Uniform Access to Distributed Name Interpretation in the V-System

David R. Cheriton and Timothy P. Mann

Computer Science Department
Stanford University

Abstract

The naming of services, objects, and operations is an important issue in the design of a distributed system. We have been exploring *distributed name interpretation* in the V-System, in which name interpretation is distributed across the system servers, each server implementing the naming of the objects and operations it provides. Access to this collection of name spaces and name-handling servers is unified by two intentionally minimal mechanisms: a *name-handling protocol* and a *context management* mechanism. The name-handling protocol provides a uniform client interface across all name-handling servers: any V server implementing name spaces or *contexts* must conform to it. The context management system provides convenient access to multiple contexts using per-user *context prefix servers*, as well as an efficient implementation of *current context*.

We argue that our approach is efficient, encourages consistency between names and named entities, and is particularly flexible in name syntax and interpretation. It is also easily extensible to existing systems and name spaces, such as computer mail, which are used in V but were developed elsewhere.*

1. Introduction

The *naming* problem in distributed systems is the problem of assigning names to services, objects, and operations, and providing a means of mapping from names to the entities they represent. For example, one needs to name files and to have a mechanism for mapping file names to files. Good system design puts forward such principles as uniformity, minimal duplication of function, extensibility, efficiency, and reliability. These principles, however, are not always compatible with one another.

Uniformity suggests that a system should be designed to provide a uniform syntax for names and uniform use of operations on these names. For example, the operation

```
Delete(object_name)
```

should delete the named object, independent of what it is, how it is implemented, and where it resides.

Minimizing the duplication of function suggests that name-handling be isolated in one name-handling module which is made available to processes in a distributed system as a server. That is, the name server executes this module in response to client requests, whether as messages or as remote procedure calls. Much previous work in naming has stressed the importance of uniform naming and explored name service as a separate service from the services that implement the objects to be named.

*This research was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431.

In contrast, we have been exploring *distributed name interpretation* in the V-System, with naming distributed across the servers, and each server implementing the naming for the objects and operations it provides. This collection of name spaces and name-handling servers is unified by two intentionally minimal mechanisms: a *name-handling protocol* and a *context management* mechanism.

The name-handling protocol provides uniform client access to all name-handling servers: any V server implementing one or more name spaces or *contexts* must conform to the name-handling protocol. The protocol imposes minimal restrictions on name syntax, and no restrictions on name interpretation or the number of name-handling servers. The context management system provides symbolic naming of multiple contexts using client-selectable *context prefix servers* as well as an efficient implementation of *current context*.

We argue that our approach is efficient, encourages consistency between names and named entities, and is particularly flexible in name syntax and interpretation. It is also easily extensible to existing systems and name spaces, such as computer mail, which are used in V but were developed independently.

The next section discusses approaches to the naming problem. We then provide an overview of the V-System. Section 4 describes V-System low-level naming, and section 5 discusses character-string naming, including the name-handling protocol and context prefix servers. Section 6 describes our current implementation and our experience with it to date, from the standpoints of both performance and functionality. We close with conclusions and an indication of future directions.

2. Approaches to Naming

The nature of a distributed system, in which servers and objects are placed at diverse locations, makes the design of a uniform and efficient name-handling mechanism a difficult problem. One difficulty consists in choosing where name mapping and interpretation is to be done. We identify and contrast two possible solutions below.

2.1. Two Models

In one model, a logically centralized *name server* provides name mapping as a service. This *centralized* model is motivated primarily by the considerations of uniformity and minimal duplication of function mentioned above. Several distributed system designs[1, 2, 3] have identified *naming* as a service in this way and provided a distinguished *name server* to perform this service. Ideally, every server, object, and service in such a system is registered with the name server, and clients present the registered names to the name server when referring to these entities.

The second, *distributed* model stores the names with the objects themselves. This approach is motivated by considerations of efficiency, reliability, and extensibility. By itself, this approach does not seem to provide a full solution to the naming problem, since it is not clear in general how to find an object given its name, if the name is stored with the object.

There are many hybrid approaches possible. For example, a logically centralized name server can be given a distributed implementation, in which definitions of names are placed close (in terms of communication costs) to the named objects. The basic model is the same, but distributed techniques are used to increase efficiency. In contrast, we have chosen to explore a hybrid based on the distributed model, with centralized techiniques used only when absolutely necessary to provide the needed functionality. Some of our reasons for taking this approach are indicated in the comparison below.

2.2. Comparison

While the centralized approach has the advantage of localizing the name-handling operations to one server and thus imposing some level of uniformity on the system, there are several advantages to the distributed approach as well.

