Lectures on distributed systems:

Naming and binding

Paul Krzyzanowski

What's in a name? That which we call a rose
By any other word would smell as sweet.
—Shakespeare,
Romeo and Juliet 2.2.43-44

Definitions

ONE DANGER IN ANY DISCUSSION of naming is that of getting carried away with both the metaphysical and semantic definitions of naming. We will attempt to avoid that here. John F. Shoch's definitions in *Inter-Network Naming, Addressing, Routing* (RFC 1498) do well in defining the basic terms:

name
A *name* identifies what you want.

address
An *address* identifies where it is.

route
A *route* identifies a way to get there.

A few useful additional definitions put forth by J.H. Saltzer in his 1978 paper include:

binding
Binding is the process of mapping a name to an address. J. H. Saltzer defines it as choosing "a lower-level implementation for a particular higher-level semantic construct." We can avoid the semantic problems of differentiating names from addresses by thinking of binding as the process of mapping a name into some lower-level name.

context
A *context* is a particular set of bindings. A name only has meaning relative to some context.

directory or **naming network**
A set of *catalogs* (name→object binding tables) that may include other directories

naming hierarchy
A naming network organized in a tree-structured form.

pathname
A multi-component name traversing a path in a naming hierarchy.

root

A starting catalog in a naming network.

indirect entry
An entry in a catalog that binds to a name instead of the underlying object.

name service
A service that provides a binding function.

A computing environment will have many names, each relevant within a specific context. On a networked system, we need names for:

services	for example, time of day, file service, finger, ….
nodes	identify a computer that can run services or programs
network attachment points	ports on a network: places where a node is attached (not exactly applicable for an ethernet)
paths	the route between network attachment points
objects within services	for example, file names on a file server

A name can be anything that is convenient or makes sense. For humans, it may be a human-readable character string. For machines, it may be a binary identifier ("address"). A service may run at one or more nodes. It may need to move from node to node without losing its identity. A node may be connected to one or more network attachment points and may need to move from one to another without losing its identity. Multiple paths may exist between network attachment points or the path might change. These conditions warn us that it is not a good idea for a name to contain an implicit binding, one that can be derived from the name alone (for example, a node name being a textual representation of the machine's Ethernet address).

Binding

Let us consider the basic mechanism of sending a data packet to a service and the bindings that are involved:

1. Find a node on which the required service resides. This requires service name resolution. The binding is *service name → node name.*
2. Find a network attachment point to which the node is connected. This requires locating the node name. The binding is *node name→ attachment point (or address).*
3. Find a path from this location to that point. This is the routing service. The binding is *address→route.*

As an example of naming and binding, we can consider the machine names on the Internet. A name of cs.rutgers.edu may bind to the IP address of 128.6.4.2. In turn, the IP address 128.6.4.2 may bind to the Ethernet address 08:00:20:1f:13:83. Note that the term *address* becomes context-dependent. We tend to think of addresses as the "lower-level" representation of a name. In essence, addresses are just names. A human is comfortable dealing with cs.rutgers.edu. An IP driver finds it easier to work with the 32-bit name 128.6.4.2, and an Ethernet driver prefers the 48-bit name 08:00:20:1f:13:83.

If we consider naming and binding in file systems, we have the user- and programmer-friendly textual names that are bound to internal names that are a function of the file system implementation. For example, a pathname of /usr/bin/ls binds to {device 32,149, inode 32623}.

An important issue in binding is that of *when* the binding takes place. Dissociation between the bound entities is highly desirable. A machine's address may change while its name remains the same, yet the services on that machine should be accessible. A service may run on a different port, yet a client should be able to locate it. The inode allocated to a file may change, yet that should not cause problems in accessing the file by name.

The least flexible binding is **static binding**. This is essentially a hard-coded binding. For example, a program may assume that SMTP mail service is always available on port 25 and simply access port 25 instead of attempting to resolve the binding through other means. Fortunately, well-followed conventions will often save that program. A more dangerous example is that of a program attempting to contact a machine by using a hard-coded IP or Ethernet address.

An alternative to static binding is **dynamic binding**. With dynamic binding, we no longer rely on a hard-coded name address binding but have some mechanism for resolving the name on demand. One form of dynamic binding is **early binding**. In this case a binding operation is actually performed, but it is performed some time before the binding is needed. For example, if a program needs to contact a server multiple times during a long period of execution, it might perform a name to IP address binding once at the start for efficiency. The danger here is that the server's address may change during the execution of this program and the program will be unable to contact the server at some point in time (or will contact the wrong server). At times, early binding is a crucial optimization. IP address to Ethernet address binding is an example of a case where it would be prohibitively expensive to resolve an IP address for *every* packet sent from a machine. It makes sense to maintain a cache (the ARP cache) of previously used bindings. Problems arise if the bindings change. With ethernet addresses, the problem is usually that of not being able to reach the destination machine and the system attempts to perform another binding. Early bindings hurt in a dynamically changing environment.

Most flexible is **late binding**, where the binding is performed on demand, just before use. Previous bindings are not cached. An example where this is useful is accessing a file system. Consider a user editing a file and frequently saving changes. Assume that the editor is implemented in such a way that it writes into a temporary file during the edit and renames the file to the permanent name upon save. In this case, a different inode (or FAT index) is allocated each time the file is saved, yet the name remains the same. The only way of assuring that the correct file is opened (say, by another process) is by performing the exernal to internal name binding at the time of open. The caveat with dynamic binding is that the name resolution process often takes quite a bit of time. If the same name has to be resolved over and over again, early binding may yield considerable performance gains.

Distributed systems issues

Of particular concern to distributed systems is the issue of where a directory of bindings resides. How do we look up names? The solution, of course, is to provide a name server for the particular binding as a network service (the domain name server, DNS, is an example). The problem then becomes recursive: how do we name (bind to) the name servers? Within a local area, it may be feasible to use a broadcast-protocol similar to RARP (reverse ARP). Within a wide area, some other means must be found. Quite often, the means is simply to agree on a standard and use a static binding to locate a name server.

Replication of directories is useful for efficiency. Caching is a form of replication, so an ARP cache (a cache of recently used IP address to Ethernet address bindings) is an example. However, replication suffers from consistency problems. If the prime directory is modified, the replicated copies will have stale bindings. Even if some form of synchronization is employed, the process may take time. The hope in cases where replication is used is that it often does not matter too much if the data is stale and the binding may be performed again.

Scalability of a name space is a challenge in a system with many entities. It becomes increasingly difficult to assure the uniqueness of names. Think about the challenge of coming up with unique machine names to the tens of millions of systems on the Internet. The general solution to scalability is to adopt a hierarchy of names. Uniqueness is not difficult to manage on a small scale. A hierarchy of names allows local name spaces to be maintained. The Internet domain system, X.500 directories, and hierarchical file systems are examples where hierarchies serve well to provide a manageable name space.

References

Inter-Network Naming, Addressing, Routing, RFC 1498, John F. Shoch, 1978 [easy and quick reading]

On Naming and Binding, J. H. Saltzer, 1978, MIT. [dated terminology and overly-philosophical, yet this is a must-read-first paper]

Distributed Systems, Sape Mullender, Ed., chapter 12: *Names* by Roger M. Needham, ©1993 Addison-Wesley [good coverage, but make sure you read Saltzer's paper first]