CHAPTER THREE
NAMING

3.1. Introduction

In a distributed system, names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. Names facilitate communication and resource sharing. A name is needed to request a computer system to act upon a specific resource chosen out of many; for example, a name in the form of a URL is needed to access a specific web page. Processes cannot share particular resources managed by a computer system unless they can name them consistently. Users cannot communicate with one another via a distributed system unless they can name one another, for example, with email addresses.

Names are not the only useful means of identification: descriptive attributes are another. Sometimes clients do not know the name of the particular entity that they seek, but they do have some information that describes it. Or they may require a service and know some of its characteristics but not what entity implements it.

The *naming facility* of a distributed operating system enables users and programs to assign character-string names to objects and subsequently use these names to refer to those objects.

The *locating facility*, which is an integral part of the naming facility, maps an object's name to the object's location in a distributed system.

The naming and locating facilities jointly form a naming system that provides the users with an abstraction of an object that hides the details of how and where an object is actually located in the network. It provides a further level of abstraction when dealing with object replicas. Given an object name, it returns a set of the locations of the object's replicas.

The naming system plays a very important role in achieving the goal of

- location transparency,
- facilitating transparent migration and replication of objects,
- Object sharing.

3.2. Names, Identifiers and Addresses

The naming system is one of the most important components of a distributed operating system because it enables other services and objects to be identified and accessed in a uniform manner. This section defines and explains the fundamental terminologies and concepts associated with object naming in distributed systems.

Names

A name in a distributed system is a string of bits or characters that is used to refer to an entity. For example, file names like */etc/password*, URLs like *http://www.google.com/* and Internet domain names like www.google.com etc.

An entity can be: resources such as hosts, printers, disks, and files, explicit names such as processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.

To operate on an entity, it is necessary to access it, for which we need an access point. An access point is yet another, but special, kind of entity in a distributed system. The name of an access point is called an address. The address of an access point of an entity is also simply called an address of that entity.

An entity can offer more than one access point. As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address. Indeed, many people nowadays have several telephone numbers, each number corresponding to a point where they can be reached. In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number (i.e., the server's transport-level address).

An entity may change its access points in the course of time. For example, when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before. Likewise, when a person moves to another city or country, it is often necessary to change telephone numbers as well. In a similar fashion, changing jobs or Internet Service Providers, means changing your e-mail address.

Addresses

An address is thus just a special kind of name: it refers to an access point of an entity. Because an access point is tightly associated with an entity, it would seem convenient to use the address of an access point as a regular name for the associated entity. Nevertheless, this is hardly ever done as such naming is generally very inflexible and often human unfriendly.

For example, it is not uncommon to regularly reorganize a distributed system, so that a specific server is now running on a different host than previously. The old machine on which the server used to be running may be reassigned to a completely different server. In other words, an entity may easily change an access point, or an access point may be reassigned to a different entity. If an address is used to refer to an entity, we will have an invalid reference the instant the access point changes or is reassigned to another entity. Therefore, it is much better to let a service be known by a separate name independent of the address of the associated server.

Likewise, if an entity offers more than one access point, it is not clear which address to use as a reference. For instance, many organizations distribute their Web service across several servers. If we would use the addresses of those servers as a reference for the Web service, it is not obvious which address should be chosen as the best one. Again, a much better solution is to have a single name for the Web service independent from the addresses of the different Web servers.

These examples illustrate that a name for an entity that is independent from its addresses is often much easier and more flexible to use. Such a name is called location independent.

Identifiers

In addition to addresses, there are other types of names that deserve special treatment, such as names that are used to uniquely identify an entity. A true identifier is a name that has the following properties:

- ✓ An identifier refers to at most one entity.
- ✓ Each entity is referred by at most one identifier.
- ✓ An identifier always refers to the same entity (i.e., it is never reused).

By using identifiers, it becomes much easier to unambiguously refer to an entity. For example, assume two processes each refer to an entity by means of an identifier. To check if the processes are referring to the same entity, it is sufficient to test if the two identifiers are equal. Such a test would not be sufficient if the two processes were using regular, non-unique, non-identifying names. For example, the name "John Smith" cannot be taken as a unique reference to just a single person.

Likewise, if an address can be reassigned to a different entity, we cannot use an address as an identifier. Consider the use of telephone numbers, which are reasonably stable in the sense that a telephone number for some time refers to the same person or organization. However, using a telephone number as an identifier will not work as it can be reassigned in the course of time.

Addresses and identifiers are two important types of names that are each used for very different purposes. In many computer systems, addresses and identifiers are represented in machine-readable form only, that is, in the form of bit strings. For example, an Ethernet address is essentially a random string of 48 bits. Likewise, memory addresses are typically represented as 32-bit or 64-bit strings.

Another important type of name is that which is tailored to be used by humans, also referred to as human-friendly names. In contrast to addresses and identifiers, a human-friendly name is generally represented as a character string.

These names appear in many different forms. For example, files in UNIX systems have character-string names that can be as long as 255 characters, and which are defined entirely by the user. Similarly, DNS names are represented as relatively simple case-insensitive character strings.

3.3. Flat Naming

Above, we explained that identifiers are convenient to uniquely represent entities. In many cases, identifiers are simply random bit strings, which we conveniently refer to as unstructured or flat names. An important property of such a name is that it does not contain any information whatsoever on how to locate the access point of its associated entity. In the following, we will take a look at how flat names can be resolved, or, equivalently, how we can locate an entity when given only its identifier.

