The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Week 9a (13s1)

Ihor Kuz, Felix Rauch, Manuel M. T. Chakravarty & Gernot Heiser

Naming

Most computer systems (in particular operating systems) manage wide collections of *entities* (such as, files, users, hosts, networks, and so on). These entities are referred to by users of the system and other entities by various kinds of names. Examples of names in UNIX systems include the following:

- Files: `/boot/vmlinuz`, `~/lectures/DS/notes/tex/naming.tex`
- Processes: `1`, `14293`
- Devices: `/dev/hda`, `/dev/ttyS1`
- Users: `chak`, `cs9243`

For largely historical reasons, different entities are often named using different naming schemes. We say that they exist in different *name spaces.* From time to time a new system design attempts to integrate a variety of entities into a homogeneous name space, and then also attempts to provide a uniform interface to these entities. For example, a central concept of UNIX systems is the uniform treatment of files, devices, sockets, and so on. Some systems also introduce a `/proc` file system, which maps processes to names in the file system and supports access to process information through this file interface. In addition, Linux provides access to a variety of kernel data structures via the `/proc` file system. The systems *Plan 9* [ATT93] and *Inferno* go even further and are designed according to the concept that "all resources are named and accessed like files in a forest of hierarchical file systems."

Basic Concepts

A *name* is the fundamental concept underlying naming. We define a name as a string of bits or characters that is used to refer to an entity. An entity in this case is any resource, user, process, etc. in the system. Entities are accessed by performing operations on them, the operations are performed at an entity's *access point.* An access point is also referred to by a name, we call an access point's name an *address.* Entities may have multiple access points and may therefore have multiple addresses. Furthermore an entity's access points may change over time (that is an entity may get new access points or lose existing ones), which means that the set of an entity's addresses may also change.

We distinguish between a number of different kinds of names. A *pure name*[1] is a name that consists of an uninterpreted bit pattern that does not encode any of the named entity's attributes. A *nonpure* name, on the other hand, does encode entity attributes (such as an access point address) in the name. An *identifier* is a name that uniquely identifies an entity. An identifier refers to at most one entity and an entity is referred to by at most one identifier. Furthermore an identifier can never be reused, so that it will always refer to the same entity. Identifiers allow for easy comparison of entities; if two entities have the same identifier then they are the same entity. Pure names that are also identifiers are called *pure identifiers.* *Location independent* names are names that are independent of an entity's address. They remain valid even if an entity moves or otherwise changes its address. Note that pure names are always location independent, though location independent names do not have to be pure names.

[1] The distinction between pure and nonpure names is due to Needham [Nee93]

System Names Versus Human Names

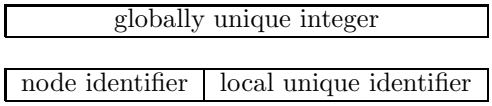
Related to the purity of names is the distinction between *system-oriented* and *human-oriented* names. Human-oriented names are usually chosen for their mnemonic value, whereas system-oriented names are a means for efficient access to, and identification of, objects.

Taking into account the desire for transparency human-oriented names would ideally be pure. In contrast, system-oriented names are often nonpure which speeds up access to repeatedly used object attributes. We can characterise these two kinds of names as follows:

- System-oriented names are usually *fixed size numerals* (or a collection thereof); thus they are easy to store, compare, and manipulate, but difficult for the user to remember.
- Human-oriented names are usually *variable-length strings,* often with structure; thus they are easy for humans to remember, but expensive to process by machines.

System-Oriented Names

As mentioned, system-oriented names are usually implemented as one or more fixed-sized numerals to facilitate efficient handling. Moreover, they typically need to be unique identifiers and may be sparse to convey access rights (e.g., capabilities). Depending on whether they are globally or locally unique, we also call them *structured* or *unstructured*. These are two examples of how structured and unstructured names may be implemented:



The structuring may be over multiple levels. Note that a structured name is not pure.

Global uniqueness without further mechanism requires a centralised generator with the usual drawbacks regarding scalability and reliability. In contrast, distributed generation without excessive communication usually leads to structured names. For example, a globally unique structured name can be constructed by combining the local time with a locally unique identifier. Both values can be generated locally and do not require any communication.

Human-Oriented Names

In many systems, the most important attribute bound to a human-oriented name is the system-oriented name of the object. All further information about the entity is obtained via the system-oriented name. This enables the system to perform the usually costly resolution of the human-oriented name just once and implement all further operations on the basis of the system-oriented name (which is more efficient to handle). Often a whole set of human-oriented names is mapped to a single system-oriented name (symbolic links, relative addressing, and so on).

