Naming and Identification in Distributed Systems: A pattern for Naming Policies

Pedro Sousa, António Rito Silva and José Alves Marques

INESC/IST Technical University of Lisbon

R. Alves Redol n°9, 1000 Lisboa, PORTUGAL

Tel: +351-1-3100287, Fax: +351-1-3145843

Pedro.Sousa@inesc.pt

This paper is submitted to the Conference on Pattern Languages of Programs (PloP '96), Allerton Park, Illinois, September 4-6, 1996.

Abstract

This paper addresses the problem of naming and identification in distributed systems. It describes an architecture centered on objects, names and contexts which support different naming policies. The pattern solution is driven taking into account the information embedded in names and the function, structure and meaning of names.

1 Problem

A traditional approach to the naming problem is to establish a one-to-one association between names and objects. Such names are known as Unique and Universal Identifiers (UUID)s.

UUIDs are used as object identifiers, to distinguish an object from all others. However there are many situations where objects need to be perceived at different levels of abstraction, and must be associated to more than one identifier.

UUIDs are also used as object references, as a way to denote a particular object. However the performance overhead in the creation and management of UUIDs makes this approach not suitable to support fine-grain objects.

Next, we will see an example in which UUIDs are not a solution neither as object identifiers nor as object references.

UUIDs as Object Identifiers

Consider the figure 1, where object A and object B are two replicas of object C, and *nA* and *nB* be the names used to reference them. Let us now consider two entities that interact with these three objects: the replica manager, that is responsible for preserving a consistency criteria between the state of objects A and B, and applications at different nodes that use objects A or B as being object C.

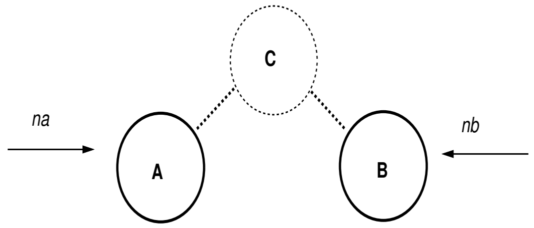

Figure 1: Object A and B are physical replicas of Object C

On one hand, the manager has to distinguish between physical objects A and B, because it must address explicitly to one or to the other. On the other hand, since replication is transparent to the applications, applications must perceive both objects A and B as being C.

The problem is then how to make objects A and B be identified as C to applications and as A and B to the manager ?

UUIDs as Object References

In systems supporting a flat object space of fine-grained objects, performance of reference manipulation and object creation/deletion is crucial to overall system's performance.

Since the creation and management of UUIDs is somewhat expensive in distributed systems, a one-to-one association between UUIDs and objects may become too expensive in such scenario. Optimizations are twofold:

- Object references should be as small as possible.
- UUIDs should not be associated to short lived and transient objects.

The problem is what names should be used as object references in such systems ?

2 Forces

In the general case, the naming problem addresses five forces: the first concerns with the function of names, the second with the structure of names, the third with the meaning of names, the forth with the information a name contains, and the fifth with the persistence of the meaning of names.

- **Functions of Names**.

  Names can have two basic roles: identification and denotation [1]. Identification allows an object to be distinguished from all others, and is normally achieved by comparing object identifiers. Denotation is a mean to refer to objects, and normally is achieved resolving the object's name.

  Assigning both functions to the same names, favors simplicity and efficiency. Assigning each function to different names favors flexibility, because the set of objects an application can refer to may differ from the set of objects it can identify.

- **Structure of Names**

  Names are structured entities, often being a concatenation of several fields. The names that applications use may have the same structure, or not.

  Imposing a fixed structure for names among applications favors code reuse, because the code handling names in different situations can be the same. Nevertheless a common structure may be too restrictive, due to the different requirements applications may have.

  For example, computation intensive applications may choose to use binary numbers as names, whereas communication intensive applications may choose to use textual names to simplify name transfer over-the-wire.

- **Meaning of Names**.

  If a name has the same meaning in every application, then it can be sent and received between different applications regardless of their current location in the system.

  Support names with a single meaning makes object sharing trivial. However, it requires cooperation among all applications to ensure that, once a meaning is given to a name in one application, all other applications cannot give another meaning to the same name.

- **Information in Names**.

  Names can contain information about the objects they denote or identify, such as their type or location.

  Under certain conditions, the inclusion of such information allows major optimizations because the holders of object's names can get that information immediately, without resolving the name. For example, the inclusion of the object location in the object's name is a common way to optimize name resolution in database systems. However, it does not promotes flexibility and evolution because, the included information becomes an invariant objects must fulfill during their lifetime.

  Names that contain no information remain valid regardless of object evolution and changes. They favor object encapsulation and system flexibility.

- **Persistence of Names**.

  The meaning of a name can persist application's runs, or it can change without the agreement of applications that have a copy of that name.

