Designing a Global Name Service[1]

Butler W. Lampson[2]
Digital Equipment Corporation[3]

Abstract

A name service maps a name of an individual, organization or facility into a set of labeled properties, each of which is a string. It is the basis for resource location, mail addressing, and authentication in a distributed computing system. The global name service described here is meant to do this for billions of names distributed throughout the world. It addresses the problems of high availability, large size, continuing evolution, fault isolation and lack of global trust. The non-deterministic behavior of the service is specified rather precisely to allow a wide range of client and server implementations.

Introduction

> *There are already enough names.*
> *One must know when to stop.*
> *Knowing when to stop averts trouble.*
> Tao Te Ching

The name service I am describing in this talk is intended to be the basis for resource location, mail addressing, and authentication in a distributed computing system. The system I have in mind is a large one, large enough to encompass all the computers in the world and all the people who use them. Of course, the amount of communication between most pairs of computers or people in such a system is small, just as the number of letters or telephone calls between most pairs of people is small. But we expect the postal system or the telephone system to handle such communication on demand, and we should expect the same from a computing system.

A name service maps a name for an entity (an individual, organization, or facility) into a set of labeled properties, each of which is a string. Typical properties are:

---

[1] This paper originated as an invited talk at the 1985 Conference on Principles of Distributed Computing, Minaki, Ontario. It was published in the proceedings of the 1986 Conference on Principles of Distributed Computing.

[2] This design was done jointly by the author, Andrew Birrell, Roger Needham and Michael Schroeder.

[3] Author's address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

```
Password = XQE$#
Mailboxes = {Cabernet, Zinfandel}
network address = 173#4456#1655476653
distribution list = {Birrell, Needham, Schroeder}
```

Grapevine [1, 3] and the Xerox Clearinghouse [4] are examples of such a name service, and they are the basis for the present design. I exclude descriptive "names" from consideration, since I don't know how to specify, much less implement, a service which maps predicates into strings and meets the other requirements of a global name service.

A name service is not a general database: the set of names changes slowly, and the properties of a given name also change slowly. Furthermore, the integrity constraints of a useful name service are much weaker those of a database. Nor is it like a file directory system, which must create and look up names much faster than a name service, but need not be as large or as available. Either a database or a file system can be named by the service, though.

The name service has its own requirements:

- Large size, to handle an essentially arbitrary number of names and serve an arbitrary number of administrative organizations.
- Long life, during which many changes will occur in the organization of the name space and the component that implement the service.
- High availability, because the system can't work when the name service is broken.
- Fault isolation, so that local failures don't cause the entire service to fail.
- Tolerance of mistrust, since a large-scale service won't have any component which is trusted by all the clients.

These requirements imply a hierarchical system; hierarchy is the fundamental method for accommodating growth and isolating faults.

In addition to the functional requirements, there is a need for a precise specification of how the service behaves, especially in the presence of faults. The designers devoted a good deal of effort to such a specification.

The system described here was designed by Andrew Birrell, Butler Lampson, Roger Needham, and Michael Schroeder. We talked extensively with Dave Oran and Tony Lauck. A toy implementation has been done by William Stoye, but no real one has yet been attempted.

The next section gives an overview of the name service, from the viewpoint first of a client and then of an administrator. Next is an explanation of the precise nature of the name space and the provisions for changing it, followed by informal but fairly precise specifications for both client and administrative levels of the service. Interesting

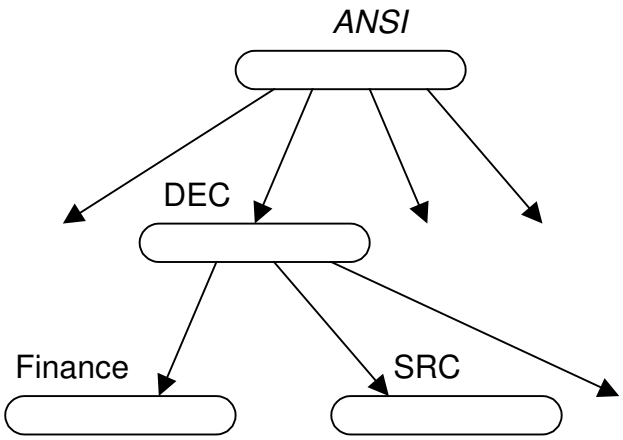

Figure 1: A directory tree

algorithms are sketched but not given in detail. Authentication is an important part of the design, sketched in the next section but discussed elsewhere [2].

The name service has a complete semi-formal specification and a detailed design; it is described informally here.

Overview

The name service supports six major abstractions, divided into two levels. At the client level there are hierarchical names and their values, with operations for reading and updating them, and facilities for protection and authentication. The fact that the database is distributed and replicated is invisible at this level. At the administrative level the copies of the database are visible, together with the mechanisms for locating copies and keeping them synchronized.

Client level

The client sees a structure much like a Unix file system. There is a tree of directories (see figure 1), each with a unique *directory identifier* (DI) and a name by which it can be reached from its parent. The arcs of the tree are called *directory references* (DRs). A DR is the value of the name; it consists simply of the DI for the child directory. Thus a directory can be named relative to a root by a path name called its *full name* (FN). In the figure, the lowest directory is named `DEC/SRC` relative to the root labeled *`ANSI`*. We write this *`ANSI`*`/DEC/SRC`, where the italic *`ANSI`* stands for the directory identifier of the root. The implications of this scheme are discussed below, in the section on the name space.