As an example of all this, consider the naming of files in UNIX. A pathname is a human-oriented name that, by means of the directory structure of the file system, can be resolved to an inode number, which is a system-oriented name. All attributes of a file are accessible via the inode (i.e., the system-oriented name). By virtue of symbolic and hard links multiple human-oriented names may refer to the same inode, which makes equality testing of files merely by their human-oriented name impossible.

The design space for human-oriented names is considerably wider than that for system-oriented names. As such naming systems for human-oriented names usually require considerably greater implementation effort.

Name Spaces

Names are grouped and organised into *name spaces*. A structured name space is represented as a labeled directed graph, with two types of nodes. A *leaf node* represents a named entity and stores

information about the entity. This information could include the entity itself, or a reference to the entity (e.g., an address). A *directory node* (also called a *context*) is an inner node and does not represent any single entity. Instead it stores a *directory table*, containing $(node - id, edge - label)$ pairs, that describes the node's children. A leaf node only has incoming edges, while a directory node has both incoming and outgoing edges. A third kind of node, a *root node* is a directory node with only outgoing edges.

A structured name space can be strictly hierarchical or can form a directed acyclic graph (DAG). In a strictly hierarchical name space a node will only have one incoming edge. In a DAG name space any node can have multiple incoming edges. It is also possible to have name spaces with multiple root nodes. Scalable systems usually use hierarchically structured name spaces.

A sequence of edge labels leading from one node to another is called a *path name*. A path name is used to refer to a node in the graph. An *absolute path name* always starts from a root node. A *relative* path name is any path name that does not start at the root node. In Figure 1 the absolute path name that corresponds to the leftmost branch is `<home,ikuz,cs9243_lectures>`. The path `<ikuz,cs9243_lectures>`, on the other hand, represents a relative path name.

Many name spaces support aliasing, in which case an entity may be reachable by multiple paths from a root node and will therefore be named by numerous path names. There are two types of alias. A *hard link* is when there are two or more paths that directly lead to that entity. A *soft link*, on the other hand, occurs when a leaf node holds a pathname that refers to another node. In this case the leaf node implicitly refers to the file named by the pathname. Figure 1 shows an example of a name space with both a hard link (the solid arrow from d3 to n0) and a soft link (the dashed arrow from n1 to n2).

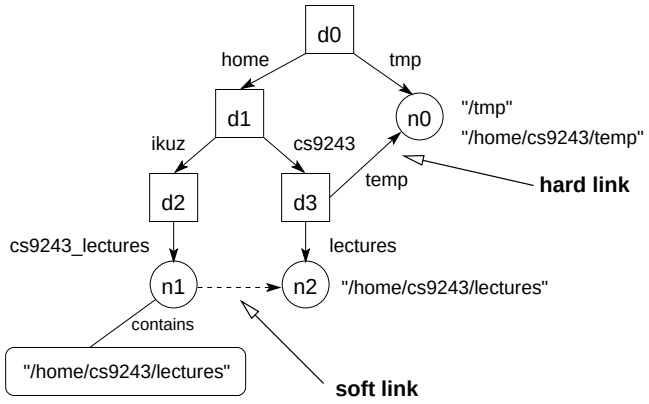


Figure 1: An example of a name space with aliasing

Ideally we would have a global, homogeneous name space that contains names for all entities used. However, we are often faced with the situation where we already have a collection of name spaces that have to be combined into a larger name space. One approach is to simply create a new name that combines names from the other name spaces. For example, a Web URL

```
http://www.cse.unsw.edu.au/~cs9243/naming-slides.ps
```

globalises the local name `~cs9243/naming-slides.ps` by adding the context `www.cse.unsw.edu.au`. Unfortunately, this approach often compromises location transparency—as is the case in the example of URLs.

Another example of the composition of name spaces is mounting a name space onto a *mount point* in a different (external) name space. This approach is often applied to merge file systems (e.g., mounting a remote file system onto a local mount point). In terms of a name space graph, mounting requires one directory node to contain information about another directory node in the external name space. This is similar to the concept of soft linking, except that in this case the link is to a node outside of the name space. The information contained in the mount point node must, therefore, include information about where to find the external name space.

Name Resolution

The process of determining what entity a name refers to is called *name resolution*. Resolving a name[2] results in a reference to the entity that the name refers to. Resolving a name in a name space often results in a reference to the node that the name refers to. Path name resolution is a process that starts with the resolution of the first element in the path name, and ends with resolution of the last element in the name. There are two approaches to this process, *iterative resolution* and *recursive resolution*.

