using the paradigm of *problem-oriented shared memory* [4]. Conceptually, a global naming facility can be realized as a single global directory that records all bindings between global names and objects. This directory can be viewed as a shared memory that is accessed by all nodes in the distributed system. In our design, the global directory is partitioned across multiple servers and accessed by multiple clients connected to a network, analogous to multiple memory boards and multiple processors connected to a backplane. Each client maintains a cache of names to reduce network traffic, with a cache miss producing a broadcast or multicast to the participating servers. Various aspects of the design, particularly cache consistency, rely on the semantics of the problem domain, in this case, naming. Besides consistency maintenance, problem-oriented aspects of the design arise in caching optimizations and handling of specialized naming operations.

We examine how the problem-oriented shared memory paradigm leads to an efficient, reliable, extensible, and network-transparent naming facility, drawing on experience with our design as implemented in the V distributed system [5]. We conclude that ours is a feasible approach to high-level naming in distributed systems, with several advantages over the more conventional approach of using dedicated name servers.

The next section describes our design in detail, while section 3 discusses general properties of the design, including its efficiency, reliablility, extensibility, and network transparency. In section 4 we present performance measurements of the design as realized in the V distributed system. Section 5 presents an application of decentralized techniques to low-level naming, and section 6 compares our approach to naming with related work. We close with conclusions and an indication of future directions.

2 Design

Conceptually, our naming facility is a system-wide global directory providing reference by high-level name to objects implemented by multiple object managers. The global directory contains a (name, object)-tuple for each binding of global name to object.[1] Each client may also have its own directory of bindings from local names (or *aliases*) to global names. The naming facility provides operations for

- Binding names to objects
- Removing name bindings
- Name mapping: finding objects bound to a given name
- Inverse name mapping: finding the name bound to a given object

In our decentralized design, the global directory is distributed across the object managers such that each object manager stores and maintains that portion of the directory corresponding to the objects it implements. Each client maintains a cache of bindings from name to object manager, as illustrated in Figure 1. When a client invokes an operation using a high-level object name, the client checks its cache for an entry that maps the name to an object manager. If a cache entry for the name is found (as is the case with *name2* in Figure 1), the operation and name are then forwarded to the object manager indicated by the cache entry. Otherwise, a query is multicast to the object managers to determine the correct object manager for the named object (as is the case for *name1* in Figure 1). If an object manager responds, a cache entry is created and the processing of the request proceeds as before, with the operation being forwarded to the responding object manager. Otherwise, the specified object name is assumed to be invalid and an error indication is returned to the client.

Inverse name mapping is simply a lookup in the global directory using an object's low-level identifier or handle in place of its high-level name. We assume that the low-level identifier provides enough information to determine which manager implements the object in question, and hence which manager stores the portion of the global directory containing its name. The same (absolute) global name is returned for a given object even if the client originally accessed the object using a local name, alias, etc. Low level identifiers are not standardized across all object types, so the inverse name mapping operators provided are manager-specific.

The design as described above follows the conventional implementation of shared memory in a multiprocessor. Each processor (client) has a cache which provides highly efficient access except on cache miss. On cache miss, a request is broadcast on the backplane bus (network) for the missing entry, with the response coming from one of possibly several memory boards (servers). Consequently, it has some of the same advantages, namely:

- **Efficiency.** With a high cache hit ratio, average access time to the global directory is fast. (Measurements indicate that our implementation in the V-System achieves a very high hit ratio; see section 4.1.1.)

[1] Note that high-level names are bound directly to objects, not to low-level names (such as globally unique numeric identifiers). Our design views high-level names as the only permanent, globally unique identifiers for objects.

<==<cliserv.press<

Figure 1: Decentralized Global Directory

- **Reliability.** The failure of an individual server (memory board) only disables access to the portion of the name space it implements.
- **Extensibility.** The global directory can be extended by simply adding new managers (memory boards).

However, consistency maintenance and the provision of specialized naming operations, such as inverse name mapping, require *problem-oriented* techniques that exploit the semantics of the naming operations on this directory "memory."

The name caches attempt to replace multicast network access with directed or unicast network access to the object manager, not to eliminate network access, for we assume that every operation on an object specified by name requires communication with the object's manager. In this respect our design differs from the general shared memory model. In the case of shared memory, eliminating bus access is possible because the cache can maintain complete local copies of the named objects (memory cells). With high-level naming of more complex and varied objects, however, caching the objects themselves is not feasible. Instead (in effect), our design caches each object's last known location.

The following subsections describe naming-specific techniques used to provide cache consistency, to ensure good cache utilization, and to implement various specialized naming facilities.

2.1 On-Use Cache Consistency

Information in one or more client caches may become invalid (or *stale*) when a change is made to the name-object mapping in the global directory. In our design, stale name cache entries are detected and corrected upon use, exploiting the fact that names are only used as part of performing some operation on the named object. To perform the operation, the client must access the object manager for the object and this object manager is, by design, the authoritative storage site for the portion of the global directory that names the object.[2] If the cache entry is stale because the object manager it specifies no longer exists, the communication system signals failure to the client when the client attempts to communicate with the manager. If the object manager exists but no longer stores the named object, the object manager returns a failure indication to the client after examining the name (or portion of the name) passed with the operation request,

When a failure is detected in either of the above two cases, the offending cache entry is deleted, a query is multicast to refresh the cache entry, and the request is tried again. If the multicast query fails or the subsequent retry of the operation fails, an error indication is returned to the invoker of the operation.

On-use cache consistency is much simpler and more efficient to implement in a network environment than cache consistency protocols that require cached information to be consistent at all times. For example, the currently-popular "snoopy" cache protocols for multiprocessors [1,10,11,12] depend on reliable broadcast to notify processors of changes to data that may be in their caches. Such protocols are not practical in our application, since as argued in [8], it is costly to achieve reliable broadcast over an unreliable network. There also a substantial cost in delivering updates to clients that do not have the changed information in their caches. One can, of course, reduce these costs by keeping a central record of which caches contain each data item and transmitting updates only to the affected caches. This approach still requires reliable multicast (or the equivalent series of unicasts), however, and imposes an additional record-keeping burden as compared with on-use consistency.

2.2 Contexts and Context Identifiers

In our design, the name space is hierarchically structured, and we refer to each internal node of the naming hierarchy as a *context* [19]. Names are *pathnames* in that they describe a path through the hierarchy, beginning at (i.e., *relative* to) some specific context. Absolute names are those that begin at the root context.

While providing this familiar hierarchical name space, our design maintains a strict division of the global directory among object managers, using a technique we call *vertical partitioning*. Each object manager implements a tree of contexts starting *at the root* of the complete name hierarchy, thus storing the absolute names of the objects it implements. Some contexts (the root in particular) are implemented by multiple object managers. Such *multi-manager* contexts are partitioned across the managers that participate in their implementation. Each participating manager stores only that subset of the context needed to name objects it manages.[3]

An example of vertical partitioning is shown in Figure 2. The root context, here represented by "/", has two descendants, $/A$ and $/B$. $/A$ is implemented by manager 1 as a *single-manager context*, meaning that all objects with prefix $/A$ are implemented by manager 1. Similarly, the context $/B/M$ is implemented by manager 2, while $/B/N$ is implemented by manager 3. In this example, $/B$ is a multi-manager context since some objects with this prefix are implemented by manager 2 and some by manager 3. Within $/B$, the binding from name M to context **M** is stored only by manager 2, while the binding for N is stored only by manager 3.

Object managers can join or leave the naming hierarchy at any time. For example, assume manager 1 in the above figure is a file server. As the load on the server grows, eventually it may become necessary to

[2] Even simple query operations that could potentially be satisfied by cached information alone, actually verify the cached information with the associated manager before returning it to the client.

[3] The partitioning is called *vertical* by analogy with "vertically integrated" manufacturing companies. Each object manager is self-sufficient, implementing a complete tree of contexts extending vertically to the top of the hierarchy.

