# Scalable Naming in Global Middleware

G. C. Ballintijn
Faculty of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

M. van Steen
Faculty of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

A. S. Tanenbaum
Faculty of Sciences
Vrije Universiteit
Amsterdam
The Netherlands

## Abstract

Middleware systems that are distributed worldwide are difficult to build due to all kinds of scalability issues. Problems already start when considering how naming should be done. It is commonly thought that organizing symbolic names into a hierarchical name space, and subsequently distributing the implementation of that space in a hierarchical fashion as well, is appropriate for worldwide naming. This solution is adopted by DNS. We argue that such approaches are not suited for naming in future global middleware systems. Instead, name space implementations should make heavily use of flexible and large-scale replication in order to exploit locality as much as possible. Current systems cannot be easily adapted to this extent. We describe a novel approach to implementing naming systems in large-scale, worldwide distributed middleware.

**Keywords**: symbolic names, naming systems, distributed systems, wide-area systems.

## 1 Introduction

To facilitate the development of distributed applications, middleware systems provide a high degree of distribution transparency. Unfortunately, current middleware systems are designed for local-area networks only. However, it becomes increasingly unrealistic to assume that a group of users is dispersed only across a relatively small area, given the emergence of global virtual communities, such as virtual enterprises and Internet social communities. These global communities need the same distribution transparency as offered by traditional middleware, but are also in need for scalable systems.

The problem is that the approaches used in current middleware systems are inherently limited in scalability. As we argue elsewhere [17], a major source of scaling problems is caused by the fact that most sys-

tems are still based on a single-server, multiple-client model. This model is not suited for wide-area communication, which is characterized by long latencies and unreliable transport mechanisms, leading to a considerable loss of performance.

Another major source of scaling problems comes from the limitations of services that form part of middleware. In this paper, we address one particular important service, namely naming. A naming service allows different users to find, access, and share distributed resources. No middleware can do without a proper naming service. Consequently, if the implementation of the naming service does not scale, it hardly makes sense to put any effort in attempting to scale other parts of the middleware system.

As we argue in this paper, *all* currently implemented naming systems lack proper scalability in the face of general naming requirements. These systems not only include those implemented for middleware such as CORBA [10] and DCE [12], but also the Internet's Domain Name System (DNS) [1, 6]. Their main problem is that a name can only be resolved at the specific, static location where it is stored. This dependency prohibits scalable name resolution for objects that are not statically tied to a single location, even if an object changes location only once in its lifetime. The location-dependency problems occur at different levels of abstraction: from low-level object references to high-level symbolic names.

In this paper, we show what is wrong with current naming systems, including those for wide-area networks, and why their implementations are unsuitable for global middleware where a general-purpose naming solution is needed. We propose a new naming architecture supporting user-defined location-independent names. An improvement over existing architectures is that it uses the location independence to store names at convenient, dynamically chosen locations, and thereby implement scalable name resolution. We further show how to implement scalable worldwide

naming systems using this architecture. To simplify our discussion, we concentrate in this paper on object-based middleware systems only.

The rest of this paper is structured as follows. Section 2 describes how naming is used in different places in middleware, and what problems can occur. Section 3 describes the name space model we support and other naming service characteristics. In Section 4, we describe the architecture of our naming service. Section 5 then discusses how our naming architecture supports scalability. In Section 6 we discuss related work and we draw our conclusions in Section 7.

## 2 Names in Middleware Systems

Names are used in middleware systems at different levels of abstraction. At the lowest level of abstraction, objects are named through local pointers. These pointers allow a client to invoke methods on an object and are usually directly interpreted by the hardware. At a higher level of abstraction, objects are named through systemwide object references. Object references allow a client process to **bind** to an object. This binding procedure effectively resolves an object reference to a local pointer.

At yet a higher level of abstraction, objects are referred to through user-defined symbolic names. Normally, these names are organized in hierarchical name spaces, such as in the UNIX file system or DNS. The symbolic names are usually bound to object references. Besides symbolic-naming systems, there are also property-based naming systems. These systems are also known as directory services or traders. A well-known example is the X.500 directory service [11].

In this paper, we do not consider directory services or local pointers. Instead, we concentrate on object references and symbolic-naming systems only.