Efficiency. Separating the name of an object from its implementation introduces the extra cost of interacting with one more server – the name server – every time a name is referenced. Caching the name in the client would introduce inconsistency problems and only benefit the few applications that reuse names. Because of this cost, there are few name servers that implement file names separate from a file server, even though the name servers implement names for many other system entities such as hosts and users.

Consistency. Separating the naming implementation from the implementation of the named entity makes it more difficult to ensure the name server's information is kept consistent with the objects being named. For example, 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 will be left inconsistent unless deletion as performed as a multi-server atomic transaction. Such solutions to the consistency problem reduce the efficiency of using name servers. Alternatively, many servers and client programs must be prepared to deal with inconsistency in the name service.

Fewer levels of naming. If objects and their names are kept together, mapping from a name to its associated object is an internal operation for the server that maintains both. A name server, on the other hand, cannot map a name to an entity, but only to another name that can be used directly with the server implementing the entity. Thus, an additional level of naming is required between the name server and other system servers. A common design is to use low-level globally unique identifiers (e.g., 48-bit values), with the view that such identifiers are efficient to communicate and manipulate.

We hold the view that unique identifiers are a generalization of identifiers used internally in, for example, file system implementations. Making such identifiers externally visible, and requiring them to be of a uniform format and globally unique, either imposes a uniform scheme of internal naming on all servers, or forces the unique identifiers to be treated as purely external names which are mapped by each server to the lower-level identifiers it uses internally.

Extensibility. A distributed system typically includes numerous different *established* name spaces, name-handling servers, and interpretations. For example, the names for mailboxes, such as "cheriton@su-score.ARPA," may be imposed by standards established outside of the system in question. Such preexisting servers fit well into a model in which names are normally interpreted by the server providing the named objects, but are difficult to accommodate in a system using a name server that translates all names into low-level universal identifiers.

Reliability: If an object's name is stored with the object, the name will always be accessible if the object itself is accessible. A name server, on the other hand, represents a central failure point, and its failure can cause a situation in which objects existing at locations where there have been no failures are inaccessible because they cannot be named.

Although the distributed model offers a number of advantages, it also has some drawbacks. The distributed model works best in the case where a class of objects is implemented by a server, and the objects are stored near the server. Files provided by a storage server are an excellent example; it is convenient to store file names in directory files on the same storage medium as the files they name, and to implement the naming within the storage server. As another example, servers that provide a small number of transient objects – for instance, virtual terminal servers – can store names and attributes of the objects in memory.

Extending the distributed model to cover the entire space of named entities in a distributed system is more difficult. It is advantageous for each server to provide name mapping for the objects it implements, for the reasons described above, but it is not clear how to define names for the servers themselves. One possibility is to provide one or more name servers to map from server names to addresses or other low-level identifiers for servers. This partially centralized method shares several of the drawbacks of fully centralized naming schemes.

Another method would be to have each server store its own name. A name mapping request could then be broadcast or multicast to a group of servers, and each server would compare the specified name with its own name. This technique introduces an additional cost, in that each server in the group receives many requests that are not directed to it, and must spend some processing time in examining and discarding them. There are also potential problems of consistency – some care is required to prevent two non-identical servers from storing identical names for themselves.

2.3. The V Naming Model

In the V naming model, we have combined aspects of the centralized and distributed models using the technique of *distributed name interpretation*. Names may have more than one component (i.e., they may be hierarchical), and different components may be interpreted by different servers. For example, the first component may be interpreted by a context prefix server, which maps the component to a low-level identifier designating a server, then forwards the remainder of the name on to the server. The context prefix server provides a central repository for the names of servers, while the names of other objects are kept close to the objects themselves.

In the case of context prefix servers, we have tried to avoid the drawbacks of centralized naming by providing multiple context prefix servers (one per user), and experimenting with a variety of mechanisms to obtain initial name definitions to be stored by the prefix servers. Planned future work includes experiments using a newly introduced group send mechanism in the V kernel[4] to map server names using the multicast technique mentioned above.

Our approach is also shaped by the recognition that the system is, in part, a distributed database of information on the entities it implements. The name of an entity is just one of its attributes. Extending the name-handling mechanism to include a query operation on objects fits naturally into our model because the server interpreting a name generally also implements the named entity, allowing it to provide additional information about the entity with little difficulty. In contrast, extending a name server to include additional information about the entities it names exacerbates the problems discussed in the previous section, particularly that of maintaining consistency,

Before presenting the details of the name-handling protocol, we describe the basic system environment.