  Preventing the meaning of a name to change without the knowledge of clients, allows them to resolve a name without testing if it's meaning hasn't changed since the last time it was resolved. However, this name persistence is very hard to support in a large scale distributed system, since the system must keep trace names and find out which clients hold a given name, which is basically the problem of distributed garbage collection.

  If a name can change its meaning without a client consent, then some identification mechanism must be provided to allow clients to confirm that the names denotes the objects they expect.

  A system supporting persistent names does not have what is normally called "dangling references". The opposite is not necessarily true.

3 Solution

UUIDs as Object Identifiers

The problem exists because both the applications and the manager denote objects A and B through the names nA and nB, and applications can only identify object C, whereas the manager can only identify objects A and B.

The solution is to keep denotation and identification assigned to different names. Names nA and nb are used solely as object references. Assuming objects have a $getId()$ method that returns its identifier, then $na \rightarrow getId()$ and $nb \rightarrow getId()$ should return IC if called from applications and should return IA and IB if called from the system, where IA, IB and IC are the identifiers of Objects A, B and C, accordingly.

The $getId()$ method is context dependent, like the traditional UNIX $open()$ system call is dependent of contextual information such as the current directory and owner rights.

Names nA and nB can have the same structure or meaning in different applications. However, they cannot contain information specific to any object, because applications would relate it to object C, whereas the system would relate it to objects A and B, accordingly.

UUIDs as Object References

The problem exists because UUIDs are large and expensive names.

The solution is to assign local names to objects, and use them to refer and to identify objects locally, within an application. Assign one UUID to each local name that is sent or received from other applications. Local names are those valid within the scope of the application.

Each time a local name is sent to another node, the corresponding UUID is sent instead, which will be then converted to the receiver's local name. This solution is traditional in distributed systems supporting fine grain objects.

Local names have different structures and meaning in different applications, and thus they can be managed autonomously by each application.

4 Participants

The naming problem can be solved using three entities:

- **Objects:** are the entities to be identified and named. Objects can have multiple names, both in the same context and in different contexts.
- **Contexts:** define the meaning of names. Each context holds the association between names and objects. Contexts can be chained: a name in a context is associated with another name in another context.
- **Names:** are values that refer to objects. A name refers a single object in a given context, but a name can designate different objects in different contexts.

5 Collaborations

There are two main scenarios for interaction between objects, names and contexts:

- Objects can be *associated/de-associated* with names within a context. The association can only succeed if that name is not associated with any other object in that context. However, the same object can be associated with different names in the same context.
- A name can be *resolved* into an object in a given context, and yields the object associated with it. If the name is associated with another name in another context, that resolution should proceed resolving that name in that context.

6 Variations

In order to introduce some variations of this pattern, it is convenient to define the concepts that embodied the forces presented in section 2.

- Names are called *identifiers* when are used for identification purposes, and *references* when used for denotation purposes.
- A *name space* defines the structure of names an applications can recognize as valid. Names are *universal* if they are valid in all applications that refer to an object, and are *local* otherwise.
- Names are *absolute* if they refer to the same object in all applications where it is valid. Names are *relative* if they refer to different objects in different applications.
- Names can be *pure*, if they contain no information concerning the objects they denote and *impure* otherwise.

The universal and absolute property of names are determined by the way contexts and name spaces are combined, as shown in figure 2.

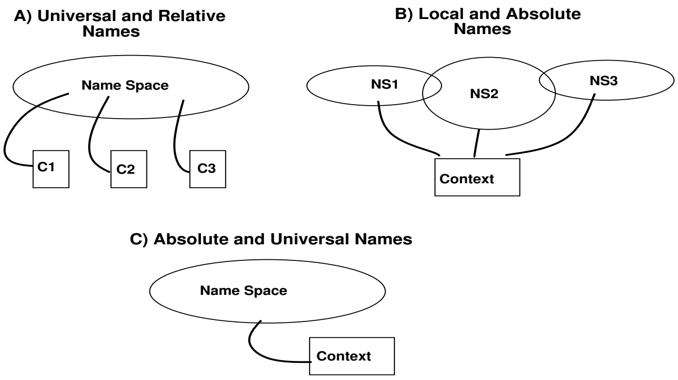

Figure 2: Names Spaces and Contexts

In scenario A, there is a single name space which is associated to several contexts. Names are universal and relative because their meaning depends on the context they are resolved.

In scenario B, there are several name spaces and a single context. Names can have a single meaning, even though they are not universal names.

In scenario C, there is a single name space and a single context. Therefore, names are universal and absolute.

7 Implementation

The support of different functionality imposes different requirements on contexts and name spaces.