In iterative resolution the *resolver* contacts each node directly to resolve each individual element of the path name. In recursive resolution the resolver only contacts the first node and asks it to resolve the name. This node looks up the node referred to by first element of the name and then passes the rest of the name on to that node. The process is repeated until the last element is resolved after which the result is returned back through the nodes to the resolver.

A problem with name resolution is how to determine which node to start resolution at. Knowing how and where to start name resolution is referred to as the *closure mechanism*. One approach is to keep an external reference (e.g., in a file) to the root node of the name space. Another approach is to keep a reference to the 'current' directory node for dealing with relative names. Note that the actual closure mechanism is always implicit, that is it is never explicitly defined in a name. The reason for this is that if a closure mechanism was defined in a name there would have to be a way to resolve the name used for that closure mechanism. This would require the use of a closure mechanism to bootstrap the original closure mechanism. Because this could be repeated indefinitely, at a certain point an implicit mechanism will always be required.

Naming Service

A *naming service* is a service that provides access to a name space allowing clients to perform operations on the name space. These operations include adding and removing directory or leaf nodes, modifying the contents of nodes and looking up names. The naming service is implemented by name servers. Name resolution is performed on behalf of clients by resolvers. A resolver can be implemented by the client itself, in the kernel, by the name server, or as a separate service.

Distributed Naming Service

As with most other system services, naming becomes more involved in a distributed environment. A distributed naming service is implemented using multiple name servers over which the name space is partitioned and/or replicated. The goal of a distributed naming service is to distribute both the management and name resolution load over these name servers.

Before discussing implementation aspects of distributed naming services it is useful to split a name space up into several layers according to the role the nodes play in the name space. These layers help to determine how and where to partition and replicate that part of the name space. The highest level nodes belong to the *global layer*. A main characteristic of nodes in this layer is that they are stable, meaning that they do not change much. As such, replicating these nodes is relatively easy because consistency does not cause much of a problem. The next layer is the *administrational layer*. The nodes in this layer generally represent a part of the name space that is associated to a single organisational entity (e.g., a company or a university). They are relatively stable (but not as stable as the nodes in the global layer). Finally the lowest layer is the *managerial layer*. This layer sees much change. Nodes may be added or removed as well as have their contents modified. The nodes in the top layers generally see the most traffic and, therefore, require more effort to keep their performance at an acceptable level.

Typically, a client does not directly converse with a name server, but delegates this to a local *resolver* that may use caching to improve performance. Each of the name servers stores one or more naming contexts, some of which may be replicated. We call the name servers storing attributes of an object this object's *authoritative name servers.*

[2]also referred to as *looking up a name*

Directory nodes are the smallest unit of distribution and replication of a name space. If they are all on one host, we have one central server, which is simple, but does not scale and does not provide fault tolerance. Alternatively, there can be multiple copies of the whole name space, which is called *full replication.* Again, this is simple and access may be fast. However, the replicas will have to be kept consistent and this may become a bottleneck as the system grows.

In the case of a hierarchical name space, partial subtrees (often called *zones*) may be maintained by a single server. In the case of the Internet Domain Name Service (DNS), this distribution also matches the physical distribution of the network. Each zone is associated with a name prefix that leads from the root to the zone. Now, each node maintains a prefix table (essentially, a hint cache for name servers corresponding to zones) and, given a name, the server corresponding to the zone with the longest matching prefix is contacted. If it is not the authoritative name server, the next zone's prefix is broadcast to obtain the corresponding name server (and update the prefix table). As an alternative to broadcasting, the contacted name server may be able to provide the address of the authoritative name server for this zone. This scheme can be efficiently implemented, as the prefix table can be relatively small and, on average, only a small number of messages are needed for name resolution. Consistency of the prefix table is checked *on use,* which removes the need for explicit update messages.

For smaller systems, a simpler structure-free distribution scheme may be used. In this scheme contexts can be freely placed on the available name servers (usually, however, some distribution policy is in place). Name resolution starts at the root and has to traverse the complete resolution chain of contexts. This is easy to reconfigure and, for example, used in the standard naming service of CORBA.

Implementation of Naming Services

In the following, we consider a number of issues that must be addressed by implementations of name services. First, a starting point for name resolution has to be fixed. This essentially means that the resolver must have a list of name servers that it can contact. This list will usually not include the root name server to avoid overloading it. Instead, physically close servers are normally chosen. For example, in the BIND (Berkeley Internet Name Domain) implementation of DNS, the resolver is implemented as a library linked to the client program. It expects the file `/etc/resolv.conf` to contain a list of name servers. Moreover, it facilitates relative naming in form of the `search` option.