3. V-System Overview

The V-System is a distributed operating system designed to run on a collection of personal workstations and server machines connected by a high-speed local-area network. It currently runs on SUN workstations[5] connected by 3 or 10 Mbit Ethernet. The basic structure of V is similar to that of a number of contemporary distributed systems. The distributed V kernel provides uniform local and network interprocess communication by messages. Most system services are implemented by dedicated server processes that provide their services through message communication. Application programs are written using a procedural interface to system services provided by a collection of *stub* routines that are part of the standard run-time library. These routines hide the message interface from applications that do not explicitly use interprocess communication.

The V kernel provides an ideal base for a distributed system because it allows software above the kernel level to be designed and implemented in the same way, independently of whether the clients it serves and the services it uses are local or distributed. For instance, we are using diskless personal workstations with all file access and program loading via IPC messages to network file servers. Adding a disk and local file server process to a workstation requires no changes to the workstation software other than adding a disk driver to the kernel.

3.1. V-System IPC Model

The following briefly describes the IPC model provided, the basic IPC protocol, and its performance. For further details, the interested reader is referred to a separate in-depth study[6].

The V interprocess communication design is derived from Thoth[7] with several extensions for distributed operation. Basic interprocess communication is in terms of three primitives – ***Send***, ***Receive***, and ***Reply*** – that implement a message transaction or request-response pair of messages, as shown in Figure 1.

The *sender* sends a message to the *receiver* and is blocked until the message is received and replied to. The time for a ***Send-Receive-Reply*** sequence using 32-byte messages between two processes on separate 10 MHz SUN workstations connected by a 3 Mbit Ethernet is 2.56 milliseconds. A message may be *forwarded* by the receiver to a third process, in which case it appears as though the sender originally sent to the third process, which is then expected to receive the message and reply to the sending process.

Large data transfers are handled by separate primitives, ***MoveTo*** and ***MoveFrom***, which move an arbitrary number of bytes between the sender and the receiver of a message while the sender is still awaiting the reply message. Basically, the recipient of a message can use ***MoveFrom*** and ***MoveTo*** to read and write the memory space of the message sender up to the point that the reply message is sent. Using ***MoveTo*** for program loading from a network file server into a diskless SUN workstation (assuming the program text is already in the file server's memory buffers), a 64 kilobyte program can be loaded in 338 milliseconds on the 3 megabit Ethernet. This performance is within 13 percent of the maximum speed at which a SUN workstation can write packets out to the network when there is no protocol overhead.

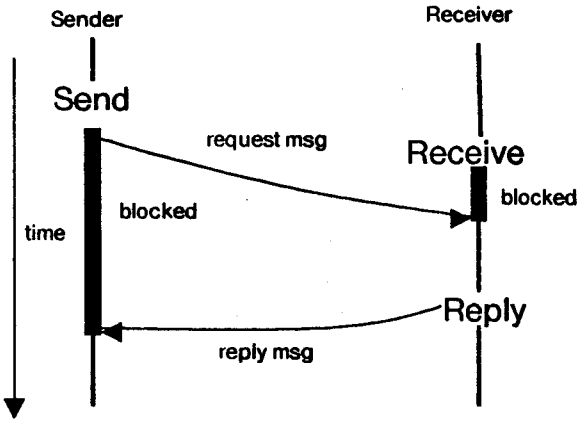

Figure 1: Send-Receive-Reply Message Transaction

Streams can be implemented efficiently using the V IPC primitives. For instance, with a disk delivering a 512 byte page every 15 milliseconds, a file can be read sequentially averaging 17.13 milliseconds per page. This is comparable to the performance of highly tuned special-purpose file access protocols[8]. With this performance, the V IPC facility is also entirely adequate as a transport level for remote terminal access and file transfer.

3.2. Message Standards

One of the key aspects of building a cohesive system on top of the V kernel is adopting standards on the use of messages so that different applications and servers can be interconnected flexibly with minimal difficulty.

Messages specifying operations, called request messages, follow a system standard message format in containing the request code specifying the operation in the first field (16-bit word) of the message. The operation code acts as a tag field in the request message, similar to tag fields in Pascal variant records, specifying the format of the rest of the message, i.e., the number and type of parameters to the operation.

Reply messages sent by servers in response to request messages have their format specified by the requested operation as well, relying on the tight coupling of reply messages to request messages in the V message primitives. A reply code (usually one of a set of standard system replies) appears at the beginning of each reply message, indicating whether the request succeeded or failed, and in the latter case, the reason for failure.