3.3.1. Simple Solutions

We first consider two simple solutions for locating an entity. Both solutions are applicable only to local-area networks. Nevertheless, in that environment, they often do the job well, making their simplicity particularly attractive.

Broadcasting and Multicasting

Consider a distributed system built on a computer network: that offers efficient broadcasting facilities. Typically, such facilities are offered by local-area networks in which all machines are connected to a single cable or the logical equivalent thereof. Also, local-area wireless networks fall into this category.

Locating an entity in such an environment is simple: a message containing the identifier of the entity is broadcast to each machine and each machine is requested to check whether it has that entity. Only the machines that can offer an access point for the entity send a reply message containing the address of that access point. This principle is used in the Internet Address Resolution Protocol (ARP) to find the data-link address of a machine when given only an IP address. In essence, a machine broadcasts a packet on the local network asking who is the owner of a given IP address. When the message arrives at a machine, the receiver checks whether it should listen to the requested IP address. If so, it sends a reply packet containing, for example, its Ethernet address.

Broadcasting becomes inefficient when the network grows. Not only is network bandwidth wasted by request messages, but, more seriously, too many hosts maybe interrupted by requests they cannot answer. One possible solution is to switch to multicasting, by which only a restricted group of hosts receives the request. For example, Ethernet networks support data-link level multicasting directly in hardware.

Multicasting can also be used to locate entities in point-to-point networks. For example, the Internet supports network-level multicasting by allowing hosts to join a specific multicast group. Such groups are identified by a multicast address.

When a host sends a message to a multicast address, the network layer provides a best-effort service to deliver that message to all group members.

A multicast address can be used as a general location service for multiple entities. For example, consider an organization where each employee has his or her own mobile computer. When such a computer connects to the locally available network, it is dynamically assigned an IP address. In addition, it joins a specific multicast group. When a process wants to locate computer A, it sends a "where is A?" request to the multicast group. If A is connected, it responds with its current IP address.

Another way to use a multicast address is to associate it with a replicated entity, and to use multicasting to locate the nearest replica. When sending a request to the multicast address, each replica responds with its current (normal) IP address.

A crude way to select the nearest replica is to choose the one whose reply comes in first.

3.3.2. Home Based Approach

The use of broadcasting and multicasting imposes scalability problems – difficult to implement efficiently in large scale networks.

A popular approach to supporting mobile entities in large-scale networks is to introduce a home location - which keeps track of the current location of an entity. Special techniques may be applied to safeguard against network or process failures. In practice, the home location is often chosen to be the place where an entity was created.

For example, Mobile IP follows Home based approach. Each mobile host uses a fixed IP address. All communication to that IP address is initially directed to the mobile host's home agent. This home agent is located on the local-area network corresponding to the network address contained in the mobile host's IP address. In the case of IPV6, it is realized as a network-layer component. Whenever the mobile host moves to another network, it requests a temporary address that it can use for communication. This care-of address is registered at the home agent. When the home agent receives a packet for the mobile host, it looks up the host's current location. If the host is on the current local network, the packet is simply forwarded. Otherwise, it is tunneled to the host's current location, that is, wrapped as data in an IP packet and sent to the care-of address. At the same time, the sender of the packet is informed of the host's current location. Note that the IP address is effectively used as an identifier for the mobile host.

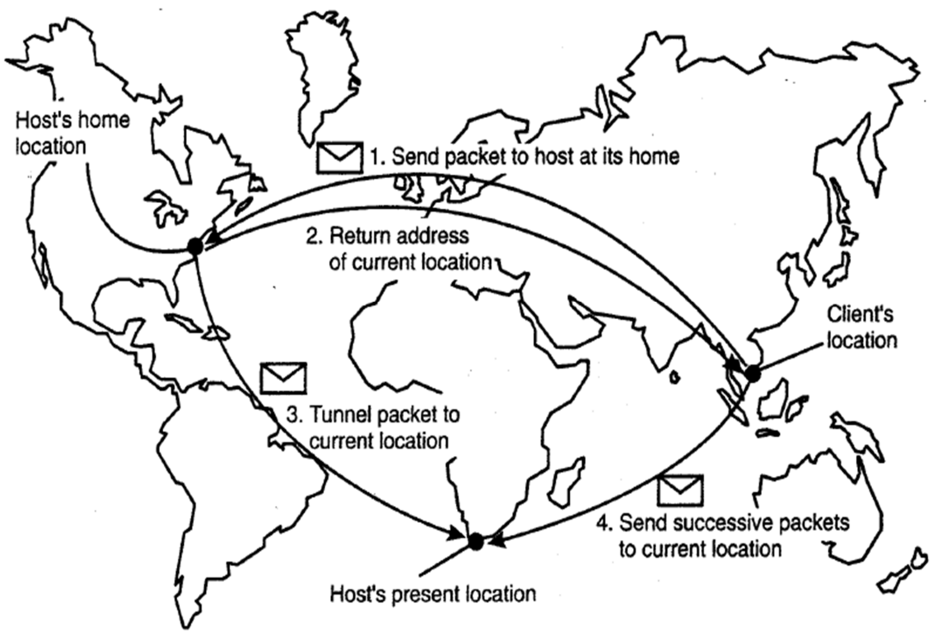

Host's home location
1. Send packet to host at its home
2. Return address of current location
Client's location
3. Tunnel packet to current location
4. Send successive packets to current location
Host's present location

Fig. 3.3.2.1. The principle of Mobile IP

Drawbacks of home based approach

- ✓ In large scale network
  - To communicate with a mobile entity, a client first has to contact the home, which may be at a completely different location than the entity itself. The result is an increase in communication latency.
- ✓ From the use of a fixed home location
  - For one thing, it must be ensured that the home location always exists. Otherwise, contacting the entity will become impossible.
  - Problems are aggravated when a long-lived entity decides to move permanently to a completely different part of the network than where its home is located.
  - In that case, it would have been better if the home could have moved along with the host.