- **Transparency to Distribution.** It is achieved when applications use names from a single name space to denote objects regardless of their location in the system. To support it, the context and name space associated with the each application must support the association between a name with any object in the system.
- **Transparency to Object Migration.** It is achieved when objects are able to change location without their movements be noticeable by their clients. To support it, the names resolution must not be based on location information included in the names. Names must be pure!
- **Transparency to Object Sharing.** Object sharing occurs when two or more applications have references for the same objects. Applications can exchange object names if they have the same structure and meaning. Thus, to support transparent sharing, object names must be universal and absolute.
- **Support for Interoperability.** Interoperability is achieved when heterogeneous application are able to cooperate. Due to the heterogeneity, applications cannot resolve the names created by other applications. Names are resolved asking the application that created that name to resolve it. Thus, applications play the role of contexts, and their names must be universal and absolute.

Generation of UUIDs

Generation of relative and local names is easy because it can be done by a single entity. Generation of UUIDs is hard because generators cannot produce repeated names, and they must have the same meaning everywhere.

Typically, UUIDs are produced by concatenating a generator label with a local counter. Assigning a unique label to each generator is the hard problem, because it implies cooperation among all possible generators. Label assignment normally follows an hierarchical structure, starting from some central decision.

Since machines have no unique labels, a combination of IP addresses and time are often used as such, for example in CORBA [2], DCE [3] and in OLE [4]. However the use of time fields has two major drawbacks: (i) time fields must be very large and UUIDs that include time are at least 128 bits long. (ii) time fields establish an upper limit to the generation ratio and time to generate a number, regardless of the processor's mips, because one must wait for the clock to change when local counter are exhausted.

To produce compact UUIDs, labels cannot be statically assigned but instead, generators should request them on demand. Each label allows the generation of a small set of names.

8 A Case Study: DNS

The Domain Name System (DNS) is the name service used in current across the internet and is probably the most known and used name service.

DNS distinguishes two types of entities: domains and objects. Domains support our notion of context, but they are also the entities that handle the administrative issues. Objects can be of any kind but they are normally computer's IP addresses. Names can be assigned and de-assigned to IP addresses in a given domain and later names can be resolved in that domain.

Now w'll see the forces resolution in DNS:

- **Function of Names**.

  In DNS names are used for denotation only. There is no way an application can find out weather two different names refer to the same object (computer or domain) without resolving them: DNS names are not identifiers.

  This decision greatly simplifies the implementation of DNS because it does not need to prevent different names to be assigned to the same computer.

  However, since DNS provides no identification mechanism for domains, it does allow changes in the hierarchical structures of domains. For example, it would not be practical to put United State domains one level below as other countries, moving domain *edu* to *us.edu* and so on. The Global Name Service solves this problem precisely by providing an identification for contexts [5].
- **Structure of Names**.

  DNS names are textual with an hierarchical structure. A name consist of one or more strings called labels separated by the delimiter ".". There is no delimiter at the beginning or end of a domain name. Since domain and object names have the same structure, DNS supports a single name space and DNS names are universal.

Such approach allows the system to support a single name resolution mechanism. Servers can resolve a name without knowing if it will resolve into a domain or a computer.

- **Meaning of Names**.

  Domain names have the same meaning for all clients: they are absolute names. However, the same computer name can denote different computers in different domains: they are relative names.

  Supporting absolute domain names allows applications in the network to interchange domain names without whoring with their meaning. Implementing absolute names is fairly easy in hierarchical structure: each domain "child" domain can only be registered in the "parent" domain.

  Computer names do not need to be absolute since that DNS servers do not recognize computer names alone. They are always presented for resolution together with a domain name. In most implementations, this is done by the library at the client that keeps a list of default domain names that are appended to the computer names automatically to any single component name before resolution.

  DNS follows the approach A) of figure 2 where each domain is a context. Computer names are resolved in these contexts but domain names are resolved in a single context. Notice that context is fiscally supported by an hierarchical structure of domain servers.

- Information in Names.

  DNS names hold no information about object's location, nor about their properties. Names in the same domain can denote widely spread entities and these entities can be domains or computers.

  This ensures the idependence between the name service and the objects whose names it manages.

- Persistence of Names.

  DNS names are not persistent entities. If a client holds a name, it cannot use the DNS service to know if the computer it is accessing is the one it expects.

  Supporting persistent names does not mean the IP address cannot change without the client knowledge. It means the IP address that the name is translated into is always the one to with that server responds to. In other words, it means the relations between names and IP addresses, and the relations between IP addresses and computers cannot be managed separately.

  This is clearly out of the scope of DNS, which does distinguish IP addresses from computers.

9 Known Uses

Object Systems

Most object systems use UUIDs both as object identifiers and as object references. The system supports a single name-space and a single context of UUIDs which is available to all applications, as shown in figure 3.

To avoid the high cost of UUIDs manipulation, relative names are often used to denote objects as aliases to UUIDs. Typically, relative names are just references (or virtual addresses) to the objects holding the UUID.

