Chapter 3

Naming and Binding

Naming and binding are fundamental ingredients of any computing system. Naming deals with the designation of the various resources that compose a system, while binding is concerned with actual access to objects through names. This chapter starts with an introduction to the basic concepts of naming. It goes on with a discussion of design patterns for distributed name services, illustrated by case studies. Then comes an introduction to binding, followed by a discussion of a general pattern for distributed binding. The chapter concludes with a presentation of the naming and binding framework of Jonathan, a kernel that provides basic tools for the construction of distributed binding services.

3.1 Names

In a computing system, a *name* is an information associated with an object (the name *designates* the object) in order to fulfill two functions:

- to identify the object, i.e., to distinguish it from other objects, so the object can be (usually unambiguously) referred to.
- to provide an access path for the object, so the object can actually be used according to its specification.

In the above definition, the term “object” has a very general meaning and could be replaced by “resource”, meaning anything that we may wish to consider as an independent entity, be it material or logical. Examples of objects, in that sense, are: a variable or a procedure in a program; a file; a software component; a memory cell, a track on a disk, a communication port; a device, either fixed (e.g., disk, printer, captor, actuator) or mobile (e.g., PDA, mobile phone, smart card); a user; a host on the Internet; a network. The specification of an object describes its interface, i.e., the set of operations that it provides to the outside world. Familiar instances of names are identifiers used in programming languages, addresses of memory cells, domain names in the Internet, e-mail addresses, file names, port numbers, IP addresses, URIs.

The two functions of a name (identification and access) impose different sets of requirements.

- A name used for identification should be permanently and unambiguously associated with the object that it identifies. For instance, [Wieringa and de Jonge 1995] state three requirements for "identifiers" (names used for identification purpose): an identifier designates at most one object; an object is designated by at most one identifier; an identifier always designates the same object (in particular, it should not be reused for a different object).

- A name used for access should allow the object to be physically located. As a consequence, such a name may change when an object moves; a new name may be created for efficiency reasons, e.g., if a copy of the object is put in a cache to speed up access. Such names are typically reused and cannot serve as unique identifiers.

Two remarks are in order: first, the requirements for identifiers are so stringent that they are almost never met in practice[1]; second, the requirements for identification and access are contradictory. As a consequence, in the naming schemes used in practical situations, an object is usually designated by two kinds of names: names that are primarily used for designation (but do not necessarily have all the uniqueness properties of an identifier; for instance, aliases are allowed, and names may be reused), and names that are primarily used for access, which typically contain some form of location-related information, e.g., a memory address, a network address, a port number, etc. Such names are usually called *references*. However, a reference to an object may not always be directly used for access (for example, in order to access a remote object, a communication protocol must be set up). Associating designation names with references, and making references usable for access, is the function of binding, the subject of Section 3.3.

Other criteria have been proposed to classify names. A common distinction is between names intended for use by humans, usually in the form of a character string (e.g., a symbolic file name, or an e-mail address) and names intended to be interpreted by some computing system (e.g., a bit string representing a memory address or an internal identifier). While intuitive, this distinction does not have deep implications. A more significant distinction is between "pure" and "impure" names [Needham 1993]. A pure name does not give any information as to the physical identity or location of the object it refers to, while an impure name does contain such information. Names used for designation may be pure or impure, while names used for access are, by definition, impure. The advantage of using an impure name is to speed up access to the named object; however, if the physical location of an object changes, an impure name that refers to it must usually be changed as well, while a pure name remains invariant. Changing a name may be problematic, because the name may have been copied and distributed, and it may be expensive or unfeasible to retrace all its occurrences. The choice between pure and impure names is a trade-off between flexibility and efficiency, and hybrid solutions are used in many cases (i.e., compound names, partly pure and partly impure).

The distinction between pure and impure names is also related to the distinction between identity and representation, as illustrated by the following examples. Using a pure name often amounts to introducing an additional level of abstraction and indirection.

[1] The naming scheme for Ethernet boards uses 48-bit identifiers, of which 24 bits uniquely identify a board manufacturer, and the 24 remaining bits are internally allocated (without reuse) by each manufacturer. These identifiers satisfy the above uniqueness properties.

Examples.

1. Usually, the mail address associated with a specific function within an organization is an impersonal one, such as `webmaster@objectweb.org`. Mail sent to such an address is redirected to the person (or group of persons) that performs the function. In addition to preserving anonymity, this simple device allows the function to be transparently transferred to a different person or group.
2. To increase availability and performance, some systems maintain multiple copies (replicas) of critical objects. The user of such an object is not usually aware of the replication; he sees a unique (abstract) object, designated by a single name. In contrast, the object management system must access and update replicas to maintain the illusion of a single copy, and therefore needs to identify each replica by a different name.

Names are essential for sharing information and other resources between different users or different programming systems. Names allow objects to be shared by reference, by embedding a name referring to the shared object in each of the objects that share it. This is usually cheaper, easier to manage, and more flexible than sharing by copy or inclusion (including the shared object, or a copy of it, in the sharing objects); for physical objects, sharing by reference is the only possibility. Many of the difficulties of binding names to objects are related to the management of such names embedded in objects.

Examples

1. Sharing by reference is the basic device of the World Wide Web, in which objects (web pages) are referred to by URIs embedded in web pages.
2. Code libraries (such as mathematical functions, string manipulation procedures, etc.) are usually shared by reference in the source programs, by embedding their (symbolic) name in the programs that use them. In the executable (compiled) programs, they still may be shared by reference (through a common address, if the system supports shared libraries), or they may be shared by inclusion (including a copy of the library programs in the executable code segment, and using its local address in that segment).

A *naming system* is the framework in which a specific category of objects is named; it comprises the rules and algorithms that are used to deal with the names of these objects. Usually, many naming systems coexist in a given application (e.g., there is a naming system for the users, another one for the machines, yet another one for the files, etc.), although some systems (e.g., Plan 9 [Pike et al. 1995]) have attempted to use a uniform naming system. In a given naming system, a *name space* defines the set of valid names, usually by providing an alphabet and a set of syntax rules.

The main issues in the design of a naming system are the following.

- Organizing the name space; this is done by means of *naming contexts*, as described in Section 3.1.1.
- Finding the object, if any, associated with a name; this process, called *name resolution*, is examined in Section 3.1.2

- *Binding* names to objects, i.e., managing the association between names and objects. Binding is the subject of Section 3.3

In a more general view of naming, an object may be characterized by some properties, i.e., assertions on some of its attributes. This characterization may or may not be unique. For example, one can look for a server delivering a specified service, identified by a set of attributes. This aspect of naming is further developed in Section 3.2.3.

3.1.1 Naming Contexts.

One could envision a name space as a uniform set of identifiers, such as the integers, or the strings on some alphabet. While conceptually simple, this scheme has practical limitations: searching in a large flat space is inefficient, and there is no simple way of grouping related objects. Therefore a name space is usually organized into naming contexts. A *naming context* is a set of associations, or bindings, between names and objects. Naming contexts correspond to organizational or structural subdivisions of a global name space, such as directories in a file system, departments in an organization, hosts in a computer network, or domains in the Internet DNS (3.2.2).

Context Graphs and Contextual Names

When dealing with names defined in different naming contexts, e.g., *NC1* and *NC2*, it is useful to define a new context *NC* in which *NC1* and *NC2* have names (e.g., *nc1* and *nc2*, respectively). This leads to the notion of a *context graph*: designating *NC1* by name *nc1* in *NC* creates an oriented arc in that graph, labeled by *nc1*, from *NC* to *NC1* (Figure 3.1a.

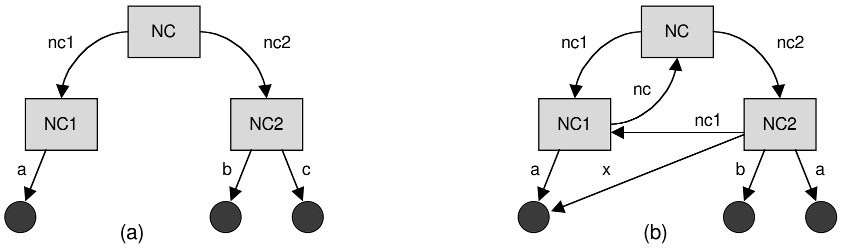

Figure 3.1. Naming contexts

A context graph may have any configuration (e.g., it may have cycles, multiple arcs between nodes, and it may be disconnected), and it may evolve dynamically when contexts and names are created or removed. For example, we may do the following operations on our context graph: create a new name, x in *NC2* for the object named a in *NC1*; create a name *nc1* in *NC2* for context *NC1*; create a name *nc* in *NC1* for context *NC*; change the name of object c in *NC2* to a. The resulting graph is shown on Figure 3.1b.

Composite names may now be defined: if a is the name of an object A in *NC1*, and b is the name of an object B in *NC2*, then *nc1.a* and *nc2.b* (where the separator "." denotes a name composition operator) respectively designate A and B in *NC* (Figure 3.1a). A name thus constructed, be it simple or composite, is called a *contextual name*: it is relative to some context, i.e., it is interpreted within that context.

The main operations on names, e.g., searching or resolution, involve navigating in a context graph by following inter-context arcs. Two notions are important here.

- The starting point, a specified context called a *root*. There may be one or several roots (multiple roots are mandatory in the case of a disconnected context graph but may be convenient in other situations).
- The current position, usually called the current context. An example is the notion of a working directory, present in most file systems.

Roots are intended to be stable reference points that seldom, if ever, change. A name relative to a root is said to be *absolute*; in the case of multiple roots, the name must specify which root it starts from. An absolute name has the same meaning in any context, as opposed to a *relative* name, which is only significant in a specified context (by default, the current context). Absolute and relative names are distinguished by syntax (e.g., in the Unix file system, absolute names start with "/", which is also the absolute name of the root).

Example.

> In the context graph of Figure 3.1b, let *NC* and *NC2* be chosen as roots. Assume that the notation for an absolute name is <*root*>:<*name*>, where <*root*> is a universally known identifier for the specified root and <*name*> is a name relative to that root. Then, *NC:nc1.a*, *NC2:nc1.a*, and *NC2:x* are absolute names for *A*, and *a* is a relative name for *A* if the current context is *NC1*.

Note that there is a need for universal (or global) denotations, that are valid in any context. Examples are the names of the root(s) and the denotations of well-known invariant entities such as integers or character strings[2].

Restructuring Context Graphs

As seen in the previous section, a context graph may evolve by creating or deleting objects and naming contexts. Even in the absence of creations and deletions, it may be useful to perform structural changes. These include renaming entities, moving objects or naming contexts within a context graph, and combining context graphs.

An object that is shared between several contexts has a name in each of these contexts. In addition, an existing object or naming context, which has a name in some context, may receive a new name in another context, for the following reasons.

- Shortening the name. In a tree-structured naming space, for example, designating an object in a "cousin" naming context entails using an absolute name, or at least

[2]There is a distinction here again between identity and representation. The denotation "7" universally designates the integer 7 as an abstract entity. However, specifying an actual representation for this entity (e.g., on 16 or 32 bits, little- or big-endian, etc.) is resolved at a different level (this issue is similar to that of the actual identity of the person(s) behind `webmaster@objectweb.org`).

going up to the first common ancestor if parent contexts can be named. Such a name may be quite long if the tree is deep. Creating a direct link allows for a shorter, more convenient name.

- Logical grouping. Objects already named in different contexts may be logically regrouped by giving them new names in a common context.
- Access speed-up. Creating a new name may speed up access to the named object (this depends, however, on the implementation of name to object bindings).

Giving a new name to an object may be done in two different ways, illustrated by the notions of "hard" and "soft" links in the Unix file system.

A hard link has exactly the same status as the original name of the object. It may be created in the same context as the original name or in a different context. A soft link, by contrast, is an indirect designation: it points to a holder that contains a name for the object. In that sense, a soft link is just a hint: if the initial name of the object changes, the soft link becomes invalid because it still points to the original name.

These notions are illustrated by a simple (slightly contrived) example.

