

# A Scalable Implementation for Human-Friendly URIs\*

Gerco Ballintijn  
Maarten van Steen  
Andrew S. Tanenbaum

June 2000

## Abstract

In the current Web, Uniform Resource Locators (URLs) are used to name resources. URLs have, however, significant scalability problems. For example, they cannot be conveniently used to refer to a replicated Web page. Likewise, if the location of a page is changed, its URL will have to be changed as well. We propose the use of Human-Friendly Names (HFNs) to solve such scalability problems. HFNs are high-level names that allow (human) users to easily deal with names in a location-independent way. We have designed and implemented a scalable HFN-to-URL resolution mechanism. Our solution makes use of the Domain Name System (DNS) and the Globe Location Service.

**Keywords:** naming, resource location, scalability, wide-area systems, DNS

## Introduction

Resources in the World Wide Web are named using Uniform Resource Identifiers (URIs). The most common and well-known form of URI is the Uniform Resource Locator (URL). A URL is used in the Web for two distinct purposes: to identify resources and to access resources. Unfortunately, combining these two leads to scalability problems, since resource identification has different requirements than resource access. Consider, for example, a popular Web page that we want to replicate to improve its availability. Currently, replicated Web pages are named by means of multiple URLs, one for each replica, as shown in Figure 1(a). However, to hide replication from users, that is, to make replication transparent, we need a name that only identifies the page. In other words, that name should not refer to a specific replica, but instead, should refer to the set of replicas as a whole.

---

\*This paper is an updated version of technical report IR-466.

Uniform Resource Names (URNs) provide a solution to these scalability problems. A URN is also a URI, but differs from a URL in that it only *identifies* a Web resource. A URN does not indicate the location of a resource, nor does it contain other information that might change in the future. A good example of a URN is an ISBN number for a book. An ISBN only identifies a book, but not any of its copies.

To access the resource identified by a URN, the URN needs to be resolved into access information, such as a URL. Using URNs to identify resources and URLs to access resources allows one URN to (indirectly) refer to many copies at different locations, as shown in Figure 1(b). This separation allows transparent replication of Web resources. Moreover, since a URN is a stable reference to the name of a resource and not its location, we can also move the resource around without changing its URN. A URN can thus support mobile resources by (indirectly) referring to a set of URLs that changes over time.

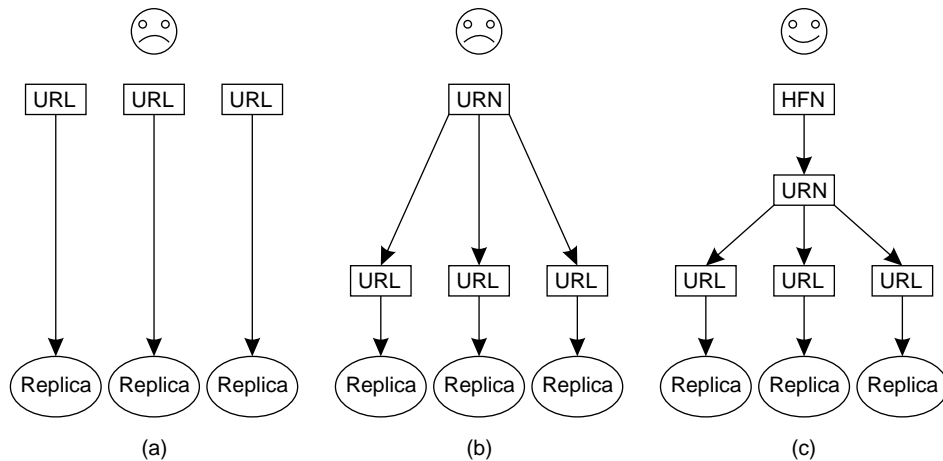


Figure 1: Naming a replicated resource. (a) Using multiple URLs. (b) Using a single URN. (c) Using an HFN combined with a URN.

Since URNs are intended to be primarily used by machines, there is no requirement to make them easy to use by humans (other than being transcribable). However, humans do need a way to name Web resources in such a way that those resources can easily be shared and looked up. To this end, there are basically two approaches.