A solution to this problem is to register the home at a traditional naming service and to let a client first look up the location of the home. Because the home location can be assumed to be relatively stable that location can be effectively cached after it has been looked up.

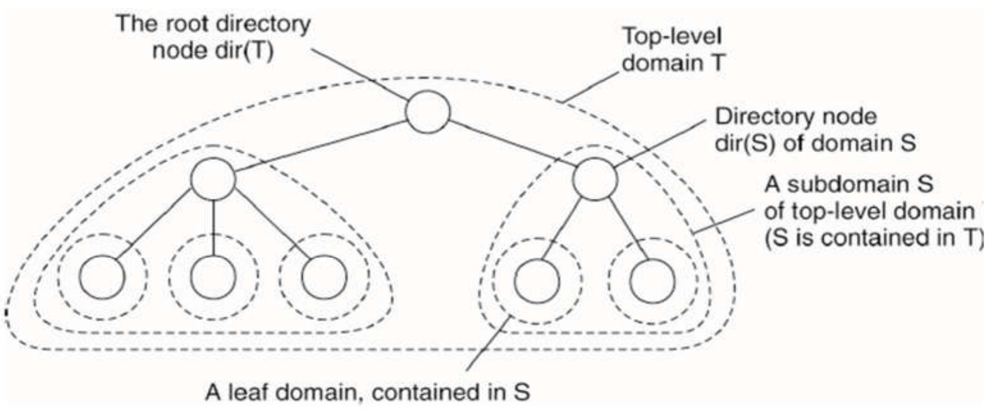
Generally, the issues with Home-Based Approaches are

- ✓ Home address has to be supported as long as entity lives
- ✓ Home address is fixed – unnecessary burden if entity permanently moves
- ✓ Poor geographical scalability (the entity may be next to the client)

3.3.3. Hierarchical Approach

In a hierarchical scheme, a network is divided into a collection of domains. There is a single top-level domain that spans the entire network. Each domain can be subdivided into multiple, smaller sub domains. A lowest-level domain, called a leaf domain, typically corresponds to a local-area network in a computer network or a cell in a mobile telephone network.

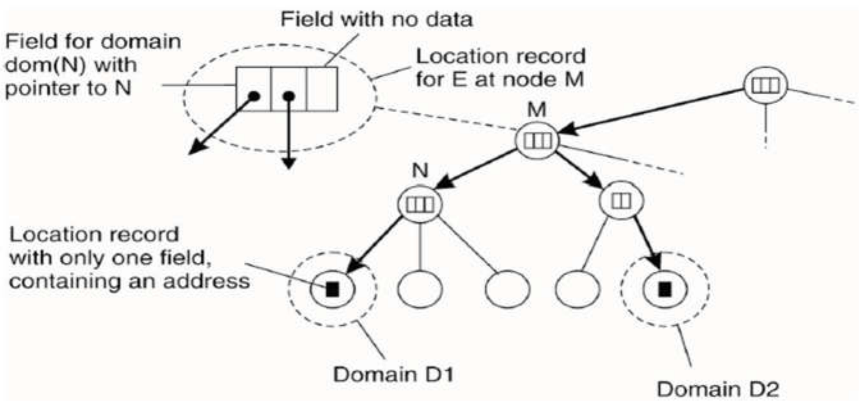
Each domain D has an associated directory node dir (D) that keeps track of the entities in that domain. This leads to a tree of directory nodes. The directory node of the top-level domain, called the root (directory) node, knows about all entities.


The root directory node dir(T)
Top-level domain T
Directory node dir(S) of domain S
A subdomain S of top-level domain (S is contained in T)
A leaf domain, contained in S

Hierarchical organization of a location service into domains, each having an associated directory node.

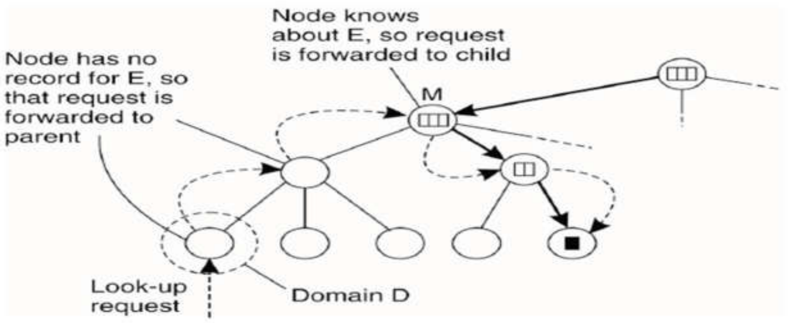
To keep track of the whereabouts of an entity, each entity currently located in a domain *D* is represented by a location record in the directory node dir (D). A location record for entity *E* in the directory node *N* for a leaf domain *D* contains the entity's current address in that domain. In contrast, the directory node *N'* for the next higher-level domain *D'* that contains *D,* will have a location record for *E* containing only a pointer to *N.* Likewise, the parent node of *N'* will store a location record for *E* containing only a pointer to *N'.* Consequently, the root node will have a location record for each entity, where each location record stores a pointer to the directory node of the next lower-level sub domain where that record's associated entity is currently located.