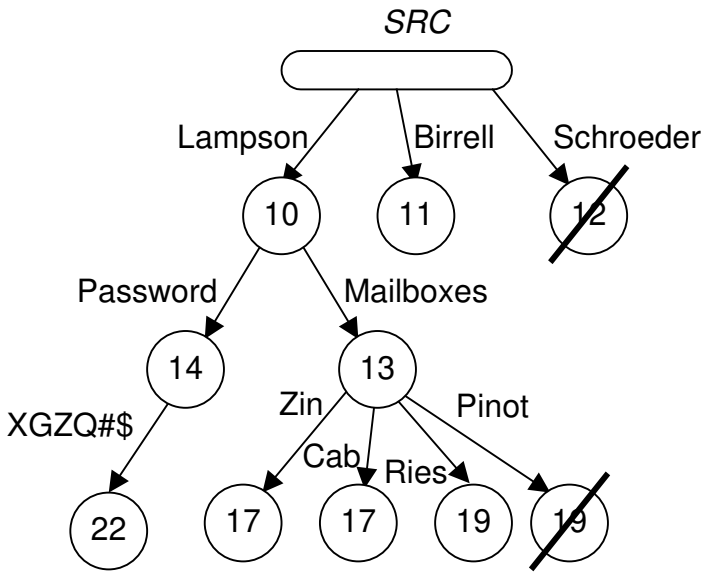

Figure 2: The values in a directory

The value of a name can also be a *link*, which is simply another full name for a directory. Thus, if the value of `ANSI/DEC/SRC/TJW` is the link `ANSI/IBM/TJW`, then `ANSI/DEC/SRC/TJW/CJS` has the same value as `ANSI/IBM/TJW/CJS`.

A directory is not simply a mapping from simple names to values. Instead, it contains a *tree* of values (see figure 2). An arc of the tree carries a label (L), which is just a string, written next to the arc in the figure. A node carries a *time-stamp* (TS), represented by a number in the figure, and a *mark* which is either *present* or *absent*. Absent nodes are struck through in the figure. A path through the tree is defined by a sequence of labels (L*); we write this sequence just like a full name, e.g., `Lampson/Password`.

For the value of the path, there are three cases:

- If the path *l**/*l* ends in a leaf that is an only child, we say that *l* is the value of *l**. This rule applies to the path `Lampson/Password/XGZQ#$3`, and hence we say that `XGZQ#$3` is the value of `Lampson/Password`.

- If the path *l**/l_i ends in a leaf that is not an only child, and its siblings are labeled l_1... l_n, we say that the set $\{l_1 ... l_n\}$ is the value of *l**. For example, `{Zin, Cab, Ries, Pinot}` is the value of `Lampson/Mailboxes`.

- If the path *l** does not end in a leaf, we say that the sub-tree rooted in the node where it ends is the value of *l**. For example, the value of `Lampson` is the subtree rooted in the node with time-stamp `10`.

An update to a directory makes the node at the end of a given path present or absent. The update is time-stamped, and a later time-stamp takes precedence over an earlier one with the same path. The subtleties of this scheme are discussed later; its purpose is to allow the tree to be updated concurrently from a number of places without any prior synchronization.

Access control is based on the notion of a *principal*, which is an entity that can be authenticated by its knowledge of some encryption key (which acts as its password). A principal is identified either by a full name or, in case the root of the full name is not trusted, by a relative name, a path through the directory tree starting at the target directory and using '..' to denote the parent; for example, in the `Finance` directory the principal *`ANSI`*`/DEC/SRC/Lampson` can also be identified by the relative name `../SRC/Lampson`. Each directory has an *access control function* which maps a principal and a path into a set of *rights* drawn from {*read, write, test*}. Each of the operations provided by the name service requires the principal that invokes it to have certain rights to the nodes involved in the operation.

For the convenience of the users, the access control function is defined by a set of triples (principal pattern, path pattern, rights); in the directory of figure 2 the triple (*`ANSI`*`/DEC/*`, `Lampson/*`, {read}) gives every principal starting with *`ANSI`*`/DEC` read rights to the subtree which is the value of `Lampson`. The triple (`../*`, `Lampson/*`, {read}) has the same effect, but the authentication of the principal must come from the parent directory.

Authentication is based on the use of encryption to provide a *secure channel* between the caller of an operation and its implementor. A directory has an *authentication function af*, which is a mapping from keys to principals; it accepts a message encrypted with key *k* as coming from principal *af*(*k*). Each directory has a few values for which *af* is defined by some external means (such as a courier). In particular, there is a secure channel for each parentchild link; the parent's *af* maps this channel's key to the child's name, and the child's *af* maps it to '`..`'.

The authentication function can be extended by a *certificate*, a message encrypted with key *k'* which says, "Key *k* authenticates the principal whose name is *N* relative to me." This allows *af*(*k*) to be defined as *af*(*k'*)/*N*. For example, suppose *`ANSI`*`/DEC` sends the *`SRC`* directory a certificate that *k* authenticates `Finance/Wright` over the secure channel from *`DEC`* to *`SRC`*, then *`SRC`*'s *af* can be extended with (*k*, `../Finance/Wright`).

A sequence of certificates can establish a secure channel between any two directories; the relative names to which the directories map the channel will depend on what other directories participated in setting it up. The details of this scheme, together with arguments for its soundness, can be found in [2].