The first approach is to use a directory service, such as those based on LDAP [5]. Such a “yellow pages” service allows a user to search for a resource based on attribute values that have been assigned to that resource. The main drawback of directory services is their limited scalability. In practice, only implementations based on local-area networks offer acceptable performance. Large-scale, worldwide directory services are yet to be developed. At best, the current implementations are

constructed as federations of local directory services, in which searches are not allowed to span multiple sites unless severely restricted.

The second approach is to make use of a (possibly hierarchical) naming graph comparable to a “white pages” service such as a telephone book. DNS is an example of a such a traditional naming service. Although naming services offer less advanced facilities than directory services, they have proven to easily scale to worldwide networks with millions of users. From this perspective, they are currently more attractive to use than the poorly scalable directory services.

To fill the gap between the current use of URNs and what humans need, a new kind of URI needs to be introduced, as also suggested by Sollins [7]. A **Human-Friendly Name (HFN)** is a high-level name tailored to be used by humans. Unlike URNs, HFNs explicitly allow the use of descriptive names. An HFN needs to be resolved to one or more URLs when the user needs to access the named resource. One way to do this, is to bind a HFN to a URN, and to bind a URN to possibly multiple URLs as described above. In effect, this approach turns HFN resolution into a two-step process. We first resolve the HFN to its associated URN, and then resolve the URN to its associated URLs, as shown in Figure 1(c).

There are many advantages to this approach. First, users can name resources anyway they feel is convenient. If a resource is replicated, or moved to another location, this will not affect the name it was given by its users. Likewise, if a user decides to change the HFN, this will not affect the placement of replicas. Moreover, a user may even decide to use several names to refer to the same resource, very much similar to the use of aliases for e-mail addresses and symbolic links in file systems.

Unfortunately, the problem of resolving HFNs to URLs in a scalable manner has hardly been addressed, and no practical solutions have yet been proposed. To address this shortcoming, we have developed an HFN-to-URL resolution mechanism, paying specific attention to two scalability issues. First, we can support a large number of resources. Second, we can support resources distributed over a large geographical area. Our HFN-to-URL resolution mechanism supports a hierarchical name space, similar to that of the UNIX file system. Our solution is based on the Domain Name System (DNS) and the Globe Location Service [9]. To the best of our knowledge, our design provides the first real solution to large-scale HFN-to-URL resolution.

## Model

For our current naming system, we restrict ourselves to highly popular and replicated Web resources, such as popular Web pages and software distributions. Support for other resource types, such as personal Web pages or highly mobile resources, is yet to be incorporated. We also assume that changes to a particular part of the name space always originate from the same geographical area. The reason for choosing this specific resource model is that it allows us to make efficient use

of the underlying DNS infrastructure. We return to the use of DNS below.

We stay close to the familiar hierarchical name space as found in the UNIX file system. An example of an HFN supported by our system is `hfn://nl/vu/cs/globe`. The name space has one root directory named `//`, and every HFN is a sequence of simple names—indicating a path in the name space—from the root down to some leaf directory. Our name space uses `/` as separator and allows only letters, digits, and hyphen (`-`) in a simple name. Names are case insensitive. It is important to realize that these restrictions result from limitations imposed by DNS, but are otherwise not fundamental to our approach.

Our security policy is minimal, just enough to prevent unauthorized changes to the HFN-to-URL mapping, but do not intend to make the HFN-to-URL mapping confidential, since we assume that HFNs will be used in much the same way as URLs are used today. The name service is also not used to provide access control.

Since the use of locality is of prime importance for scalability, we want the name service and its various components to use locality when possible. When resolving a name, locality should be used in two distinct ways. First, the name resolution process should provide us with access to the nearest replica. This type of locality is needed for a scalable Web system.

A second, and harder, form of locality is that the name resolution process itself should also use nearby resources when possible. For example, assume we have a client located in San Francisco who wants the DNS name `vu.nl` to be resolved. In the current DNS, name resolution normally proceeds through a root server, the name server for the `nl` domain (which is located in The Netherlands), and the name server for the `Vrije Universiteit` (which is located in Amsterdam). If the resource named by `vu.nl` happened to be replicated and already available in San Francisco, the lookup request will have traveled across the world, to subsequently return an address that is close to the requesting client. In this case, it would have been better if the name resolution process itself could have taken place using only name servers in the proximity of the client. This second type of locality is not addressed by any existing naming system.