### 2.1 Scalability Problems with Object References

To use an object, we first need to resolve its object reference to a local pointer. The resolution process results in the creation of a local implementation of that object's interface. A pointer to the implementation is subsequently returned. We assume, for this discussion, without loss of generality, that local implementations are mere proxies that communicate with a remote implementation of the object.

To make resolution of the object reference simple and efficient, an object reference generally contains all information needed to contact the referred object. In CORBA [10], for example, object adapters generate references for the objects they manage. In these references, they encode the network address of the server at which they reside. As we argue in [16], encoding location-dependent contact information in an object reference can never scale worldwide.

The main problem with encoding location information is that once the object moves to another location the reference becomes invalid. Middleware systems use techniques such as forwarding pointers or broadcast to deal with this situation (see, for example, [2, 9, 13]). However, both techniques have inherent scalability problems that make them unsuitable for global middleware. It is also unclear how forwarding pointers can deal with heavily-replicated objects.

Our solution to these problems is the introduction of persistent location-independent object identifiers. To use an object, we let a *location service* resolve the object's object identifier to what we call contact addresses. A contact address describes exactly how and where an object can be contacted, and is similar to an object reference in systems like CORBA. The mapping of object identifier to contact address is transient, it can change over time.

An important property of our location service is that when it looks for an object's contact addresses, it starts to search in the proximity of the client. It basically uses an efficient expanding-ring search algorithm. Using this property, our location service finds the nearest contact address first, thus exploiting locality and improving scalability. The implementation of our wide-area location service is described in [18].

### 2.2 Scalability Problems with Symbolic Names

Existing naming systems that support symbolic names have a number of scalability problems that make them unsuitable for global middleware systems. The problems are not immediately obvious, and are best illustrated by taking a look at DNS.

DNS provides an extensible hierarchical name space which is primarily used to name Internet hosts. In the naming hierarchy, more general naming authorities delegate responsibility for parts of their name space (subdomains) to more specific naming authorities. For example, the naming authority responsible for the `.com` domain, delegates the responsibility for the `intel.com` domain to the Intel company. This allows Intel to add to its domain what ever subdomain or hostname it wants.

Resolving a hostname in DNS consists, conceptually, of contacting a sequence of name servers. The domains stored by this sequence of name servers are

increasingly specific, allowing the resolution of an increasing part of the hostname. For example, to resolve the hostname `www.intel.com`, the resolution process visits, in turn, the name servers responsible for the root (i.e., "."), `com`, and `intel.com` domain, respectively. The last name server will be able to resolve the complete hostname.

DNS is claimed to scale worldwide if it can be assumed that name-to-address mappings do not frequently change. The reason is that name servers can effectively cache mappings and intermediate results during name resolution. This cached information can be used to resolve names more efficiently the next time. This approach has indeed shown to work for the current name space.

However, the efficiency of DNS relies not only on the assumption that mappings hardly change. It also relies on the implicit assumption that the *owner* of a named object is in the vicinity of the name server that stores the object's address. This assumption is generally correct for Internet hosts, and allows for an efficient and scalable distribution of the name space implementation. Under this assumption, an update operation—which is carried out by the owner of an object—is then a local operation.

The assumption that a name can be stored in one fixed location is wrong when talking about general-purpose middleware systems. The assumption is particularly wrong when considering long-lived objects. When an object moves a great distance because its owner moves, it will be useful if the name would "follow" the object and user in the sense that resolving that name would still exhibit locality. For example, when a user moves from Amsterdam to New York, its home directory should be moved to New York as well. The same reasoning about locality applies to replicated objects. An object's name should be stored near every replica of the object to provide local name resolution. For example, if a web-site has replicas in London and Tokyo, the directory that names the pages in the web-site should have replicas in London and Tokyo as well, allowing names to be resolved locally.

The problem we are thus faced with is to invent a naming service that can continue to support locality in name resolution in the presence of object mobility and replication.

## 3 Naming Model

The goal of our global middleware system is to support $10^{12}$ objects, distributed worldwide. Our naming system therefore has to deal with a large number of names as well as large geographical distances. The objects themselves have two important characteristics that also influence the design of the naming service: (1) objects might be replicated; (2) objects are not statically bound to a single location. Even though (most of) the problems created by these characteristics are solved through location-independent object identifiers and the use of a location service (as discussed in Section 2.1), the previous section clearly shows this is not enough. A naming service has to be designed with these characteristics in mind.

