Scalable Human-Friendly Resource Names *

Gerco Ballintijn
Maarten van Steen
Andrew S. Tanenbaum

June 2001

Abstract

In the current Web, Uniform Resource Locators (URLs) are used to name and access resources. However, URLs pose a significant scalability problem in the Web since they cannot be used to refer to replicated Web pages. We propose a new URI scheme called Human-Friendly Names (HFNs) to solve this scalability problem. HFNs are high-level names that are easy-to-use by humans and name Web resources in a location-independent way. We describe the design of a scalable HFN-to-URL resolution mechanism that is based on URNs and 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 type 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 a scalability problem 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.

Uniform Resource Names (URNs) provide a solution to this scalability problem. A URN is also a type of 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

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

contain other information that might change in the future. A good example of a URN is an ISBN number for a book. An ISBN number 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 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.

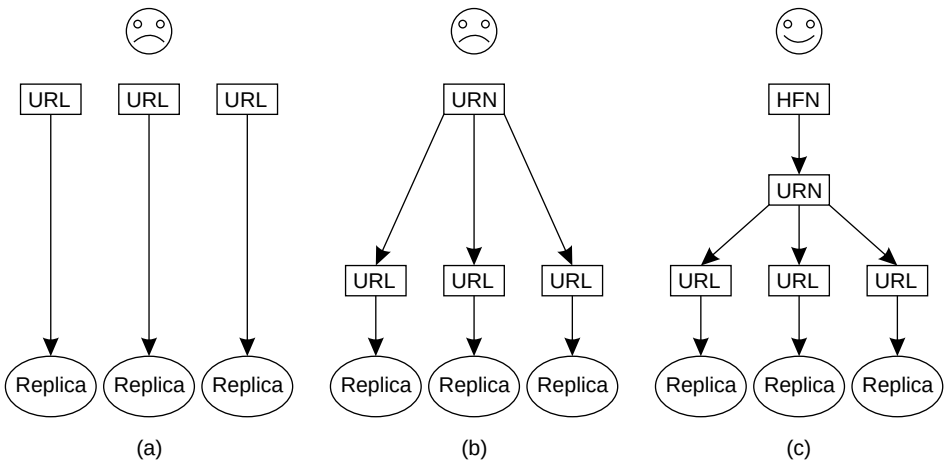

Figure 1: Naming a replicated resource. (a) Using multiple URLs. (b) Using a single URN. (c) Using an HFN combined with a URN.

Since URNs are intended to be primarily used by machines to identify resources, there is no requirement to make them easy-to-use or remember by humans. The only requirement on URNs regarding humans, as stated in RFC 1737, is that URNs are human transcribable. For instance, ISBN numbers can easily be written down and copied by humans, but are not easily remembered. However, humans do need a way to name Web resources in such a way that those names can easily be shared and remembered.

To fill the gap between what URNs provide and what humans need, a new kind of URI is needed, as suggested in RFC 2276. We propose the introduction of a new URI scheme, called **Human-Friendly Names** (**HFNs**), to meet this need. HFNs are tailored to be convenient to use by humans and therefore explicitly allow the use of descriptive names, unlike URNs. There are different approaches to human friendly naming. Two well-known approaches are the “yellow pages” and “white pages” services.

The "yellow pages" approach is to use a directory service, such as those based on LDAP [6]. Such a 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 "white pages" approach is to make use of a (possibly hierarchical) naming graph, such as used in file systems. The Domain Name System (DNS) is the prime example of traditional naming services. 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, we choose to base our HFNs on a hierarchical name space implemented using the DNS. The hierarchical name space provides users a convenient and well-known way to name resources. We return to our choice for DNS later.

Like a URN, an HFN needs to be resolved to one or more URLs when the user needs to access the named resource. We propose a two-step process to HFN resolution. In our approach, we bind an (hierarchical) HFN to a URN, and bind a URN to possibly multiple URLs, as described above. HFN resolution then consists of first resolving the HFN to its associated URN, and then resolving the URN to its associated URLs, as shown in Figure 1(c).

There are many advantages to this two-step approach. If a resource is replicated, or moved to another location, this will not affect the name it was given by its users. Likewise, a user is free to change the HFN since this will not affect the placement of replicas. Moreover, a user may even decide to use several names to refer to the same resource, similar to the use of symbolic links in file systems.

Our HFN-to-URL resolution mechanism pays 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. To the best of our knowledge, our design provides the first solution to large-scale HFN-to-URL resolution.