## Architecture

In its general form, the HFN-to-URL mapping is an N-to-M relation. In other words, multiple HFNs may refer to the same set of URLs. This mapping may possibly change regularly. For example, a resource may be given an extra name, a replica is added, or moved to another location. To efficiently store, retrieve, and update the HFN-to-URL mapping, we split it into two separate mappings as discussed before. We introduce the **object handle** to provide a stable, unique name for every object. The first mapping is the HFN-to-object handle mapping. The second mapping is the object handle-to-URL mapping. By splitting the HFN-to-URL mapping in two separate mappings, we have an N-to-1 relation and a 1-to-M relation, which are each far easier to maintain compared to a single N-to-M

relation. The object handle is a URN, as described previously. It identifies a Web resource and is globally unique.

The main purpose of the HFN-to-object handle mapping is to uniquely *identify* a resource by providing its object handle. The HFN-to-object handle mapping is maintained by a **name service**. The object handle-to-URL mapping is maintained by a **location service** whose sole purpose is to *locate* a resource. HFN resolution thus consists of two steps. In the first step, the HFN is resolved to an object handle by the name service, and in the second step, the object handle is resolved to a URL by a location service.

To use our naming system, we add three new elements to the normal setup of Web browsers and HTTP servers: an HFN-to-URL proxy, a name service, and a location service. It is the task of the HFN-to-URL proxy to recognize HFNs and resolve them by querying the name and location service. As such, it operates as a front end to these two services. With the URL obtained from the location service, the proxy accesses the named resource. In our present implementation, the proxy is implemented as a separate process that interacts with any standard Web browser. We have also integrated the proxy into Mozilla, the open-source version of the Netscape™ browser.

Figure 2 shows the setup we propose to retrieve Web resources named by HFNs. When a user enters an HFN in the Web browser, the browser contacts the HFN-to-URL proxy to obtain the Web resource named by the HFN (step 1). The proxy recognizes the HFN and contacts the name service in step 2. The name service resolves the name to an object handle, and returns it to the proxy (step 3). The proxy then contacts the location service in step 4. The location service resolves the object handle to a URL, and returns it to the proxy (step 5). The proxy can now contact the HTTP server storing the named resource in step 6, which returns an HTML page in step 7. The proxy then returns this HTML page to the Web browser (step 8).

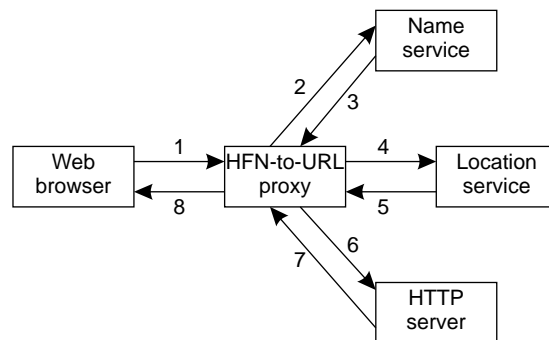


Figure 2: The setup to retrieve Web resources named by HFNs.

## **Name Service**

We use the Domain Name System (DNS) to store the mapping from an HFN to an object handle. DNS is at the moment primarily used to name Internet hosts and email destinations. We can, however, reuse the existing DNS infrastructure for HFNs with only minimal changes, as we explain next.

### **The Domain Name System**

DNS provides an extensible hierarchical name space, in which 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 Corporation. A naming authority is responsible for providing the resources needed to store and query a DNS name, and can decide for itself which names to store in its subdomain. The Intel Corporation can thus create whatever host name or email destination it wants in its subdomain.

Resolving a host name in DNS consists, conceptually, of contacting a sequence of name servers. The domains stored by the sequence of name servers are increasingly specific, allowing the resolution of an increasing part of the host name. For example, to resolve the host name www.intel.com, the resolution process visits, in turn, the name servers responsible for the root, com, and intel.com domains, respectively. The last name server will be able to resolve the complete host name.

To enhance its performance, DNS makes extensive use of caching. When a name server is asked to resolve a DNS name recursively, it will contact the sequence of name servers itself to resolve the name. The name server can then cache the intermediate and end results of the resolution process. This procedure avoids having to contact the sequence of name servers a second time when the same or a similar name is looked up. However, for effective caching, DNS needs to assume that the name-to-address mapping does not change frequently.