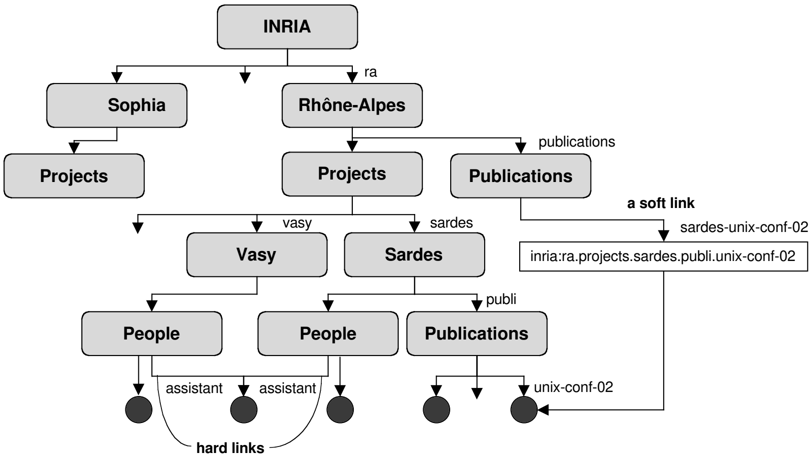

Figure 3.2. Hard and soft links

Figure 3.2 shows part of a name space representing research projects at INRIA. Projects Sardes and Vasy happen to share a project assistant. This person is designated by the two names *inria:ra.projects.sardes.people.assistant* and *inria:ra.projects.vasy.people.assistant*, which have the same status, i.e., a hard link. On the other hand, suppose INRIA Rhône-Alpes decides to group all the publications of its projects in a common naming context, *inria:ra.publications*. One way of doing it is to create soft links in this context to all existing publications of the projects. Thus, in this context, a link named *sardes-unix-conf-02* is made to point to a holder containing the original name *inria:ra.projects.sardes.publi.unix-conf-02*. However, if the original name is changed because of a restructuring of the Sardes publication context, the link becomes invalid and should be explicitly reestablished to the new name.

Some situations call for combining (parts of) previously separated context graphs. This happens when physically connecting separate devices or subsystems, each of which has its own context graph. Another instance is merging two companies or reorganizing an administration.

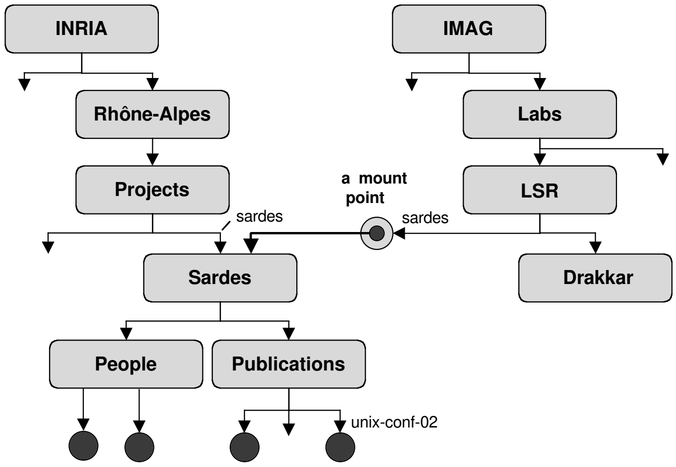

Figure 3.3. Mounting

Following up on the previous example, suppose that project Sardes of INRIA becomes a joint project of INRIA and another research institute, IMAG. This may be done in the following fashion:

- Create a new entry of name *sardes*, called a *mount point*, in the context *imag:labs.lsr* (the new context in which the project should be known). This is essentially a holder for a link to another context.

- Connect the existing context *inria:ra.projects.sardes* to this holder; this is done through a *mount* operation.

From now on, project Sardes may be named in the IMAG context graph exactly like IMAG's own projects. Nothing in the name's structure reveals the existence of the link through the mount point (compare *imag:labs.lsr.sardes* with *imag:labs.lsr.drakkar*).

In addition to name management, there is another aspect to mounting: access must be guaranteed to the mounted object. This aspect is considered in Section 3.3

Mounting was initially introduced to incorporate removable devices (e.g., a removable disk stack) to the Unix file system, by making the device's directories and files part of the file system. It was generalized to integrate the file systems of several networked machines in a single unified file system, the Network File System (NFS) [Sandberg et al. 1985].

Another way of combining separate context graphs is to give them names in a common root context. This was done, for instance, in Unix United [Brownbridge et al. 1982], the first attempt towards unifying separate file systems. This technique, however, entails changing all absolute names of existing objects. The GNS system [Lampson 1986] also allows upwards extensions, but proposes a remedy to the name change problem, by defining an imaginary super-root which has all directories as its children and uses global identifiers. Any name may then be made absolute by prefixing it with the global identifier of an appropriate directory[3], and does not need to be changed if the context tree is extended upwards. When a directory is created as an upward extension, or as a common root of existing context trees, a mapping is set up between the global identifiers of its directory children and their local names in the new directory, so that the old (global) names may be correctly interpreted in the new context. Note that the problem is now to define and to manage unique global identifiers in the super-root directory.

Conclusion

In summary, a naming scheme based on naming contexts has the following benefits.

- Contexts are convenient for grouping related objects; devices like current context, relative naming, symbolic links, allow for shorter names;
- Names may be chosen independently in different contexts, and nothing prevents the same name from being reused in different contexts (e.g., in Figure 3.1b name *a* exists in contexts *NC1* and *NC2* and refers to different objects; name *nc1* exists in contexts *NC* and *NC2* and refers to the same object, context *NC1*).
- Contexts may be used to speed up navigation and searching in the name space.
- The naming system may be indefinitely extended by creating new contexts, and by linking existing contexts together.

3.1.2 Name Resolution.

In order to determine the object, if any, referred to by a valid name, a process called *name resolution* must be carried out. Name resolution starts in an initial naming context, and proceeds in steps. At each step, a component of the name (a label in the current context) is resolved. This operation either delivers a result, the *target*, or fails (if the current label is not bound to any object in this context). Three cases may occur.

- The target is a typed value (not a name), i.e., a couple [`T`, `V`], where `T` is a type and `V` a value of that type. The type does not have to be explicit, i.e., it may be determined by the naming context or derived from the name.
- The target is a primitive name (also called an address), i.e., a name that cannot be further resolved. If the designated object is a physical entity (e.g., a memory cell, a disk drive, a host, a mobile device), the address identifies the object. If the designated object is a data structure (e.g., a record, a Java object, a file), the address identifies a physical container or location for the object[4].

[3] this may be done automatically by the system using the notion of a current context.

[4] It often happens that the designated object has a complex structure and is not physically identified by a single address. In that case, the target is a data structure called a *descriptor* (a form of reference), which contains the information on the physical layout of the object, together with other information (e.g., protection). An example of a descriptor is the *i*-node, a data structure that describes a file in the Unix operating system, and includes the disk addresses of the blocks that contain the file. Another example is a couple (network address, port number), which refers to a service on a network.

- The target is a non primitive name (in some naming context). The resolution process is then called again on the target name in that context.

We now propose a general scheme for the name resolution process. Assuming that an initial naming context is known (we shall see later how it is determined), we represent names and contexts as objects (in the sense of object-oriented programming, i.e., the association of a state and a set of methods), and the resolution process then takes the form of the following method call:

`target = context.resolve (name)`[5]

in which `context` is the naming context, `resolve` is a method that embodies the name resolution algorithm (the resolver) in this context, and `name` is the name to be resolved. If the resolution succeeds, `target` is the result, which may have different forms:

- a value associated with the name.

- another name, together with a new context.

In the latter case, the name must in turn be resolved in the new context by calling its resolver. The resolution process is repeated[6] until either it fails (in which case the original name cannot be resolved) or it delivers a value, i.e., either the object itself or the name of a physical container for it (e.g., a memory address, a descriptor). Note that the resolution may be repeated using either iteration or recursion. Let ($name_{i+1}$, $context_{i+1}$) the couple (name, context) delivered by the resolver of $context_i$.

- In the iterative scheme, the results are delivered to the initial resolver, which keeps control of the resolution process, by successively calling the resolver of $context_i$ on the $name_i$, for increasing values of i.

- In the recursive scheme, control goes from one resolver to the next: $context_i$ calls the resolver of $context_{i+1}$ on the name $name_{i+1}$, and so on.

The recursive scheme generates less messages than the iterative scheme. However, there is a heavier load on the servers, and recovery from failure is more complex, since the initial resolver does not keep track of the successive calls. See [Tanenbaum and van Steen 2006], 4.1.3) for a detailed discussion.

As an example, consider the resolution of an URI in the World Wide Web (Figure 3.4), which may be summarized as follows. The initial naming context is determined by examining the protocol part (i.e., the first string of the URI, up to the first ":"). For instance, if the protocol is `http` or `ftp`, the context is the Internet; if it is `file`, the context is the local file system; etc.

[5] A logically equivalent scheme would be `target = name.resolve (context)`, i.e., the `resolve` method may as well be borne by the name as by the context. See the description of the Jonathan platform (Section 3.4) for practical examples.

[6] If the context graph contains cycles, the resolution process may enter an infinite loop. In practice, termination is ensured by setting an upper limit to the number of resolution steps.

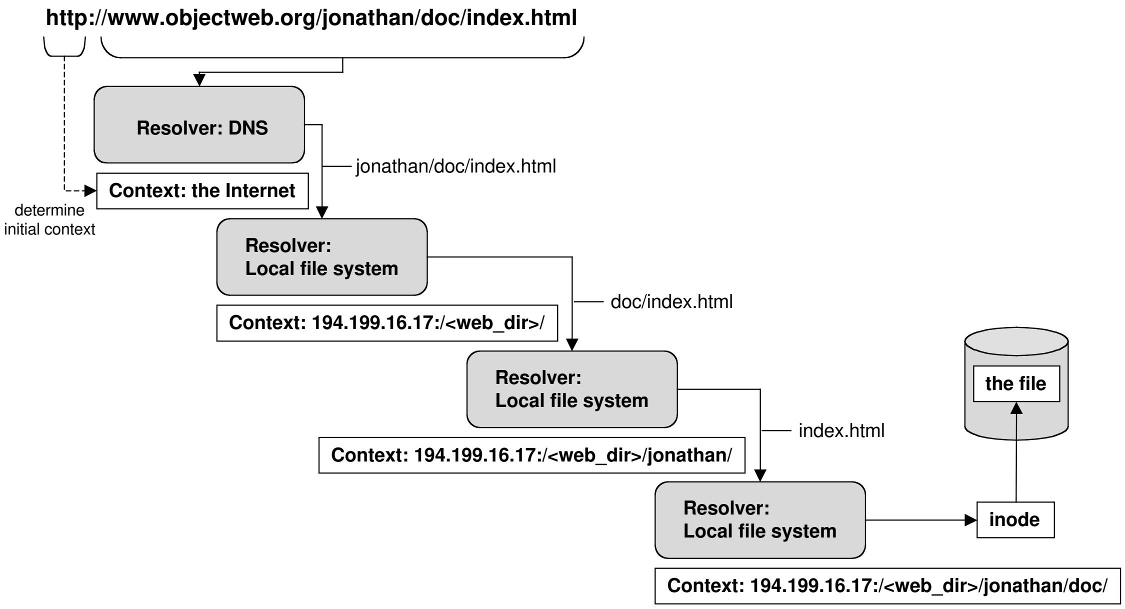

Figure 3.4. Resolving a name in the World Wide Web

In our example, the protocol is `http`. Therefore the string between `//` and the first following `/`, i.e., `www.objectweb.org` is assumed to be the domain name of a server, and is resolved in the context of the Internet (the resolver in this context is the DNS, see 3.2.2). Let us assume this resolution succeeds, delivering the IP address of a host (194.199.16.17). Its result is a new context (the web server directory on the host, noted `/<`*web_dir*`>`), together with a new name (the remaining part of the URI, i.e., whatever follows the `/` that ends the host name). The new name is now resolved in the new context. This in turn may lead to a chain of resolutions, going down the file hierarchy on the server. Assume the named object is a file, like in our example (*index.html*). Then the resolution ends with the descriptor of that file, from which the file itself may be retrieved from the disk. If the string following the last `/` contains a `?`, then the substring on the left of `?` is interpreted as the name of the resolve procedure in the current context, and the sub-string on the right as the name to be resolved. This is used, in practice, to perform an attribute-based look-up (the equivalent of a query in a database, to find objects satisfying a given predicate). In this example, the structure of the name reflects the chain of contexts in which the resolution takes place (the name of the object explicitly contains the names of the contexts). This, however, needs not be the case (e.g., the original name may designate a descriptor which contains both the name of the new context and the name to be resolved in this context, and so on down the chain).