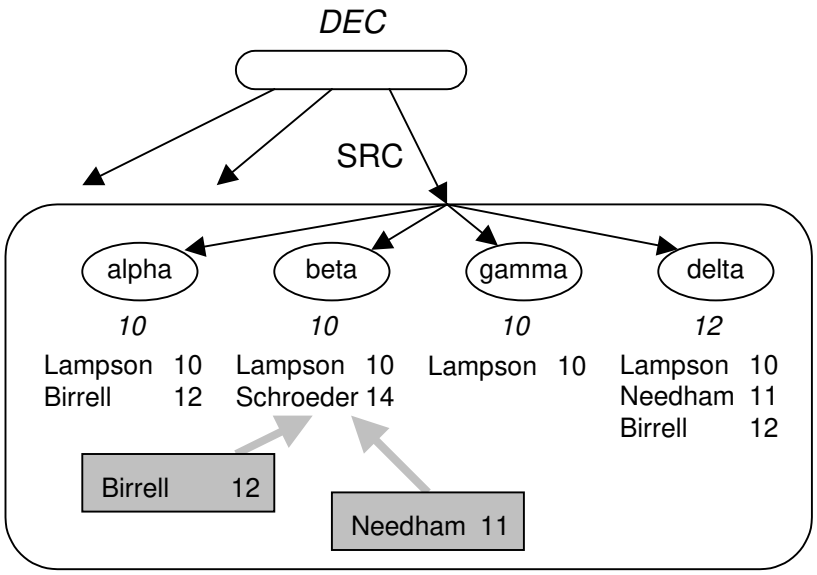

Figure 3: Directory copies with different contents

Administrative level

The client sees a single name service and is not concerned with the actual machines on which it is implemented or the replication of the database that makes it reliable. The administrator allocates resources to the implementation of the service and reconfigures it to deal with long term failures. Instead of a single directory, she sees a set of directory copies (DC), each one stored on a different server (S) machine. Figure 3 shows this situation for the directory *DEC*/SRC, which is stored on four servers `named alpha, beta, gamma, and delta`. A directory reference (DR) now includes not just the DI of the directory, but also a list of the servers that store its DCs. A lookup can try one or more of the servers to find a copy from which to read.

The copies are kept approximately but not exactly the same. The figure shows four updates to *SRC*, with timestamps 10, 11, 12 and 14. The copy on `delta` is current to time 12, as indicated by the italic *12* under it. This is called its *lastSweep* time. The others have different sets of updates, but all have *lastSweep* = 10. Each copy also has a *nextTS* value (not shown), the next time-stamp it will assign to a new update; this value can only increase.

An update originates at one DC, and is initially recorded there. The basic method for spreading updates to all the copies is a *sweep* operation, which visits every DC, collects a complete set of updates, and then writes this set back to every DC. The sweep has a time-stamp *sweepTS*. Before it reads from a DC it increases that DC's *nextTS* to *sweepTS*; this ensures that the sweep collects all updates earlier than *sweepTS*. After it writes back to a DC, it sets that DC's *lastSweep* to *sweepTS*. Figure 4 shows the state of *SRC* after a sweep at time 14.

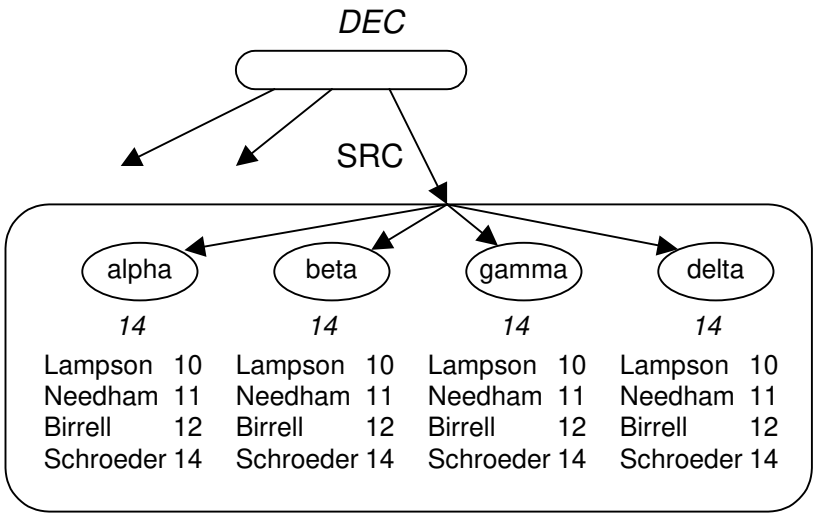

Figure 4: The directory of figure 3 after a sweep

In order to speed up the spreading of updates, any DC may send some updates to any other DC in a message. Figure 3 shows the updates for `Birrell` and `Needham` being sent to server beta. I expect that most updates will be distributed in messages, but it is extremely difficult to make this method fully reliable. The sweep, on the other hand, is quite easy to implement reliably.

A sweep's major problem is to obtain the set of DCs reliably. The set of servers in the DR stored in the parent is not suitable, because it is too difficult to ensure that the sweep gets a complete set if the directory's parent or the set of DCs is changing during the sweep. Instead, all the DCs are linked into a *ring*, shown by the fat arrows in figure 5. Each arrow represents the name of the server to which it points. The sweep starts at any DC and follows the arrows; if it eventually reaches the starting point, then it has found a complete set of DCs. Of course, this operation need not be done sequentially; given a hint about the contents of the set, say from the parent DR, the sweep can visit all the DCs and

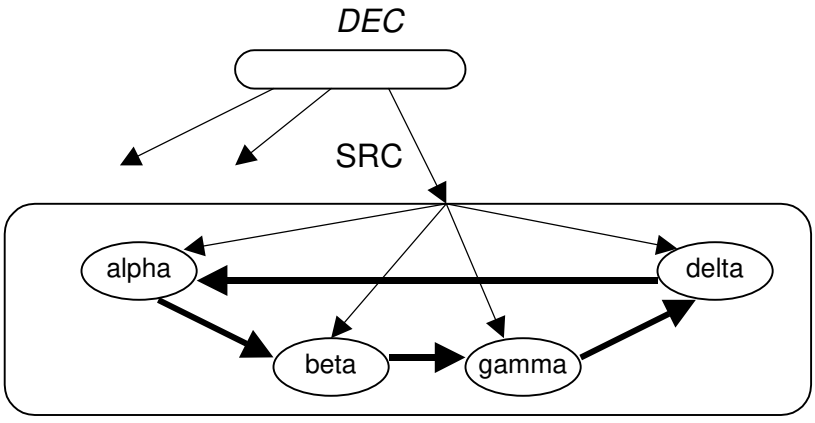

Figure 5: The ring used for a sweep

read out the ring pointers in parallel. DCs can be added or removed by straightforward splicing of the ring.

If a server fails permanently, however (say it gets blown up), or if the set of servers is partitioned by a network failure that lasts for a long time, the ring must be reformed. In the process, an update will be lost if it originated in a server that is not in the new ring and has not been distributed. Reforming the ring is done by starting a new *epoch* for the directory and building a new ring from scratch, using the DR or information provided by the administrator about which servers should be included. An epoch is identified by a time-stamp, and the most recent epoch that has ever had a complete ring is the one that defines the contents of the directory. Once the new epoch's ring has been successfully completed, the ring pointers for older epochs can be removed. Since starting a new epoch may change the database, it is never done automatically, but must be controlled by an administrator.