An entity may have multiple addresses, for example if it is replicated. If an entity has an address in leaf domain D_1 and D_2 respectively, then the directory node of the smallest domain containing both D_1 and D_2, will have two pointers, one for each sub domain containing an address.


Field with no data
Field for domain dom(N) with pointer to N
Location record for E at node M
M
N
Location record with only one field, containing an address
Domain D1
Domain D2

An example of storing information of an entity having two addresses in different leaf domains.

In hierarchical approach a look up operation proceeds as follows. A client wishing to locate an entity *E,* issues a lookup request to the directory node of the leaf domain D in which the client resides. If the directory node does not store a location record for the entity, then the entity is currently not located in D. Consequently, the node forwards the request to its parent. Note that the parent node represents a larger domain than its child. If the parent also has no location record for *E,* the lookup request is forwarded to a next higher level, and so on.


Node knows about E, so request is forwarded to child
Node has no record for E, so that request is forwarded to parent
M
Look-up request
Domain D

Looking up a location in a hierarchically organized location service.

As soon as the request reaches a directory node *M* that stores a location record for entity *E,* we know that *E* is somewhere in the domain dom(M) represented by node *M. As shown i*n the above figure, *M* is shown to store a location record containing a pointer to one of its sub domains. The lookup request is then forwarded to the directory node of that sub domain, which in turn forwards it further down the tree, until the request finally reaches a leaf node. The location record stored in the leaf node will contain the address of *E* in that leaf domain. This address can then be returned to the client that initially requested the lookup to take place.

3.3.4. Forwarding Pointers

Another popular approach to locate mobile entities is to make use of forwarding pointers. The principle is simple: when an entity moves from “A” to “B”, it leaves behind in “A” a reference to its new location at “B”. The main advantage of this approach is its simplicity:

as soon as an entity has been located, for example by using a traditional naming service, a client can look up the current address by following the chain of forwarding pointers.

There are also drawbacks. First, if no special measures are taken, a chain for a high mobile entity can become so long that locating that entity is prohibitively expensive. Second, all intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed. A third drawback is the vulnerability to broken links. As soon as any forwarding pointer is lost, the entity can no longer be reached.

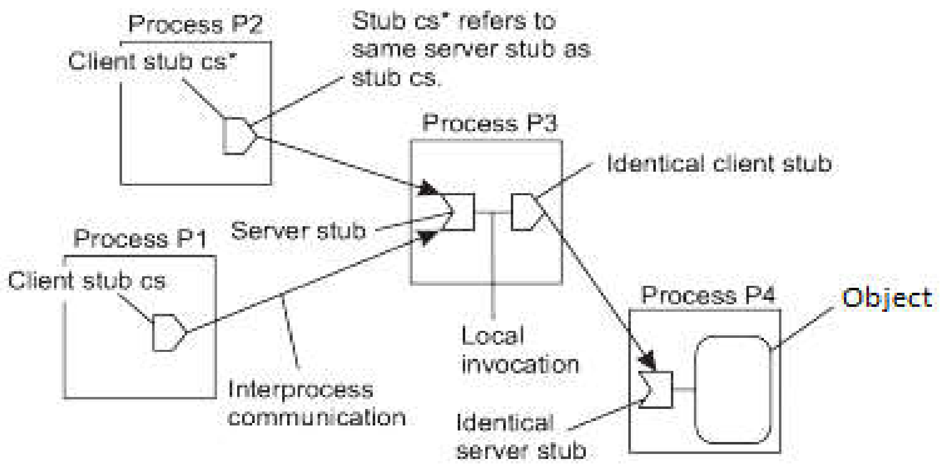


Figure 3.3.4.1: The principle of forwarding pointers using (Client stub, server stub) pairs

3.3.5. Distributed Hash Table (DHT): Chord System

Chord uses an m-bit identifier space to assign randomly chosen identifiers to nodes as well as keys to specific entities. The later can be virtually anything: files, processes, etc. The number m of bits is usually 128 or 160; depending on the hash function is used. An entity with key *k* falls under the jurisdiction of the node with the smallest identifier id >= *k*. This node is referred to as the successor of k and denoted as *succ(k)*.

The main issue in DHT-based systems is to efficiently resolve a key *k* to the address of *succ*(k). An obvious non-scalable approach is to let each node p keep track of the successor *succ*(p+1) as well as its predecessor *pre*(p). In that case, whenever a node p receives a request to resolve key k, it will simply forward the request to one of its two neighbors-unless *pred*(p)<*k*≤p in which case node p should return its own address to the process that initiated the resolution of the key k.

Instead of this linear approach toward key lookup, each Chord node maintains a finger table containing s ≤ m entities. If FT_p denotes the finger table of node p, then

$$FT_p[i]=succ(p+2^{i-1}) \bmod (128 \text{ or } 160)$$

the i-th entry points to the first node succeeding p by at least 2^{i-1}. To lookup a key *k*, the node p will then immediately forward the request to node q with index j in the p's finger table where:

q= FTp[j] ≤ k < FT_p[j+1] or

q= FTp[1] where p < k < FTp[1].