`<==<dtree.press<`

Figure 2: Example of a Vertically Partitioned Name Space

move some of its files to a new server. Assume that the subtree beginning with $/A/C$ is copied to the new file server and deleted from manager 1. Now the new file server can join the naming hierarchy, sharing the implementation of context $/A$ with manager 1, and that of / with all the other object managers. Adding a new object manager in this way is analogous to mounting a new UNIX [17] file system as a new entry in an existing directory. The difference is that each object manager in our design maintains its own knowledge of where it is "mounted" in the global name space—the previous implementors of the existing directory (context) do not store the new entry, nor is it recorded in a global "mount table."

A context can be referenced by its absolute name, or by its *context identifier*: a compact low-level identifier that is effectively a pointer into the name space, providing direct, efficient access to the object manager(s) implementing the context. Referencing an object using a context identifier plus relative name allows the name lookup to start at the identified context rather than from the global root, thereby reducing the need for multicast and reducing the length of the name that must be looked up by the object managers. In V, a context identifier is structured as a (*manager-id*, *specific-context-id*) pair, where the *manager-id* is a process identifier or process group identifier specifying the object manager(s) that implement the context, and the *specific-context-id* is mapped by the identified manager(s) to one of the contexts they implement.

When a context is renamed, its old context identifier becomes invalid and another is assigned. Thus, in effect, a context identifier is bound to a context *name*, not to the context object itself.

2.3 Prefix Caching

Our design exploits the hierarchical nature of the name space by using *prefix caching*—the caching of name prefixes rather than full names. With a prefix cache, a cache hit occurs when a cache entry matches some prefix of the name and not necessarily the entire name. Thus, one cache entry serves as a "hit" for a potentially large set of names. For example, if the root context consists of 10 immediate subcontexts that are strictly partitioned across 10 object managers, then a client cache containing these 10 entries eliminates use of multicast for valid names entirely, independent of the total number of names in the system.

Entries in the prefix cache map from name prefixes to context identifiers, allowing the mapping of an absolute name that hits in the cache to proceed with the efficiency of relative name mapping. For example, referring again to Figure 2, if the cache contains an entry for the prefix $/B/N$, all operations using a name with that prefix are unicast directly to manager 3 rather than requiring multicast to locate the correct manager. An operation specifying the name $/B/N/V/Q$ results in the suffix V/Q being transmitted to manager 3 together with the context identifier for $/B/N$.

2.4 Context Identifiers as Hints

Context identifiers are provided as *hints*. That is, a context identifier is allowed to become invalid even if the corresponding character-string name is still bound to the same context. For example, if a V object manager crashes and is restarted under a different process identifier, all its old context identifiers become invalid (since they contain the manager's process identifier as a subfield), even if all the objects it manages are recovered.

This treatment of context identifiers has several advantages. First, a context identifier can contain information that makes it easy to find the context's *current* manager (e.g., its process id), since the identifier is permitted to change when the manager changes. Further, context identifiers need not be allocated for contexts that have never been queried, and need not be kept in stable storage. In particular, the object manager can use in-memory descriptors to identify the context name bound to each context identifier it has returned in response to a name query. The descriptors can be discarded across object manager crashes, or more frequently if memory is limited.[4] In-memory descriptors have proven particularly convenient for object managers implemented at guest level in an existing operating system, such as one we have implemented to provide access to UNIX files from the V-System.

Invalidation of context identifiers introduces another way in which prefix cache entries can become stale. As before, such stale cache entries are detected and corrected on use. An invalid context identifier is detected either by the communication subsystem identifying the manager-id portion as invalid, or by the identified manager reporting the specific-context-id portion as invalid. When a client receives such an error response, it proceeds by retrieving its stored copy of the context's absolute name, multicasting a query for a new context identifier, and retrying the name mapping.

We require object managers to maximize the recycle time for context identifiers, so that a context identifier is very unlikely to be bound to a new context name when one or more client caches still contain a (stale) binding of that identifier to some other name. Maximizing the recycle time is facilitated by providing a large space of possible context identifiers. For example, a V context identifier is 64 bits wide: 32 bits for the object manager identifier and 32 bits for the specific context identifier. This identifier space is large enough that, for all practical purposes, a client cache will never contain an identifier for a previous object manager when the manager identifier is reused for a new object manager,[5] nor will it ever contain an old specific context identifier that is reused within an object manager.

[4] In practice, we find that each manager has sufficient memory that the invalidation of context identifiers is only necessary when an object manager is rebooted, and therefore has little impact on performance.

[5] Creation of new object managers is an infrequent event.

2.5 Relative Names and Current Context

Context identifiers provide a simple implementation of a *current context* for interpreting relative names, analogous to the *current working directory* in UNIX and other systems. In our design, the current context of a program is represented by a context identifier, stored by run-time routines within the program's address space. Each name that is syntactically recognized as a relative name by these run-time routines is augmented with the context identifier of the current context.[6] The context identifier and name suffix are then forwarded to the object manager(s) specified by the identifier, as in the case of a prefix cache hit. The client also stores the absolute name for the current context. If the context identifier for the current context has been invalidated, the client remaps the absolute name for the context, as described previously.

This implementation of relative names and current context provides a facility that works efficiently across the network, requires minimal support in the servers or managers, and provides much the same mechanism as that enjoyed in UNIX. There is a subtle difference in semantics, however—a consequence of our treatment of high-level names as the only permanent identifiers for objects. In V, setting one's current context is logically equivalent to storing a constant absolute name prefix that is prepended to all relative names. If a client's current context is renamed or deleted, all relative names it uses from that point are invalid, unless a new context is subsequently created with the same name as the old one (or the client switches to a new current context). In UNIX, setting one's working directory causes the directory name to be mapped *at that time* only. If the directory is renamed, the client's focus of attention effectively follows it to its new location; if it is deleted, relative names are invalid until the client resets its current context, regardless of whether a new context is created with the old name.

Another way to specify context for a relative name is to prepend a locally-defined name prefix, as described in the next section.

2.6 Local Aliases as Cache Entries

An *alias* is a name bound to another name, such that when the alias is used, it maps to the referent of the second name. A *local alias* is an alias defined locally to a single client, whose interpretation may thus vary from client to client. For example, in the V-System the name prefix *[home]* is locally defined in each program to refer to the home directory of the user who invoked the program.

Local aliases for context names are implemented in our design as additional name cache entries, inserted by explicit client operations. The cache entry for a local alias contains, in place of a context identifier, a pointer to another cache entry containing the name to which it has been equated; this entry may be either another alias or a global name. When the cache lookup algorithm encounters a local alias, pointers are followed until a global name and cached context identifier are encountered. With this structure, if the named context changes its context identifier, the cache management routines can recover by remapping the saved absolute name to obtain the new identifier. Further, if A is defined as an alias for B, and B is also an alias, the meaning of A will change as it should if B is redefined.

Storing local aliases as part of the prefix cache allows both alias expansion and prefix mapping to be performed in a single lookup. Also, the aliases have the correct scope—local to a single program, with inheritance by programs it creates—as discussed in the next section.

2.7 Cache Loading

Client caches are *preloaded* with a number of entries when the cache is created, rather than relying entirely on cache misses to fill the cache. For instance, when the V program executive starts a new program, it preloads the new program's name cache with a set of useful entries, including some standard local aliases. In fact, any program invoked by another program inherits a copy of its parent's name cache, including aliases.

[6] Naming systems typically use some special syntax to distinguish between absolute and relative names; absolute names; for example, a special initial character (e.g., "/" in UNIX) may be used to identify absolute names, with other names being relative to the current working context. The V-System naming implementation uses "[" for this purpose.

Preloading gives much of the benefit of a shared cache, together with the efficiency and flexibility of a per-program cache. That is, there are few cache misses as part of program startup compared to starting with an empty cache, yet a program can efficiently access the cache information since this information is stored in its own address space.

Storing the name cache in every program also allows programs to be executed remotely and migrated [23] without additional communication cost or change of name semantics. The name cache migrates along with the program and the binding of absolute names and context identifiers remains unchanged, since both are location independent.

Clients can also explicitly load particular entries into their caches. In the V implementation, this facility is used primarily to define new local aliases.