The servers are themselves named in the data base, by *server names* (SN) that are simply full names. The value of a SN is a unique *server identifier* (SI) and the network address of the server.

The name space

This section deals with the structure of names, provisions for expanding the name space and changing its structure without destroying the usefulness of old names, and ways of safely caching the result of a name lookup.

Names

A name has two parts: the full name (FN) of a directory, and the name of an entity registered in that directory. For example, in the context of figures 1 and 2 *ANSI*/DEC/SRC/Lampson is a name for Lampson in the directory *ANSI*/DEC/SRC. This name has a Password property with the value XGZQ#$, and a Mailboxes property with the value {Zin, Cab, Ries, Pinot}. An arbitrary node has a two part name; for example, (*ANSI*/DEC/SRC, Lampson/Mailboxes). This division reflects the radically different implementation of the two parts; it also makes it simple to represent a DR as a value within its containing directory; thus (*ANSI*/DEC, SRC/DR) names the DR for *SRC* in the *ANSI*/DEC directory.

The value part is expected to be short, since the entity being named is distinguished by its first component, and each entity has only a few relatively simple properties (though some distribution lists might have hundreds of members). An entire directory should contain no more than a few hundred names and a few thousand nodes. Furthermore, a directory copy is stored on a single server; all of it can be accessed under a single lock if necessary, and there is no possibility of losing track of part of it.

In contrast, the directory tree may be very large. It can only be modified using special operations to add or delete a leaf node, or move a subtree; these operations deal with the

multiple copies which represent the directory, the multiple servers involved, and the need to avoid losing a directory or forming a cycle in the tree. Because the tree is large, it will have to evolve gradually over a long time. Finally, the unit of replication is the directory.

Growth

The basic mechanism for growth of the name space is its hierarchical structure. Each directory is a context within which names can be generated independently of what is going on in any other directory. Thus names with the prefix *ANSI*/DEC, for example *ANSI*/DEC/SRC or *ANSI*/DEC/Finance, can be assigned without any concern for what names exist with the prefix *ANSI*/IBM, such as *ANSI*/IBM/TJW or *ANSI*/IBM/Finance. Each directory can have its own administrator, and they do not have to coordinate their actions. Since new directory names can be created as easily as new entity names, the organization of the name space can readily be made broader (by adding *ANSI*/DEC/Personnel or *ANSI*/DEC/Legal) or deeper (with *ANSI*/DEC/SRC/Finance or *ANSI*/DEC/SRC/Theory).

Growth by combining existing name services, each with its own root, is a little trickier. The basic idea is obvious: add a new root, making the existing roots its children. Figure 6 illustrates the creation of a new *ANSI* root above *DEC* and *IBM*. Now the name *ANSI*/DEC/SRC/Lampson is a synonym for *DEC*/SRC/Lampson. Both these names appear at the bottom of the figure; the directory identifiers (beginning with # signs) are written explicitly.

Note the difference between a full name and a file name in a hierarchical file system. A file system name is normally relative to the working directory (part of the state of the running process), or to the root of the file system (either a constant, or also part of the process state as in Unix, although it is unusual to change the root from the default). A full name begins with the DI of the root directory for that name. For example, the meaning of the name #311/SRC/Finance is to start with the unique directory that has DI #311, look up SRC there to obtain another directory, and look up Finance there to obtain yet a third

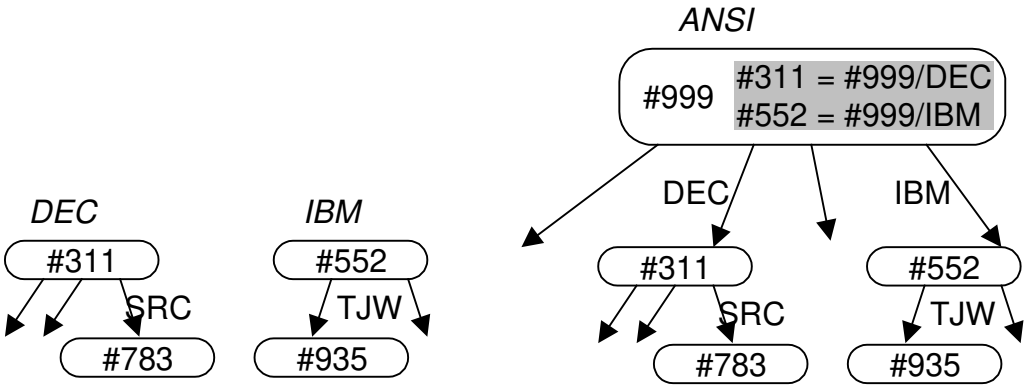

Figure 6: Growing the name space by combining two trees under a new root

directory. Thus the DIs act as names in an imaginary super-root which has all the directories as its children; an FN behaves like a file system name relative to the superroot.

There are two obvious questions: how do users type FNs, and how can a directory be found from its DI among millions of directories scattered all over the world?

The first question has the same answer as it does in a file system: a user will have a working root, which is prefixed to any name she types. Many variations are possible. For example, as in Unix the user could have a working root which is prefixed to typed names that start with `/` and a working directory which is prefixed to typed names that do not. Of course the user is also free to type an initial DI explicitly, or to define named links to various roots in his working directory.

The second question is more subtle. At any time, an instance of the name service has a single root, and there are data structures maintained by the administrative level that allow a copy of the root to be found from any server; these are discussed later. Taken without qualification, this means that only FNs beginning with the root's DI can be looked up, which is fine when the root is created first and growth occurs at the leaves. To handle the growth by combination shown in figure 6, the root keeps an ersatz super-root, in the form of a table of well-known directories that maps certain DIs into links which are FNs relative to the root. Thus in the figure the well-known DIs in *`ANSI`* (shown in gray) are `#311` and `#552`, the DIs for *`DEC`* and *`IBM`*. Now when a lookup reaches the root, it can consult the well-known table and replace the FN's DI with a path that starts at the root itself. Thus `#311` is replaced by `#999/DEC`, and hence `#311/SRC/Lampson` becomes `#999/DEC/SRC/Lampson`, which can be looked up starting at the *`ANSI`* root. When combining name services, it is prudent to make the old roots well-known in the new root, so that old names can still be looked up.