Model

For our naming system, we restrict ourselves to naming only highly popular and replicated Web resources. 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 restricted resource model is that it allows us to make efficient use of the existing DNS infrastructure. Given these restrictions, our HFN scheme is currently not appropriate as a replacement for URLs in general.

Since our HFNs are implemented using DNS, their syntax closely follows the structure of domain names. An example of a HFN that refers to the source code of the current stable Linux kernel is hfn:stable.src.linux.org. The hfn: prefix identifies our URI scheme; the rest is the actual name of the resource. Our security policy is minimal, we just want to prevent unauthorized changes to the HFN-to-URL mapping. We do not make the HFN-to-URL mapping confidential since we assume that HFNs will be shared in the open, in much the same way as URLs are shared today.

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

The second form of locality requires that the name resolution process itself should also use nearby resources when possible. For example, assume we have a user 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 happens 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 user. In this case, it would have been better if the name resolution process itself would have used only name servers in the proximity of the user.

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 change regularly. For example, a resource is 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. The first mapping is the HFN-to-URN mapping. The second mapping is the URN-to-URL mapping. We use URNs to provide us with a stable, globally unique name for every resource. 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 main purpose of the HFN-to-URN mapping is to uniquely *identify* a resource by providing its URN. The HFN-to-URN mapping is maintained by a **name service**. The URN-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 a URN by the name service, and in the second step, the URN is resolved to a URL by a location service. The type of URN used

in our naming scheme is determined by the 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 design, we chose the proxy to be a separate process that can interact with any standard Web browser. However, a plug-in module can introduce the same functionality directly into a Web 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 a URN, and returns it to the proxy (step 3). The proxy then contacts the location service in step 4. The location service resolves the URN 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).

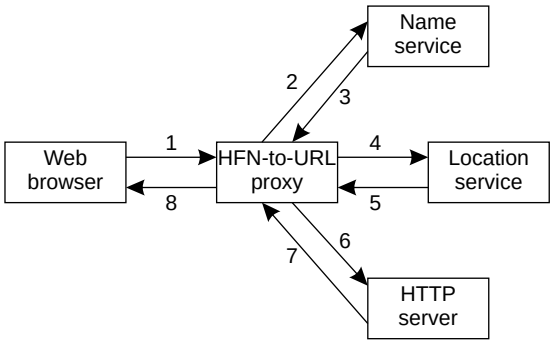

Figure 2: The setup to retrieve Web resources named by HFNs.

Name Service

We use the DNS to store the mapping from an HFN to URN. 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

We introduce a new type resource record to store the association of a URN with a DNS name. When a user introduces a new HFN, we create a resource record to store the URN associated with that HFN. This record will subsequently be inserted into the DNS name space. The proper name server to store the record is the one responsible for the parent domain of the HFN. For instance, to insert the HFN hfn:devel.src.linux.org, we need to contact the server responsible for the src.linux.org domain. The actual insertion at that server can be done dynamically using the DNS *update* operation, as described in RFC 2136.

Location Service

We use the Globe location service [9] to resolve URNs into URLs. It allows us to associate a set of URLs with a single URN. Since the location service uses so-called *object handles* to identify resources, we use these object handles as URNs in

our two-level naming scheme. However, to ease our discussion, we will continue to use the term URN. The location service offers, in addition to a lookup operation for URNs, two update operations: insert and delete. The insert and delete operation are used to modify the set of URLs associated with a URN.

Architecture

To efficiently update and look up URLs, we organize the underlying wide-area network (i.e., the Internet) into 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, administrative, or network-topological regions. For example, a lowest-level domain may represent a campus-wide network of a university, whereas the next higher-level domain represents the city where that campus is located. Another important difference is that the domain hierarchy is a completely internal structure, unlike DNS it is not visible to users.

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 slightly 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, but storing URLs at intermediate nodes may, in the case of highly mobile resources, lead to considerably more efficient lookup operations, as discussed below. However, since our current model excludes (highly) mobile resources, we can safely assume that all URLs are always stored in leaf nodes.

Figure 3 shows as an example the contact records for one URN. In this example, the root node has one forwarding pointer for the URN, 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.