Finally, caches are automatically loaded with a new entry after every cache miss, providing the name is valid. Each such miss generates an additional cache entry. We have not considered cache replacement because we do not expect client caches to grow large enough for it to be necessary to remove old entries to make room for new ones.[7] However, each client is free to provide its own cache management. As an extreme example, a simple client (the V bootstrap program) runs with no cache at all, relying entirely on the multicast mechanism for each name.

2.8 Prefix Cache Refinements

The prefix caches in our design include some refinements beyond what has been presented so far. First, since the cache can include prefixes for multi-manager contexts, several prefixes of varying lengths may match a given name. In this case the longest matching prefix must be returned to get maximum benefit from the cache. For example, if the name */a/b/c* is presented and the cache contains entries for */a* and */a/b*, the latter is returned.

A match in which the given name extends beyond the matched prefix and the prefix identifies a multi-manager context is considered a *"near miss"* rather than a hit, since a multicast query is still needed to narrow the search for the named object down to a single manager. A near miss is still helpful, however, as the resulting query need only be directed to the subgroup of object managers identified in the cache entry, not to every object manager. For example, referring to Figure 2, if the name */B/M/K* were presented to the cache, a match on the cached prefix */B* would result in a multicast to servers 2 and 3 only.

Some care is needed to determine when a sufficiently long prefix has been matched and a name query is no longer necessary. When the matched prefix maps to a single-manager context, there is little additional benefit in caching longer prefixes, so such cache entries are considered *non-extendible*. It is also possible for a context to be implemented by a group of processes that split up the work using some criterion other than a name prefix—for example, the user identification of the requestor. Such contexts are also non-extendible. The V implementation provides a flag bit in the specific-context-id field of each context identifier to indicate whether the corresponding context is extendible.

The cache also contains entries for *relative prefixes*, i.e., prefixes of relative names. Relative prefix entries are generated and used when a client attempts to map relative names after having set its current context to a multi-manager context below the root. For example, in Figure 2, the client might set its current context to */B*, then map such names as *M/L* or *N*. Relative prefix cache entries actually map from (context identifier, prefix) pairs to context identifiers. To obtain the longest matching prefix for some names input to the cache, several relative prefixes may need to be chained together and possibly appended to an absolute prefix. As a further optimization, the information returned from a query that extends an extendible prefix is stored as a relative prefix, allowing it to contribute to cache hits for both relative and absolute names.

The full algorithm for processing name queries in the object managers is also slightly more complex than described so far. Upon receiving such a query, each object manager parses the received name left to right until it either (1) reaches the end of the name, (2) reaches a single-manager context that it implements, or (3) reaches a context it does not implement—that is, encounters an undefined name component within

[7] Stale entries are of course deleted upon detection.

a multi-manager context it participates in. Each object manager which reaches case (1) or (2) returns an indication of how far the parse proceeded and which manager (or managers) implement names beginning with that prefix. Object managers that reach case (3) do not reply. For example, referring again to Figure 2, a query specifying the name */B/N/V/S* causes manager 3 to return an indication that the prefix */B/N* maps to manager 3. (Information on shorter prefixes (e.g., */B*) could also be returned if convenient for the manager.) A query specifying the name */B* returns an identifier for the group of managers that implement context */B*; both managers 2 and 3 respond with the same information in this case.

2.9 Generic Names and Group Names

A *group name* is a name that refers to a group (i.e., set) of objects, which need not all be implemented by the same object manager. A *generic name* refers to one member selected from such a group according to a rule associated with the name. Such names are useful in supporting replication for objects or services, as illustrated in Figure 3 below. A request for the current time would be directed to the generic name */time/any*, indicating that a response from any of the three time servers **A**, **B**, or **C** would be acceptable. The same request sent to */time/all* would return the current time value from every time server.

`<==<groupname.press<`

Figure 3: Replication Using Generic Names

In this example, the same set of object managers respond to name query operations on both the generic name */time/any* and the group name */time/all*. However, the context identifier returned from a query on a group name contains an identifier for the entire group of managers implementing the prefix and is marked non-extendible, while each object manager responds to a query on a generic name with an identifier of its own, possibly marked as extendible. Thus, generic name requests generated by a client program are mapped to a random matching cache entry, if any, otherwise to the first object manager to respond to the multicast query. In contrast, group name requests are sent to all the object managers that implement a portion of the name space that matches. A request specifying a group name may result in multiple responses being returned. In V, this is supported by the ability to receive multiple replies in response to a request message [8].

Figure 3 also illustrates replication of some other types of objects. In the example, a user request to run the Pascal compiler would trigger loading of the program code from */sys/bin/pascal*, where */sys* is a generic name prefix for the tree of standard system files, replicated across several storage servers. Output requests intended for the local graphics display—that is, the display attached to the host the client program is running on—would use the generic name */display/local*, while an announcement that should be displayed on every user's screen would go to */display/all*.

To prevent unpredictable behavior when mapping relative names, the standard run-time routines do not permit a client to set its current context directly to a generic name. If a generic name is presented to the *change context* operation, its selection rule is applied to choose a single member context from the group it represents, and that context's absolute name is used as the stored prefix.

It is outside the scope of the naming facility to ensure consistency among replicated files or other objects. The naming facility simply provides support for referencing the replicas by a common name; the object managers implementing them are responsible for maintaining whatever consistency is required. This division of responsibility is appropriate since each type of object has different consistency requirements. For example, replicated files should ideally be locked against parallel update to ensure that the replicas appear identical at all times, whereas replicated time servers must update their times in parallel and can only guarantee approximate consistency—and graphics displays or other devices "replicated" on each workstation do not maintain consistency with one another at all.

2.10 Support for Upward References

An *upward reference* is a name component that refers to the parent of the context it is interpreted in, as with the UNIX ".." notation. If the parent context is implemented by the same manager that implements the interpretation context, the upward reference is handled directly by that manager.

If the parent context is not fully contained in the object manager, however, the operation must be forwarded to the object manager or group of object managers that implement the parent context. This technique is illustrated in the following example. Referring to Figure 2, a client specifies the name *../M/L* in context **N**, which is equivalent to the absolute name */B/M/L*. The client's stored context identifier indicates that context **N** is implemented by manager 3, but the object named by *../M/L* is implemented by manager 2. Manager 3 modifies the client's request by stripping off the name prefix it was able to map and replacing the identifier specifying mapping in context **N** with one specifying context **B**. It then forwards the request to the group of managers that implement context **B** (excluding itself).

Alternatively, the object manager could return an indication to the client to retry the operation in the parent context with a modified name. Using this approach, in the above example manager 3 would return to the client an indication that it mapped the first component of the given name (*../*) to context **B**, specifying the latter by its context identifier. The client would then strip off the prefix and reissue the name mapping request, this time relative to context **B**.

The forwarding technique is preferable in systems that permit forwarding of client requests from one object manager to another, as it is more efficient, simplifies the implementation of clients, and is no more complex to implement in object managers. (The V implementation uses the **Forward** message-passing primitive [5] to implement this technique.)

2.11 Predefined and Administratively Defined Names

Some portion of the hierarchical name space needs to be predefined. At minimum, the name of the root context must be defined and correspond to a particular context identifier. For instance, in the V implementation, the root is named "[", and its context identifier is a static value known to all programs, consisting of a manager-id subfield that designates the group of all object managers providing named objects, and a reserved value for the specific-context-id subfield.

In addition, certain classes of services or categories of objects can be predefined. In V, for example, the context name *[storage]* serves as the root of the subtree of files provided by file managers (or "storage servers"), *[team]* as the context in which the program manager (or "team server") on each workstation defines its name, and so forth. Each of these context names is assigned a fixed context identifier including a statically-assigned process group identifier for the group of managers participating in the context.

These static definitions are among those preloaded into each name cache when it is created. Thus, for example, a cache miss on the name *[storage/user/cheriton/...* is multicast to the set of all storage object managers, not every name-handling manager in the system. Note that logically, only the root name "[" and its context identifier need be known to a client program for it to begin issuing name mapping requests; preloading the other static names serves only to improve performance.