The goal of our naming service is to bind symbolic names to persistent location-independent object identifiers. In our model, an object identifier will subsequently be further resolved to a contact address by means of a location service. As we discuss later, in our model, the location service is also used for the implementation of the naming service. However, since the location service deals with name changes due to mobility and replication, we can safely assume that name-to-object identifier mappings are relatively stable.

The structure of our name space is a directed graph with labeled arcs. The structure is based on the directory concept, as found in the UNIX file system and the Prospero naming system [7]. Interior nodes of the graph represent directories, whereas leaf nodes represent user objects such as files. A directory is basically a table of references, or more specifically, object identifiers. Each reference points to another directory or a user object, of which a contact address can be looked up by means of our location service. Each reference in a directory is indexed by a simple name (i.e., a label).

A name in the name space is a sequence of labels. The labels indicate a path through the naming graph. Name resolution thus consists of step-wise traversing the graph, each step resulting in a reference. Name resolution starts at a predefined starting directory. At this directory, the head label of the sequence is used to look-up the reference to the next directory. Resolution recursively continues with the rest of the sequence. The last label of the sequence finally maps to a reference to the object requested by the client.

We do not give a complete description of our name space model in this paper, but define only those parts needed for understanding our naming architecture. For instance, an important aspect left unspecified is closure, i.e. how to choose the starting directory needed to resolve a name. Other important aspects left out deal with the question whether to restrict the naming graph to a tree, or more mundane questions regarding character sets.

# 4 Implementation Architecture

The naming architecture presented in this section may look similar to architectures presented in previous work. It differs, however, significantly in that it provides a *loose* coupling between directories and the servers that store them. A loose coupling allows the naming system to move directories to servers where they can provide the best performance. Previous architectures, such as DNS, have always used a strong coupling between directories and name servers, tying a directory to a fixed location. A second important distinction with previous work, is that our architecture allows every directory to have its own replication strategy (tailor made, if necessary). In the following discussion, we ignore security aspects for the sake of brevity.

## 4.1 Directories

Directories are implemented using Globe's Distributed Shared Object (DSO) paradigm, as described in [17]. In this paradigm, a set of replicas work together to provide the notion of one conceptual (distributed) object (see Figure 3). The DSO is identified by an object identifier. Replicas provide the contact points needed to contact the DSO. To allow a replica to be run anywhere, its implementation is completely self-contained, that is, a replica contains all the code that controls its behavior.
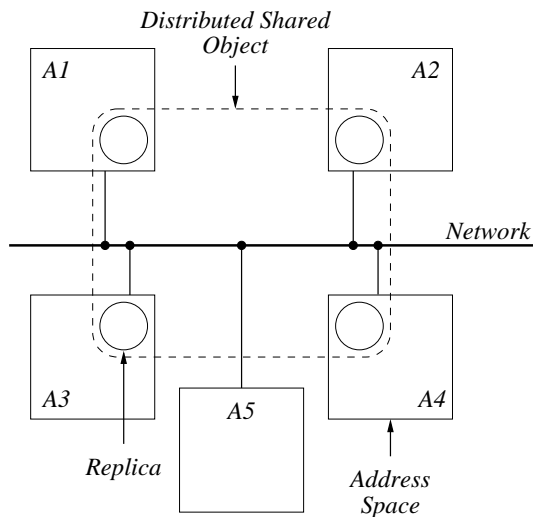


Figure 1: A distributed shared object in a network

To invoke operations on a directory, a client must first **bind** to that directory. An important part of the binding process is finding a nearby replica. Here we use our location service. Remember that our location service exploits locality by starting its search in the proximity of the client. Consequently, in our naming approach, when a client wants to bind to a directory, the location service returns the address of the nearest contact point, that is, the address of the replica that is closest to the client. The client then uses this contact point to contact the directory.

The binding process finishes with the creation of a proxy for the directory in the client's address space. This proxy implements the same interface as the directory, allowing the client to remotely invoke operations on the directory.