Operations

When a user 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 user provides the resource's URN as a parameter. The lookup operation starts by checking whether the leaf node has a contact record for the URN. If it has a contact record, the operation returns

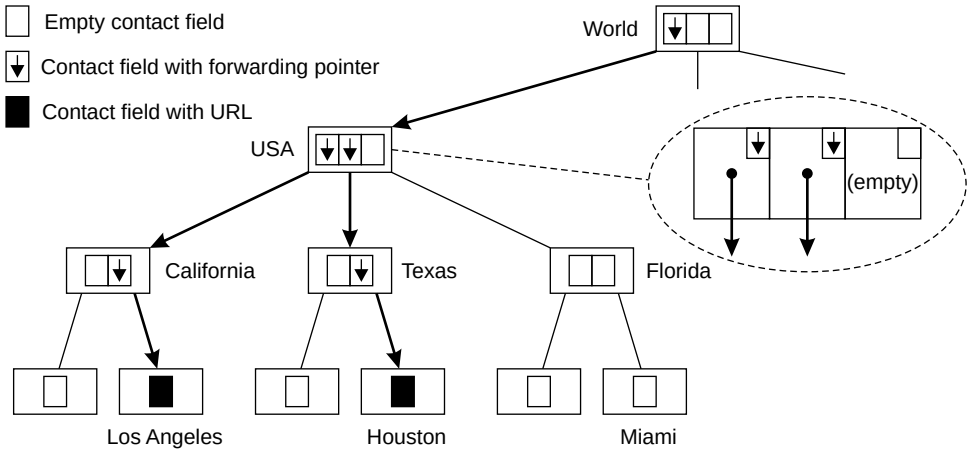

Figure 3: The organization of contact records in the tree for a specific resource.

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 URN is unknown to the location service.

As an example, consider a user located near the leaf node of Miami, as shown in Figure 3. When the leaf node is contacted by the user 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 URN, and will, in turn, forward the request to its parent, the USA node. The USA node does know about the URN, 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 thus resembling 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 replica in a leaf domain, the URL of the new replica is inserted 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].

Improvements

The basic search tree described so far obviously does not scale yet. In particular, higher-level directory nodes, such as the root, pose a serious problem. They have to store a large number of contact records and handle a large numbers of requests. Our solution is to partition an overloaded directory node into multiple directory *sub*nodes. Each subnode is responsible for only a subset of the contact records originally stored at the directory node, and therefore has a much smaller load. We use a hashing technique to decide at which subnode to place a contact record. The hashing technique determines the subnode using only the contact record's URN.

A second way to alleviate the load on higher-level nodes is to make use of caches. We cannot effectively use a scheme in which URLs are cached since URLs can easily change in the presence of mobility. We therefore devised a caching scheme called *pointer caches*. Assume that a resource changes its URL mainly within a domain *D*, but hardly ever moves outside that domain. In that case, it makes sense to let the directory node for *D* store the URL, and subsequently let other nodes cache a pointer to the directory node for *D*. Since the resource will hardly ever move outside domain *D*, a cached pointer will remain valid despite that the URL of the resource may change regularly.

In this approach, whenever a lookup operation finds a URL at node *N*, it returns the URL as well as a pointer to *N*. All nodes visited during the lookup will then subsequently store the pointer to *N* in their local pointer cache. The next time a lookup operation visits any of these nodes, it can be immediately directed to node *N*. In this way, the lookup operation avoids visits to higher-level nodes. Details on this caching scheme can be found in [1]. The effects of both improvements are described below.

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-URN mappings. The current DNS infrastructure supports in the order of 10^8 host names and email destinations. By supporting only popular Web resources, we do not significantly increase the number of names stored in DNS, and we thereby ensure that we do not exceed its capacity.

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-URN mapping will be stored in the caches of name servers, providing users located near the cache with local access to the HFN-to-URN mapping. A DNS query to obtain the URN 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-URN 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 (i.e., subdomain) of the name space always originate from the same geographical area, we can place the name server responsible for that part in or near the area where the changes originate.

With the restrictions discussed above, DNS is an attractive name service, given its existing infrastructure. Unfortunately, if we want to drop those restrictions on the 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, caching will be ineffective, and DNS might become overloaded. If we want to support mobile resources, the caching mechanism might cache mappings in the wrong place. Therefore, 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 name service in [2]. Note that we do not criticize DNS: it has never been designed to support the HFNs as we propose and it can be argued that we are actually misusing the system.

Location Service

The problem of storing a large number of URN-to-URL mappings in the Globe location service can be divided into a storage and a processing problem. We can show by a simple computation that the storage requirements of the location service are not a problem. Consider, for example, the root node, and assume that a single contact record has a size of 1 KByte at the root. This 1 KByte of data should contain the URN, the forwarding pointers, some local administrative information, and still leave space for future additions, like cryptographic keys. If we assume a worst-case scenario, where our system supports in the order of 10^8 resources (as discussed above), this would mean that the root node has to store 100 GByte. Using the partitioning scheme mentioned earlier, we can distribute the 10^8 contact records over, say, 100 subnodes, resulting in 1 GByte per subnode. Using our partitioning scheme, storage requirements are clearly not a problem.