To illustrate this lookup, consider resolving k=26 from node 1 as shown in the Figure 3.5.1. First, node 1 will look up k=26 in its finger table to discover that this value is larger than FT_1[5], meaning that the request will be forwarded to node 18= FT_1[5]. Node 18, in turn, will select node 20, as FT_{18}[2] ≤ k < FT_{18}[3]. Finally, the request is forwarded from node 20 to node 21 and from there to node 28, which is responsible for k=26. At that point, the address of node 28 is returned to node 1 and the key has been resolved.

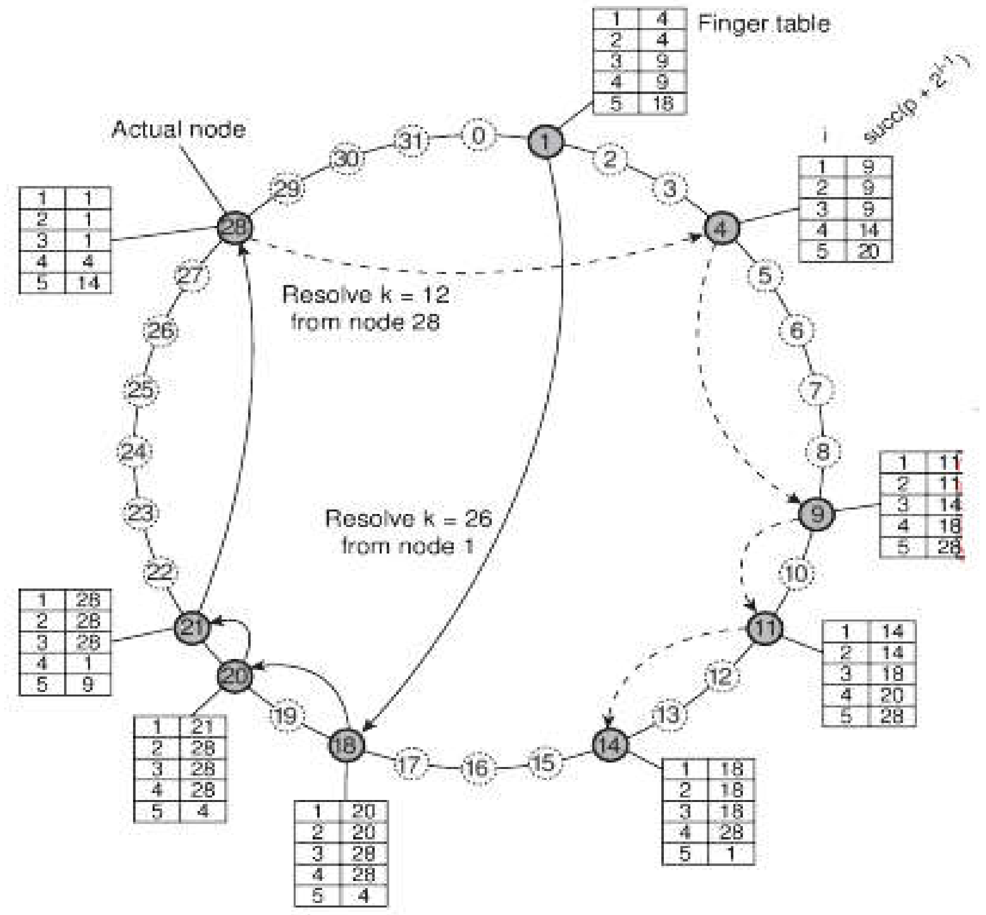


Figure 3.5.1. Resolving key 26 from node 1 and key 12 from node 28 in a Chord System

3.4. Structured Naming

Flat names are good for machines, but are generally not very convenient for humans to use. As an alternative, naming systems generally support structured names that are composed from simple, human-readable names. Not only file naming, but also host naming on the Internet follow this approach.

3.4.1. Name Space

Names are commonly organized into what is called a name space. Name spaces for structured names can be represented as a labeled, directed graph with two types of nodes. A leaf node represents a named entity and has the property that it has no outgoing edges. A leaf node generally stores information on the entity it is representing-for example, its address-so that a client can access it.

Alternatively, it can store the state of that entity, such as in the case of file systems in which a leaf node actually contains the complete file it is representing.

In contrast to a leaf node, a directory node has a number of outgoing edges, each labeled with a name, as shown in the figure shown below. Each node in a naming graph is considered as yet another entity in a distributed system, and, in particular, has an associated identifier. A directory node stores a table in which an outgoing edge is represented as a pair *(edge label, node identifier).* Such a table is called a directory table.

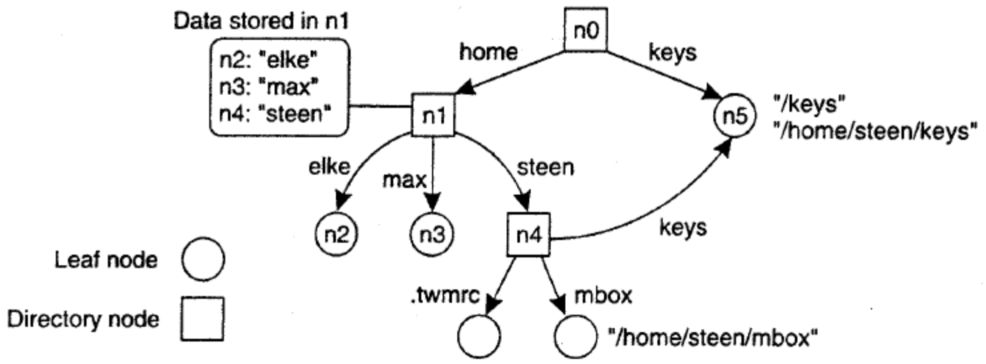


Fig.3.4.1.1 A general naming graph with a single root node

The naming graph shown in fig. 3.4.1.1 has one node, namely *n0,* which has only outgoing and no incoming edges. Such a node is called the root (node) of the naming graph. Although it is possible for a naming graph to have several root nodes, for simplicity, many naming systems have only one. Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as

N:<label-1, label-2... label-n>

where *N* refers to the first node in the path. Such a sequence is called a path name. If the first node in a path name is the root of the naming graph, it is called an absolute path name. Otherwise, it is called a relative path name.