The question of how the first context is determined remains to be settled. There are several possibilities.

- It may be inferred from the name to be interpreted (e.g., in the example of URIs, where the initial context is determined by the name of the protocol).

- It may be found in a predefined location (e.g., in the case of a virtual address, where the context is determined by the contents of the MMU and of the page tables, which are addressed by a predefined register).
- It may be part of the current environment (e.g., the "working directory" determines the context for file name interpretation in an operating system).

A name may be valid in several different contexts, and one may wish to select a given context for a specific execution. One solution in that case is to explore the set of possible contexts according to a specified search rule (e.g., linear order in a list, depth first in a directed graph, etc.) until the name can be resolved. The set of contexts, together with the search rules, defines a *search path*; by changing the order of the contexts in the search path, a given name may be resolved into different objects. This technique is used by linkers for resolving the names of library routines. It allows (for example) a user to supply his own version of a routine instead of that provided by the system, by including his private version in a directory and placing that directory at the head of the search path.

3.2 Patterns for Distributed Name Services

The function of a *name service* is to implement name resolution for a given name space. In this section, we present the main design principles and patterns applicable to name services in a distributed environment. We use a few examples for illustration, but we do not intend to present full case studies of working name services. These may be found in the provided references.

Section 3.2.1 presents the main problems presented by the design of a large scale distributed naming service, and some general principles and techniques used to solve them. While these apply to all naming services, different situations occur in practice. Broadly speaking, the main distinctive criterion is *dynamism*, the rate at which change occurs in the structure and composition of the system.

In section 3.2.2, we examine the case of a naming system for a relatively stable environment, in which the rate of change is slow with respect to the rate of consultation of the naming service. This situation is that of the global Internet, and the Domain Name Service is a representative example of this class of systems.

Section 3.2.3 deals with the additional constraints introduced by highly dynamic environments, characterized by mobility, attribute-based search, and rapidly changing composition of the system. A typical application is *service discovery*, in which the object being looked up is a service provider.

3.2.1 Problems and Techniques of Distributed Naming

The requirements of a naming service are similar to those of any distributed application. However, some of them are made more stringent by the function and environment of a naming service.

- *Availability.* Availability is defined as the fraction of time the service is ready for use (more details in Chapter 11). Given the central role of the name service in any working environment, availability is perhaps its most important requirement.

- *Performance.* The main performance criterion for a naming service is latency. This is because name resolution introduces an incompressible delay in the critical path of any request for a remote service.
- *Scalability.* A service is scalable if its performance remains acceptable when its size grows. Measures of size include the number of objects, the number of users, and the geographical extent. A naming service may potentially manage a large number of objects and its span may be worldwide. In some applications, new objects are created at a very high rate; merging several applications may give rise to a large name space. Therefore scalability is also an important requirement for a name service.
- *Adaptability.* A distributed name service often operates in a changing environment, due to the varying quality of service of the communication network, to the mobility of objects and users, and to evolving requirements. The service therefore needs to be adaptable, in order to maintain its performance in spite of these variations.

To meet these requirements, a few design principles and heuristics have shown to be efficient.

- Avoid an organization that involves a single point of decision. This is detrimental both to performance (bottleneck) and availability (single point of failure).
- Use *hierarchical decomposition* as a guide for partitioning the work and for delegating responsibility.
- Use *redundancy*; arrange for several independent ways to get an answer.
- Use *hints* in order to speed up the search. A hint may be invalid, but there should be a way to get up to date information if needed. *Caching* is the main relevant technique.
- Most information becomes stale with time; use *timeouts* to discard out of date information.
- Use *late binding* when adaptability is needed.
- *Avoid broadcast*, which does not scale well; however, some use of broadcast or multicast is unavoidable in dynamic environments; in that case, use *partitioning* to restrict the range.

3.2.2 Naming Services in a Slowly Changing World

In this section, we review the design principles of a naming service for a relatively stable environment, in which the frequency of change is small with respect to that of consultation. The Domain Name System (DNS), our first case study, was designed under this hypothesis (however, relative stability does not preclude change: the Internet has actually expanded at an exponential rate since the introduction of the DNS). We then examine the techniques used to deal with mobility, still at a relatively slow rate.

The Domain Name System

The Domain Name System [Albitz and Liu 2001] is the name service of the Internet. In this section, we briefly examine the principles of its organization and operation.

The name space is organized as a hierarchy of spaces called *domains*, as shown on Figure 3.5. The notation for DNS global names is similar to that of files names in a hierarchical file system, except that the local names are concatenated in ascending order, separated by dots, and that the name of the root (denoted by a dot) is usually omitted; thus the hierarchy is actually considered as a forest, starting from a set of top-level names. For example, `turing.imag.fr` denotes a leaf object (in this case a machine in the domain `imag.fr`), while `research.ibm.com` denotes a domain.

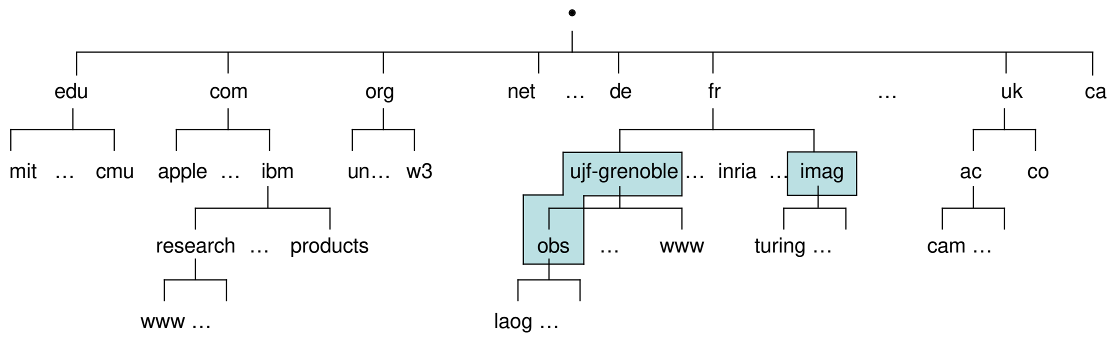

Figure 3.5. The DNS domain hierarchy

There are a few hundreds top-level domains, which may be generic (e.g., `com`, `edu`, `org`, etc.) or geographical (e.g., `ca`, `de`, `fr`, `uk`, etc.). These names are allocated by a global authority, the ICANN (Internet Corporation for Assigned Names and Numbers), an internationally organized, non-profit company. The management of lower level domains is delegated to specific authorities, which themselves delegate the management of their sub-domains. Thus the name `ujf-grenoble.fr` was allocated by AFNIC, the French authority in charge of the top-level `fr` domain, and the `ujf-grenoble.fr` domain is managed by the local administration of Université Joseph Fourier (UJF).

The physical organization of the DNS relies on a set of name servers. As a first approximation, one may consider that each server manages a domain. The real situation deviates from this ideal scheme on the following points.

- There is no server for the domain ".". The root server manages the first two levels of the hierarchy, i.e., it knows all the servers for the second-level domains such as `mit.edu`, `ibm.com`, `inria.fr`, etc.

- Below the second level, some sub-domains may be grouped with their parent domain to form a *zone* (for example, a university department that has its own domain may rely on the system administrators of the university for the management of that domain). Thus a zone may include a single domain or a subtree of domains (examples of zones are shown in gray on Figure 3.5). A zone is a management unit, and a server is associated with a zone.

- The servers are replicated, to ensure both scalability and fault tolerance. Each zone is managed by at least two servers. The root server has currently a dozen of replicas.

A server maintains a set of records for the objects included in its zone. Each record associates a name and some attributes of the object that the name designates (in most cases, the IP address of a host). Resolving a name, i.e., finding the object associated with the name, follows the general scheme outlined in 3.1.2, and illustrated on Figure 3.6. The search starts form a local server (the address of at least one local server is part of any client's environment). If the local server does not hold a record for the requested name, it interrogates the root server (i.e., it consults the closest replica of that server). If the root server cannot answer the query, it finds the address of a server that is more likely to do it, and so on until the name is resolved or declared as unknown. The search usually combines an iterative and a recursive scheme: all requests involving the root server are iterative[7], i.e., the root server returns the IP address of a server instead of querying that server, and the requests further down the chain are usually recursive.

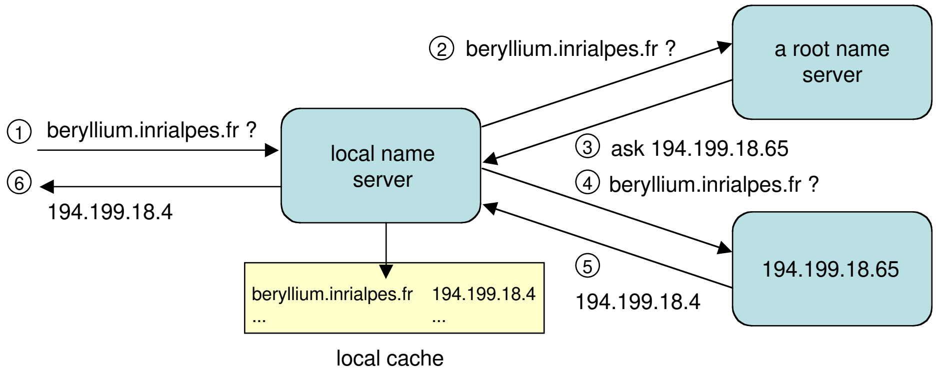

Figure 3.6. Resolving a name in the Domain Name System

In order to speed up name resolution, each server maintains a cache containing the most recent resolved records. The cache is first looked up before any search involving the server. However, an answer coming from a server's cache is only indicative, since the caches are not directly updated after a change, and may thus contain obsolete data. An answer coming from a record held by a zone server is said to be *authoritative* (it can be trusted), while an answer coming from a cache is *non-authoritative* (it is only a hint). Caching greatly improves the efficiency of the search: a vast majority of DNS requests are satisfied in at most two consultations.

The DNS is a highly successful system: its structure and operation are still those defined by its initial design [Mockapetris and Dunlap 1988], while the size of the Internet has scaled up by five orders of magnitude! This success may be ascribed to the following features of the design.

- A highly decentralized structure, following the domain hierarchy. There is no single point of decision. Adding a new domain (usually as a new zone) automatically adds a server, which helps scalability.

[7]in order to reduce the load on the root server.

- Server replication, both for performance and for fault tolerance. The servers located at the higher levels of the domain hierarchy are the most heavily replicated, since they are consulted often; this does not induce a high consistency maintenance cost, since changes are less frequent at the higher levels.
- Intensive use of caching, at all levels. Since locality of reference also applies to names, the performance of caching is good.

As mentioned above, the DNS is representative of a situation in which the universe of names evolves slowly. Another popular name service in this class is LDAP (Lightweight Directory Access Protocol) [LDAP 2006], a model of directory services for large scale enterprise systems, based on the X.500 standard [X.500 1993].

Techniques for "Slow" Mobility

The DNS was designed to deal with name space extension, but not with mobility. The association of a name with an address is assumed to be permanent: to change the association, one should remove the target object, and recreate it with the new address.

Here we examine some techniques used to deal with mobile targets, still assuming relative stability with respect to the consultation rate. We consider two typical situations.