Another V-System standard is the V I/O protocol[9], which provides uniform connection of program input and output to a variety of data sources and sinks, including disk files, terminals, pipes, network connections, graphics pointing devices, and memory arrays. The I/O protocol uses the IPC provided by the kernel as a transport layer. It is a presentation protocol in specifying conventions on the format of messages and a session protocol in specifying the legal sequence of operations for opening a file,

reading and writing the file and closing the file. Basically, it provides a data transfer protocol between any two processes, useful when the data can be viewed as a file and communication between processes can be structured with one process, the server, implementing the file and the other process, the client, reading and writing the file.

The use of the I/O protocol by all servers that support file-like objects has been of utmost importance in the cohesiveness of V. Building on the experience with the I/O protocol over the past 3 years, we have designed and implemented a name-handling protocol that provides uniform client-server interaction for handling names while allowing flexible distributed name-handling by a variety of different servers.

4. V-System Low-Level Naming

V-System naming is structured as three levels: IPC, service, and object. In this section, we discuss the low-level names used for these entities. Because V is built on a message-based distributed kernel, IPC is the most basic level.

4.1. IPC Naming

Communication in V takes place between processes, with senders of data specifying the recipient as a process, as opposed to a port or mailbox.* Similarly, the sender of data is identified to the receiver by its process identifier. A V process identifier (or *pid*) is a 32-bit value that uniquely identifies a process within one V domain.** A *V domain* is a set of logical hosts running the distributed V kernel, usually machines connected by one local network, over which kernel operations are transparent with respect to machine boundaries. A V domain is basically one V-System installation.

Process identifiers are structured as two 16-bit subfields: *logical host* and *local process identifier*, as shown in Figure 2.

Figure 2: Process Identifier Subfields

The logical host field is mapped to a particular host address and the local pid is mapped to a process on the machine at that address. This structuring provides an efficient means of locating a process. It also allows each logical host within a V domain to independently generate unique process identifiers without danger of conflict. Finally, one can efficiently determine from a process identifier whether the named process is local or remote, an important issue for some servers.

Process identifiers are the only absolute names in a V domain. All other names are relative to either a process identifier, typically the identifier of a server process implementing a service, or a "logical process identifier" for a service, as described below.

*Alternatively, one can think of a process as having exactly one mailbox which is permanently associated with that process.

**A process identifier is spatially unique, but not unique in time, as it may be reused at some point after the process it currently names is terminated. We attempt to maximize the time before reuse.

4.2. Service Naming

V-System services include storage, printing, time, context management, virtual graphics terminals, program loading, and exception handling, among others. Most V-System services are provided by dedicated server processes. To name the *service* (as opposed to the process that implements the service), the kernel supports a simple naming facility that allows processes to register as providing a particular service, and allows client processes to determine the process identifier of a registered server.

```
SetPid(service, pid, scope)
```

registers the process specified by *pid* as providing the specified service within the given scope. *Scope* is one of "local" to this machine, "remote" or "both local and remote." We have found it important to distinguish between simple local servers and remotely-available "public" servers, and even to allow both simultaneously for the same service.

```
pid = GetPid(service, scope)
```

returns the process identifier, or *pid*, of a process registered as providing the particular service within the specified scope. In response to a ***GetPid***, the kernel checks its local table and, if that fails and the scope is not *local*, broadcasts to query other kernels on the network.

Using this mechanism, programs are written in terms of services, and the binding of service to server process occurs at time of use. With simple services like time, the client typically translates from service to real server pid on each operation. With file access, the pid for a server process is acquired when the file is opened and used subsequently without remapping from the service name. This avoids the extra overhead of an indirection from service to process on every communication, as arises in a system using ports or mailboxes. It does, however, complicate rebinding when a server fails, although this presumably rare event can then be handled without imposing overhead on normal operation.

A service naming mechanism separate from process naming is required because a process identifier can only identify the process currently implementing the service, and not the service itself. The process may change while the service remains essentially the same. For instance, if a storage server is recreated after a crash with a different process identifier, it is still the same service from the client's point of view, even though now implemented by a different process.

An alternative design for service naming is to make a distinguished name server "well-known" and use it to name all other services. This design can still be implemented using our kernel, but the ***SetPid-GetPid*** mechanism would still be needed to obtain the name server's pid, since the use of even one "fixed" process identifier with the V kernel does not suit its design, i.e., process identifiers are always allocated randomly.

4.3. Object Naming

In general, many services consist of implementations of one or more types of *objects* (in the data abstraction sense), with the operations associated with the service being the operations on these objects, i.e., the server is an abstract data type manager.

There are basically two categories of server objects with respect to naming: temporary and permanent. Temporary objects typically only exist during the execution of the program that requested their creation. A temporary object is named by a short, numeric identifier (low-level name) generated by the server when the object is created. The identifier need not be chosen by the user because it is only used internally by the program and the server. A temporary object is

uniquely specified by the server pid, type of object, and object identifier. Normally, the type of object is derived from the type of the request message, as specified by the operation code.