In addition, most object managers are administratively assigned their portion of the name space. For instance, a program manager uses the name of the workstation on which it is running (e.g., Teton) and after booting responds to the *[team/teton/...* portion of the name space. Each object manager checks at initialization time that it is not in conflict with some existing manager by multicasting a query for its portion of the name space. If there is no response, the new object manager assumes there is no conflict. (Two object managers initializing simultaneously for the same portion of the name space would receive responses from each other.) In the case of conflict, the object manager produces an error indication and exits, since the situation calls for administrative intervention.

2.12 Dynamic Name Definition

Dynamic definition of new names takes place in two stages. First, it is necessary to select the object manager that will implement the newly named object, and send it a request to define the new name. If a new object is being created, a default location may be implicit in the new name. For example, if *[storage/user/mann* is currently a single-manager context implemented by a specific storage server, a newly created file with the name *[storage/user/mann/naming.mss* would by default be located on the same manager. The client can also choose to override the default and submit its request to some other manager. If an existing object is being renamed, by default it continues to be implemented by the same manager. Overriding the default in this case implies copying or moving the object to another manager.

Next, the selected object manager checks the request for validity, then records the new definition. Assume manager **M** is attempting to establish a binding for a previously undefined (absolute) name. Call the longest context prefix of the new name that specifies an existing context the *defined prefix*.[8] There are three cases for the manager **M** to consider.

1. The defined prefix specifies a single-manager context $\mathbf{C}_M$ managed by **M**. In this case, the name to be bound is in a part of the name space implemented exclusively by **M**, so it can simply define the name locally.

2. The defined prefix specifies a multi-manager context $\mathbf{C}_G$. In this case, **M** joins the groups implementing $\mathbf{C}_G$ and each of its ancestors in the tree (if it has not already done so), then adds the name suffix to $\mathbf{C}_G$ after querying the other managers in the context to be sure the suffix is not already defined elsewhere.[9] Any contexts **M** creates below $\mathbf{C}_G$ are initially made single-manager.

3. The defined prefix of the name specifies a single-manager context $\mathbf{C}_N$ on another manager **N**. In this case, $\mathbf{C}_N$ (and possibly some of its ancestors) must be expanded to a multi-manager context. For each context to be expanded, **M** chooses a new context identifier (with embedded group identifier), and sends it in a request message to **N**. If **N** agrees to share management of the context, it responds affirmatively to the request, and both managers begin using the new context identifier. If all the expansion requests succeed, **M** defines the name suffix as in the previous case. If not, the name binding attempt fails.

For each new name to be bound, **M** identifies the defined prefix and determines which of the three cases holds using local naming information plus the same query operations used by clients to gather cache information.

[8] The defined prefix is never null since it always includes at least the initial component specifying the root context

[9] As with the manager initialization check, the manager must be prepared to respond to such queries from the other managers concurrently with its query, to prevent race conditions if two managers attempt to define the same name simultaneously.

Our model is that contexts are generally created as single-manager and are expanded to include more managers when needed. For example, if only manager **M** implements objects with name prefix */a/b/c*, the context with that name need only be single-manager. If at a later time manager **N** agrees to implement an object named */a/b/c/d*, context */a/b/c* must become multi-manager with both **M** and **N** participating in its implementation. However, not every object manager need be willing to implement more of the name space than was initially statically assigned to it, nor need every manager be willing to give up exclusive control over parts of the name space it initially implemented as single-manager contexts. For example, the program manager on host Teton would not be willing to implement an object named *[time/gmt]*, nor would it be willing to share implementation of the context of teams running on Teton (*[team/teton]*) with any other object manager.

2.13 Context Directory Listing

In our design, each context in the naming hierarchy has an associated *context directory*: a set of object descriptors, one for each object named in the context. A context directory provides the ability to list the names defined in the associated context and to identify the type and properties of each named object. As with most hierarchical naming designs, only objects (including subcontexts) that are named by a single name component relative to the given context are included in its context directory. I.e., if there is a subcontext, it appears as a single object in the parent context directory, and objects named within the subcontext do not appear. Following other file system designs (again the UNIX file system is a familiar example), a context directory is read by opening it as a file and reading each entry as a data block. Unlike UNIX, however, a directory entry in our design contains a full descriptor for the object, not just a name and internal identifier for the object.

A multi-manager context directory is implemented as multiple context directory files, one per manager in the context. To list a multi-manager context directory, the client opens the context directory for each object manager in the context and then merges the object entries into a single list, as illustrated in Figure 4 below. All the context directories for a context are opened in parallel using multicast, using the name of the context as a group name (section 2.9). To compensate for the inherently unreliable delivery of multicast messages and responses, a followup message containing the list of managers from which replies were received can be multicast to the object managers. Only omitted object managers respond to the followup message. For full reliability, additional followup messages can be transmitted until no more replies are received.

This approach requires the client software to handle the distributed nature of the context, both in locating all object managers and in merging the entries from the set of context directories; however, it has the advantage of allowing the object managers to remain relatively autonomous. They need only cooperate to avoid duplicate names.

3 Discussion

Typical requirements placed on a high-level naming facility include transparency, extensibility, reliability, and efficiency. In this section, we describe how our design meets these requirements, then go on to discuss some other key issues.

3.1 Name Transparency with Local Aliases

The design provides a single global name space in which objects are named with *name and location transparency* [24]. That is, an absolute name specifies the same object, independent of the client using the name. An object manager can be moved within the system without changing the names of any of the objects it implements, since there is no fixed binding between names and network addresses. Further, individual objects can be relocated to different object managers with no change of name because contexts can be split across multiple managers.

<==<mmcontext.press<

Figure 4: Listing a Multi-Manager Context Directory

At the same time, local names are defined on a per-program basis, minimizing interference between programs. Further, a current context for each process is supported, similar to the MULTICS and UNIX concept of working directory. Since these definitions are stored in each program's address space, they migrate with the program and are easily (selectively) passed on to subprograms.

There is, however, some cost associated with naming objects completely independently of their locations. For example, if objects named $/a/b/c/d/e$ and $/a/b/c/d/f$ are stored by different object managers, $/a/b/c/d$ becomes a multi-manager context, along with each of its parent contexts. Listing or adding names to these contexts then become multi-manager operations that require retransmissions and a time delay to ensure reliable communication with all participating managers. Hence there is a performance advantage in grouping objects implemented by a single manager into a subtree of single-manager contexts where possible.

3.2 Extensibility

The naming space is easily extensible to accommodate new objects and new object types, since any object manager can provide global names for the objects it implements by joining the necessary process groups and implementing a system-standard protocol. No changes are required in existing object managers or clients when a new manager is added in this way.[10]

New objects can be added to the name space by adding additional branches or whole subtrees to the hierarchy, as with any hierarchical naming scheme. As discussed in section 2.12 above, if the new branch point is already a multi-manager context, a new object manager can begin implementing names in it simply

[10] Of course, when a new type of object is introduced, clients that need to manipulate such objects may need new type-specific code, but this requirement would arise regardless of the naming mechanism used.

by joining the group that implements the context; if it is a single-manager context, the new manager can convert it to be multi-manager after obtaining permission from the existing manager.

Independently designed name spaces can also be added as subtrees within the global naming hierarchy. A common name syntax for different contexts is not required; only prefixes for different managers must be unique. For example, *[mail/csnet]* could designate the context within which names are interpreted as CSnet mailbox names. Each manager must, of course, implement the standard name operations, particularly the name query operation. For existing (foreign) systems and object managers that need to be accessible from a distributed system following our design, a *facade* can be implemented that handles the required naming operations and passes other operations on to the foreign system. The facade strips off the global prefix on a name before passing on the operation request to the foreign system to be interpreted in its name space. In V, we have one such facade that provides access to UNIX file systems.

3.3 Fault Tolerance

Failures in a distributed system can disable client nodes, server (object manager) nodes, or the connecting network. If a client node fails, it has no effect on the naming mechanism for the rest of the system, since a client node only implements the name cache and name handling for its local programs. In fact, client *programs* can fail individually without detrimental effects on other programs, at least with respect to the naming mechanism, since each program has its own name cache.