It is important to realize that names are always organized in a name space. As a consequence, a name is always defined relative only to a directory node. In this sense, the term "absolute name" is somewhat misleading. Likewise, the difference between global and local names can often be confusing. A global name is a name that denotes the same entity, no matter where that name is used in a system.

In other words, a global name is always interpreted with respect to the same directory node. In contrast, a local name is a name whose interpretation depends on where that name is being used. Put differently, a local name is essentially a relative name whose directory in which it is contained is (implicitly) known.

This description of a naming graph comes close to what is implemented in many file systems. However, instead of writing the sequence of edge labels to represent a path name, path names in file systems are generally represented as a single string in which the labels are separated by a special separator character, such as a slash *(/).* This character is also used to indicate whether a path name is absolute. For example, in Fig. 3.4.1.1, instead of using *n0:<home, steen, mbox>,*that is, the actual path name, it is common practice to use its string representation /*home/steen/mbox.* Note also that when there are several paths that lead to the same node, that node can be represented by different path names. For example, node *n*5 in Fig. 3.4.1.1 can be referred to by */home/steenlkeys* as well as */keys.*

There are many different ways to organize a name space. As we mentioned, most name spaces have only a single root node. In many cases, a name space is also strictly hierarchical in the sense that the naming graph is organized as a tree.

This means that each node except the root has exactly one incoming edge; the root has no incoming edges. As a consequence, each node also has exactly one associated (absolute) path name.

Name space Distribution

Name spaces for a large-scale, possibly worldwide distributed system, are usually organized hierarchically. As before, assume such a name space has only a single root node. To effectively implement such a name space, it is convenient to partition it into three logical layers.

The *global layer* is formed by highest-level nodes, that is, the root node and other directory nodes logically close to the root, namely its children. Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed. Such nodes may represent organizations, or groups of organizations, for which names are stored in the name space.

The *administrational layer* is formed by directory nodes that together are managed within a single organization. A characteristic feature of the directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrational unit. For example, there may be a directory node for each department in an organization, or a directory node from which all hosts can be found. Another directory node may be used as the starting point for naming all users, and so forth. The nodes in the administrational layer are relatively stable, although changes generally occur more frequently than to nodes in the global layer.

Finally, the managerial layer consists of nodes that may typically change regularly. For example, nodes representing hosts in the local network belong to this layer. For the same reason, the layer includes nodes representing shared files such as those for libraries or binaries. Another important class of nodes includes those that represent user-defined directories and files. In contrast to the global and administrational layer, the nodes in the managerial layer are maintained not only by system administrators, but also by individual end users of a distributed system.

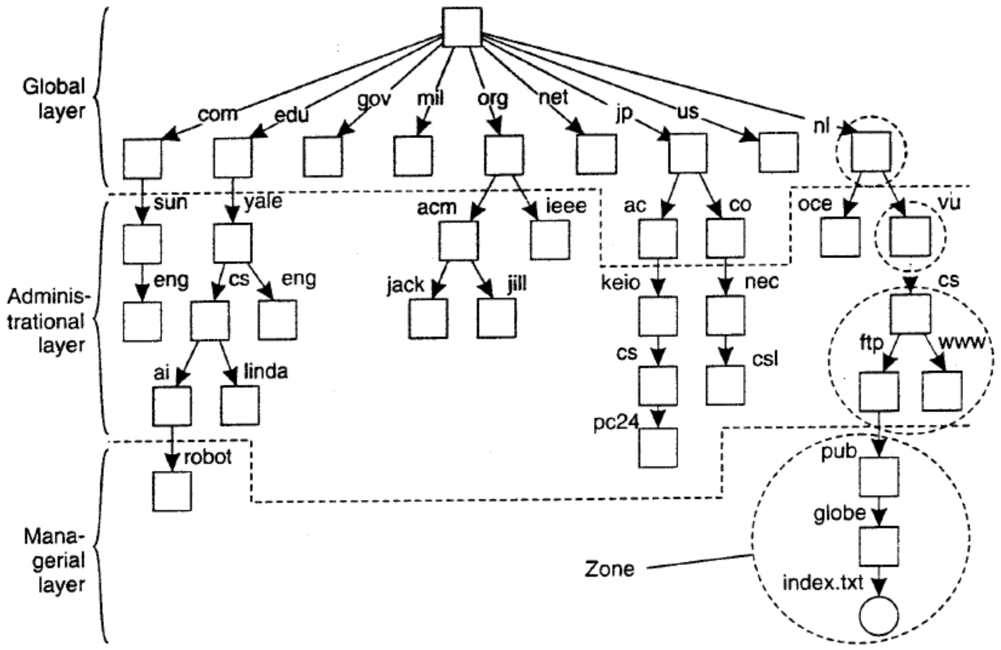


Fig.3.4.1.2 An example partitioning of the DNS name space, including Internet-accessible files, into three layers.