DNS uses **resource records** to store name mappings at name servers. A DNS name can have zero or more resource records. There are two kinds of resource records. The first kind stores user data, like the resource records for naming Internet hosts and email destinations. This kind of record associates an IP address or a mail server with a DNS name. The second kind is the name server resource record, which is used internally by DNS to implement the name space delegation. This resource record associates another DNS server with a DNS name, indicating another name server at which to continue name resolution.

### **Using DNS to Store HFNs**

To store HFNs in DNS, we first need to define a translation between our name space and the DNS name space. A name in our name space is a sequence of simple names, where the simple names at the start of the name refer to directories closer to the root. In DNS, simple names at the end of a name refer to directories (domains)

closer to the root. We thus need to reverse the order of simple names in an HFN to store it in DNS. A second difference is that we prefer to use the slash (i.e., “/”) as a separator, while DNS uses a dot (i.e., “.”). We therefore also need to change slashes into dots. As an example, we will thus transform the HFN `hfn://nl/vu/cs/globe` into the DNS name `globe.cs.vu.nl`, and use the latter to store the mapping associated with `hfn://nl/vu/cs/globe`.

To resolve an HFN, the HFN-to-URL proxy starts by removing the `hfn` tag of the HFN. The resulting name then needs to be translated to a proper DNS name. During the translation, the order of the simple names is reversed and slashes are changed into dots. The resulting DNS name can be used to query the DNS name service. The query will specify the DNS name and the request for a resource record holding an object handle.

To store the association of an object handle with a DNS name, we introduce a new resource record. This resource record will contain an object handle encoded as an ASCII character string. When a user introduces a new HFN, we create a resource record and store the object handle associated with that HFN. This record will subsequently need to be inserted into the DNS name space. The proper name server for this purpose is the one responsible for the parent domain of the HFN. For instance, to insert the HFN `hfn://nl/vu/cs/globe`, we need to contact the server responsible for the `cs.vu.nl` domain. The actual insertion at that server can be done dynamically using the DNS *update* operations as described in RFC 2136.

## Location Service

To resolve an object handle into a URL, we use the Globe location service [9]. The Globe location service allows us to associate a set of URLs with a single object handle. The service offers, in addition to a lookup operation for object handles, two update operations: insert and delete. Insert and delete operations are used to modify the set of URLs associated with an object handle. Since the location service is distributed across a wide-area network, it should deal gracefully with server failures, network partitions, and (long) communication delays.

## Architecture

To efficiently update and look up URLs, we organize the underlying wide-area network (i.e., the Internet) as a hierarchy of **domains**. These domains are similar to the ones used in DNS. However, their use is completely independent of DNS domains, and they have been tailored to the location service only. In particular, the domains in the location service represent geographical or network-topological regions. For example, a lowest-level domain may represent a campuswide network of a university, whereas the next higher-level domain represents the city where that campus is located. Each domain is represented in the location service by a **directory node**. Together the directory nodes form a worldwide search tree.

A directory node has a **contact record** for every (registered) resource in its

domain. The contact record is divided into a number of **contact fields**, one for each child node. A directory node stores either a **forwarding pointer** or the actual URLs in the contact field. A forwarding pointer indicates that URLs can be found at the child node. Contact records at leaf nodes are different: they contain only one contact field storing the URLs from the leaf domain.

Every URL stored in the location service has a path of forwarding pointers from the root down, pointing to it. We can thus always locate a URL starting at the root node and following this path. In the normal case, URLs are stored in leaf nodes. However, storing URLs at intermediate nodes may, in the case of highly mobile resources, lead to considerably more efficient lookups. However, to simplify our present discussion, assume that URLs are always stored in leaf nodes.

Figure 3 shows as an example the contact records for one object handle. In this example, the root node has one forwarding pointer for the object handle, indicating that URLs can be found in its left subtree, rooted at the USA node. The USA node, in turn, has two forwarding pointers, pointing to the California and Texas nodes, respectively. Both of these nodes have a forwarding pointer to a leaf node where a URL is actually stored.