## 4.2 Directory Replicas

As with any other Globe object, a directory replica is implemented by means of four local subobjects, as shown in Figure 2. The **semantics subobject** is a local subobject that implements the actual functionality of the directory and contains its current state. It contains a table storing the mapping between labels and object identifiers. The semantics subobject contains no code that is related to how its content is distributed across a network.
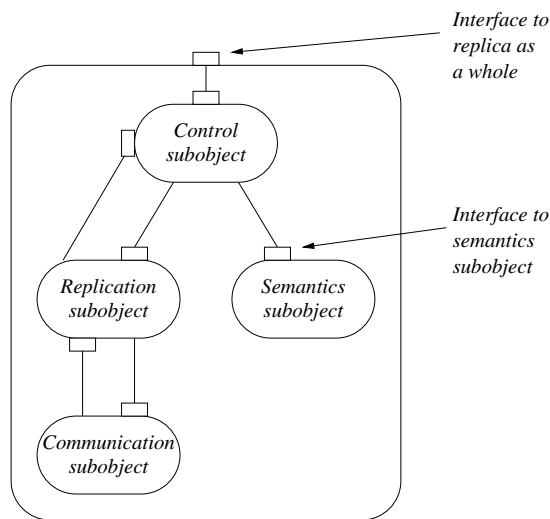


Figure 2: Internal structure of a directory replica

The state of the directory DSO is made up of the state of the semantics subobjects of its replicas. Semantics subobjects may be replicated for reasons of fault tolerance or performance. It is the **replication subobject** that is responsible for keeping these replicas consistent according to some (directory-specific)

coherence strategy. A key observation is that different directory DSOs may have different replication subobjects, implementing different replication algorithms. Having the replication algorithm encapsulated in a subobject (with standardized interfaces) allows to easily change the replication algorithm, when needed.

The **communication subobject** is a system-provided subobject that offers a standard interface to the underlying network, somewhat comparable to sockets and their implementation. However, this subobject can provide more functionality if needed, such as reliable multi-casting, or reliable datagram services.

The **control subobject**, finally, takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. The interfaces of the replication subobject are standard and independent of those of the semantics subobject. Consequently, the control subobject is needed to bridge the gap between the user-defined interfaces of the semantics subobject, and the standard interfaces of the replication subobject.

## 4.3 Directory Operations

Every directory provides two standard directory interfaces: the look-up interface and the update interface. The look-up interface allows clients to query the directory. Its main method is the `resolve` method that resolves a given name to an object identifier. The `resolve` method traverses the naming graph to find the object identifier of the named object. The traversal can be implemented in two ways, using a recursive or an iterative approach, but hybrid approaches are also possible.

In the recursive fashion, a client binds to the starting directory to invoke the `resolve` method to resolve the complete name. The starting directory performs the name resolution by retrieving the object identifier of the first label and binding to the directory it designates. Using this new binding the starting directory (recursively) invokes the `resolve` method with the rest of the name. The recursion stops when a directory is requested to recursively resolve a single label (the last one). The directory can simply return the label's object identifier, since it is the object identifier we are looking for.

Unfortunately, recursive name resolution is also relatively expensive, since it requires the directory to make and keep track of bindings to other directories. The resources used to manage these bindings might be better used to store more directories and handle more (iterative) name resolution requests. For this reason directories are allowed to *refuse* recursive name resolu-

tion and support only iterative name resolution. DNS top-level servers typically refuse to recursively resolve names for the same reason.

In the iterative fashion, a client also starts by binding to the starting directory, but in this case invokes the `resolve` method with the first label of the name only. This invocation results in an object identifier. The client then releases its binding to the starting directory, and binds to the directory designated by the object identifier. This new binding allows the client to resolve the second label of the name. The client binds to a new directory for every label of the name. The object identifier resulting from the method invocation with the last label of the name is object identifier we are looking for.

The update interface consist of two methods: `insert` and `delete`. The `insert` operation allows a client to insert a pair consisting of a label and an object identifier into a directory. The `remove` operation deletes a `(label,identifier)`-pair from a directory given the label. The replica('s) at which the `insert` or `delete` is actually performed during an invocation is determined by the replication subobjects of the directory. These replication subobjects also determine how state changes are subsequently propagated between replicas.