In V, temporary objects are named using short (16-bit) numeric identifiers, called *object instance identifiers*. Servers attempt to maximize the time before reusing a temporary object identifier after the object has been destroyed.

Permanent objects typically exist beyond the execution of the program that created them. Thus, the name of the object must be stored in a file or remembered by the user. Therefore, permanent objects are named by users or applications for their convenience.

A permanent V object is named using a high-level *character string name*. Character-string naming is discussed in the next section.

5. Character String Naming

V-System standards for character string naming are specified by the *V-System Name Handling Protocol*. The protocol specifies a number of conventions for request messages that use character string names to identify objects. The protocol includes:

- A method for specifying the context in which a name is to be interpreted, using a numeric identifier.
- Specification of several common fields for all messages that contain character string names.
- A standard procedure for mapping names, which may involved forwarding partially interpreted names from one server to another.
- A mechanism for determining the objects named in a given context, including general query and modification operations on such objects.
- Several other operations that must be implemented by all servers participating in the protocol, and some that are optional.

We first define some terms, then discuss the protocol itself. The section concludes with a discussion of context prefix servers.

5.1. Character String Names

A V-System character string name (or *CSname*) is a sequence of zero or more bytes, of a specified length or else terminated by a null byte. Although CSnames may contain arbitrary bytes, they are usually meaningful human-readable ASCII strings.

The term *character string name handling server (CSNH server)* refers to any server that performs character string name mapping as specified by the name-handling protocol, regardless of what else it does. We use the term *CSname request* to refer to any request that contains a character string name that must be mapped to an object as part of the requested operation.

5.2. Contexts

In general, the interpretation of a character string name depends on the *context* in which the name is used. Formally, a context is a set of (name, object)-tuples. A context can have an arbitrary set of members in theory.

Multiple contexts on a single server can arise from servers implementing more than one type of object. For example, a file server may implement both files and user accounts. Multiple contexts are also used to modify the interpretation of names within a structured name space such as a hierarchical file system. One can regard the directories in a conventional file system as defining multiple contexts. For example, the filename "naming.mss" could refer to a file named "/ng/mann/naming.mss," "/ng/cheriton/naming.mss," or perhaps other files, depending upon which fileserver the name was mapped by and which directory was current at the time, i.e., upon which context it was interpreted in.

In the V-System, a context is specified by the pair *(server-pid, context-identifier)*. The *server pid* is a process identifier, specifying the process which is to interpret the name. The *context identifier* is a numeric identifier, specifying a particular context or name space implemented by the server.

Ordinary context identifiers are server-assigned and valid only so long as the server process continues to exist. We have also found it convenient to define several *well-known* context identifiers with fixed values. These are used to specify generic name spaces such as "home directory" and "standard program directory." Also, when a server implements only one context, the context identifier has little meaning and uses a standard default value of 0.

Thus, a *fully-qualified* CSname includes a server pid, a context identifier, and the byte string. Given such a specification, the interpretation of the name is fully specified independent of the operation requested.

5.3. Message Format

Each CSname request specifies the name, length of name, index into the name at which interpretation is to begin (or continue), and a context identifier specifying the context in which to interpret it. The server-pid portion of the context is implictly specified by sending the message directly to the server in question.

This standard format serves as a skeleton for any request message type that contains a CSname. The standard fields are a fixed part of the message structure, always appearing in the same place, while the rest of the message is a variant part whose format depends on the request's operation code. Thus, a CSNH server can perform some processing on any CSname request, even if it does not understand the operation code. The importance of this is indicated below.

5.4. Name Mapping Procedure

A CSNH server follows the following algorithm in handling a request containing a CSname.

If the server does not provide pointers to contexts in other servers as part of its name space, it may interpret the name in any way it chooses.

Otherwise, the server begins by looking at the name itself, not the operation code. Since this request may have been directed to another server (to which it will eventually be forwarded by this algorithm), the operation code is irrelevant at this point.

Names are ordinarily interpreted left-to-right, if the server implements hierarchical naming, though this is not required. The server initializes the variable `CurrentContext` to the context id specified in the request. As each component of the name is parsed, it is looked up in the current context. If the name specifies a context, the variable `CurrentContext` is updated. If the new context is implemented by some other server, the name index field in the request message is updated to point to the first character of the name not yet parsed, the context id field is set to the value of `CurrentContext`, and the request is forwarded to the server that implements the context.