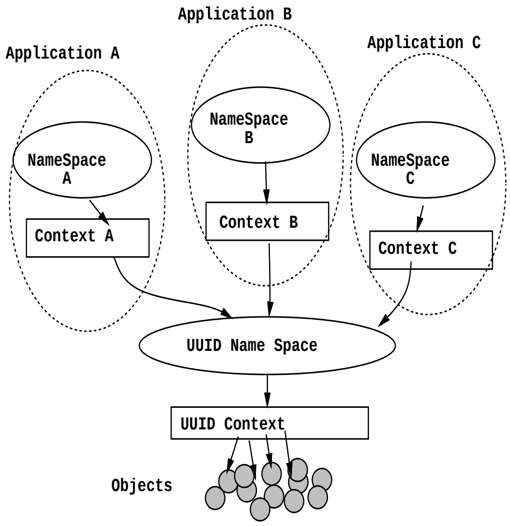

Figure 3: UUIDs in Object Systems

This is the traditional approach in distributed systems, such as [6, 7, 8], in persistent systems, such as [9, 10, 11] and in both distributed and persistent systems [12, 13]. Notice that since UUIDs are also used for identification purposes, each object must be associated with one UUID.

In persistent systems, where object migration is not supported, the resolution contexts can be encoded in the name, and thus resolution can be speeded up. Thus, in these systems UUIDs are often impure names because they include information about the cluster the object belongs to, such as in [14] or [15]. Persistent names are normally found in persistent object systems with support for automatic garbage collection such as [12, 13].

Internet Addresses

Internet addresses are absolute and universal names, since they have the same meaning and structure everywhere, but do not provide host identity because the same host may have different IP addresses and IP addresses change when a machine moves in the net.

IP addresses are generated hierarchically having the Network Information Center organization as root. Thus IP addresses are allocated in large chunks, which make the IP name space sparsely used.

Distributed File Systems

In file systems, there is a clean separation between file denotation and file identification. UUIDs are used as file identifiers, but they are known solely within the system. Users and applications only handle file references, which are textual and hierarchically structured names.

The properties of file names (file references) changes with the system. In NFS [16], file names are relative, impure and universal names. Relative, because their meaning depends on the particular mount points; Impure, because two files in the same directory must be in the same location; Universal, because all names follow a single structure. In AFS [17], file names are also universal, but they can be relative or absolute, depending if they are in the private or shared volume.

Telephone Numbers

Conceptually, telephone numbers are hierarchically structured by country codes, city codes and local phone number. However they are not absolute names.

Phone numbers are relative numbers because there is no agreement on international access code. In practice phones have different numbers in different countries. Each country represents a different context.

References

[1] Pedro Sousa, António Rito, and José Alves Marques. Object Identifiers and Identity : A Naming Issue. In *Proc. of the Int. Workshop on Object Orientation in Operating Systems - IEEE*, Lund, Sweend, August 1995.

[2] OMG. *CORBA 2.0/Interoperability*. OMG TC Document 95.3, March 20 1995. Revision 1.8 jm.

[3] OSF. *DCE Application Development Guide*. Open Software Foundation, Grenoble, March 1991.

[4] Kraig Brockschmidt. *Inside OLE2*. Microsoft Press Programming Serie, 1993.

[5] B. W. Lampson. Designing a Global Naming Service. In *Symp. on Principles of Distributed Computing*, pages 1–10. ACM, August 1986.

[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of Summer Usenix*, July 1986.

[7] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[8] Marcel Schelvis and Eddy Bledoeg. The Implementation of a Distributed Smalltalk. In *ECOOP '88*, pages 212–232, Oslo, August 1988.

[9] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[10] Andrew Straw, Fred Mellender, and Steve Riegel. Object Management in a Persistent Smalltalk System. *Software - Practice and Experience*, 19(8):719–737, August 1989.

[11] Benzaken, Delobel, and Harrus. Clustering Strategies in O2: An Overview. In François Bamcilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-Oriented Database System - The Story of O2*, pages 385–410. Morgan Kaufman Publishers, 1994.

[12] Antonio Albano, G. Ghelli, Maria Eugenia Occhiuto, and Renzo Orsini. A Strongly Typed Interactive Object Oriented Database Programming Language. *ACM Transactions on Database Systems*, June 1985.

[13] Pedro Sousa, André Zúquete, Nuno Neves, and José Alves Marques. Persistence in a Heterogeneous Distributed Oriented Environment. *The Computer Journal*, 37(4), 1994.

[14] Carey et al. Object and File Management in the EXODUS Extensible Database System. In *Proc. of the 12th Int. Conf. on Very Large Databases*, 1986.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[16] Sun. *Network File System Protocol Specification*. Sun Microsystems, February 1986. Revision B.

[17] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, and A. Spector. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.