Restructuring

Sometimes what is wanted is not growth but restructuring. Suppose that DEC buys IBM. The subtree rooted in the *`IBM`* directory should be moved under the *`DEC`* directory, as shown in figure 7. Moving a subtree is the only restructuring operation; as long as it doesn't form a cycle (not allowed), it preserves the tree structure of the service.

The obvious problem with this move is that all the names that begin *`ANSI`*`/IBM` no longer work, since *`IBM`* is no longer a child of *`ANSI`*. The solution is familiar from the telephone system: when a number changes, a call to the old number elicits the response, "The number you have reached has been changed. The new number is...." Similarly, an entry in *`ANSI`* for *`IBM`*, with the link *`ANSI`*`/DEC/IBM` as its value, gives names beginning *`ANSI`*`/IBM` the same meaning they had before the takeover.

Caching

Name lookup is not likely to be especially cheap. Indeed, if the servers that store the name or its parent directories are far away in the network, lookup may be quite expensive. Hence it is very desirable for a client to be able to cache the result of a lookup for a while,

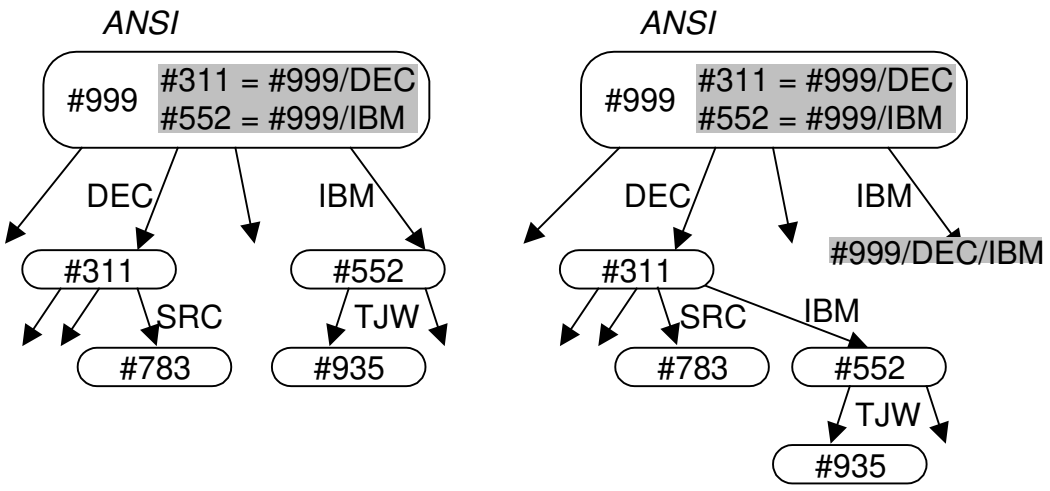

Figure 7: Restructuring a name space by moving the *IBM* node

rather than repeating it every time the value is needed. Since it is impractical for the service to keep track of clients that are doing this and notify them when there is a change, caching must be paid for either by enforcing a slow rate of change on the naming database, or by tolerating some inaccuracy in the cached information. The latter requires no work from the service, but the former does.

The enforcement mechanism is an *expiration time* (TX) on entries in the data base, and in particular on parent-child arcs in the directory tree and on links. The rule is: an arc or link may not be changed until its TX has expired, except that an arc may be deleted by a subtree move if it is replaced by a link to the moved subtree; e.g., see figure 7. With this restriction, the result of a directory lookup can be safely cached until the minimum TX of any arc or link that was followed. In figure 8, for example, the result of looking up `ANSI/DEC/SRC` is valid until 15 Sept 1985, which is the minimum of the two TX values encountered.

One important client for caching is the name service itself: directories are expected to cache their names from the root, so that a lookup which encounters a server storing the `SRC` directory need not find one that stores `ANSI` in order to look up `ANSI/DEC/SRC`. Without this mechanism, access to `ANSI` might well become a bottleneck.

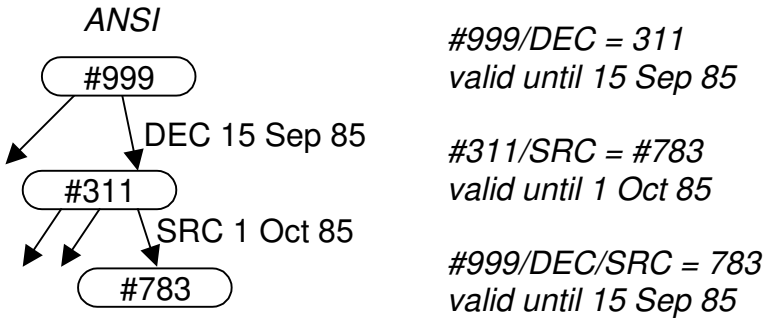

Figure 8: Using expiration times to validate caches

Authentication is another client of caching, since "key authenticates principal" is the result of a name lookup.

The name service interface

The following table gives the procedures in the interface to the name service. I have included it to give a feeling for the complexity of the system when viewed from the outside; many programming details are missing.

As you can see, the interface is based on remote procedure calls. It is organized according to four main abstractions: values and directories at the client level, and directory copies and servers at the administrative level. Procedures that create a directory have a server name argument; although this isn't logically necessary, allowing the service to choose the server would be impractical.

A few types are used in the table as abbreviations. A *path* is a sequence of labels on arcs in the value tree. A *TSpath* is a sequence of (label, time-stamp) pairs. A *value designator* (VD) is a pair (full name, path) that designates a node in the value tree. A *tree* is either a mark (present or absent), or a set of (label, tree) pairs; it is a representation of a value tree without the time-stamps.