1. Mobility of a target software object between different hosts of a network.
2. Mobility of a host itself.

In the first case, the successive nodes hosting the target object provide a support that can be used for tracking the target. In the second case, there is no such support, so a new fixed element (the so-called home) is introduced.

Consider a situation in which an object, initially located on host $N1$, moves to host $N2$. The technique of *forwarding pointers* [Fowler 1986] consists in leaving a pointer to host $N2$ in the place of the initial target on host $N1$. If the object moves further, a new pointer to the new location $N3$ is placed on host $N2$. This is illustrated on Figure 3.7 (a and b).

However, after a number of moves, the chain becomes long, which has a cost in access efficiency (multiple indirections to remote hosts) and availability (the target may be lost if one of the intermediary hosts crashes). One may then shorten the chain, by updating the initial reference to the object (Figure 3.7c). The remaining pointers are no longer reachable and may be garbage collected. Note that this technique implies the existence of a descriptor for the object, which serves as a unique access point.

The approach to host mobility is different, but still relies on indirection. The problem is to allow a mobile host to attach to different subnetworks of a fixed internetwork, while keeping the same network address in order to ensure transparency for the applications running on this host.

The Mobile IP protocol [Perkins 1998], designed for mobile communication on the Internet, works as follows. The mobile host has a fixed address $< home\ network, node >$. On the home network (a local area network that is part of the Internet), a *home agent*

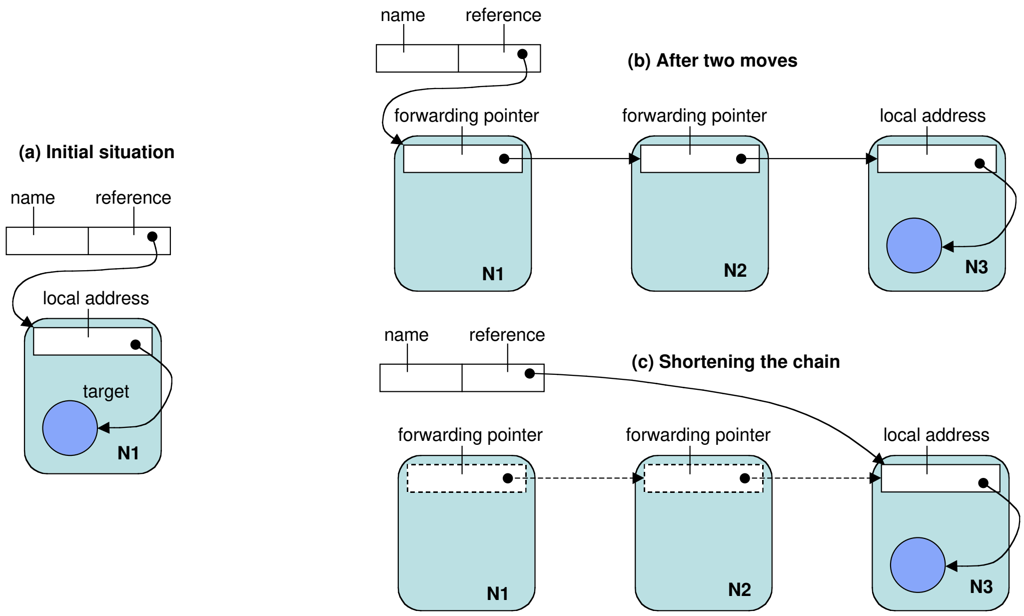

Figure 3.7. Forwarding pointers

running on a fixed host is in charge of the mobile host. All messages directed to the mobile host go to the home agent (this is done by setting up redirection tables on the routers of the home network). If the mobile host moves and connects to a different network, it acquires an address on that network, say $< foreign\ network, node1 >$. A foreign agent located on a fixed host of the foreign network takes care of the mobile host while it is connected to that network. The foreign agent informs the home agent of the presence of the mobile host on the foreign network. Messages directed to the mobile host still go to the home agent, which forwards them to the foreign agent using a technique called *tunneling* (or encapsulation) to preserve the original address of the mobile host.

This is an extremely schematic description, and various details need to be settled. For example, when a mobile host connects to a foreign network, it first needs to find a foreign agent. This is done by one of the service discovery techniques described in 3.2.3, for instance by periodic broadcast of announcements by the available foreign agents. The same technique is used initially by a mobile host to choose its home agent.

3.2.3 Dynamic Service Discovery Services

Distributed environments tend to become more dynamic, due to several factors: increasing mobility of users and applications, favored by universal access to wireless communication; development of new services at a high rate, to satisfy new user needs; service composition by dynamic aggregation of existing services and resources.

This situation has the following impact on naming services.

- The objects being searched are *services* rather than servers, since the location or identity of a server delivering a specific service is likely to change frequently.

- A service is usually designated by a set of attributes, rather than by a name. This is because several servers may potentially answer the need, and their identity is not necessarily known in advance. The attributes provide a partial specification of the requested service.

- If the communication is mainly wireless, additional constraints must be considered, such as energy limitation and frequent disconnection.

In the rest of this section, we first introduce a few basic interaction schemes, which may be combined to build patterns for service discovery. We next present the principles of a few current service discovery systems.

Interaction Patterns for Service Discovery

In the context of service discovery, we define the state of a client as its current knowledge about the available services. In an environment that is changing at a high rate, with a high probability of communication failures, the so-called *soft state* approach is preferred for state maintenance. The notion of soft state has initially been introduced for network signaling systems [Clark 1988]. Rather than relying on a explicit (*hard state*) procedure to propagate each change of state, the soft state approach maintains an approximate view of the state through periodic updates. These updates may be triggered by the clients (*pull* mode), by the services (*push* mode), or by a combination of these two modes. In addition, in the absence of updates after a preset time interval, the state is considered obsolete and must be refreshed or discarded.

The soft state maintenance scheme, which combines periodic update and multicast, is called *announce-listen* [Chandy et al. 1998]. It may be implemented using several interaction patterns, as explained below.

Service discovery involves three types of entities: *clients* (service requesters), *services* (short for "service providers"), and *servers* (short for "directory servers"). The clients need to find available services that match a certain description. Service description is briefly examined at the end of this section.

There are two kinds of interactions: announcement (a service declares itself to a server) and search (a client queries a server for a service matching a description). Both may use the pull or push mode. These interaction patterns are summarized on Figure 3.8

We assume, when needed, that each client and each service knows the address of at least one server (we describe later how this is achieved). A service registers its description and location with one or several servers (Figure 3.8a), using single messages or multicast. A client's query may be satisfied either in pull mode, by interrogating one or several servers (Figure 3.8b1 and b2), or in push mode, by listening to periodic servers' broadcast announcements (Figure 3.8c). Since the number of clients is usually much larger that the number of servers, the push mode should be used with care (e.g., by assigning each server a restricted multicast domain).

In a small scale environment, such as a LAN, the servers may disappear, and the services directly interact with the clients. This again may be done in pull mode (Figure 3.9a) or in push mode (Figure 3.9b).

Two issues remain to be examined;

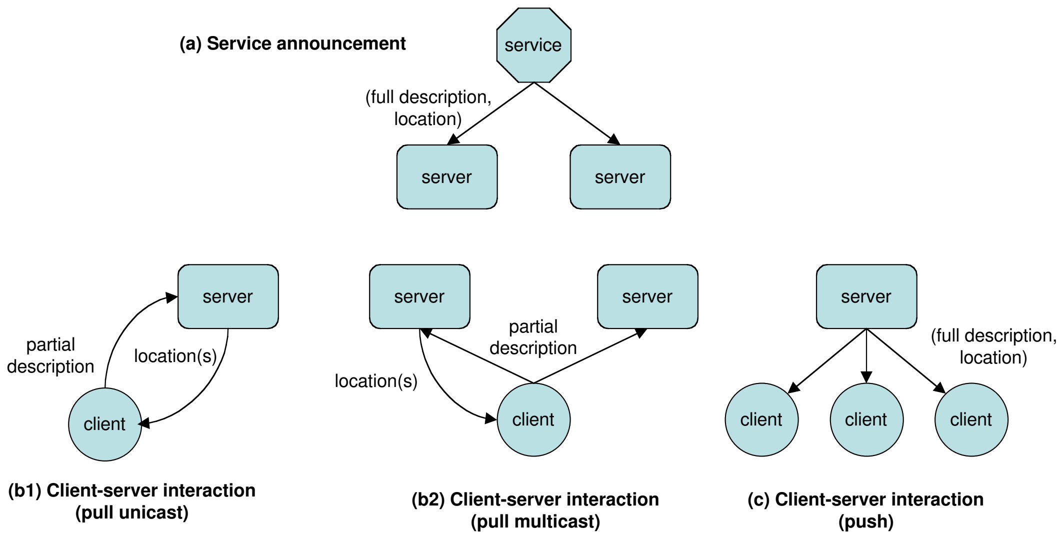

Figure 3.8. Interaction patterns for service discovery

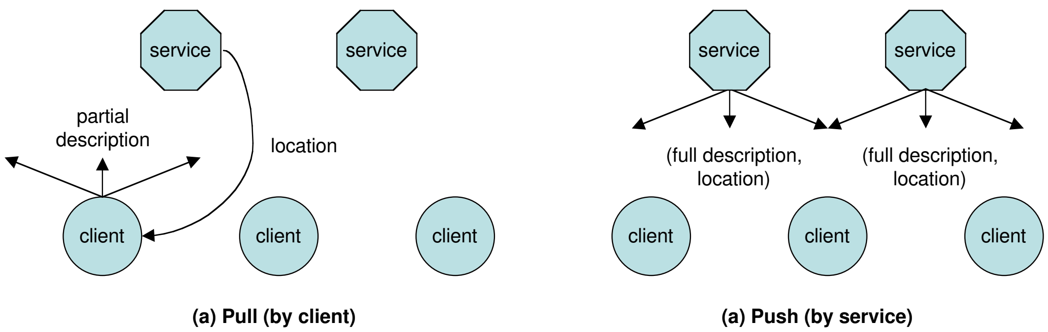

Figure 3.9. Interaction patterns for service discovery (without servers)

- *Describing the service.* In the simplest case, a service description is a couple (attribute, value), such as used in the early trading servers. In the current practice, a description is a complex ensemble of such couples, usually organized as a hierarchy, and often described using the XML formalism and the associated tools. A query is a pattern that matches part or whole of this structure, using predefined matching rules. In pull mode, the matching process is done by the server; in push mode, it is done by the client.

- *Finding servers.* A client or a service may again use pull (broadcasting a query for servers) or push (listening for server announcements) to find the servers. This is illustrated on Figure 3.10. Again the broadcast range should be restricted to a local environment.

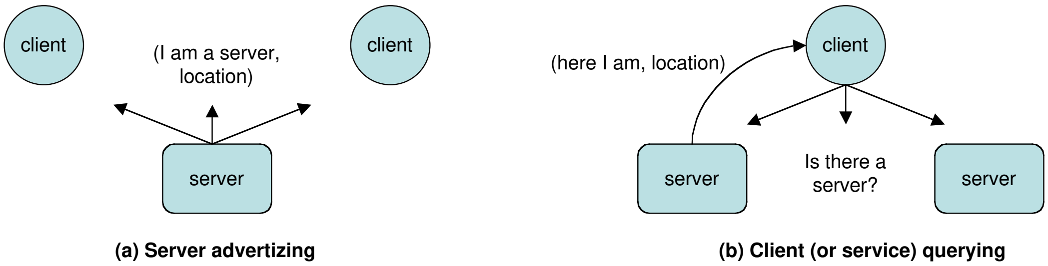

Figure 3.10. Discovering a server

Finally, in an open environment, security must be ensured to prevent malicious agents from masquerading as bona fide servers or services. This is achieved through authentication techniques (Chapter 13). The case studies presented below provide trusted service discovery.

Examples of Service Discovery Services

Many service discovery services have been proposed in recent years, both as research projects and as standards proposals. As an example, we briefly describe the principle of the Service Location Protocol (SLP), a standard proposal developed by an Internet Engineering Task Force (IETF) working group.