Name Caches

Name resolution is expensive. For example, studies found that a large proportion of UNIX system calls (and network traffic in distributed systems) is due to name-mapping operations. Thus, caching of the results of name resolution on the client is attractive:

- High degree of locality of name lookup; thus, a reasonably sized name cache can give good hit ratio.
- Slow update of name information database; thus, the cost for maintaining consistency is low.
- On-use consistency of cached information is possible; thus, no invalidation on update: stale entries are detected on use.

There are three types of name caches:

- *Directory cache:* directory node data is cached. Directory caches are normally used with iterative name resolution. They require large caches, but are useful for directory listings etc.
- *Prefix cache:* path name prefix and zone information is cached. Prefix caching is unsuitable with structure-free context distribution.

- *Full-name cache:* full path name information is cached. Full-name caching is mostly used with structure-free context distribution and tends to require larger cache sizes than prefix caches.

A name cache can be implemented as a process-local cache, which lives in the address space of the client process. Such a cache does not need many resources, as it typically will be small in size, but much of the information may be duplicated in other processes. More seriously, it is a short-lived cache and incurs a high rate of start-up misses, unless a scheme such as *cache inheritance* is used, which propagates cache information from parent to child processes. The alternative is a kernel cache, which avoids duplicate entries and excessive start-up misses, but access to a kernel cache is slower and it takes up valuable kernel memory. Alternatively, a shared cache can be located in a user-space cache process that is utilised by clients directly or by redirection of queries via the kernel (the latter is used in the CODA file system). Some UNIXvariants use a tool called "name server cache daemon" (nscd) as a user-space cache process.

Example: Domain Name System (DNS)

Information about DNS, the main concepts, the model, and implementation details can be found in RFCs 1034 [Moc87a] and 1035 [Moc87b].

Attribute-Based Naming

Whereas names as described above encode at most one attribute of the named entity (e.g., a domain name encodes the entity's administrative or geographical location) in *attribute-based naming* an entity's name is composed of multiple attributes. An example of an attribute-based name is given below:

```
/C=AU/O=UNSW/OU=CSE/CN=WWW Server/Hardware=Sparc/OS=Solaris/Server=Apache
```

The name not only encodes the location of the entity (`/C=AU/O=UNSW/OU=CSE`, where `C` is the attribute *country*, `O` is *organisation*, `OU` is *organisational unit* — these are standard attributes in X.500 and LDAP), it also identifies it as a Web server, and provides information about the hardware that it runs on, the operating system running on it, and the software used. Although an entity's attribute-based name contains information about all attributes, it is common to also define a *distinguished name (DN)*, which consists of a subset of the attributes and is sufficient to uniquely identify the entity.

In attribute-based naming systems the names are stored in *directories*, and each distinguished name refers to a *directory entry*. Attribute-based naming services are normally called *directory services*. Similar to a naming service, a directory service implements a name space that can be flat or hierarchical. With a hierarchical name space its structure mirrors the structure of distinguished names.

The structure of the name space (i.e., the naming graph) is defined by a *directory information tree (DIT)*. The actual contents of the directory (that is the collection of all directory entries) are stored in the *directory information base (DIB)*.

Directory Service

A directory service implements all the operations that a naming service does, but it also adds a `search` operation that allows clients to search for entities with particular attributes. A search can use partial knowledge (that is, a search does not have to be based on all of an entity's attributes) and it does not have to include attributes that form part of a distinguished name. Thus, given a directory service that stores the entity named in the previous example, a search for all entities that have Solaris as their operating system would return a list of directory entries that contain the `OS=Solaris` attribute. This ability to search based on attributes is one of the key properties that distinguishes a directory service from a naming service.

Distributed Directory Service

The directory service is implemented by a directory server. As with naming services, directory servers can be centralised or distributed. Even more than name services, centralised directory services run the risk of becoming overloaded, thus distributed implementations are preferable when scalability is required. Distributed implementations also increase the reliability of the service.

As in naming services, the directory service can be partitioned or replicated (or partitioned *and* replicated). Partitioning of directory services follows the structure of the DIT, with the tree being split up over available directory servers. Generally the nodes are partitioned so that administratively or geographically related parts of the DIT are placed on the same servers. Figure 2 shows an example of the administrative partitioning of a DIT.