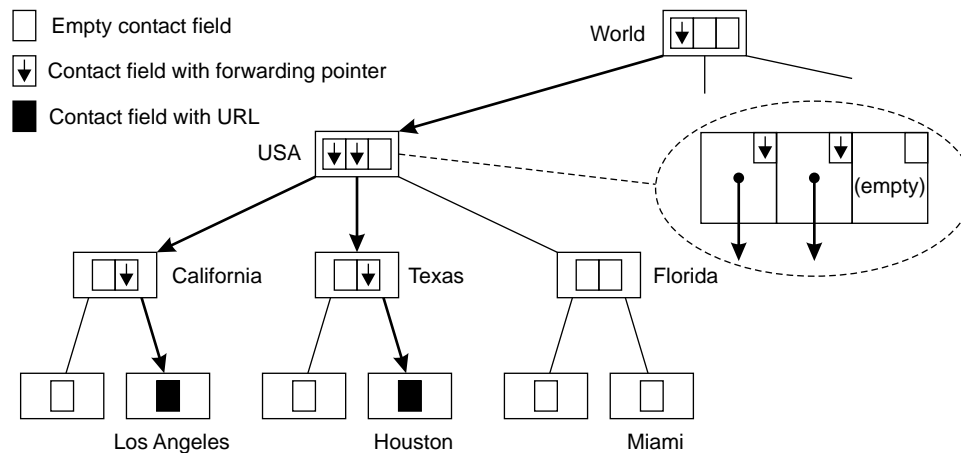


Figure 3: The organization of contact records in the tree for a specific resource.

The search tree described so far obviously does not yet scale. In particular, higher-level directory nodes have to store a large number of contact records and handle a large numbers of requests, since there is a path of forwarding pointers from the root node down for every object handle. Our solution is to partition a directory node into one or more directory subnodes, such that each subnode is responsible for only a subset of the records originally stored at the directory node. We use a hashing technique on the object handles to identify subnodes at parents and children.

## Operations

When a client wants to know the URL of a resource, it initiates a lookup operation at the leaf node of the domain in which it resides. The client provides the resource's object handle as parameter. The lookup operation starts by checking whether the leaf node has a contact record for the object handle. If the leaf node has a contact record, the operation returns the URL found in the contact record. Otherwise, the operation recursively checks nodes on the path from the leaf node up to the root. If the lookup operation finds a contact record at any of these nodes, the path of forwarding pointers starting at this node is followed downwards to a leaf node where a URL is found. If no contact record is found at any of the nodes on the path from the leaf node to the root, the object handle is unknown to the location service.

As an example, consider a client located near the leaf node of Miami, as shown in Figure 3. When the leaf node is contacted by the client with a request for a URL, it will forward the request to its parent, the Florida node, since it does not contain a contact record. The Florida node also does not know about the object handle, and will, in turn, forward the request to its parent, the USA node. The USA node does know about the object handle, and forwards the request to one of its children indicated by a forwarding pointer. The lookup operation then follows the path of forwarding pointers to one of the leaf nodes, for instance, the Houston leaf node. By going higher in the search tree, the lookup operation effectively broadens the area that is searched for a URL, and resembles search algorithms based on expanding rings.

The goal of the insert operation is to store a URL at a leaf node and create a path of forwarding pointers to the leaf node. When a resource has a new URL in a leaf domain, the object inserts this new URL at the node of the leaf domain. The insert operation starts by inserting the URL in the contact record of the leaf node. The insert operation then recursively requests the parent node, the grandparent node, etc., to install a forwarding pointer. The recursion stops when a node is found that already contains a forwarding pointer, or otherwise at the root. The delete operation removes the URL and path of forwarding pointers analogous to the insert operation. Further technical details can be found in [10].

## Discussion

An important aspect of our HFN-to-URL resolution scheme is its scalability. As explained in the introduction, we can distinguish two types of scalability: the support of a large number of resources and the support for resources that are distributed over a large geographical area. For our resolution scheme to be scalable, both kinds of scalability need to be addressed in the name service and in the location service.

## **Name Service**

The first form of scalability requires our name service to deal with a large number of resources, that is, deal with a large number of HFN-to-object handle mappings. The current DNS infrastructure supports in the order of  $10^8$  host names. We therefore assume only that the number of popular resources does not significantly increase the number of names stored in DNS.