All these procedures can also return various errors, such as "value not present."

Values and updates		
Snapshot	VD → (mark, TSpath)	give status and path
DoUpdates	(VD, tree) → time-stamp	add the updates
Enumerate	VD → set of labels	give all VD's children
GetValue	VD → tree	give all of VD's value
SetValue	(VD, tree) → time-stamp	replace VD's value
Directories		
FirstRoot	server address → DI	
NewRoot	(SN, FN, name) → DI	old root FN, called name in new root
NewD	(SN, FN) → DI	named FN
MoveSubtree	(VD, FN) → ()	give VD name FN
Directory copies		
NewDC	(SN, FN) → ()	copy of FN at SN
RemoveDC	(SN, FN) → ()	
Sweep	(FN, SN) → time-stamp	start at SN
NewEpoch	(DI, set of SN) → epoch	
Baptise	(FN, epoch, SN) → ()	add FN on SN to ring

Servers		
NewServer	() → server address	initialize a server
RemoveS	SN → ()	
SetInSN	(FN, bool) → ()	can FN point to servers
SetUpOn	(SN, SN) → ()	turn on server
CheckS	(server address, SN) → ()	verify server invariants

Specifications

The name service is specified by a collection of predicates. There are two kinds:

- Invariants that hold between the execution of any two atomic actions.
- Post-conditions that specify what is true after successful execution of a procedure call.

The pre-conditions are all true, since the service must work even if its clients are not well-behaved. Since this is a concurrent system, a post-condition usually relates input and output values of the procedure, or is true once the procedure has returned and forever thereafter.

The model of computation is arbitrary interleaving of atomic actions in separate processes. Atomic actions are the primitive computing steps. They take place entirely on one server, where the atomicity can be maintained by a local lock.

A program in the system may refer to any of the variables in the server on which it is running. It may also make remote procedure calls to other servers. The design is such that it is never necessary to hold a lock while making such a call.

A predicate may refer to any of the variables in the system, and also to *auxiliary variables* which are not part of the program. The most important auxiliary variable is the database (*DB*), which is a function DI → directory that defines the current contents of the database. Since this information is distributed over all the servers, there is no way for a program to refer to it directly. *DB* could only be determined by stopping the system and reading it out, and there is no way to do this. However, the specification defines precisely how *DB* changes as a result of procedure calls on the service, and it also defines how the results of procedure calls are related to *DB*. These definitions are completely precise, and they are also intuitive in the sense that *DB* corresponds to a natural intuition about what the database ought to be.

Another auxiliary variable used in the predicates is *root*, the DI of the root directory.

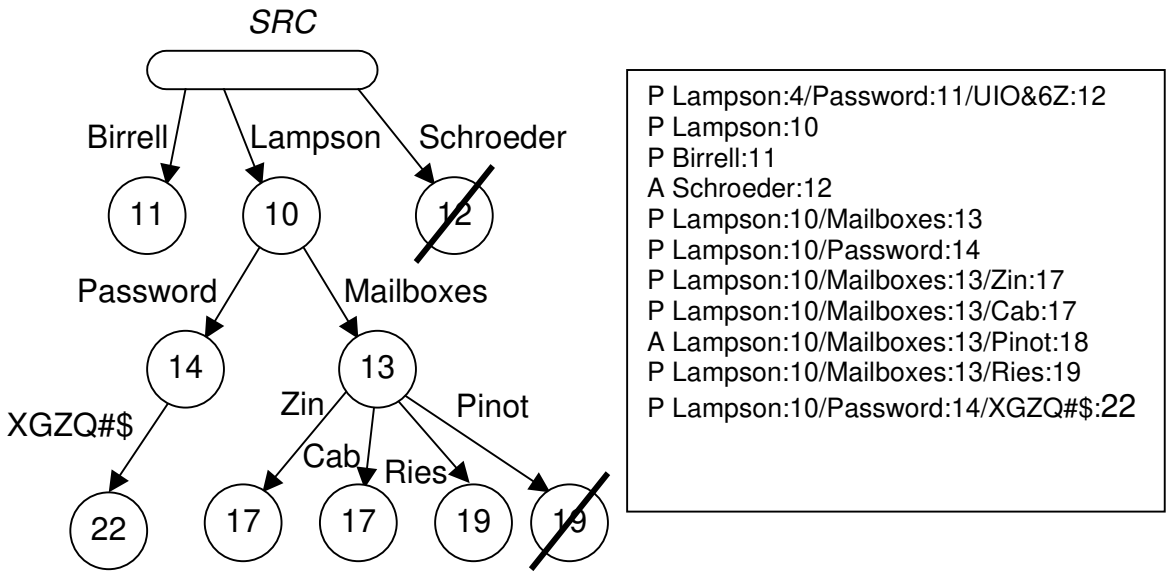

Figure 9: A directory value determined by a set of updates

Values and updates

As we saw earlier, a value is a tree with labeled arcs and time-stamped nodes; a leaf node in the tree is marked present or absent. The left side of figure 9 illustrates.

A value *v* is modified by an update operation which

- names a node by giving a path *p* in the tree, including the time-stamp at the end of each labeled arc (this is the *TSpath* defined in the previous section), and
- says whether to make that node present or absent.

This update modifies the value in the following way.

— Find the longest prefix of *p* for which *v* has arcs and nodes with labels and time-stamps matching *p*; this prefix defines a node (and hence a subtree) *v'*.

— If the prefix is all of *p*, then if the update says present do nothing; if it says absent, replace *v'* with an absent leaf node.

— Otherwise consider the next element of *p*; call it (*l, ts*).

— If there is an arc labeled *l* from *v'* to a node with time-stamp greater than *ts*, do nothing.

— Otherwise, remove any subtree of *v'* reached by an arc labeled *l*, and add the subtree defined by the rest of *p*.

This complicated definition serves to make the order of updates immaterial to the result. Why is this important?