5.5. Object Descriptions

One important operation provided on objects is a query operation to get a description of the object, where the description is a list of properties or attributes of the object. Since the type of the object need not be specified by its name or known to the client, a query operation returns a description record with the first field being a tag field specifying the format of the record, similar to the technique used with request messages. The tag field also allows the application to check that the object is of the type it might be expecting. An example description is shown in Figure 3.

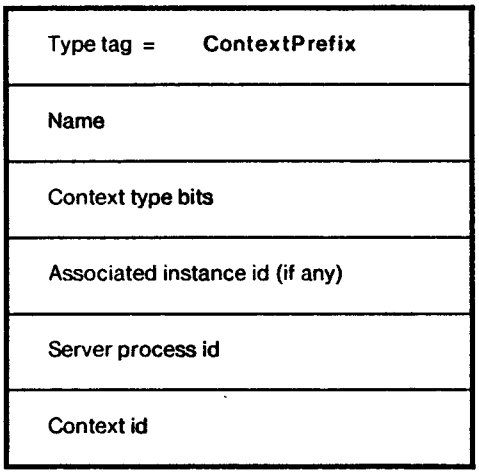

Figure 3: Typical Object Description Record

Some of the fields of a description record are typically names of other system objects, such as name of the owner. Thus, a user-level description of an object can entail query operations not only on this object but also on objects mentioned in the description record for the original object.

It is also important in many cases to be able to modify some fields of an object's description, for example, the access control bits associated with a file. The protocol provides a uniform modification operation, which takes a description record of the proper type, and "overwrites" the original description. Servers are free to ignore changes to any fields which it makes no sense to change in this way.

The query and modification operations manipulate descriptions of individual objects. We introduce the concept of context directories for accessing descriptions of some or all of the objects in a context.

5.6. Context Directories

A *context directory* is logically a file consisting of a sequence of description records, one for each object in the associated context. A client process can open and read a context directory in the same way it opens a file. The description records returned are identical to those returned if the client had instead invoked a query on each object in the context. Writing a description record has the same semantics as invoking the modification operation on the corresponding object.

The directory allows the client to access each object description without knowing the name of each object in the context. It also provides efficient access for the client and the server when the context contains many objects of interest to the client.

A server need not store information about objects in the same format as it is presented to the client in the context directories. Instead, server data structures should be organized to maximize server performance on critical operations with context directory entries dynamically fabricated on demand. For example, a file server may store file names separate from their descriptions with an association maintained by internal indices, such as the "i-node numbers" in Unix, but return both the name and the description in response to a query operation or context directory read. Of course, the cost of dynamically fabricating these records must be considered as part of designing the server.

An alternative to this approach would be to provide an operation that enumerates (or lists) the names of objects in a context. The client would use the list of names in conjunction with the object query operation to simulate the reading of a context directory. We argue that our approach is preferable because with an increasing diversity of objects in our distributed system, even the listing of names of objects in a context requires some indication of the type of each object. Thus, a straight enumeration of names is rarely sufficient and requires an additional operation for each object at considerable cost over the context directory approach. We also view our approach as more consistent with the underlying model in which names are attributes in object descriptions.

The cost of our approach is the collation and transmission of information that may not be required in all cases. Consequently, we have been considering extensions to context directories such as pattern matching, which would cause the server to only include objects that match the given pattern in the returned context directory.

5.7. Standard Request Types

Besides the description query and modify operations just mentioned, three standard operations and two optional ones are specified as part of the name-handling protocol. As explained above, there is no limit to the number of request message types that may contain CSnames.

There is a standard operation to map a CSname that specifies a context into a *(server-pid, context-id)* pair, which is returned in the reply message. There are also standard operations to perform an inverse mapping from a *(server-pid, context-id)* pair to a CSname, or from a *(server-pid, object-instance-id)* pair to a CSname.

There are also operations to add and delete names for an existing context from another existing context. These are optional, and are ordinarily implemented only in context prefix servers (described below).

5.8. Context Prefixes for CSnames

Many servers (particularly file servers) implement a hierarchically structured space of names. Some others may implement arbitrary directed graphs, but we will speak in terms of trees here. A flat name space may be considered as a degenerate tree in which all nodes are sons of the root. Thus, we may view the V-System name space as a forest, where each tree is associated with a server, as illustrated in Figure 4. The forwarding conventions in the naming protocol allow any server to include a pointer to a context on another server as part of its name space (curved arrow in the figure), but in practice, we do not expect enough such pointers to exist at any one time to unify the forest into a single graph.