If a server node fails, only the object managers on that node are disabled, so only the portion of the name space corresponding to those managers is affected. An object cannot be made unavailable by the failure of an independent entity, such as a global name server. In fact, because a name and the corresponding object are implemented by the same object manager, *objects are available precisely when they are nameable.*

Storing names together with the named objects also minimizes the danger of inconsistency between the name directory and the objects. For example, with a separate name server, deleting a named object requires notifying the name server that its name for the object is invalid. If one of the servers crashes during the operation, the system would be left inconsistent unless deletion were performed as a multi-server atomic transaction. Such expensive solutions to the consistency problem are not required with decentralized naming.

Storing the name and object together also means that, if the object is replicated across multiple managers for reliability, the name binding is automatically replicated the same number of times, thus maintaining a comparable level of reliability for the name binding.

Finally, if the network fails and partitions the system, all objects whose managers are reachable within a given partition remain nameable and available within that partition. Of course, if the network fails completely, each node is left with only the functionality it implements locally. For instance, a personal workstation with its own local disk would continue to have its file system available even though the rest of the distributed file system was unavailable.

During partitioned operation, we do not prevent definition of the same name to refer to different objects in different partitions. When this occurs, an inconsistency can result if the partitions are later reconnected.[11] In environments where such partitioning is expected to occur, each object manager must periodically multicast a query for the root of each of its trees of single-manager contexts, to ensure that no other manager has begun to implement an object of the same name during a period of partitioned operation. Dynamically created multi-manager contexts must also be checked in this way, since two groups of managers could have created contexts of the same name, but with conflicting context identifiers. If a multiply-defined name is detected, in some cases the problem can be corrected automatically—for example, two identically-named contexts could be merged into one multi-manager context—but in general, human intervention is required to determine which object should retain its name and which should be renamed or deleted.

[11] Actually, after the partitions are reconnected, the multiply-defined name acts as a generic name for the set of objects that are bound to it. If this behavior is acceptable, there is no reason to consider the system inconsistent.

There are two other possible sources of reliability problems. First, correctly detecting stale cache entries is dependent on not reusing context identifiers too quickly. In the V implementation, specific context identifiers are selected from a large space (32 bits), allowing the interval between reuses to be long. In fact, an object manager is unlikely to reuse any specific context identifiers during its lifetime between restarts. When restarted, the object manager is assigned a new manager identifier (also selected from a 32-bit space), thereby invalidating all its previous context identifiers and allowing it to start assigning new specific context identifiers beginning at 0.

Second, reliably concluding that a name is undefined is dependent on correct behavior of the communication subsystem, in particular the multicast mechanism.[12] To determine with absolute reliability that a name is undefined, it is in principle necessary to query all object managers that could hold a definition for it and receive a negative response from each. Any object manager from which no response is received must be assumed to have failed to receive the query, and could potentially hold a conflicting definition. Thus, fully reliable multicast is required for absolute reliability of the naming; however, this level of reliability is costly to implement.

Instead, our design adopts a technique that is less costly but still maintains acceptable reliability. A name query is unreliably multicast to the group of all object managers that could hold a definition for it. Only those object managers that recognize the name being queried respond. If no response is received after some number of retransmissions, say N, the client concludes that the name is undefined.[13] The client need not (and does not) know the identities or even the number of group members it is querying. Negative replies would thus carry no useful information; they are therefore omitted to reduce network traffic and avoid systematic errors at clients with inadequate buffering for incoming replies.

With this technique, repeated network failures can cause a client to conclude that a name is undefined when it is in fact defined, but the probability of this error can be made as low as desired. If we assume that the probability of a packet being delivered to a given node is p and is an independent event, the probability of falsely concluding that a name is undefined is $(1 - p^2)^N$, which approaches 0 as N is increased.

Clearly, there is a trade-off between reliability and efficiency here since higher reliability requires a larger number of retransmissions. In the V implementation, retransmissions are separated by a time period (R) longer than the expected response time for the object managers, so concluding a name is invalid requires $N \cdot R$ seconds. This worst-case time period can be shortened somewhat by retransmitting N times before waiting R seconds, but only at the cost of additional network traffic in the average case.[14] The time can also be reduced by careful tuning of the retransmission period: although reducing R may cause some responses to be missed, effectively reducing p and requiring an increase in N to achieve the same reliability, $N \cdot R$ does not necessarily remain constant as R is varied.

Another source of failure in the naming design arises if an object manager or some other entity responds to a portion of the name space that it is does not implement, or is not authorized to implement. We treat this as a security violation.

3.4 Security

Security is commonly divided into *mandatory* security and *discretionary* security. With mandatory security, we assume that different security levels are cleanly separated at the communication level so, for instance, a query at the top secret level is not receivable at the confidential level. Thus object managers at one security level are totally isolated from other security levels. Multi-level object managers must be verified to respond to requests from different security levels in such a way that security is not compromised. In this terminology, multi-level object managers are part of the *trusted computing base*. Clients can only access that portion

[12] Note that incorrectly concluding a name is undefined may not only cause clients to fail but may also allow multiply defined names to arise in the system, since the check for duplicate names in a multi-manager context involves checking whether the name is already defined among several object managers.

[13] The V implementation of this mechanism uses the so-called 1-reliable multicast provided by the V kernel [8].

[14] Also, closely spaced retransmissions may not provide the same level of reliability because the receiving end does not retransmit the reply in response to retransmissions until it has generated a response.

of the name space at their security level since they can only communicate with object managers at their security level. However, there is a slight security problem in avoiding multiply defined names across different security levels. That is, suppose a client at the one level wishes to create an object with a name that conflicts with an existing name at a higher security level. A simple solution is to have separate basic prefixes for each security level, i.e. *[unclassified/storage/...*, *[confidential/storage/...*, etc.

With discretionary security, there is a need to restrict both clients and object managers to that which they are authorized. Object managers can individually respond to clients according to the authorization of the clients. We assume that the communication system provides the identity of the requesting client and the authorization of the client can be determined from this identity.

There are three approaches to enforcing object manager authorization so that an object manager only responds to name prefixes to which it is authorized to respond. First, one can restrict the object managers that can hear and thus can respond to multicast requests to particular parts of the name space. For instance, one can restrict *[storage]* so that only "system" object managers can participate. In the V implementation, we use *restricted process groups*, whose membership is restricted to a set of processes all associated with the same user account. However, this technique precludes providing different file servers under different accounts or authorizations. For example, users of personal workstations may have local file systems that they would like to include in the global name space. To solve this problem, one can provide separate multicast groups for each authorization, but clients then need to know which multicast groups to use, leading to the second approach.

The second approach is to rely on the client to avoid or detect unauthorized participants. For example, one could have a subcontext for each authorization and a well-known mapping from each such subcontext to a multicast group. That is, *[storage/cheriton* could be used by the set of storage servers run under authorization of the user account "cheriton". In V, clients are able to receive multiple responses to a multicast query plus identify the authorization or account associated with the responder. In the case of accessing a object with prefix *[storage/cheriton*, the client would discard all responses not associated with the "cheriton" user account. In general, one can place the onus on the client to detect and reject bogus responses if there are criteria available to clients to recognize bogus responses. Encoding this information in the name is but one relatively simple and convenient scheme. Within this client-based scheme, some additional security derives from the ability for a client to first select an object manager by multicast and then invoke the operation on a particular object manager. The alternative, sending the operation to the group, does not allow the client to control the recipients of the information in the operation invocation request.

Finally, separate network monitoring facilities can be used to watch the network for object managers transmitting unauthorized responses. Again, the multicast query followed by object manager selection allows the network monitor to disable the unauthorized responder before the client actually makes use of the responder.

These ideas go well beyond our current V implementation. However, we regard it as important to consider how these security issues can be addressed with our design and hope to explore them further in our ongoing development of V.