## 4.4 Name Servers

To run directory replicas, we introduce a set of generic name servers. These name servers place the aforementioned four subobjects in their address spaces and allow them to run. Name servers provide directory replicas with the necessary resources and support. They provide, for instance, the necessary means for a replica to communicate with other replicas, and offer an interface to the Globe location service. In addition, a name server can provide fault-tolerant persistent storage (if needed).

The name servers are distributed across the network in such a way that every client always has one or more servers located nearby. As an example, Figure 3 shows seven name servers 1–7, with three directories, a, b, and /. Directory a has replicas at name server 5, 6, and 7. Directory b is replicated at 2 and 4. Directory / consists of a single replica at name server 1.

Layered around the grid of name servers are naming service **resolvers**, implementing the interface to the naming service as a whole. Resolvers are responsible for implementing iterative name resolution, when directories refuse recursive resolution. Placing this responsibility at the resolvers makes implementing name server clients easier. Figure 3 shows three resolvers.
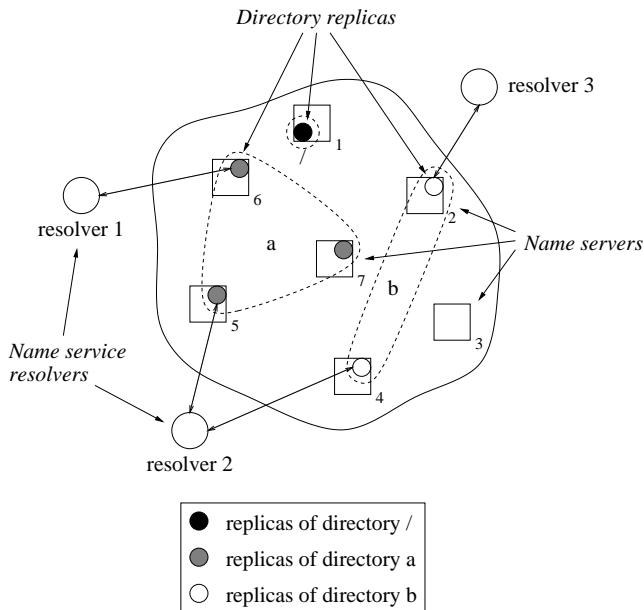
Figure 3: The name server grid.

Resolver 1 and 3 are each using one directory (a and b, respectively), while resolver 2 is using both directories.

# 5  Scalability of our Approach

In this section we focus on how the various aspects of our naming architecture enhance scalability.

## 5.1  Efficient Look-up Operations

The desire to make name resolution exploit locality has played an important part in the design of our name service architecture. Using locality where possible is the primary means to make name resolution scalable. The architecture preserves the locality of look-up operations by using the location service when binding to directories. By using the location service, traversing the name space always involves contacting nearby directory replicas. In particular, if a directory is needed to resolve part of a full name, and that directory has a replica located in the neighborhood, precisely that replica will be used.

Consequently, regardless of whether name resolution is done iteratively or recursively, our approach ensures that each next step of the resolution process is carried out at a name server that is as close as possible to the server where the current step had been carried out. Optimally, the next step is carried out the same server as the current one. In contrast, DNS can ensure locality *only* through caching. Without caching, resolution always proceeds toward the location of the named object.

To make our point clear, first consider the situation where resolver 3 wants to recursively resolve the name /a/b/c in Figure 3. Resolver 3 will start by binding to the only replica of directory / located at name server 1. Resolver 3 will then use the binding to invoke the resolve method, with a/b/c as parameter. To resolve the name, the replica of / at 1 uses the first label a to retrieve the object identifier of directory a. Using the object identifier it will then bind to directory a.

The location service ensures that the replica of / at 1 binds to nearest replica of directory a, namely the one located at name server 6. The replica of / will then invoke the resolve method with b/c as parameter. The replica of directory a will be bound to the nearest replica of directory b to resolve the last label c. The nearest replica of b is located at name server 2. One can see that the name servers used in this example are all located near each other.

Now consider the situation that recursion is *iterative*. In that case, it can be seen that resolver 3 will first bind to the replica of / at server 1. It will then bind to a replica of directory a that is nearest to itself, which is the replica at 7, to finally bind to the replica of directory b at server 2.