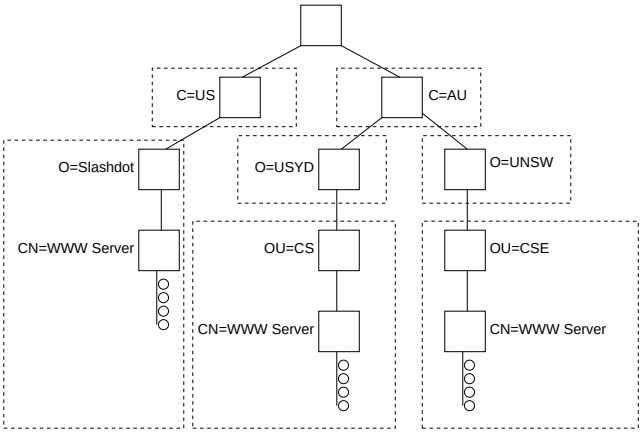


Figure 2: A partitioned DIT

Replication in a directory service involves either replicating the whole directory, or replicating individual partitions. This replication is generally more sophisticated than in naming services, and as a result a distributed directory service usually provides both read-only and read/write replicas. Furthermore, caching (e.g., caching of query results) is also used to improve performance.

Lookup in a distributed directory service is similar to lookup in a distributed naming service, it can be done iteratively (called *referral*) and recursively (called *chaining*). Search operations are also handled using referral or chaining.

Because a search can be performed on any attributes, performing a search may require the examination of every directory entry to find those containing the desired attributes. In a distributed directory service, this would require that all directory servers be contacted and requested to perform the search locally. As this is inherently unscalable and incurs a high performance penalty it is necessary for users to reduce the scope of searches as much as possible, for example, by specifying a limited part of the directory tree in which to search. Another approach to increase the performance for commonly performed searches is to keep a *catalog* at each directory server. The catalog contains a subset of each directory entry in the DIB. Catalog entries generally contain the distinguished attributes and some of the most searched for attributes. This way a search can easily be resolved by searching through the local catalog and finding all entries that fulfill the search criteria. It is necessary to tune the set of attributes stored in the catalog so that the catalog remains effective, but does not become too large.

Example: X.500 and LDAP

X.500 and LDAP are examples of widely used directory services. An overview of X.500 and LDAP can be found in [How95]. More technical details about both can be found in RFCs 1309 [WRH92], 1777 [YHK95], and 2251 [WHK97].

Address Resolution of Unstructured Identifiers

Unstructured identifiers are almost like random bit strings and contain no information whatsoever on how to locate the access point of the entity they refer to. Because of this lack of structured information, we have the problem of how to find the corresponding address of the entity. Examples of such unstructured identifiers are IP numbers in a LAN or hash values.

A simple solution is to use broadcasting: The resolver simply broadcasts the query to every node and only the node that has the access point answers with the address. This approach works well in smaller systems. However, as the system scales, the broadcasts impose an increasing load on the network and the nodes, which make it impractical for larger systems. As a practical example, the address resolution protocol (ARP) uses this approach to resolve IP addresses of local nodes or routers in a network to MAC addresses.

A more complicated and scalable approach is to use *distributed hash tables* (DHT). DHTs are constructed as overlay networks and allow the typical operations of hash tables such as `put()`, `get()` and `remove()`. The details of a DHT implementation called "Chord" can be found in [SMLN+03]. An advantage of such DHTs is that lookups of keys and their values can be done in $O(\log n)$ (where n is the number of nodes in the DHT), which makes DHTs very practical even in very large-scale systems. A well known application of DHTs are peer-to-peer file-sharing networks, where DHTs are used to store meta-information about the filenames or keywords of the files in the network.

References

[ATT93] AT&T Bell Laboratories, Murray Hill, NJ, USA. *Plan 9 from Bell Labs Second Release Notes*, 1993.

[How95] Timothy A. Howes. The lightweight directory access protocol: X.500 lite. Technical Report 95-8, University of Michigan CITI, July 1995.

[Moc87a] P. Mockapetris. Domain names – concepts and facilities. RFC 1034, November 1987. `http://www.ietf.org/rfc/rfc1034.txt`.

[Moc87b] P. Mockapetris. Domain names – implementation and specification. RFC 1035, November 1987. `http://www.ietf.org/rfc/rfc1035.txt`.

[Nee93] R. Needham. Names. In S. Mullender, editor, *Distributed Systems, an Advanced Course*. Addison-Wesley, second edition, 1993.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoekz, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, December 1997. `http://www.ietf.org/rfc/rfc2251.txt`.

[WRH92] C. Weider, J. Reynolds, and S. Heker. Technical overview of directory services using the X.500 protocol. RFC 1309, March 1992. `http://www.ietf.org/rfc/rfc2251.txt`.

[YHK95] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. RFC 2251, March 1995. `http://www.ietf.org/rfc/rfc2251.txt`.