3.5 Efficiency

The efficiency of our design is highly dependent on a high cache hit ratio (which is achieved in practice, as documented by the next section). For instance, the expected elapsed time C of a naming operation (that specifies a valid absolute name) is

$$C = pU + (1-p)(M_k + U) + B$$

where p is the probability of a cache hit, U is the time for unicast communication with a manager, M_k is the time for multicast communication with all k object managers, and B is the time for the operation.[15]

[15] For simplicity, the possibility of a "near miss" in the cache permitting multicast to fewer than k servers is ignored here.

Assuming a high cache hit ratio, and ignoring cache lookup overhead, the expected time approximates $U + B$, the minimal cost. These formulas may also be adapted to compute expected system overhead, as developed further in the next section.

A high cache hit ratio also nicely distributes the processing overhead of name lookup between clients and servers. In the extreme, the complete name is contained in the client cache so the context identifier provided to the object manager eliminates the entire name lookup overhead in the server. This transfer of processing load from servers to clients is appealing, given the multi-client load placed on each server, as a shared facility.

In contrast, a design based on a centralized name server requires two messages to perform each naming operation or $2U + B$, one to the naming server and a second to the object manager. While it would appear that providing a client cache would reduce this to approximately $U + B$, the same as for our design, this realistically requires partial name lookup facilities in each object manager. Otherwise, the client cache must contain the complete names for a large number of individual objects to achieve a high hit rate; i.e., prefix caching is not applicable. In this case, the centralized name server only serves to implement the top levels of the name hierarchy. While such a minimal top-level name server with client caches should have comparable performance to our design, a variety of management problems arise with a server that does so little, yet is so crucial. In our experience with similar servers in V, it is difficult to keep servers of this nature available at all times. It seems necessary to replicate the servers for availability and load balancing, but such replication raises problems of multiple copy consistency that are research issues in themselves.

3.6 Dependence on Multicast

Our decentralized design is dependent on multicast for efficient query of a group of object managers. In the case of a cache miss, querying k object managers using multicast costs M_k, which it is typically much less than $k \cdot U$, the cost of unicast communication with each manager in turn. When the object managers run on distinct processors, multicasting also reduces the elapsed time to complete a request, due to parallelism. In the absence of multicast support, our design could be implemented by a series of unicast communications with the object managers assuming the client knew their identities; however, this technique significantly increases the cost of a cache miss.

Our design also depends on multicast as a binding mechanism. Clients request communication with groups of object managers using a single group identifier. The communication subsystem handles the details of delivering each request message to all group members (but need not guarantee full reliability). A client can begin operation with no *a priori* knowledge of specific server identifiers; only the well-known context identifier for the root context "[" is required.

The total absence of multicast or broadcast at the communication level would make our design less attractive. However, such an environment also makes any name service implementation difficult. For instance, with a centralized name server, the client has the problem of determining the identifier or network address for the name server. Worse yet, if the name server is replicated (as it should be for availability), each client must know the identity of many replicas. Clearly, it is preferable that clients not have these addresses "hardwired" into the code or hardware. While a simple broadcast mechanism would solve this problem, the high cache hit ratios presented in the next section suggest that a simple broadcast mechanism is also adequate for our design without significant loss of efficiency.

4 Performance

We have implemented our decentralized naming design as part of the V distributed operating system. In considering the performance of this implementation, there are three primary measures of interest: the elapsed times for naming operations, the system load imposed by these operations, and the space cost of the implementation. In this section we report performance data gathered on the V installation in the Computer

Science Department at Stanford. We also discuss the possible effect of scaling the system size on these measurements.

4.1 Measurements

Our V installation consists of about 35 Sun and MicroVax II workstations and two Sun-based file servers running the V kernel, plus two 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, text processing, and remote access to other hosts on the DARPA Internet.

4.1.1 Elapsed Time

We first performed a series of experiments to determine the average elapsed time required to map a name. We began by timing a trivial operation performed on an object referenced by name, making separate measurements for each of the three primary cases of interest (Table 1). In the *cache hit* case, a cache hit allowed the operation to be completed in a single unicast message transaction. In the *cache miss* case, the given name missed in the cache but was recognized by some object manager. Completing the operation required a multicast query to fill the cache plus the same unicast message transaction as in the cache hit case. In the *unimplemented name* case, no prefix of the given name was recognized by any object manager. Attempting to map the name generated an unsuccessful multicast query, consisting of several retransmissions and finally a timeout on the client machine.

Case	Elapsed Time (ms)
Cache hit	9.23 ± 0.24
Cache miss	47.7 ± 9.2
Unimplemented name	5379 ± 92

Table 1: Elapsed Time Measurements

The timing tests were run on Sun-2/50 workstations with 10 MHz MC68010 processors and Ethernet interfaces based on the Intel 82586 chip. The test program used the standard client name caching code, while the referenced objects were files managed by a standard in-memory file server commonly used for temporary file storage on the diskless workstations in our system. A *test run* measured the total time for 100 to 10000 operations referencing an object by name, dividing by the number of repetitions to obtain the average time. The table gives the means and sample standard deviations of the times obtained on four test runs.

The measured times break down into four parts—time spent in the client's name caching code, communication time, time spent by the server mapping the name, and time spent by the server performing the operation. The latter time is negligible for these tests, since the requested operation was a trivial one (**GetContextId**). Communication time makes up a substantial fraction of the remainder. For example, the cache hit case required one unicast message including the name (approximately 32 bytes long) as "appended data" [9]. A separate measurement of communication time alone for messages with 32 bytes of appended data gave a time of 5.71 ms (± 0.10), accounting for 62% of the total time.[16]

The average elapsed time for name mapping depends on the cache miss ratio and the frequency of occurrence of unimplemented names seen in actual use of the system. Accordingly, we have instrumented the standard library routines for name management used by each program in our system to collect such statistics. With the modified library in place, each program reports the statistics gathered during its run just before exiting. A system statistician process running on each workstation maintains a running total of

[16] Additional V kernel performance figures, including some measurements of group communication, are available in [8,9,26].

the statistics for programs run there, plus the statistics for the other system processes on that workstation. Periodically, a master statistician program multicasts a request for statistics to the local statisticians, which respond with their current totals, then clear them.

Experimental period:	Oct 17 through Nov 9, 1985
Workstation-seconds:[17]	$6.033 \cdot 10^7$
Total names mapped:	386626
Successful multicast queries:	780 (0.20%)
Failing multicast queries:	380 (0.10%)
No query required:	385466 (99.70%)

Table 2: Summary of Statistics

Referring to Table 2, we see that, using the cache, multicast queries were necessary on only 0.30% of all operations involving name lookups. Of these, one third (0.10% of the total) failed because the given name fell within an unimplemented portion of the name space, while the remainder (0.20%) were true cache misses.

Using these statistics and the elapsed time measurements of Table 1, we can compute an approximate average name mapping time of 14.7 ms. The unimplemented-name case increases this average significantly; the average time for mapping an *implemented* name is 9.31 ms.[18]

These figures allow us to quantify the speed advantage of decentralized naming as compared with centralized name servers (introduced in section 3.5 above). Our implementation of decentralized naming performs operations on named objects in an average of $9.31 + B$ ms, where B is the basic cost of the operation. We can assume that a similar implementation of centralized naming would require about the same amount of time to perform the name mapping as in the cache hit case, plus an additional message transaction to contact the object manager once the name was mapped. Taking 5.71 ms as the cost of the additional message transaction, the total cost per operation would be $15.02 + B$ ms, a significant difference when B is small. Of course, this analysis assumes a totally unloaded name server; under load, the queuing delay in a centralized name server acting as a front end to multiple object managers could be substantial.

Our name-mapping scheme shows its worst-case performance when called upon to map an unimplemented name, taking more than 5 seconds of elapsed time to time out and return an error indication to the client. This time is determined primarily by two characteristics of the underlying communications medium—reliability and delay—as outlined previously in section 3.3. Timing out an unacknowledged transmission requires $N \cdot R$ seconds, with N determined by the required reliability as compared with the reliability of the communication medium and R determined by the expected time to receive an acknowledgement. In our local network environment, 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 at process level on our UNIX systems.