The second form of scalability requires our name service to deal with names distributed over a large geographical area. We tackle this second problem by ensuring the use of locality in lookup and update operations. The locality of lookup operations in DNS is provided by caches. If a resource named by an HFN is popular, its HFN-to-object handle mapping will be stored in the caches of name servers, providing clients located near the cache with local access to the HFN-to-object handle mapping. A DNS query to obtain the object handle can thus be answered directly, without the need to contact a name server located far away. By assuming the use of popular Web resources and a stable HFN-to-object handle mapping, we ensure that caching remains effective. Update operations in the name service exploit locality as well. Since we assume that changes to a specific part of the name space always originate from the same geographical area, we can always place the name server that stores the resource records for that part near the area where the changes originate.

If we can assume that changes to a particular part of the name space always originate from the same domain, DNS is an attractive name service, given its existing infrastructure. Unfortunately, if we want to drop those assumptions in favor of a less restrictive resource model, scalability problems could arise in DNS that prevent our HFN resolution mechanism from scaling further. If we want to support resources that are unpopular, DNS might become overloaded. If we want to support mobile resources, the caching mechanism might cache mappings in the wrong place. If we want to support a more general resource model, we need to replace DNS with a more scalable name service. We describe the design of such a naming service in [1].

A different problem with using DNS as the name service in our scheme, is that it places restrictions on the syntax of the HFN name space. Since we need to translate our HFN names to DNS names, we can support only the small subset of the ASCII character set that is allowed by DNS. Given the increasing number of Web users speaking languages not supported by the ASCII character set, this limitation will become a problem in the future.

## **Location Service**

The Globe location service deals with a large number of object handle-to-URL mappings in two ways. By virtue of the distributed search tree, contact records are distributed over different servers. The higher-level directory nodes form potential bottlenecks, as they may need to support many objects and their associated opera-

tions. However, servers for these nodes can be off-loaded by applying straightforward partitioning techniques. For example, the root node can be partitioned into two or more subnodes, where each subnode is responsible for its own subset of object handles. These subnodes can thus run independently on separate machines, which are possibly located far away from each other.

The location service deals with URLs distributed over a large geographical area by using locality through its distributed search tree and related lookup algorithm. By starting the lookup operation at the leaf node to search the nearby area first, and continuing at higher nodes in the tree to search larger areas, the location service avoids using remote resources when a URL can be found using local resources only.

## **Implementation**

We have implemented our HFN resolution scheme, using the software created by the BIND project to implement the name service part. We have implemented a prototype of the location service and the HFN-to-URL proxy ourselves. The software is currently used in an experimental setup to validate our approach. For example, we have built an instant messaging application that is currently running on our local-area network, and which we intend to soon deploy worldwide. To store object handles, we chose to use the existing TXT resource record, instead of implementing a new resource record. Using the TXT resource record allows us to use the BIND software in an unmodified form.

For the moment, we use the standard DNS authorization scheme to prevent unauthorized update operations on the name space. In this scheme, a DNS name server performs update operations only when they are initiated from hosts trusted by the server. In the future, we could use the new security features that have been added to DNS. Security in the Globe location service is part of ongoing research.

## **Related Work**

The URI research community has so far shown little interest area of human friendly names. Most current work concentrates either on URLs or URNs. The URL work deals mainly with assignment and specification of new access schemes.

The IETF URN working group has created several RFC documents describing the concepts behind URNs. Their focus thus far has been on the definition of a URN name space (RFC 1737), and a general architecture (RFC 2276). In this architecture, the URN name space actually consists of several independent URN name spaces, and every URN name space has (potentially) its own specific URN resolver. Therefore to resolve a URN, the first thing to do is to select the appropriate name resolver. This selection of name resolver is done by a Resolver Discovery Service (RDS).

There are currently two proposals for an RDS. Daniel and Mealling propose to build an RDS using DNS [2]. In their proposal, DNS contains resource records specifying rewrite rules. When a URN needs to be resolved, these rewrite rules are applied to the URN, resulting in a resolver that can resolve the complete URN, or possibly even the resource itself. Slottow [6] proposes to build an RDS as a distributed database, using a distributed B-tree. Within the nodes of the B-tree, replication is used to maintain high availability. Our work has not included a resolver discovery system since we are looking in our research at one specific URN space, that is, the object handle space.