A value is determined by the sequence of update operations that have been applied to an initial empty value. An update can be thought of as a function that takes one value into another. Suppose the update functions have the following properties:

- Total: it always makes sense to apply an update function.
- Commutative: the order in which two updates are applied does not affect the result.
- Idempotent: applying the same update twice has the same effect as applying it once.

Then it follows that the *set* of updates that have been applied uniquely defines the state of the value.

It can be shown that the updates on values defined earlier are total, commutative and idempotent. Hence a set of updates uniquely defines a value. This observation is the basis of the concurrency control scheme, as explained in the next subsection. The right side of figure 9 gives one set of updates that will produce the value on the left.

The presence of the time-stamps in *p* ensures that the update is modifying the value that the client intended. This is significant when two clients concurrently try to create the same name. The two updates will have different time-stamps, and the earlier one will lose. The fact that later modifications, e.g. to set the password, include the creation time-stamp ensures that those made by the earlier client will also lose. Without the time-stamps in *p* there would be no way to tell them apart, and the final value might be a mixture of the two sets of updates.

Directories (D)

The main post-condition for the value of a directory depends on the property of value updates established above. It makes precise the notion that the result of a name lookup depends on which updates have reached the directory copy being read.

(D1) A read operation (*Snapshot*, *Enumerate*, or *GetValue*) on a directory *d* with identifier *d.di* returns a result determined by the state of *d* after some set *S* of updates which is a subset of the updates in *DB*(*d.di*). *S* includes:

- All the updates with a time-stamp less than *d.lastSweep*, the time of the last completed sweep of *d*.
- An arbitrary subset of the updates with a time-stamp greater than *d.lastSweep*.

This is a fairly weak post-condition, since *S* is chosen non-deterministically; one might well ask why it isn't stronger. The reason is that it is sufficient for the needs of a name

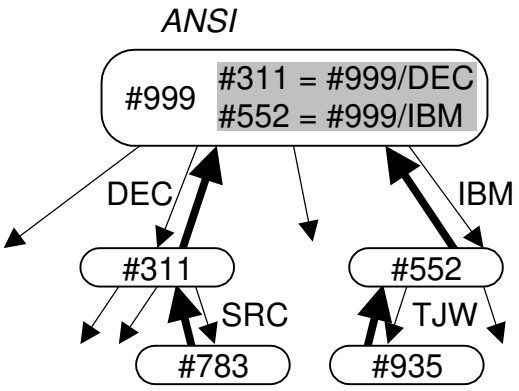

Figure 10: Back-references in the directory tree

service client, allows both read and update operations even if only one copy of the directory is accessible, and admits of a simple and robust implementation.

In addition to this condition, there are the obvious postconditions on the update operations: that they modify *DB* appropriately.

The other directory predicates have to do with the tree structure. The next one gives a condition under which looking up a FN is guaranteed to succeed.

(D2) Looking up the FN $di/n_1/.../n_k$ yields a directory if for the entire duration of the lookup operation

- *di* is the root, and each n_j is defined in the directory $di/n_1/.../n_{j-1}$ and always yields the directory reference dr_j (in spite of the non-determinism of (D1)), or
- *di* is in the root's well-known table with value $di'/n_1'/.../n_l'$, and $di'/n_1'/.../n_l'/n_1/.../n_k$ satisfies the conditions of (D2).

Of course a lookup can also succeed if some of the prefixes yield links, or if the directory structure is changing during the lookup, but this is the fundamental rule.

There are two invariants to ensure that the tree structure of the directories remains well-formed. They are based on the observation that a collection of nodes forms a tree if every node (except for one called the root) has a single *back-reference* (BR) to another node, provided the backreferences form no cycles. The BRs are the child-parent arcs in this tree. We therefore take the BRs as the primary structure defining the tree, and view the DRs as secondary. Figure 10 illustrates. Thus *MoveSubtree* simply changes the BR, and then adjusts the DR to agree.

(D3) The D's defined in *DB* form a tree rooted in root whose arcs are DRs that are the reverse of the BR backpointers.

(D4) Each DR is pointed to by a BR with a longer TX.

Note that because of (D1) the tree you see by doing lookups can shift around if the BRs change during the lookups, or faster than sweeps can keep up.

Actually (D3) is a bit oversimplified. Since *MoveSubtree* cannot be atomic, each directory has a set of BRs; *MoveSubtree* adds the new parent to the set, adjusts the DR, and finally removes the old parent. (D3) should say that there is some BR in each set which satisfies the predicate above.

In addition to these invariants, there are the obvious post-conditions on the procedures that modify the tree: *NewRoot* establishes a new value for *root*, and so forth.

Directory copies (DC)

The actual implementation of values and directories is based on directory copies stored in servers, as described earlier. The main invariant relates the contents of the copies to the value of *DB*.

(DC1) The database value of a directory *DB*(*di*), is equal to the union of the updates in all the DCs for *di*, and each DC has at least all the updates with timestamps earlier than its *lastSweep*.

From this it is easy to deduce that reading from any copy satisfies (D1). It is also not hard to show that the sweep operation, which increases *lastSweep*, maintains this invariant.

There are no new invariants for the tree. The intended implementation of procedures that change the tree is somewhat subtle, however. Since these procedures all involve changes at more than one server, they cannot be implemented atomically. Instead, each procedure makes an atomic change at one server, and then a *cleanup* procedure propagates the consequences implied by this change to the other servers involved. If some server crashes during this process, the cleanup procedure restarts. It is constructed in such a way that it maintains the invariants, and when it finally completes successfully it has completed the treechanging operation, if that is possible. The D invariants take account of the intermediate states during cleanup by allowing the back-reference to be a set, as the previous subsection points out.

There are obvious post-conditions for the procedures that add and subtract directory copies, and for the *Sweep* procedure (it leaves all the *lastSweep* values at least as late as the time-stamp that it returns).

Two invariants govern the epoch mechanism.

(DC2) There is always at least one complete ring in the set of DCs for a directory.

(DC3) There are never two complete non-intersecting rings.