## 5.2  New Replicas

To have local name resolution requires that there are directory replicas located at name servers nearby. The naming service architecture allows us to easily create new directory replicas nearby, by providing a loose coupling between directories and the name servers. If a directory becomes popular in an area where it has no replica, a new replica can be created at a name server in that area. This new replica can then provide local access. Name resolver provide the perfect place to recognize popular directories. Each of them can thus take the appropriate actions to create a new local replicas, when needed.

When two directories are logically closely related, the creation of a new replica for one directory, might result in the creation of a new replica for the other directory as well. Consider, for instance, the following situation in Figure 3. Assume that directory a becomes popular around name server 3. Since there is currently no replica of a at 3, the name service creates one at 3. Furthermore, assume that the only way to access directory a is via the / directory, which is cur-

rently located only at name server 1. In that case, it makes sense to create a replica of directory / at 3, as well. It is clear that the name service needs to obtain and maintain information about these kind of relations, if it wants to exploit locality for scalable name resolution.

## 5.3 Efficient Update Operations

While adding replicas increases locality, it comes at the price of an increase in resources. We can limit the resource usage by choosing an efficient replication strategy for the directory at hand. A replication strategy is efficient if it provides the desired combination of consistency and resource usage. The actual combination is determined by the type of directory.

Consider, for example, a personal home directory. Such a directory will often contain only a few entries, and there will be relatively few clients. In this case, it might be easiest to use master-slave replication, where the complete directory state is shipped from the master replica (probably near the owner) to slave replicas. However, when considering a "root" directory, like the `.com` domain in DNS, a different picture emerges. Such a directory will contain many entries, and will be accessed by a vast number of clients worldwide. In this case, a form of active replication might be useful. With active replication, instead of shipping state, only the update operations on the directory are forwarded to all replicas. In our example, immediate propagation of updates may not be necessary. Instead, updates may be batched and periodically multicast to all replicas.

The fact that a replica's implementation is self-contained allows us to use new replication methods when they come available. By allowing efficient update operations, directory-specific replication strategies enhance the scalability of our naming service.

## 5.4 Caching

To avoid traversal through the name servers, the name service can cache both intermediate and final results (directory and user-object identifiers, respectively). The architecture provides two places where results can be cached: in directories and in the resolver.

If the `resolve` method is executed recursively, the name service resolver and directories in the path traversed during name resolution, have the possibility to cache the object identifier of the resolved name. If extra information is returned after name resolution, as in DNS, intermediate results could also be cached. It is thus seen that recursion naturally leads

to the most convenient places to cache results. Unfortunately, recursion is relatively expensive and directories can choose not to provide it, as explained in Section 4.3.

If the `resolve` method is implemented iteratively, the name service resolver becomes the only place to cache intermediate and final results. By combining intermediate results, name resolver can effectively build short-cuts in the naming graph. Unfortunately, these short-cuts will be used only by clients of the same resolver.

## 6 Related Work

There is already a considerable body of experience with large-scale naming systems. The Internet Domain Name System (DNS) is perhaps the best-known example [1, 6]. We described the DNS naming and resolution model in Section 2, and have already argued that DNS cannot be used as a general-purpose naming system for global middleware. Also note that DNS is used to name $56 \times 10^6$ hosts in July 1999.[1] which is significantly less than the $10^{12}$ objects we intend to support. We doubt that DNS can scale to very large numbers of hosts.

The Prospero file system supports a naming model similar to ours [7]. However, the focus of the Prospero system is completely different. The goal of Prospero is to examine how scale affects users, specifically how scale affects the usability of a large system. To enhance usability, Prospero allows users to build their own personal virtual system by customizing their view of the name space. The customization uses, what are called, *filters* and *union links*. Since these methods are based on directories, the Prospero naming model can thus be seen, and possibly implemented, as an extension of our naming model. Prospero uses forwarding pointers to deal with directories that changed location. Forwarding pointers scale poorly, as demonstrated, for example, in the Web.