Therefore, V makes available standard *context prefix servers*, which provide each user with locally defined character string names

for contexts on servers of interest. The context prefix server is a CSNH server that allows users to add and delete names for contexts in other servers, using the optional operation codes mentioned above. A context prefix is simply the part of the CSname that is parsed by the context server to determine where to forward the request. The syntax is: any CSname starting with '[', with the prefix terminated by a closing ']'.

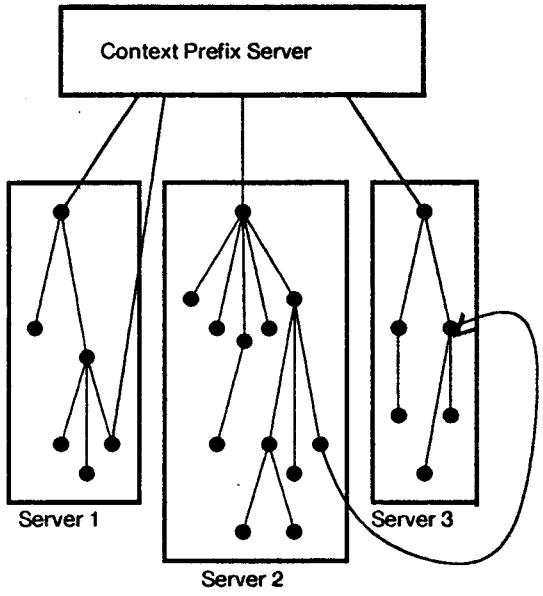

Figure 4: The V Naming Forest

The standard V run-time routines check for such prefixes and route any CSname request containing one to the context server in its default context. This procedure is described more fully in the next section.

6. Implementation

The model described in the previous section has been implemented and in use in the V-System for several months. In this section, we describe some details of interest regarding the implementation and discuss our experience to date.

Our current configuration consists of several diskless SUN workstations (about 30 counting those using V, but not necessarily part of our project) connected by 3 and 10 megabit Ethernet. There are 7 VAX/UNIX* systems running our file server software, providing program loading and file access for the workstations. We are in the process of bringing up a V kernel-based file server on a SUN. There is also a V kernel-based laser printer server, and a Internet server that runs a V kernel-based implementation of IP/TCP. Each workstation also runs one or more simple local server processes, including a virtual graphics terminal server[10], exception server, program manager, and context prefix server.

All of the servers that deal with CSnames implement the name-handling protocol described in the previous section. In particular, the file server software maps context identifiers onto directories that act as starting points for interpreting relative pathnames, similar to the current working directory in Unix[11]. A pathname is interpreted as a context prefix specifying the directory with the final file name component being interpreted in the context defined by the directory.

The context prefix server on each workstation defines context prefixes for the contexts being used by the user of the workstation. Normally these include some standard context prefixes and some corresponding to the file servers being used, plus some special contexts within the file servers, such as home directory, etc. It is common procedure for a workstation user to be accessing several file servers simultaneously, with several context prefixes defined for each server. Because each user has his own context prefix server, the top-level context prefixes can be user-specified and different for each user, although in practice considerable use is made of standard prefixes.

The context prefix server allows names to be defined for both ordinary *(process-id, context-id)* pairs, and also *(logical-pid, well-known-context-id)* pairs. For the latter type of name, the server performs a ***GetPid*** operation each time the name is used. It has proven useful to be able to give character string names to generic services in this way, and several of the standard, predefined prefixes are of this type.

The system run-time routines provide several types of support for the system naming conventions. When a new program is executed, it is passed a process identifier and context identifier specifying its *current context*. It may change this during the course of execution using a function that is analogous to the "change directory" function in Unix. When the program executes an ***Open*** call to access a file-like object, the ***Open*** routine checks whether the name specified starts with the standard context prefix character, '['. If so, it sends an ***Open*** request message to the workstation context prefix server which parses the context prefix, modifies the message to indicate the context corresponding to the prefix, and forwards the message to the server implementing this context. If not, ***Open*** specifies the current context identifier in the message and sends to the request directly to the server implementing the current context. All other CSname-handling routines operate similarly, including routines for removing, renaming, querying, and changing the current context. (The code that checks for the '[' character is localized in a single common routine.)

The time for an ***Open*** (including creating the message, sending to the server, and receiving and processing the reply, but excluding the server-specific actions on ***Open***) is 1.21 milliseconds in the current context with the server local and 3.70 milliseconds in the current context with the server remote. When a context prefix is specified and the ***Open*** request thus goes through the context prefix server, the time increases to 5.14 milliseconds with the server local, and 7.69 milliseconds with the server remote. The difference is identical within the limits of experimental error in both cases (3.94 vs. 3.99 milliseconds), because it reflects the processing time in the context prefix server, which is always local, even though the server performing the ***Open*** may be local or remote.