The latter condition is essential to ensure that a sweep cannot complete without seeing all the updates. However, when an administrator invokes *NewEpoch* to construct a new ring,

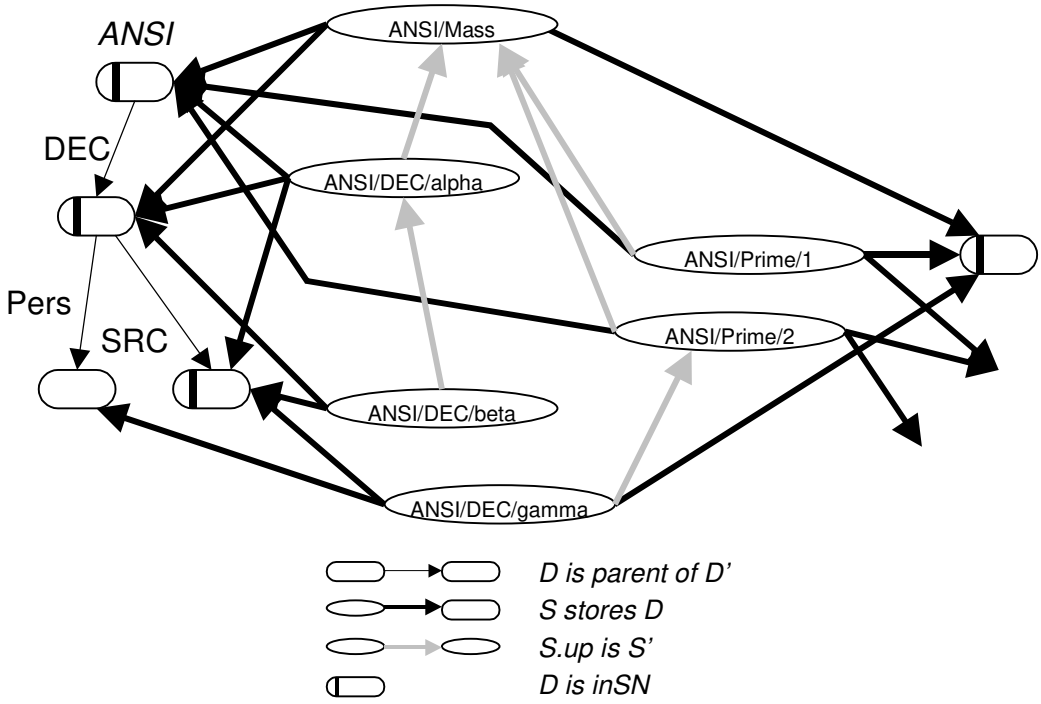

Figure 11: Distribution of directories among servers to satisfy the server invariants

(DC3) can be broken if she specifies a set of servers disjoint from those that contain an earlier ring. This could happen, for example, if a network partition separates the servers for a directory into two groups that cannot communicate. There doesn't seem to be any way to prevent this without unacceptably restricting the ability of administrators to cope with disasters. Hence, *NewEpoch* is the exception to the rule that name service interface procedures do not have pre-conditions.

Servers (S)

The invariants for servers arise from the fact that servers are identified in directory references by server names (SN), which are themselves name service names. This is very convenient, since it allows the network address of a server to be maintained using the mechanisms of the name service itself. However, it introduces the unpleasant possibility that the process of looking up a server name will result in an infinite loop: a directory needed to look up the server name may be stored only on servers whose names include that directory.

For an example, look at figure 11, ignoring the fat and gray arrows for the moment. Suppose that the *DEC* directory is stored only on the servers *ANSI*/DEC/alpha and *ANSI*/DEC/beta. Then it will be impossible to look up these server names, since in trying to get from *ANSI* to *ANSI*/DEC in order to look up alpha and beta, we must look up either *ANSI*/DEC/alpha or ANSI/DEC/beta; this leads to a loop.

The invariants that prevent this situation depend on a Boolean *inSN* in each directory. A directory cannot be part of a server name unless its *inSN* is true.

(S1) Every directory *d* on a direct path from *root* to an entry that stores a server address has *d.inSN* = true.

(S2) If *d.inSN* = true, then either *d* is the root, or a copy of *d* is stored on a server with a name shorter than any direct name of *d*. (This copy will be accessible without using *d* itself to look up its server name.)

The fat arrows in figure 11 show a placement of directories on servers satisfying (S1-2). Note that each directory with *inSN* true has a fat arrow entering from above.

This is not the whole story for servers, however. The model for client use of the name service is that the client must locate a server using some lower-level resource location facility provided by the network; Ethernet broadcast is an obvious example. From that point, however, the service should ensure that the client can do any operations that are authorized. Since some operations require access to a copy of the root, this means that it must be possible to locate a copy of the root from any server. To ensure this, each server that doesn't store the root already has an *up* pointer to another server that is closer to the root.

(S3) Every server *s* either stores the root, or *s.up* is a shorter name of another server, and *s* stores a copy of the directory for *s.up*.

The final clause ensures that we can go from the server name in the *up* pointer to the actual address of the server. (D2) ensures that following *up* pointers will terminate. The gray arrows in figure 11 show a possible set of *up* pointers. The rising fat arrows are the ones required by the final clause of (S3).

Conclusion

I have informally described a design for a global name service. Its most interesting aspects are the provisions for both stability and growth of the name space, the high availability allowed by the design, the precise specification of the non-deterministic lookup semantics, the ability to name servers using the service itself, and the methods for authentication without global trust.

References

1. Birrell, A.D. et.al. Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4 (Apr 1982), 260-274.
2. Birrell, A.D. et. al. A global authentication service without global trust. *Proc. 1986 IEEE Symposium on Security and Privacy* (Apr 1986), 223-230. IEEE Computer Society order number 716.
3. Schroeder, M.D. et. al. Experience with Grapevine. *ACM Trans. Computer Sys*. **2**, 1 (Feb. 1984), 3-23.
4. Xerox Corporation. *Clearinghouse Protocol*, XSIS 078404. Stamford, Conn., 1984.