The intended range of SLP is a local network or an enterprise network operating under a common administration. The SLP architecture [Guttman 1999] is based on the three entities defined in the above discussion, namely clients (called User Agents, or UA), services (called Service Agents, or SA), and servers (called Directory Agents, or DA). UAs and SAs can locate DAs by one of the methods already presented (more details below). In the standard mode of operation, an SA registers its service with a DA, and an UA queries a DA for a specified service; both registration and query use unicast messages. The response to a query contains a list of all SAs, if any, that match the client's requirements. A service is implicitly de-registered after a specified timeout (it thus has to be re-announced periodically).

The above mode of operation works well on a local area network. For a larger scale environment, consisting of interconnected networks, scalability is ensured through the following techniques.

1. *Multiple Directory Agents*. An SA registers with all the DAs that it knows. This favors load sharing between the DAs, and increases availability.

2. *Scoping*. A *scope* is a grouping of resources according to some criterion (e.g., location, network, or administrative category); SAs may be shared between scopes. A scope is designated by a single level name (a character string). Access of UAs to scopes is restricted, and an UA may only find services in the scopes to which it has access. This limits the load on the directory agents.

SLP can also operate without DAs, in which case an UA's request is multicast to the SAs (pull mode), which answer by a unicast message to the UA.

At system startup, UAs and SAs attempt to discover the DAs using a protocol called multicast convergence. This protocol is also used by the UAs if no DA is found. It works as follows. An agent attempting to discover DAs (or services, in the case of an UA operating without DAs) multicasts a request in the form of a *Service Request* message. In response, it may receive one or more unicast messages. It then reissues its request after a wait period, appending the list of the agents that answered the first request. An agent receiving the new request does not respond if it finds itself on the list. This process is repeated until no response is received (or a maximum number of iterations is reached). This ensures recovery from message loss, while limiting the number and size of the messages exchanged.

Security is ensured by authenticating the messages issued by SAs and DAs through digital signatures. Authentication is provided in each administrative domain.

Another example, developed as a research project, is the Ninja Service Discovery Service (SDS) [Czerwinski et al. 1999]. SDS shares many features of SLP, but has investigated the following additional aspects.

- *Scalability*. While the name space of SLP scopes is flat, SDS servers (the equivalent of DAs) are organized in a hierarchy, each server being responsible for one or several domains. In addition, a server may spawn new servers to cope with a peak of load. Servers use caching to memorize other server's announcements.

- *Security*. In contrast with SLP, SDS provides cross-domain authentication, and authenticates clients (user agents), to control clients' access to services. Access control is based on capabilities.

Other service discovery services are part of industrial developments such as Universal Plug and Play (UPnP) [UPnP] and Jini [Waldo 1999]. A comparison of several service discovery protocols may be found in [Bettstetter and Renner 2000, Edwards 2006].

3.3 Binding

Binding is the process of interconnecting a set of objects in a computing system. The result of this process, i.e., the association, or link, created between the bound objects, is also called a binding. The purpose of binding is to create an access path through which an object may be reached from another object. Thus a binding associates one or several sources with one or several targets.

We first present in 3.3.1 a few instances of binding in various contexts, and the main properties of binding. We then introduce in 3.3.2 a general model for distributed binding. This is the base of the main pattern for binding, the subject of 3.3.3. We conclude in 3.3.4 with an overall view of the place of binding in distributed services.

3.3.1 Examples and Basic Techniques

Usual instances of bindings are the following:

- Associating a name with an object in a context creates an access path through which the object can be accessed using the name. A typical example is language level binding (e.g., associating the identifier of a variable with a storage location containing its value) performed by a compiler and a linking loader.
- Opening a file, an operating system primitive, creates a link between a file descriptor (an object local to the calling process) and the file, which resides in secondary storage. This involves setting up internal data structures, e.g., a local cache for accessing a portion of the file and a channel between that cache and the disk locations that contain the file. After opening a file, read and write operations on the local descriptor are translated into actual data transfers from and to secondary storage (data present in the cache is read from there).
- Compiling an interface definition for a remote procedure call creates client and server stubs (relays for transmission) and a network connection, allowing the procedure to be executed remotely as a result of a local call.
- Setting up a chain of forwarding pointers to a mobile object is a binding, which is preserved by updating the chain when the object migrates to a new location.

Binding may be done at the language level, at the operating system level, or at the network level; most frequently, several levels are involved. Language level binding typically takes place within a single address space, while operating system and network bindings usually bridge several different address spaces. Binding may be done in steps, i.e., the access chain between two objects may be partially set up, to be completed at a further stage.

In addition to setting up an access path, binding may involve the provision of some guarantees as to properties of this access path. For example, binding may check access rights (e.g., at file opening), or may reserve resources in order to ensure a prescribed quality of service (e.g., when creating a channel for multimedia transmission, or when creating a local copy of a remote object to speed up access).

An important notion is that of *binding time*, i.e., the point, in the lifetime of a computing system, at which binding is completed. Typical examples follow.

- Binding may be static (e.g., the denotation of a constant in a program is statically bound to that constant's value).

- Binding may occur at compile time (a compiler binds a variable identifier to an offset in a memory segment; a stub compiler binds a procedure identifier to the location of a stub, itself bound to a channel to a remote location).

- Binding may occur at link time (a linker explores a search path to find the objects associated with the names that remain unresolved after compilation; a component interconnection language is interpreted by a script that sets up access paths from a component to remote components).

- Finally, binding may be dynamic, i.e., deferred until execution time. Dynamic binding is performed when an unbound object is accessed; it may be done at first access or at each access. Binding a virtual address to a storage location in a paged virtual memory is an example of dynamic binding, which occurs at each page fault; another example is the setting up of a forwarding pointer at object migration.

Delayed binding allows greater flexibility, since the decision about how to reach the target of the binding is made at the last possible moment and may then be optimized according to the current situation. Dynamic binding also simplifies evolution: in an application made of many components, only the modified components need to be recompiled, and there is no global relink. Virtual memory binding uncouples the management of virtual addresses from that of physical storage, allowing each management policy to be optimized according to its own criteria.

Two basic techniques are used for binding: name substitution and indirection. These techniques are usually combined, and may be used recursively, to create a chain of bindings.

- *Name substitution* consists in replacing an unbound name by another name containing more information on the target of the binding. This is the main technique used for language level bindings; for example, after compilation, all occurrences of a variable identifier in the text of the program are replaced, in the executable file, by the address of the memory location allocated to the variable. This may be done in steps (e.g., if the output of the compilation is a relocatable file, then an offset is provided, not a complete address).

- In the *indirection* technique, an unbound name is replaced by (the address of) a descriptor that contains (or points to) the target object. This technique is well suited to dynamic binding, because all that is needed is to change the contents of the descriptor (there is no need to locate all the occurrences of the original name). On the other hand, each access to the target incurs the cost of at least one indirection. Forwarding pointers used in distributed systems to locate mobile objects (Section 3.2.2) are an example of this technique.

Note that name substitution creates an efficient binding at the expense of the loss of the original name. If that name is still needed, e.g., for debugging, then it is necessary to explicitly keep an association between the original name and its binding (for example in the form of a symbol table in language level binding).

3.3.2 A Model for Distributed Binding

Distributed systems are built by interconnecting hardware and software components located on different hosts over a network. There is an important difference between centralized and distributed systems as regards binding. In a centralized system, a memory address may be directly used to access an object located at that address. In a distributed system, a reference to a remote object (the equivalent of an address), such as [*host network address*, *port number*] is not directly usable for access. One first needs to actually create a binding to the remote object by building an access chain involving a network protocol, as shown in the following simple example.

Consider the case of a client to server connection using sockets. The server associates a server socket with a port number corresponding to some provided service, and makes this socket wait for incoming requests, through an `accept` operation (Figure 3.11 a). Assume that the client knows the name of the server and the port number associated with the service. Binding consists in creating a socket on the client host and connecting this socket, through a `connect` operation, to the above server socket. The server socket, in turn, creates a new socket linked to the client socket (Figure 3.11 b). The name of the client socket is now used for accessing the server, while the server socket remains available for new connections. This is an instance of binding by indirection: the sockets on the client and server site act as the intermediate descriptors between the client and server processes.

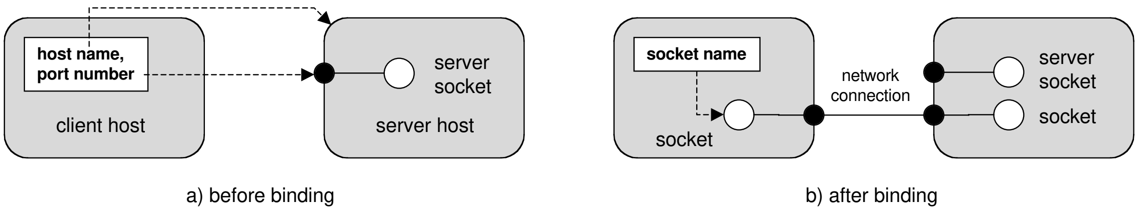

Figure 3.11. Client to server binding using sockets

It should be noted that the binding may only be set up if there is actually a server socket accepting connections on the target port. As a rule, the binding must be prepared, on the target's side, by an operation that enables binding and sets up the adequate data structures.

Binding Objects

To introduce distributed binding, we use the main notions defined in the Reference Model of Open Distributed Processing[8], a general framework for the standardization of Open Distributed Processing (ODP) jointly defined by ITU and ISO [ODP 1995a, ODP 1995b].

[8]We do not attempt, however, to give a full detailed account of this model. In particular, the framework considers five viewpoints according to which the notions may be presented. We are only concerned with the so-called computational and engineering viewpoints, which deal, respectively, with the functional decomposition of a system, and with the mechanisms needed to support distributed interaction between the parts of a system.

The ODP model is based on objects. An object is defined by one or more interface(s), each of which specifies the set of operations that may be performed on the object, together with other properties (e.g. quality of service) that define a contract between the object and its environment. Therefore a binding between objects actually sets up a connection between the objects' interfaces.

In order to provide a uniform view of binding, the ODP model considers the bindings themselves as objects: a *binding object* is an object that embodies a binding between two ore more objects. A binding object has a distinct client or server interface for each object to which it is connected. In addition, a binding may have a specific control interface, which is used to control the behavior of the binding, e.g., as regards the quality of service that it supports or its reactions to errors. The type of a binding object is defined by the set of its interfaces.

Each interface of the binding object acts as an interface to one or several bound objects (e.g., in a client-server system, the interface provided by the binding object to the client is that of the server object, and vice-versa). Figure 3.12 represents an overall view of a binding object.

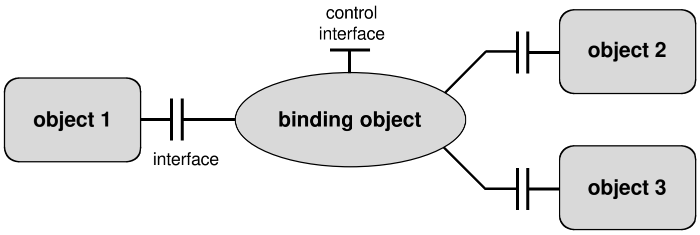

Figure 3.12. A binding object

We have presented the computational view (in the ODP terminology), i.e., a description in terms of functional components. Let us now consider the engineering view, still in ODP terminology, i.e., the implementation of the above mechanism in a distributed setting. The objects to be bound reside on different sites interconnected by a network.

A binding operation (Figure 3.13) typically takes as parameter an unbound reference to an object (the target of the binding), and delivers a *handle* to the object, in the same naming context (recall that a reference is a form of name that refers to the physical location of the object). The reference itself may be obtained through various means, e.g., by looking up a name server using an identifier for the object. The handle may take a variety of forms, e.g., the address of the object, the address of a local copy of the object, or the address of a local proxy for the object, such as a stub.

In some cases, a multiple binding may be set up, i.e., the target may consist of several objects.

The socket example can also be described in this framework, although it is implemented at a lower level than objects. The binding object, in this example, is composed of the client socket, the server socket, and the network connection set up between these sockets.