3.4.2. Name Resolution

Name resolution is the process of mapping an object's name to the object's properties, such as its location. Since an object's properties are stored and maintained by the authoritative name servers of that object, name resolution is basically the process of mapping an object's name to the authoritative name servers of that object.

Once an authoritative name server of the object has been located, operations can be invoked to read or update the object's properties.

Each name agent in a distributed system knows about at least one name server a prior.

To get a name resolved, a client first contacts its name agent, which in turn contacts a known name server, which may in turn contact other name servers.

Implementation of Name Resolution

The distribution of a name space across multiple name servers affects the implementation of name resolution. To explain the implementation of name resolution in large-scale name services, we assume for the moment that name servers are not replicated and that no client-side caches are used. Each client has access to a local name resolver, which is responsible for ensuring that the name resolution process is carried out.

Referring to Fig. 3.4.1.2, assume the (absolute) path name *root: «nl, vu, cs, ftp, pub, globe, index.html>* is to be resolved. Using a URL notation, this path name would correspond to *ftp://ftp.cs.vu.nl/pub/globe/index.html.* There are now two ways to implement name resolution.

In *iterative name resolution*, a name resolver hands over the complete name to the root name server. It is assumed that the address where the root server can be contacted is well known. The root server will resolve the path name as far as it can, and return the result to the client. In our example, the root server can resolve only the label *nl,* for which it will return the address of the associated name server.

At that point, the client passes the remaining path name (i.e., *nl: <vu, cs, ftp,pub, globe, index.html>* to that name server. This server can resolve only the label *vu,* and returns the address of the associated name server, along with the remaining path name *vu:<cs, ftp, pub, globe, index.html>.*

The client's name resolver will then contact this next name server, which responds by resolving the label *cs,* and subsequently also *ftp,* returning the address of the FTP server along with the path name *ftp:<pub, globe, index.html>*. The client then contacts the FTP server, requesting it to resolve the last part of the original path name. The FTP server will subsequently resolve the labels *pub, globe,* and *index.html,* and transfer the requested file (in this case using FTP). This process of iterative name resolution is shown in Fig. 3.4.2.1. (The notation #<*cs*> is used to indicate the address of the server responsible for handling the node referred to by <*cs*>.*)*

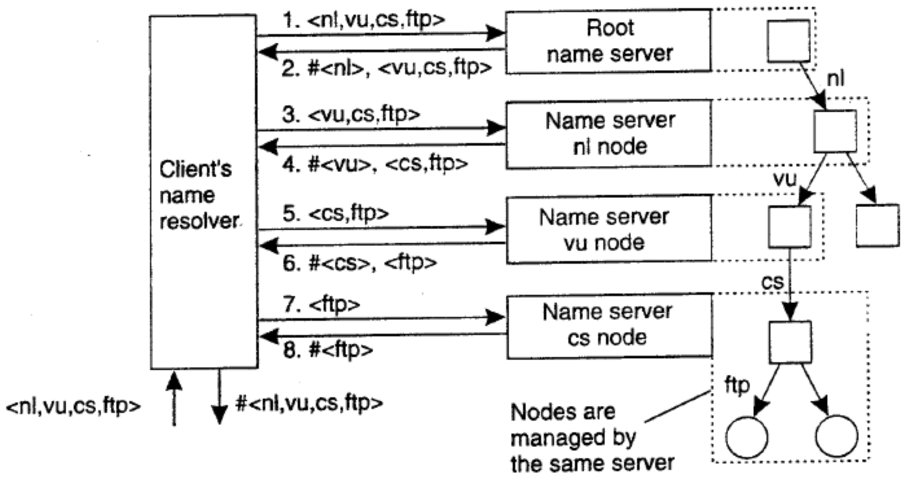


Fig. 3.4.2.1 Principle of Iterative Name Resolution

In practice, the last step, namely contacting the FTP server and requesting it to transfer the file with path name *ftp:<pub, globe, index.himl>,* is carried out separately by the client process. In other words, the client would normally hand only the path name *root: <nl, vu, cs, ftp>* to the name resolver, from which it would expect the address where it can contact the *FTP* server, as is also shown in Fig. 3.4.2.1.

An alternative to iterative name resolution is to use *recursion* during name resolution. Instead of returning each intermediate result back to the client's name resolver, with recursive name resolution, a name server passes the result to the next name server it finds. So, for example, when the root name server finds the address of the name server implementing the node named *nl,* it requests that name server to resolve the path name *nl:<vu, cs, ftp, pub, globe, index.html>*. Using recursive name resolution as well, this next server will resolve the complete path and eventually return the file *index.html* to the root server, which, in turn, will pass that file to the client's name resolver.

Recursive name resolution is shown in Fig. 3.4.2.2. As in iterative name resolution, the last resolution step (contacting the FTP server and asking it to transfer the indicated file) is generally carried out as a separate process by the client.