Kangasharju et al. [3] describe a location service (called LDS) that is based solely on DNS. Their goal is to find a set of HTTP servers, both mirror sites and caches, that store the resource normally found at a certain URL. To this end, they transform a URL into a proper DNS name, similar to the way we transform an HFN. However, their system maps URLs to IP addresses, whereas in our approach, HFNs are mapped to URLs.

The main difference between LDS and our work is the place where the location information is stored. In LDS, IP addresses of the servers are directly stored in DNS, while we store an object handle in DNS and use a separate service to provide a set of URLs for the named resource. Since LDS stores IP addresses in DNS, the DNS server needs to be updated every time a replica server is added or removed, making their system more dynamic and caching thus less useful than in our system. Since our system has a separate location service structured to reflect network distances, we can easily and efficiently provide the URL that is nearest to the client. This is not the case in LDS. Instead, a client is handed a list of IP addresses from which one must be chosen. No heuristics are provided on what the best choice is.

The Handle system [8] is a URN resolution scheme, and thus similar in function to the location service part of our HFN-to-URL resolution system. It maps a handle consisting of a prefix and suffix to access information, for instance a URL. The prefix of the handle specifies a naming authority, and the suffix specifies a name under that naming authority. Resolving a handle consists of contacting a *Global Handle Registry* to find a *Local Handle Registry*, where the handle can be fully resolved. The Handle system supports scalability by allowing both the global and local handle registries to be replicated. It does, however, not try to ensure that the access information it provides refers to resources local to the client, nor does the handle resolution process use local resources when possible. The Handle system does support Unicode character set, and thus allows names in languages other than English.

Lampson [4] has designed a global name service with a focus on scalability, high availability, and continuing evolution. 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 directories allow further name resolution to retrieve the

values we are interested in. Like DNS, caching provides the only form of locality.

The location of servers storing replicated Web resources is an integral part of the commercial content delivery system of Akamai and Sandpiper. In both systems the original URL of the replicated resource needs to be changed to point to servers of the delivery system. Akamai uses a modified Web server to redirect clients to servers, while Sandpiper uses a DNS-based solution. Both system are said to take both the client location as the current network condition in to account when providing the client with a Web server. While both systems provide local access to the Web resources they support, their naming system is not local.

## Conclusions and Future Work

As part of our research into wide-area distributed systems, we have developed a location service that, together with DNS, can be used to resolve HFNs to URLs in a scalable fashion. Scalability is achieved by using two distinct mappings, one for naming resources and one of locating them. Using this separation, we can apply techniques specific to the respective services to obtain scalability. An important part of our design is the reuse of the existing DNS infrastructure. This provides us with benefits in the form of an existing infrastructure and experience using it. Future work will consist of using our naming scheme in larger experimental setups to gain more experience. We intend to eventually replace DNS in our system with an implementation of our own name service.

## References

- [1] G. Ballintijn and M. van Steen. Scalable Naming in Global Middleware. In *Proc. Sixth Int'l Conf. on Parallel and Distributed Computing Systems*, Las Vegas, Aug. 2000.
- [2] R. Daniel and M. Mealling. Resolution of Uniform Resource Identifiers using the Domain Name System. RFC 2168, June 1997.
- [3] J. Kangasharju, K. W. Ross, and J. W. Roberts. Locating Copies of Objects Using the Domain Name System. In *Proc. Fourth Web Caching Workshop*, San Diego, CA, Apr. 1999.
- [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] P. Loshin, editor. *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. Morgan Kaufman, San Mateo, CA., 2000.
- [6] E. C. Slottow. Engineering a Global Resolution Service. Master's thesis, MIT, Cambridge, MA, June 1997.
- [7] K. Sollins. Architectural Principles of Uniform Resource Name Resolution. RFC 2276, Jan. 1998.
- [8] S. X. Sun and L. Lannom. Handle System Overview. <http://www.handle.net/overview-current.html>, Feb. 2000. Work in progress.

- [9] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Commun. Mag.*, 36(1):104–109, Jan. 1998.
- [10] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic Design of the Globe Wide-Area Location Service. *The Computer Journal*, 41(5):297–310, 1998.