The context prefix server is 4.5 kilobytes of code plus 2.6 kilobytes of data (mostly space reserved for its context directory) when compiled for the Motorola 68000. This space cost is not significant on SUN workstations, which typically have one or two megabytes of main memory.

Each CSNH server also implements one or more context directories. The initial type tag in each entry specifies one of several standard entry formats. A single "list directory" command lists the

*VAX is a trademark of Digital Equipment Corporation; UNIX is a trademark of Bell Laboratories.

objects in any one of several different contexts, including programs in execution, disk files, virtual terminals, TCP connections, and context prefixes. This program and the typed representations it relies on appear to extend well to the new contexts we are currently implementing.

Requests are provided in the servers for determining the name of a context from its context identifier as well as the name of, for example, a file from its instance identifier. This provides the required support for a program to determine the CSname of its current context as well as the "absolute" name of, for example, an open file. Unfortunately, this is the inverse mapping of a many-to-one function so the CSname may not be the one that was in fact used. In fact, there is no guarantee that there is an inverse mapping, given that, for example, the context prefix may no longer be defined. The difficulty of inverting the name mapping to determine the CSname for an object is made worse by the possibility that a name request was forwarded from one server to another during the course of name interpretation. For example, a component of the name in one file server may be logically linked to a directory on another file server. It is difficult, if not impossible, to determine which server forwarded the request when working backward from the object.

In general, this reverse mapping to extract a name appears quite important and quite difficult. The pathological cases of strange or non-existent reverse mappings occur very rarely and thus are difficult to motivate fixing. Still, they occasionally cause confusion for users.

7. Conclusions

We have presented a model of naming which illustrates how distributed system naming can be treated as an aspect of most services rather than as a separate service in itself. We argue that this approach has advantages in efficiency, consistency, extensibility, and reliability because of the distributed interpretation of a name done local to the implementation of the named object. Uniform access, which is one motivation for separated standard name servers, is provided by standard conventions on message formats and name formats plus a context management system that logically unifies different name spaces via the context prefix server.

This model has been implemented and in use in the V-System for several months, running on multiple diskless SUN workstations. The performance is good. The functionality matches well with our multiple window and executive system and the flexibility they provide.

The current deficiencies lie in reverse name mapping, e.g., determining the name of a file from an open file, and handling error conditions in name mapping. As an example of the latter case, if a name lookup fails after the name has been forwarded through a series of servers, it is difficult to properly inform the user, especially when the failure occurred at a level which is outside his model of the system operation.

Our future work in the naming area is focusing on context directories and handling of queries for a variety of different types of objects. We are also hoping to develop a concise semantic model of the V-System naming. A near-term project is to replace the low-level service naming using ***GetPid*** and ***SetPid*** with a mechanism based on multicast ***Send***.[4] Using this mechanism, a single context could be implemented transparently by a group of servers working in cooperation.

In general, we continue to see naming as an important aspect of a distributed system design. We put forward our model as one that has produced a successful implementation and provides greater uniformity and extensibility than we have enjoyed in any other system.

References

1. R.M. Needham and A.J Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley, 1982.

2. D.C. Oppen and Y.K. Dalal, "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment," Tech. report OPD-78103, Xerox Office Products Division, 1981.

3. R.W. Watson, "Identifiers (naming) in distributed systems," in *Distributed Systems - Architecture and Implementation: An Advanced Course*, Springer-Verlag, 1981.

4. D.R. Cheriton and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System," *Proceedings of the SIGCOMM 84 Symposium: Communications Architectures and Protocols*, ACM, 1984.

5. A. Bechtolsheim, F. Baskett, V. Pratt, "The SUN Workstation Architecture," Tech. report, Computer Science Department, Stanford University, January 1982.

6. D.R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the 9th Symposium on Operating System Principles*, ACM, 1983, pp. 129-140.

7. D.R. Cheriton, *The Thoth System: Multi-process Structuring and Portability*, American Elsevier, 1982.

8. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G.Rudisin, G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating Systems Principles*, ACM, December 1981, pp. 169-177.

9. D. Cheriton, "Distributed I/O using an Object-based Protocol," Tech. report 81-1, Computer Science, University of British Columbia, 1981.

10. K.A. Lantz, D.R. Cheriton and W.I. Nowicki, "Third Generation Graphics for Distributed Systems," Tech. report STAN-CS-82-958, Department of Computer Science, Stanford University, February 1983, To appear in *ACM Transactions on Graphics*.

11. D. M. Ritchie and K. Thompson, "The UNIX timesharing system," *Comm. ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.