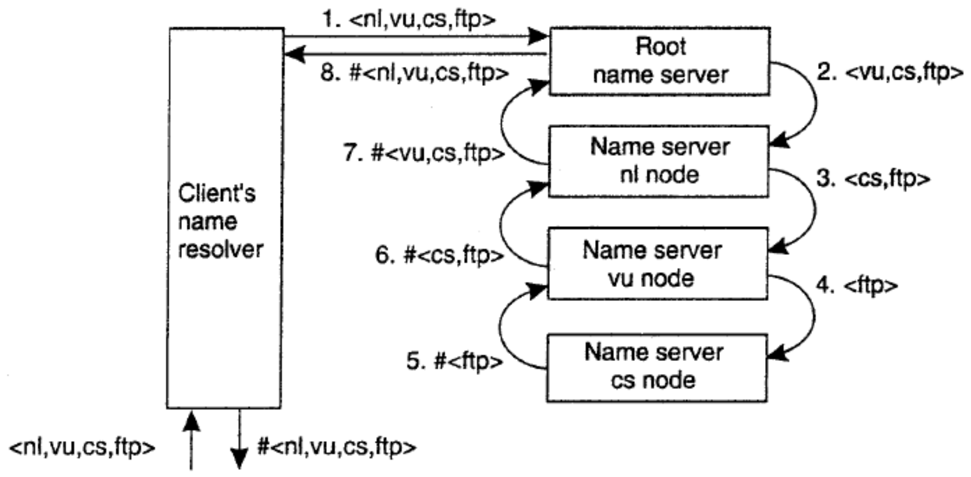


Fig. 3.4.2.2 Principle of recursive Name Resolution

The main drawback of recursive name resolution is that it puts a higher performance demand on each name server. Basically, a name server is required to handle the complete resolution of a path name, although it may do so in cooperation with other name servers. This additional burden is generally so high that name servers in the global layer of a name space support only iterative name resolution.

There are two important advantages to recursive name resolution. The first advantage is that caching results is more effective compared to iterative name resolution. The second advantage is that communication costs may be reduced.

3.4.3.Name Services and the Domain Name System

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming *contexts*: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name.

General name service requirements

Name services were originally quite simple, since they were designed only to meet the need to bind names to addresses in a single management domain, corresponding to a single LAN or WAN. The interconnection of networks and the increased scale of distributed systems have produced a much larger name-mapping problem.

Grapevine was one of the earliest extensible, multi-domain name services. It was designed to be scalable in the number of names and the load of requests that it could handle.

The Global Name Service, developed at the Digital Equipment Corporation Systems Research Center is a descendant of Grapevine.

Two examples of name services that have concentrated on the goal of scalability to large numbers of objects such as documents are the Globe name service [van Steen *et al.*1998] and the Handle System [www.handle.net]. Far more familiar is the Internet Domain Name System (DNS), which names computers (and other entities) across the Internet.

3.4.4. Domain Name System

The Domain Name System is a name service design whose main naming database is used across the Internet. The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes – and what we have referred to in this chapter as naming domains are called simply *domains* in the DNS.

Thousands of servers, installed in many different locations, provide the services we use daily over the Internet. Each of these servers is assigned a unique IP address that identifies it on the local network where it is connected.

It would be impossible to remember all of the IP addresses for all of the servers hosting services on the Internet. Instead, there is an easier way to locate servers by associating a name with an IP address.

The Domain Name System (DNS) provides a way for hosts to use this name to request the IP address of a specific server.

The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

- ✓ *com* – Commercial organizations
- ✓ *edu* – Universities and other educational institutions
- ✓ *gov* – US governmental agencies
- ✓ *mil* – US military organizations

- ✓ *net* – Major network support centres
- ✓ *org* – Organizations not mentioned above
- ✓ *int* – International organizations

New top-level domains such as *biz* and *mobi* have been added since the early 2000s. A full list of current generic domain names is available from the Internet Assigned Numbers Authority [www.iana.org I].

In addition, every country has its own domains:

- ✓ *us* – United States
- ✓ *uk* – United Kingdom
- ✓ *fr* – France
- ✓ *et* – Ethiopia etc.

A DNS server contains a table that associates hostnames in a domain with corresponding IP addresses. When a client has the name of server, such as a web server, but needs to find the IP address, it sends a request to the DNS server on port 53. The client uses the IP address of the DNS server configured in the DNS settings of the host's IP configuration.

When the DNS server receives the request, it checks its table to determine the IP address associated with that web server. If the local DNS server does not have an entry for the requested name, it queries another DNS server within the domain. When the DNS server learns the IP address, that information is sent back to the client. If the DNS server cannot determine the IP address, the request will time out and the client will not be able to communicate with the web server. Client software works with the DNS protocol to obtain IP addresses in a way that is transparent to the user.

3.5. Attribute-Based Naming

As more information is being made available, it becomes important to effectively search for entities. This approach requires that a user can provide merely a description of what he is looking for.

There are many ways in which descriptions can be provided, but a popular one in distributed systems is to describe an entity in terms of (attribute, value) pairs, generally referred to as attribute-based naming. In this approach, an entity is assumed to have an associated collection of attributes. Each attribute says something about that entity. By specifying which values a specific attribute should have, a user essentially constrains the set of entities that he is interested in. It is up to the naming system to return one or more entities that meet the user's description. Attribute-based naming systems are also known as directory services, whereas systems that support structured naming are generally called naming systems. With directory services, entities have a set of associated attributes that can be used for searching. In some cases, the choice of attributes can be relatively simple. For example, in an e-mail system, message can be tagged with attributes for the sender, recipient, subject and so on.

Further reading assignments

1. How nodes join and leave in a Chord system?
2. Other DHT protocols: Gnutella , Tapestry, Kademlia, Pastry
3. How Skype works?
4. Resource Description Framework (RDF) in Distributed System
5. What is LDAP and how it works?