Lampson has designed a global name service with a focus on scalability, high availability, and continuing evolution [4]. The name service uses a tree-shaped name space like DNS, but distinguishes at the implementation level between local and global directories. The global directories guide name resolution from the root directory down to local directories. The global directories are replicated and maintain consistency through the use of a *sweep* operation that propagates state changes between replicas. The local di-

---

[1]Source: Internet Software Consortium (http://www.isc.org/)

rectories allow further name resolution to retrieve the values we are interested in. The use of locality (outside caching) or mobility of directories is not considered.

The CORBA naming specifications provide a high-level description of the CORBA naming model [8]. This model is compatible with the naming model we use. In the CORBA model, directories are distributed objects and name resolution means traversing a graph of directory objects. A scalable implementation of the specification will depend heavily on the scalability of the underling object request broker (ORB). We are not aware of an implementation having the same scalability capabilities as our approach.

Our approach to naming and locating objects has strong links to the development and deployment of Uniform Resource Names (URNs) [5, 14]. A URN is a location-independent naming scheme, in which names are to be resolved into URLs. However, discussions concerning the way that URNs should be implemented, and in particular how they should be resolved, hardly address the scalability concerns we have presented in this paper. In fact, proposals actually ignore the issue by concentrating only on how appropriate resolvers should be identified [3, 15]. We claim that our approach will provide a unique and scalable solution to implementing URNs.

## 7 Conclusion and Future Work

Global middleware requires the implementation of a scalable naming service. For a naming service to scale, it needs to take the use of locality and directory-specific replication strategies into account. We have designed a naming architecture that makes use of these two design principles, and shown how they enhance its scalability.

Our future work consists of extending the naming model described in this paper to be complete, and using the specification to implement a naming service. We are currently laying the ground work for the name server implementation. A location service prototype is already implemented. Using the specification we can implement the directory semantics subobjects, and start experimenting with different replication subobjects.

## References

[1] P. Albitz and C. Liu. *DNS and BIND*. O'Reilly & Associates, Sebastopol, CA., 3rd edition, 1998.

[2] A. Black and Y. Artsy. "Implementing Location Independent Invocation." *IEEE Trans. Par. Distr. Syst.*, vol. 1, nr. 1, pp. 107-119, Jan. 1990.

[3] R. Daniel and M. Mealling. "Resolution of Uniform Resource Identifiers using the Domain Name System." RFC 2168, June 1997.

[4] B. Lampson. "Designing a Global Name Service." In *Proc. Fourth Symp. on Principles of Distributed Computing*, pp. 1-10, Minaki, Ontario, 1986. ACM.

[5] R. Moats. "URN Syntax." RFC 2141, May 1997.

[6] P. Mockapetris. "Domain Names - Concepts and Facilities." RFC 1034, Nov. 1987.

[7] B. C. Neuman. "The Prospero File System: A Global File System Based on the Virtual System." *Computing Systems*, vol. 5, nr. 4, pp. 407-432, 1992.

[8] Object Management Group. "CORBAservices: Common Object Services Specification." OMG Document 98-12-09, OMG, Dec. 1998.

[9] ObjectSpace Inc. *Voyager 2.0 User Guide*, 1998.

[10] OMG. "The Common Object Request Broker: Architecture and Specification, revision 2.2." OMG Document 98-07-01, Object Management Group, Feb. 1998.

[11] S. Radicati. *X.500 Directory Services: Technology and Deployment.* International Thomson Computer Press, London, 1994.

[12] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilly & Associates, Sebastopol, CA., 1992.

[13] M. Shapiro, P. Dickman, and D. Plainfossé. "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection." Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.

[14] K. Sollins and L. Masinter. "Functional Requirements for Uniform Resource Names." RFC 1737, Dec. 1994.

[15] K. Sollins. "Architectural Principles of Uniform Resource Name Resolution." RFC 2276, Jan. 1998.

[16] M. van Steen, F. Hauck, P. Homburg, and A. S. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, vol. 36, nr. 1, pp. 104-109, Jan. 1998.

[17] M. van Steen, P. Homburg, and A. S. Tanenbaum. "Globe: A Wide-Area Distributed System." *IEEE Concurrency*, vol. 7, nr. 1, pp. 70-78, Jan. 1999.

[18] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. "Algorithmic Design of the Globe Wide-Area Location Service." *The Computer Journal*, vol. 41, nr. 5, pp. 297-310, 1998.