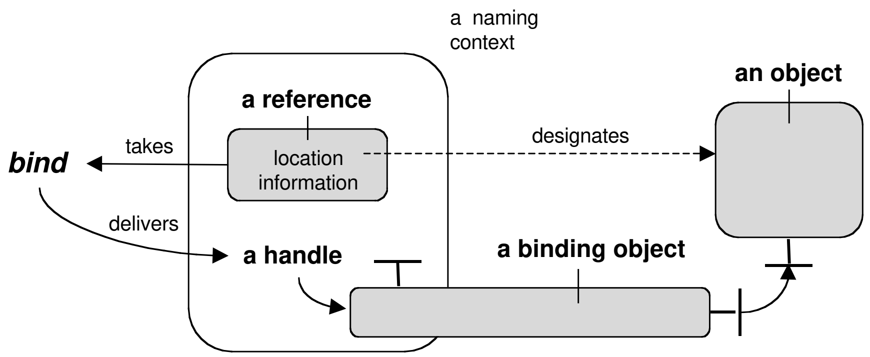

Figure 3.13. The binding process

Binding Factories

The notion of a binding factory has been introduced to define a systematic way of setting up bindings. A *binding factory* is an entity responsible for the creation and administration of bindings of a certain type. Since the implementation of a binding object is distributed, a binding factory may itself be distributed, and usually comprises a set of elementary factories dedicated to the creation of the different parts that make up the binding object, together with coordination code that invokes these factories.

For example, in the simple case of a client-server system using a remote procedure call, the task of the binding factory is split between the stub compiler, which generates client and server stubs using an interface description, the compiler, which compiles the stubs, and the linker, which binds these stubs with the client and server programs, together with the needed libraries. The coordination code consists of a script that calls these tools, using for instance a distributed file system to store intermediate results.

Detailed examples of binding factories may be found in the case studies of the present chapter (3.4) and of Chapters 4, 5 and 8.

3.3.3 The `export-bind` Pattern

The function of a binding factory may be described by a design pattern, which consists of two generic operations called `export` and `bind`.

1. The `export` operation takes as parameters a naming context and an object. Its function is to make the object known in the naming context, by creating a name for it in the context. Since the intent is to allow for a future binding, a side effect of `export` is usually to prepare data structures to be used for binding the object.

2. The `bind` operation takes as parameter a name of the object to be bound. The object must have previously been `export`ed. The operation delivers a handle that allows the object to be accessed. As noted in Section 3.3.2, the handle may take a variety of forms.

The socket example in Section 3.3.2 may be described in terms of this pattern: on the server site, `accept` is an instance of `export`, while on the client site, `connect` is an instance of `bind`. The following two examples give an overview of the use of the pattern for communication and for client-server binding. They are developed in more detail in Chapters 4 and 5, respectively.

Communication Binding

A communication service is implemented by a stack (or acyclic graph) of protocols, each one using the services of a lower-level protocol, down to an elementary communication mechanism. A protocol manages sessions: a session is an abstraction of a communication object, which provide primitives for message sending and receiving.

In our example, The concrete implementation of a session for a client-server communication consists of two objects, one on the server side, the other one on the client side. These objects communicate through a lower level mechanism, which we call a connection (Figure 3.14).

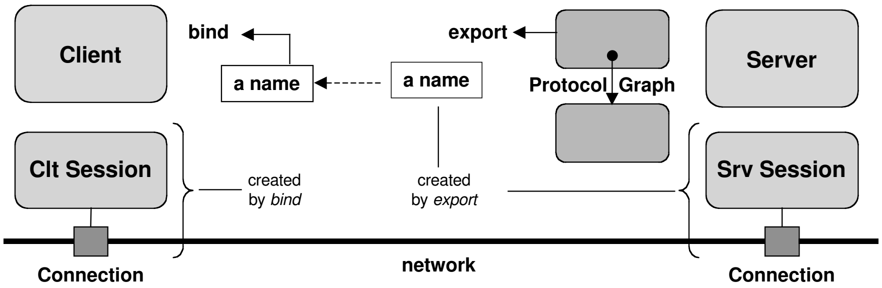

Figure 3.14. Using the `export-bind` pattern for communication binding

The server side session object is created by an `export` operation on the protocol graph, which acts as a session factory. The client uses a distribution-aware name (e.g., a name that contains the address and port number of the server) to create a client-side session object through a `bind` operation. The name may be known by convention, or retrieved from a name server.

Client-server Binding

We present a simplified view of the binding phase of Java RMI, an extension of RPC to Java objects. Java RMI allows a client process to invoke a method on a Java object, the target, located on a remote site. Like RPC, Java RMI relies on a stub-skeleton pair. Binding proceeds as follows (Figure 3.15).

On the server site (a), the target object is exported in two steps.

1. An instance of the target object is created, together with the skeleton and a copy of the stub (to be later used by the client).

2. The stub is registered in a name server under a symbolic name

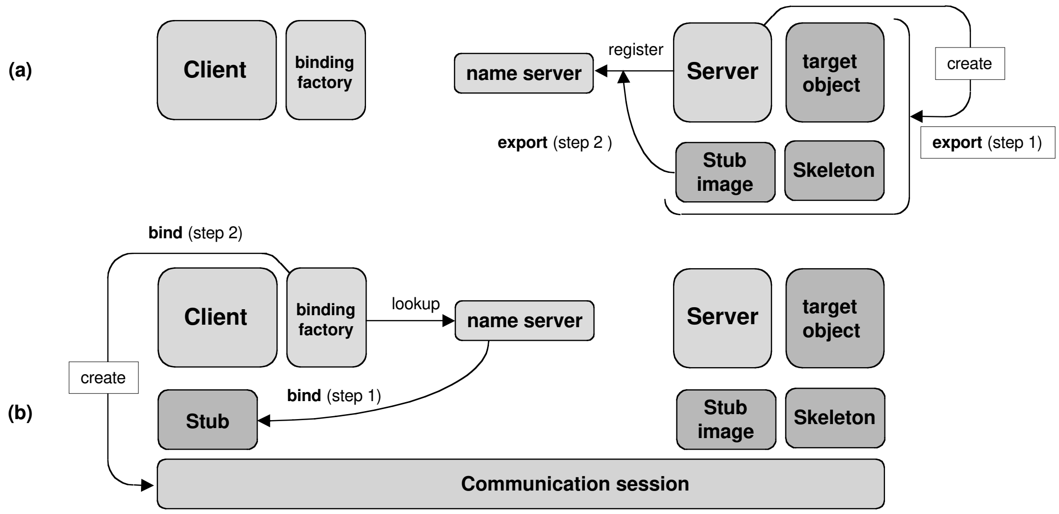

Figure 3.15. Using the `export-bind` pattern for client-server binding

On the client side (b), the client calls a binding factory, which also proceeds in two steps to complete the binding.

1. The stub is retrieved from the name server using the symbolic name.
2. A communication session is created, to be used by the client to invoke the remote object through the stub (the information needed to create the communication session was written into the stub during the `export` phase)

Note that the target object is exported twice: first to a local context on the server site, then to the name server. This is a usual situation; actually `export` is often called recursively, on a chain of contexts.

Also note that a phase is missing here: how the symbolic name is known by the client. This relies on an external mechanism (e.g., naming convention, or explicit transmission from the server to the client, etc.).

The Jonathan binding framework, described in Section 3.4, is based on the `export-bind` pattern. Further detailed examples may be found in Chapters 4 and 5.

Another view of binding may be found in [Shapiro 1994].

3.3.4 Putting it All Together: A Model for Service Provision

We may now present an overall view of the process of service provision, involving three parties: the service provider, the service requester, and the service directory. The process is organized along the following steps (Figure 3.16).

1. *Service creation.* The service provider creates a concrete implementation of the service (this may be a servant object that implements the service interface).

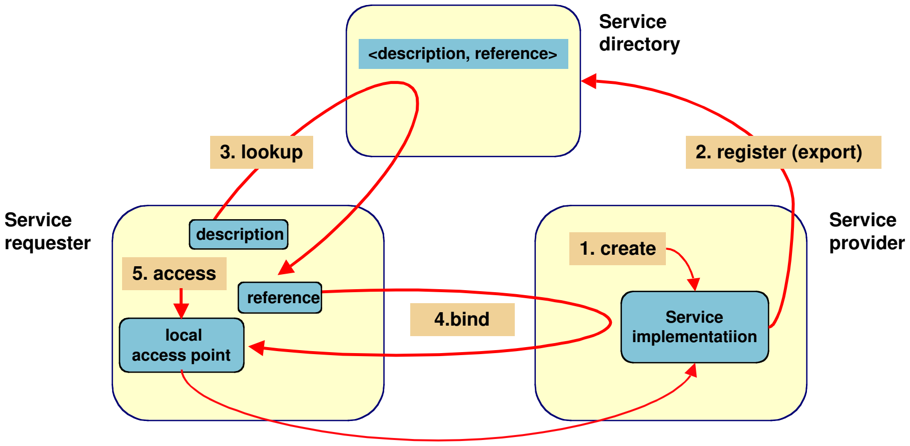

Figure 3.16. Global view of service provision

2. *Service registration.* The service provider registers the service with the service directory, by providing a link to the service location together with a name (or a set of attributes). This step actually implements an `export` operation.

3. *Service lookup.* The requester looks up the directory, using a description of the service. This description may be a name or a set of attributes. If successful, the lookup returns a reference to the service (or possibly a set of references, among which the requester may choose, e.g., according to QoS criteria).

4. *Service binding.* The requester performs the `bind` operation on the reference, thus obtaining a handle, i.e., a local access point to a binding object connected to the server. Depending on the implementation, lookup and binding may be implemented as a single operation.

5. *Service access.* Finally, the requester invokes the service, by a local call through the handle. The binding object forwards the call to the service implementation and returns the result to the caller.

This is a general scheme, which is defined in terms of abstract entities. This scheme is embodied in many different forms (e.g., Java RMI, CORBA, Web Services), in which the abstract entities are have specific concrete representations. Examples may be found in the next chapters.

3.4 Case Study: Jonathan: a Kernel for Distributed Binding

The notions related to naming and binding in an object-oriented environment are illustrated by the example of Jonathan, a framework for the construction of distributed middleware that may embody various binding policies. Jonathan is entirely written in Java and is available, as open source, on the site of the ObjectWeb consortium, at http://forge.objectweb.org/projects/jonathan.

3.4.1 Design Principles

Jonathan [Dumant et al. 1998] is a framework for building Object Request Brokers (ORBs), which are central components of middleware systems. The development of Jonathan was motivated by the lack of openness and flexibility of the currently available middleware systems. In order to facilitate the construction of ORBs adapted to specific run time constraints or embodying specific resource management policies, Jonathan provides a set of components from which the various pieces of an ORB may be assembled. These components include buffer or thread management policy modules, binding factories, marshallers and unmarshallers, communication protocols, etc. In addition, Jonathan includes configuration tools that facilitate the task of building a system as an assembly of components and allow the developer to keep track of the description of a system.

Jonathan is organized in four frameworks, each of which provides the Application Programming Interfaces (APIs) and libraries dedicated to a specific function. The current frameworks are the following.

- *Binding.* The binding framework provides tools for managing names (identifiers) and developing binding factories (or extending available ones). Different inter-object binding models may be managed, allowing for example the use of different qualities of service. This framework is based on the `export-bind` pattern presented in Section 3.3.3.
- *Communication.* The communication framework defines the interfaces of the components implied in inter-object communications, such as protocols and sessions. It provides tools for composing these pieces to construct new protocols. The communication framework is presented in Chapter 4.
- *Resources.* The resource framework defines abstractions for the management of various resources (threads, network connections, buffers), allowing the programmer to implement new components to manage these resources, or to reuse or extend existing ones.
- *Configuration.* The configuration framework provides generic tools to create new instances of components, to describe a specific instance of a platform as an assembly of components, and to create such an instance at boot time.

The main components of Jonathan (i.e., the classes that make up the frameworks) have themselves been developed using uniform architectural principles and a common set of patterns and tools (factories, helpers).