Fortunately, unimplemented 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.

---

[17] The *workstation-seconds* figure gives the total number of seconds of workstation running time for which statistics were reported; it is approximately the product of the number of workstations in the network and the length of the experimental period.

[18] These averages omit some uncommon cases that have little impact on performance, including retries due to stale cache data. The interested reader is referred to [13] for additional data.

4.1.2 System Load

To measure the system load imposed by name mapping, the experiments reported in Table 1 above were repeated, this time measuring the average CPU time consumed on the client workstation requesting the operation, on the server containing the named object, and on another server implementing a different portion of the common name space (a "bystander"). The results of this experiment are reported in Table 3. Again, the mean and standard deviation were computed over four test runs.

Case	Client (ms)	Server (ms)	Bystander (ms)
Cache hit	3.38 ± 0.13	3.89 ± 0.082	—
Cache miss	26.7 ± 5.5	11.6 ± 0.30	6.42 ± 0.21
Unimplemented name	16.0 ± 1.1	—	9.29 ± 0.75

Table 3: System Load Measurements

The most significant figures in this table cases are the 6.42 ms load imposed by a miss and 9.29 ms by an unimplemented name on each bystander reached by the multicast query. Assuming there are enough servers that most servers are bystanders even on successful queries, and using the statistics of Table 2, 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.[19] During the experimental period, we observed 386626 name mapping operations 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 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 rate was only 0.025% and the invalid rate 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.

4.1.3 Space Cost

With decentralized naming, all clients and servers must contain a certain amount of naming code. The space cost of this code is small, however, particularly for modern, medium to high performance workstations.

In V-System servers, the space cost for naming support is insignificant relative to the overall size of the servers. For example, in our disk file server, the C code to implement naming compiles to less than 4000 bytes of object code on the MC68010, less than 5% of the total size of the file server. This figure includes the code for name lookup, name query, inverse name mapping, context directory listing, and context identifier management required to participate in the global naming, plus the implementation of hierarchical naming internal to the file system.

Simpler servers require less naming code. A minimal server that implements only a single object and supports only forward name mapping would require less than 100 lines of C code for its naming implementation. The space cost in client programs is also small in comparison with their total size, typically less than 3000 bytes of code and data space. This space includes a set of standard library routines that handle the details of prefix caching, queries, and retries, presenting a simple interface to the user-level programmer. A typical prefix cache contains less than 20 entries, each of which occupies 22 bytes of memory plus the length of the prefix. 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**) compiles to 1276 bytes on the MC68010.

[19] This figure is conservative, since queries resulting from "near misses" are not received by all servers.

The effort to write and maintain this code is significant, but still manageable. There is only one implementation of the client naming code to be maintained, as all clients use a common set of library routines. The server code is rather similar across different types of servers, and much could be abstracted out into a common library of routines, though this has not yet been done.

4.2 The Effect of Scale

Decentralized naming has demonstrated excellent performance at low cost in our installation, a moderately large distributed system by today's standards. In this section we speculate on how cost and performance would be affected if the system were scaled up to include many more clients and servers.

4.2.1 Elapsed Time

With the cache miss rates we observed, only three operations in a thousand required multicast, resulting in excellent elapsed-time performance in the average case. Several factors could change the cache miss rate in a larger system, however. First, it is possible that with more servers available, each client would access a larger selection of different servers, thereby driving the miss rate up. However, one would expect the widely-observed *locality of reference* property to hold; that is, the number of different servers accessed by a typical client would not increase as rapidly as the number available, so the effect on the cache miss rate would be small.

On the other hand, it appears that in our installation, most misses occur as a result of a client workstation being rebooted (thereby losing its cache contents), or a server process being restarted (thereby invalidating information cached by clients). Since our workstations are used to test experimental system software, such restarts are rather frequent. A large production system would likely be more stable, and would therefore display a lower miss rate.

The elapsed time in the cache hit case, being dependent mostly on the speed of unicast communication, would change little as the system is scaled up unless the increase in size requires a change in network technology—such as from high-speed local networks to an internet incorporating low-speed, long-haul links—thereby increasing the error rate or round-trip delay time for communications. The elapsed times for multicast, either successful or unsuccessful, vary similarly.

In scaling to very large systems, however, (for example, a world-wide internet) some refinements in the use of multicast may be necessary. For example, upon cache miss, one could initially use *scoped multicast* [6] to query only nearby object managers, retransmitting the query throughout the internet only if no response is received after a few tries. If even occasional multicasts to the entire network are infeasible, one could adopt a hybrid architecture in which decentralized naming with multicast is used within local clusters, with a global registration service used to locate object managers in other clusters.

4.2.2 System Load

As observed above, the load imposed on bystanders is proportional to the number of clients, the level of client activity (names mapped per second), and the cache miss rate, and is not affected by the number of servers. Thus in systems with an sufficiently large number of clients, a server would spend 100% of its time receiving and discarding multicast name queries—and this problem could not be alleviated by increasing the number of servers.

Fortunately, with the miss rate and activity levels we measured, bystander processing (even at peak loads) does not reach this level until the number of clients exceeds 700,000. It does not even consume 1% of each server's CPU until the number of clients exceeds 7000.

4.2.3 Space Cost

Scaling up the system should have little effect on space costs. If, as conjectured above, clients access a wider variety of servers when more servers are available, the average number of entries in a client cache would increase as the number of servers is increased. No additional code is needed as the system is scaled up, nor are servers required to store any additional data.

5 Decentralized Low-Level Naming

The design we have presented is also useful in low-level naming; the management of process identifiers in the V kernel [5] is an example. Process identifiers in V are structured as two-level hierarchical names, with the top level being a host identifier (*logical host number*), and the second level identifying a specific process on the logical host. Each kernel maintains a prefix cache mapping logical host numbers to physical host addresses, loaded dynamically as interkernel packets are received. A cache miss results in a multicast to all V kernels. The caches achieve a high hit rate in practice due to locality in communication patterns.

Here as in our high-level naming scheme, stale cache data is detected and corrected on use. (Cache entries can become stale when a logical host migrates or is destroyed.) Whenever a kernel packet must be retransmitted due to lack of response, the destination logical host's cache entry is flushed and the retransmission is done as a multicast. This scheme has been in use in the V kernel for about 2 years, and has proven to be practical and efficient.

A similar technique has been proposed for mapping host group identifiers in an internetwork environment [6].

6 Related Work

In this section, we discuss several systems that are representative of important contemporary styles of naming architecture, contrasting them with our decentralized naming design.

6.1 Network Name Servers

The Clearinghouse [16] is representative of a class of naming systems that are an outgrowth of simple network name servers used for host naming, extended to function across a large internet and to name other objects such as mailboxes or services.

In contrast to our work, which provides an absolute naming mechanism for objects, the Clearinghouse is used primarily for naming and locating object *managers*. Each object manager must then implement a separate name space for its own objects. For instance, one might use the Clearinghouse to locate the file server containing a desired file, but the actual filename would be interpreted by that server and meaningful only to it.

Another difference between the Clearinghouse design and ours is that entities named by the Clearinghouse do not participate in their own naming. Thus, it does not share the intrinsic availability properties of decentralized naming (section 3.3); instead, high availability is achieved by replicating the name registry across many Clearinghouse servers. An advantage of this approach is that the Clearinghouse "remembers" that a given entity exists even when the entity is temporarily unavailable.

Clearinghouse-like object location services seem well suited to large internetworks with a mixture of high-speed and low-speed links. In such an environment, one might use decentralized naming as a high-performance name mapping mechanism within each local cluster of hosts, with a Clearinghouse-like registration service available as a facade through which services in other clusters can be located.

Other examples of systems in this class include the registration service of Grapevine [2,21] and the Domain Name service recently adopted in the DARPA Internet [14,15].