Jonathan includes two specific ORBs ("personalities"), which have been developed using the above frameworks. These personalities are Jeremie (5.4), an implementation of Java RMI, and David (5.5), an implementation of the OMG Common Request Broker Architecture (CORBA). The personalities essentially include implementations of binding factories, together with common services.

3.4.2 Naming and Binding in Jonathan

We first describe the structure of naming contexts, and then present the binding process and the binding factories.

Identifiers and Naming Contexts

Jonathan provides the notions of *identifier* and *naming context*, embodied in the `Identifier` and `NamingContext` interfaces. These notions are closely related: an identifier is created by a naming context, and this naming context remains associated with the identifier. Since Jonathan is a framework for building distributed object-based middleware, the entities designated by identifiers are objects. More precisely, since objects may only be accessed through an interface, an identifier created in a specific naming context is associated with an object type, defined by a (Java) interface. For example, the identifiers created by a protocol (a kind of naming context, see Chapter 4) designate sessions, which are communication objects defined by an interface associated with the protocol.

Note that the identifiers, as defined in Jonathan, do *not* have the uniqueness properties defined in 3.1 (e.g., an object may be designated by several identifiers). Identifiers may usually be viewed as (unbound) references.

To allow the construction of compound contexts (e.g., hierarchical contexts, such as described in Section 3.1.1), identifiers may be associated with chains of naming contexts. Such a chain may be (conceptually) represented as, for instance, `a.b.c.d` in which `a.b.c.d` is an identifier in the first context of the chain, `b.c.d` an identifier in the next context, etc. Note, however, that the identifiers do not necessarily explicitly exhibit this concatenated form, which is only presented here as an aid to understanding. Two operations, `export` and `resolve`, are respectively used to construct and to parse such chains. It should be noted that these operations may be used in a wide variety of situations; therefore, their signature and specification may differ according to the environment in which they appear.

- `id = nc.export(obj, hints)` is used to "export" an object `obj` to the target naming context, `nc`. This operation returns an identifier that designates object `obj` in the naming context `nc`. The initial designation of `obj` may have various forms, e.g., `obj` may be an identifier for the object in a different context, or a low-level name such as a Java reference. The `hints` parameter may contain any additional information, e.g., the name of another context to which `obj` should be exported. In most cases, `export` also has the side effect of building additional data structures that are subsequently used when binding to object `obj` (see Section 3.4.2). The `unexport` operation cancels the effect of `export` and precludes further use of the target identifier.

- `next_id = id.resolve()` is used to find the "next" identifier in a chain. This operation is the inverse of `export`: if `id1 = nc.export(obj, nc1)` exports `obj` to contexts `nc` and `nc1`, then `id1.resolve()` returns `id`, the identifier of `obj` in `nc`.

Using the conceptual concatenated representation, if identifier `id1` is represented as `a.b.c.d`, then `id1.resolve()` returns `id` represented as `b.c.d`, etc. Conversely, calling `nc.export(id1)` where `id1` is represented by the chain `x.y.z` returns an identifier `id` associated with `nc` and represented as `w.x.y.z`. Calling `resolve()` on an identifier that is not a chain returns `null`.

Since identifiers may need to be transmitted on a network, they may have to be encoded into a byte array, and decoded upon reception. The operations `encode` and `decode` respectively perform the encoding and decoding. While encoding is borne by the `Identifier` interface, decoding must be borne by each destination naming context since it returns identifiers that are only valid in that context.

- `byte_array` = `id.encode()` encodes `id` into a byte array `byte_array`.
- `id` = `nc.decode(byte_array)` decodes `byte_array` into an identifier `id` in the target naming context `nc`.

Bindings and Binding Factories

A *binding* is a connection between an identifier and the object that it designates (the target of the binding). A *binding factory* (or *binder*) is a special form of a naming context, which can create bindings. The form of the binding depends on the way of accessing the target from the binder. If they are in the same address space, the binding may take the form of a simple reference; if they are in different address spaces, the binding usually involves intermediate objects (known as proxies, delegates, or stubs), and possibly one or several communication objects if the target is remote.

Consider an identifier `id` in a naming context. If the naming context is also a binder, then a binding may be set up by invoking `id.bind()`, possibly through a chain of binders. If not, then the identifier may be `resolved`, returning another identifier associated with another context, and the resolution is iterated until the naming context associated with the identifier is a binder (Figure 3.17). An identifier that may neither be resolved nor bound is said to be *invalid* and should not be used.

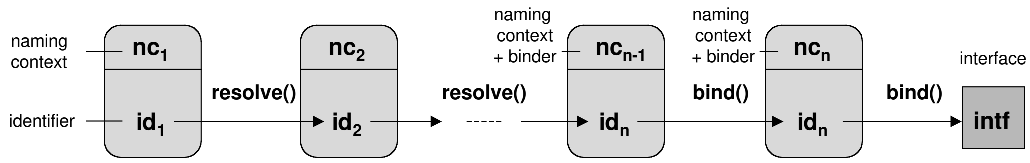

Figure 3.17. Resolving and binding identifiers in Jonathan

If the naming context of an identifier `id` is also a binder, then

`s = id.bind()`

returns an object `s` (the handle) through which the target of the binding may be accessed (the interface of `s` is therefore conform to that of the target). The target is an object designated by the identifier. The returned object `s` may be the target itself, but it is usually a representative of the target (a proxy). A special form of a proxy is a stub, which essentially holds a communication object to be used to reach the target. The `bind` operation may have parameters that specify e.g., a requested quality of service.

In order for `bind` to work, the target must have previously been `export`ed. The `export-bind` pattern is central in the Jonathan binding process and is applied in a wide variety of situations, e.g. communication protocols or remote object invocation.

3.4.3 Examples

The main examples illustrating the use of the `export-bind` pattern may be found in Chapters 4, 5, and 8.

In this section, we present two simple examples which will be used in more elaborate constructions.

Minimal Object Adapter

An *adapter* (further discussed in Chapter 5) is used to encapsulate an object in order to use it with a different interface, using the *Wrapper* design pattern (2.3.3). In the present case, the adapter is used on the server side of an ORB to encapsulate a set of servants. A servant is an object that implements a specific service. Requests arrive to the server in a generic form, such as `invoke(obj_ref, method, params)`, where `obj_ref` is a name which designates an object managed by the server. The function of the adapter is to convert this request into an invocation on a specific servant, using that servant's interface. This operation may be fairly complex, as explained in Chapter 5.

In this example, the adapter (called `MinimalAdapter`) is reduced to it simplest form: a binder that manages a set of servants, implemented as Java objects. Let `adapt` be a instance of `MinimalAdapter` and `serv` a servant. Then the operation `adapt.export(serv, hints)` creates a name `id`, of type `MoaIdentifier`, which designates the object `serv` in the context of the adapter. The parameter `hints` may be used to specify another naming context to which the object may be exported, thus allowing recursive exportation. The operation `adapt.unexport(id)` cancels the effect of `export` by disconnecting `id` from the object that it designates.

The operation `id.bind()`, where `id` is a `MoaIdentifier`, returns a reference on a servant object (in this specific implementation, a Java reference), if the name `id` is actually bound in the adapter.

In this example, the adapter is essentially a table of Java objects, implemented as a set of holders managed through a hash-coding scheme. Each holder contains a couple (identifier, object reference). The implementation includes some optimizations: there is a single holder pool for all adapter instances; holders that have been freed (by `unexport`) are kept on a reusable list, which reduces the number of holder creations. In order to prevent the holder table from being garbage collected (since no permanent reference points to it), a waiting thread is associated with the table when the first holder is allocated and disconnected when the last holder is freed.

When a new object is exported, a reusable holder is selected; if none is available, a new holder is created. In both cases, the holder is filled with a new instance `id` of `MoaIdentifier` and the reference of the target object. If the `hints` parameter specifies a naming context `nc`, the object is also exported to that naming context. Its identifier in this context, `id_nc`, is returned (and also copied in the holder). The `unexport` operation returns the holder to the reusable list and calls `unexport` on `id_nc` (thus recursively unexporting the object from a context chain).

The `bind` operation on a `MoaIdentifier` tries to find a matching holder using the hash code. The operation returns the object reference if a match is found, and returns `null` otherwise.

Domain

We now consider the implementation of a *domain* in Jonathan. Domains are used to solve the following problem arising in distributed computing. When an identifier is transmitted over a network, it is encoded (marshalled) on one side and decoded (unmarshalled) on the other side. In order to decode an identifier, one needs to know the naming context in which it has been encoded (because `decode` is borne by that context). If multiple protocols coexist, each protocol defines its own naming context and a mechanism is needed to "pack" an identification of this context together with an encoded identifier.

For example, let `id` be an identifier to be sent over a network, and let `nc` be its naming context. Then it is not enough to send the value `encoded_id = id.encode()` because `nc` is needed, at the destination site, to retrieve `id` by calling `nc.decode(encoded_id)`.

The solution consists in using a domain as a naming context, available on all sites, that identifies other naming contexts and wraps their identifiers in its own identifiers. `JDomain` is the standard implementation of a domain in Jonathan.

Let us consider again the problem of sending `id` (of naming context `nc`) over a network (Figure 3.18). Instead of `id.encode()`, the value actually sent is `encoding = jident.encode()`, where `jident = JDomain.export(id)`. `jident` is of type `JId` (the identifier type managed by `JDomain`). At the receiving end, `jident` is first retrieved as `JDomain.decode(encoding)`, and `id` is finally obtained as `jident.resolve()` (recall that `resolve` is the inverse operation of `export`).

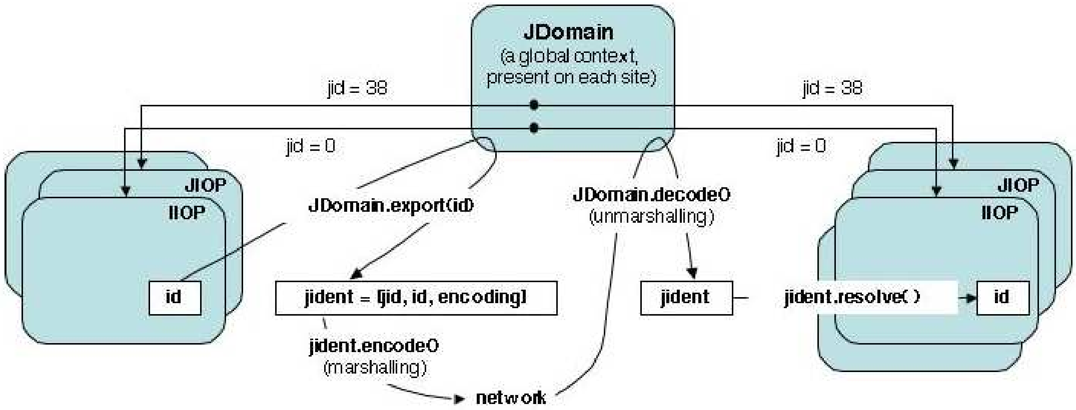

Figure 3.18. Transmitting an identifier over a network

We describe the implementation of `JDomain` in some detail, because it provides a good practical illustration of the main concepts related to identifiers in Jonathan. The inner working of `JDomain` is quite simple: each different naming context is associated with an integer (e.g., the naming context of the IIOP protocol is associated with 0, etc.). A `JId` encapsulates the identifier `id`, the integer value `jid` associated with the identifier's context, and an `encoding` of the pair (`jid, id`). Depending on how it was generated, a `JId` may actually contain (`jid, id`) or (`jid, encoding`). As is shown below, `id` can be generated from `encoding` and vice versa, if `jid` is known.

The main data structure is a linked list of holders (Figure 3.19). A holder contains an integer value (`jid`), a reference to a naming context, and the type (i.e., the name of the Java class) of this context. When a domain is created, this list is empty. An initial context (whose value is specified in the configuration phase) describes the predefined associations between binder classes and integer values.

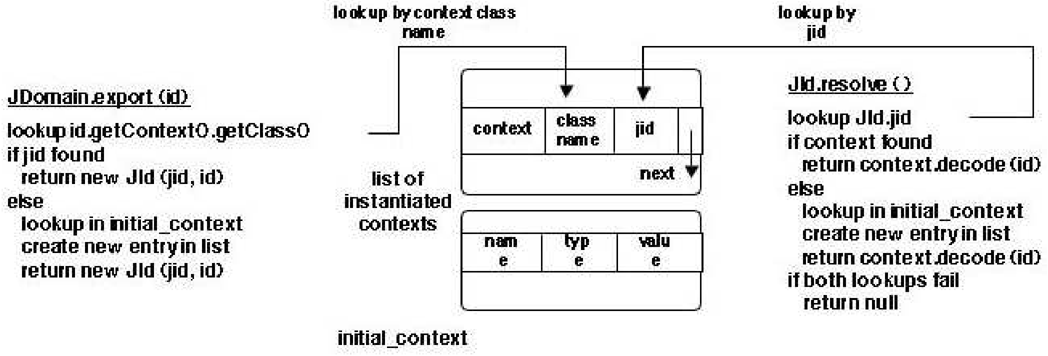

Figure 3.19. The implementation of a domain in Jonathan

A summary of the main operations follows.

- `export`. Exporting an identifier `id` to `JDomain` results in "wrapping" it into a `JId`. When `id` is exported, the context list is looked up for an entry (a holder) containing `id`'s class name in order to retrieve the corresponding `jid`. If no such entry is found, the initial context is looked up for a value of `jid`, and a new entry (`id`'s context, `id`'s class name, `jid`) is created in the list.

- `resolve`. Calling `resolve` on a `JId` returns `id`, which is found either directly or by decoding the encapsulated `encoding`. In the latter case, the context of `id` must be retrieved in order to call `decode`. This is done by looking up the context list using `jid` as a key; if this fails, the initial context is looked up and a new entry (`id`'s context, `id`'s class name, `jid`) is created in the list.

- `encode`. Calling `encode` on a `Jid` returns the value of the encapsulated encoding; if this value is null, an encoding of (`jid, id`) is generated (prefixing `id`'s encoding, generated by `id`'s encode method, with a 4-byte encoding of `jid`).

- `decode`. Two situations may occur. In the simplest case, the parameter of `decode` is a buffer containing the `encoding`. In that case, the value of `jid` is decoded from the first 4 bytes of `encoding` and a new `JId` is created with (`jid, encoding`). In the other case, `decode` is called on an unmarshaller, i.e., data delivered by a communication protocol; the data consists of `jid` and `encoding`. In that case, the identifier has been encoded by a remote system and a local naming context must be found to decode it. This is done by looking up the context list using `jid` as a key; if no context is found, then again the initial context is looked up.

- `bind`. This operation is provided for completeness, because it is not needed for identifier transmission. The operation `JDomain.bind(jid)` returns `nc`, the naming context associated with id.

An illustration of the use of `JDomain` for marshalling and unmarshalling identifiers is given in Chapter 5.

3.5 Historical Note

Names (be it in the form of identifiers or addresses) have been used since the very beginning of computing. However, a unified approach to the concepts of naming and binding in programming languages and operating systems was only taken in the late 1960s and early 1970s (e.g., [Fraser 1971]). A landmark paper is [Saltzer 1979], which introduces dynamic binding, drawing on the experience of the Multics system. Early work on capabilities [Dennis and Van Horn 1966, Fabry 1974] links naming with protection domains.

Distributed naming and binding issues are identified in the design of the Arpanet network [McQuillan 1978], and an early synthetic presentation of naming in distributed systems appears in [Watson 1981]. Later systematic presentations include [Comer and Peterson 1989], [Needham 1993], and Chapter 4 of [Tanenbaum and van Steen 2006].

In the early 1980s, with the development of networking, the issue of scale becomes central. Grapevine [Birrell et al. 1982], developed at Xerox PARC, introduces the main concepts and techniques of a scalable naming system: hierarchical name space, replicated servers, caching. Other advances in this area include the Clearinghouse [Oppen and Dalal 1983], GNS [Lampson 1986], and the design described in [Cheriton and Mann 1989] (but available in report form in 1986). At that time, the redesign of the Internet (then Arpanet) naming service is underway, and the Domain Name System [Mockapetris and Dunlap 1988] benefits from the experience acquired in all of these projects.

In the context of the standardization efforts for network interconnection, the X.500 standard [X.500 1993] is developed. Its main application is LDAP (Lightweight Directory Access Protocol) [LDAP 2006], a model that is widely used for enterprise directory services.

Trading systems use attributes, rather than names, as a key for finding distributed services and resources. Notable advances in this area are described in [Sheldon et al. 1991] and [van der Linden and Sventek 1992].

In the 1990s, flexibility, dynamism and adaptability are recognized as important criteria for open distributed systems. The issue of dynamic binding is reexamined in several research efforts (e.g., [Shapiro 1994]). The RM-ODP standard [ODP 1995a] defines a framework for flexible binding, which stimulates further research in this area, among which Jonathan [Dumant et al. 1998], Flexinet [Hayton et al. 1998], OpenORB [Coulson et al. 2002].

The trend toward flexible and adaptable naming and binding is amplified in the early 2000s by the wide availability of wireless networks and the advent of Web services. The notion of a service discovery service combines trading (search by attributes), fast dynamic evolution, flexibility, and security. There is no established standard in this area yet, but several research and development efforts (e.g., [Czerwinski et al. 1999, Adjie-Winoto et al. 1999, Guttman 1999, Waldo 1999]) have helped to identify relevant design principles and implementation techniques.

References

[Adjie-Winoto et al. 1999] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. (1999). The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 186–201, Kiawah Island Resort, near Charleston, South Carolna.

[Albitz and Liu 2001] Albitz, P. and Liu, C. (2001). *DNS and BIND*. O'Reilly. 4th ed., 622 pp.

[Bettstetter and Renner 2000] Bettstetter, C. and Renner, C. (2000). A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In *Proceedings of the Sixth EUNICE Open European Summer School*, Twente, Netherlands.

[Birrell et al. 1982] Birrell, A. D., Levin, R., Schroeder, M. D., and Needham, R. M. (1982). Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274.

[Brownbridge et al. 1982] Brownbridge, D. R., Marshall, L. F., and Randell, B. (1982). The Newcastle Connection — or UNIXes of the World Unite! *Software–Practice and Experience*, 12(12):1147–1162.

[Chandy et al. 1998] Chandy, K. M., Rifkin, A., and Schooler, E. (1998). Using Announce-Listen with Global Events to Develop Distributed Control Systems. *Concurrency: Practice and Experience*, 10(11-13):1021–1028.

[Cheriton and Mann 1989] Cheriton, D. R. and Mann, T. P. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183.

[Clark 1988] Clark, D. D. (1988). The design philosophy of the DARPA Internet protocols. In *Proc. SIGCOMM'88, Computer Communication Review, vol.18(4)*, pages 106–114, Stanford, CA. ACM.

[Comer and Peterson 1989] Comer, D. and Peterson, L. L. (1989). Understanding Naming in Distributed Systems. *Distributed Computing*, 3(2):51–60.

[Coulson et al. 2002] Coulson, G., Blair, G. S., Clarke, M., and Parlavantzas, N. (2002). The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126.

[Czerwinski et al. 1999] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., and Katz, R. H. (1999). An Architecture for a Secure Service Discovery Service. In *Proceedings of Mobile Computing and Networking (MobiCom '99)*, pages 24–35, Seattle, WA.

[Dennis and Van Horn 1966] Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155.

[Dumant et al. 1998] Dumant, B., Horn, F., Tran, F. D., and Stefani, J.-B. (1998). Jonathan: an Open Distributed Processing Environment in Java. In Davies, R., Raymond, K., and Seitz, J., editors, *Proceedings of Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, The Lake District, UK. Springer.

[Edwards 2006] Edwards, W. K. (2006). Discovery systems in ubiquitous computing. *IEEE Pervasive Computing*, 5(2):70–77.

[Fabry 1974] Fabry, R. S. (1974). Capability-based addressing. *Communications of the ACM*, 17(7):403–412.

[Fowler 1986] Fowler, R. J. (1986). The complexity of using forwarding address for decentralized object finding. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–120.

[Fraser 1971] Fraser, A. G. (1971). On the meaning of names in programming systems. *Communications of the ACM*, 14(6):409–416.

[Guttman 1999] Guttman, E. (1999). Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80.

[Hayton et al. 1998] Hayton, R., Herbert, A., and Donaldson, D. (1998). Flexinet: a flexible, component oriented middleware system. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal.

[Lampson 1986] Lampson, B. W. (1986). Designing a Global Name Service. *Proceedings of the 5th. ACM Symposium on the Principles of Distributed Computing*, pages 1–10.

[LDAP 2006] LDAP (2006). Lightweight Directory Access Protocol: Technical Specification Road Map. RFC 4510, The Internet Society. http://tools.ietf.org/html/rfc4510.

[McQuillan 1978] McQuillan, J. M. (1978). Enhanced message addressing capabilities for computer networks. *Proceeedings of the IEEE*, 66(11):1517–1526.

[Mockapetris and Dunlap 1988] Mockapetris, P. and Dunlap, K. J. (1988). Development of the Domain Name System. In *SIGCOMM '88: Symposium Proceedings on Communications Architectures and Protocols*, pages 123–133, New York, NY, USA. ACM Press.

[Needham 1993] Needham, R. M. (1993). Names. In Mullender, S., editor, *Distributed Systems*, chapter 12, pages 315–327. ACM Press Books, Addison-Wesley.

[ODP 1995a] ODP (1995a). ITU-T & ISO/IEC, Recommendation X.902 & International Standard 10746-2: "ODP Reference Model: Foundations".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[ODP 1995b] ODP (1995b). ITU-T & ISO/IEC, Recommendation X.903 & International Standard 10746-3: "ODP Reference Model: Architecture".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[Oppen and Dalal 1983] Oppen, D. C. and Dalal, Y. K. (1983). The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Transactions on Information Systems*, 1(3):230–253.

[Perkins 1998] Perkins, C. E. (1998). Mobile networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69.

[Pike et al. 1995] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254.

[Saltzer 1979] Saltzer, J. H. (1979). Naming and binding of objects. In Hayer, R., Graham, R. M., and Seegmüller, G., editors, *Operating Systems. An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 99–208. Springer-Verlag, Munich, Germany.

[Sandberg et al. 1985] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA).

[Shapiro 1994] Shapiro, M. (1994). A binding protocol for distributed shared objects. In *Proc. Int. Conf. on Distributed Computing Systems*, Poznań (Poland).

[Sheldon et al. 1991] Sheldon, M. A., Gifford, D. K., Jouvelot, P., and O'Toole, Jr., J. W. (1991). Semantic File Systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25.

[Tanenbaum and van Steen 2006] Tanenbaum, A. S. and van Steen, M. (2006). *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition. 686 pp.

[UPnP] UPnP. Universal Plug and Play. The UPnP Forum. http://www.upnp.org.

[van der Linden and Sventek 1992] van der Linden, R. J. and Sventek, J. S. (1992). The ansa trading service. *IEEE Comput. Soc. Tech. Comm. Newsl. Distrib. Process.*, 14(1):28–34.

[Waldo 1999] Waldo, J. (1999). The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82.

[Watson 1981] Watson, R. W. (1981). Identifiers (naming) in distributed systems. In Lampson, B. W., Paul, M., and Siegert, H. J., editors, *Distributed Systems – Architecture and Implementation (An Advanced Course)*, volume 105 of *Lecture Notes in Computer Science*, pages 191–210. Springer-Verlag, Berlin.

[Wieringa and de Jonge 1995] Wieringa, R. and de Jonge, W. (1995). Object identifiers, keys, and surrogates: Object identifiers revisited. *Theory and Practice of Object Systems*, 1(2):101–114.

[X.500 1993] X.500 (1993). International Telecommunication Union – Telecommunication Standardization Sector, “The Directory – Overview of concepts, models and services”, X.500(1993) (also ISO/IEC 9594-1:1994).