6.2 Network File Systems

Another class of naming system arises in distributed systems that are built by tying together a number of instances of existing centralized systems. Some recent examples include the *Newcastle Connection* [3] and

Sun Microsystems' *Network File System* [20]; an older example is *Cocanet UNIX* [18]. The cited systems link together a network of UNIX hosts using an extension of the UNIX *mount* system call.[20] Each host in the network has its own root file system, in which foreign file systems are locally mounted. Thus the name of file */usr/spool/news* on host Laurel, as viewed from host Hardy, might be */laurel/usr/spool/news*. Name mapping in such a system is efficient, since the initial components of each name are mapped locally, the remainder by the host storing the file. Another advantage of this technique is that it is relatively simple to implement as a modification to UNIX.

Responsibility for name management and mapping in these systems is decentralized, but they differ from ours in that there is no global absolute name space, as there is no guarantee that the multiple root file systems are exact replicas of one another. Thus the same object may have different "absolute" names when viewed from different hosts. This lack of absolute naming can cause difficulties for distributed application programs, since 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 filename 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 host's file system mounted under the name */hardy*.

Another drawback of this technique is that it does not work well with diskless workstations, which do not have local storage to implement their own root filesystems. The Sun NFS solves this problem by assigning each diskless workstation some private space on a file server for its root filesystem. This solution reduces the efficiency of name mapping, however, requiring the diskless workstation to communicate with both its root file server and the file server that stores the named object, which may be on different hosts.

6.3 Integrated Distributed File Systems

Locus [24] represents a third class of naming architecture, in which a centralized file system is extended across a network to form the backbone of a tightly-integrated distributed system.

Like ours, the Locus naming facility provides a global, absolute name space, partitioned across multiple servers (in this case, primarily file servers). The method of partitioning the name space is quite different, however. The Locus name space is split into *file groups* that can be independently assigned to different storage sites, as illustrated in figure 5 below. Each file group contains naming information for the files in that group, relative to the root of the group. One file group (marked *group 0* in the figure) is distinguished as the root, and name mapping is performed by traversing the naming hierarchy, searching a context directory at each node to map the name of the next context in the path.

One advantage of this method of partitioning the name space is that, once the root file group is located, there is no need for broadcast or multicast. To achieve acceptable efficiency and reliability, however, it is necessary to replicate the naming information in the root file group—otherwise, the host storing it would likely become a performance bottleneck, and failure of that host would make all files in the system unavailable.[21] The Locus design solves these problems using a powerful, general mechanism for replicating files and file groups. This mechanism is, of course, not without cost in terms of complexity and the communication time required to keep each replica up to date.

Another difference between the Locus design and ours is that Locus directories (contexts) are never partitioned among multiple servers. As a result, only one server must be contacted to list any directory or to determine whether a name component is defined in a given directory. However, a directory that is partitioned in our design must generally be replicated in Locus (as discussed above), thus requiring updates to the directory to be performed as multi-server atomic transactions to maintain consistency among the replicas. Thus, we do not see either approach as being clearly superior in this respect.

[20] The Newcastle Connection actually simulates this extension outside the UNIX kernel.
[21] In fact, this argument applies to any directory with subdirectories stored on multiple hosts.

`<==<ltree.press<`

Figure 5: Partitioning Into File Groups

In Locus, moving a file from one host to another without changing its name requires either moving its entire file group, establishing a (partial) replica of the file group on the new host (including naming information for the migrated file), or possibly splitting the file group. In either of the latter two cases, some additional replication of naming information may be required to provide efficient and reliable name mapping for the migrated object. This cost is similar to that incurred when migrating an object in our system: one can either move the object manager along with all the objects it implements, or move the object individually. In the latter case, some contexts that were formerly single-manager may be converted to multi-manager.

A recent extension to Locus is to cache the contents of remote directories that are read in the process of name mapping [22]. This form of caching improves performance in much the same way as the prefix caching used in our design; however, it is somewhat more complex. Servers are required to know where each of their directory pages are cached remotely and to notify the clients holding copies of a directory page when that page changes. An advantage of this technique is that cached information at clients is always up-to-date, so it can be used without checking with the server it was obtained from. Thus a client program can list the names defined in a locally-cached directory without contacting the server that stores it.

6.4 Prefix Tables

Welch and Ousterhout [25] describe an extension of the UNIX file system to distributed operation, using *prefix tables* to locate file servers. Prefix tables are essentially the same as our prefix caches and provide similar performance benefits. The underlying name space structure is quite different, however. There are no multi-manager contexts; instead, 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 (Figure 5). 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.

6.5 Earlier V-System Work

Our current work in naming is an extension of the design described in [7]. Our earlier system also distributed the responsibility for object naming among the system's object managers and used a similar name-mapping protocol, but 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. Our current system replaces these with the multicast name mapping mechanism and per-client name caches described in this paper.

7 Conclusions

We have described a decentralized naming facility based on the paradigm of problem-oriented shared memory. The naming facility is decentralized in that, although a central (human) authority may make decisions on how to split up the name space among different object managers and object types, no single entity within the system stores the results of these decisions. Instead, each object manager stores the binding from absolute name to object for the objects it implements. An object's name is its only permanent, globally-unique identifier. We have argued that this approach has advantages in efficiency, reliability, extensibility, and network transparency. Our claims of efficiency are supported by performance measurements taken on a complete implementation.

The efficiency of our design derives largely from its use of per-client prefix caches with a high hit rate. The division of labor among multiple clients and servers, together with the use of multicast in the event of a cache miss, are also contributing factors.

There are two aspects to reliability: availability and correctness. Our design contributes to high availability of resources by vertically partitioning the naming hierarchy among object managers, ensuring that objects are available precisely when they are nameable, i.e., no object can be made unavailable by the failure of an independent entity (such as a global name server). Reliably correct name mapping is achieved (in the absence of persistent network failures) by checking and correcting cached naming information on each use.

Extensibility is provided by the decentralized nature of the name mapping mechanism. Object managers can join or leave the naming hierarchy at any time. Nevertheless, the object managers cooperate sufficiently to implement a single, uniform, network-transparent name space.

In general, we continue to see naming as an important research topic in distributed systems. Our continuing work in this area includes further investigation of the issues of generic and group naming, security, and dynamic extension of the name space. We are also attempting to develop a concise semantic model of high-level naming. For the present, we put forward our design as one that has produced a successful implementation and provides great flexibility and uniformity.

Acknowledgements

We wish to thank the other members of the Distributed Systems Group at Stanford for their patience in serving as a user community for our implementation of decentralized naming in V. A number of group members provided comments on the ideas presented in this paper or assisted in the implementation of object managers that participate in the naming facility. Ross Finlayson, Michael Stumm, Marvin Theimer, Keith Lantz, Joe Pallas, Omur Tasar, Steve Tepper, and Lance Berc were of particular assistance. Rob Nagler, Paul Roy, Bill Nowicki, Kenneth Brooks, and Bruce Hitson also participated in the implementation of the early version of our naming design described in [7].

References

[1] O. P. Agrawal and A. V. Pohm.
Cache memory systems for multiprocessor architectures.
In *Proceedings of the AFIPS National Computer Conference*, AFIPS, June 1977.

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

[3] D. R. 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.

[4] D. R. Cheriton.
Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems.
Operating Systems Review, 19(4):26–33, April 1985.

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

[6] 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).

[7] 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.

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

[9] D. R. Cheriton and W. Zwaenepoel.
The distributed V Kernel and its performance for diskless workstations.
In *Proceedings of the 9th Symposium on Operating System Principles*, pages 129–140, ACM, 1983.

[10] R. Katz et al.
Implementing a cache consistency protocol.
In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, ACM SIGARCH, June 1985.
Also *SIGARCH Newsletter* 13(3), 1985.

[11] S. J. Frank.
Tightly coupled multiprocessor system speeds memory-access times.
Electronics, 164–169, January 1984.

[12] J. R. Goodman.
Using cache memory to reduce processor memory traffic.
SIGARCH, 11(3):124–131, June 1983.

[13] T. P. Mann.
Decentralized naming in distributed systems.
PhD Thesis, Stanford University, to appear.

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

[24] 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).

[25] 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.

[26] W. Zwaenepoel.
Message Passing on a Local Network.
PhD thesis, Stanford University, 1984.
Available as CSL TR 85-283.