The processing of lookup requests poses a more serious threat. We can ignore update requests since they are rare compared to lookup requests. Our partitioning scheme clearly also increases the lookup processing capacity, but what if it still is not enough? To investigate the processing load further, we calculated the effect of replicated resources and simulated the effects of pointer caching on the lookup

processing load.

As a metric of the scalability of our location service, we introduce the *lookup length*. The lookup length is the number of nodes visited during a lookup operation, and provides an intuitive measure of the processing load in the tree. A large value means that many nodes have been visited, resulting in a load increase in all those nodes. It also means, in general, that nodes higher up in the tree (i.e., the more centralized nodes) have been visited. In essence, we would like to keep the lookup length as small as possible.

We first investigate the effect of resource replication on the location service. When a resource becomes more popular, invariably, more replicas of the resource will be added. This results in more URLs being stored in the location service. To provide optimal local access, the replicas will be distributed far away from each other, and this results in a tree in which the paths of forwarding pointers from the root down to the different URLs, will meet only in the *root* node. Assume each node in the tree has a fanout of N and that M replicas have been created, evenly distributed across the leaf domains. In this case, we can expect that M of the N children of the root node will have registered a replica in their respective domain. As a consequence, M out of N lookup requests will no longer need to be forwarded to the root node. If replicas are evenly distributed across leaf domains, the load on the root node thus decreases linearly with the number of replicas until $M = N$, and the root node is no longer used by lookup operations.

To investigate the effects of our pointer cache system, we conducted a simulation experiment. The basic idea is that with an increasing number of lookup operations, pointer caches should incur higher hit ratios, in turn, decreasing the average lookup length. In our simulation, we built a search tree of height four with a fanout of 32, leading to just over a million leaf nodes. The simulation consists of inserting a single URL at an arbitrary leaf node, and initiating lookup operations at randomly chosen leaf nodes. Each operation makes use of pointer caches, possibly creating new entries, as explained above. For each lookup operation, we compute its length by counting the number of nodes visited, and in the end, compute an average length. This average lookup length should decrease with the number of performed operations.

Figure 4 shows the result of our simulation, and confirms that with an increasing number of lookup operations the lookup length decreases, putting less load on the higher nodes in the tree. More importantly, the figure also shows that this effect is already present will small numbers of lookup operations. Since we only support popular Web resources, we know pointer cache entries will be reused, and caching will therefore be effective.

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 areas 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. Given our goal to support popular replicated Web resources, there is always

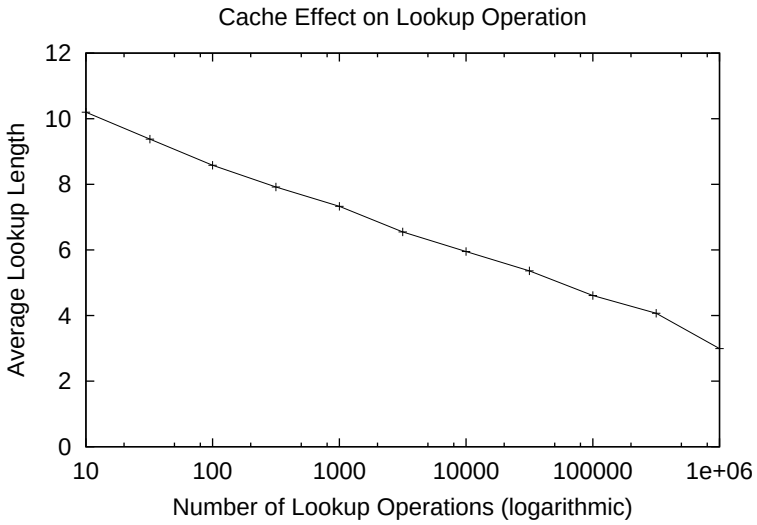

Figure 4: The average lookup length of a lookup operation.

a replica nearby.

Related Work

Most work regarding URIs is done within the working groups of the Internet Engineering Task Force (IETF). The URN working group has been primarily responsible for defining URNs. For instance, it has defined the overall URN name space in RFC 2141, provided an example URN namespace for IETF documents in RFC 2648, and outlined a general architecture to resolve URNs in 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. Resolving a URN thus requires the selection of the appropriate URN resolver. This selection of URN resolver is done by a Resolver Discovery Service (RDS). Daniel and Mealling propose to build an RDS using DNS [3]. 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. Our research has not included an RDS since we focused on one specific URN namespace, that is, the object handle space.

Another related working group of the IETF is the Common Name Resolution Protocol (CNRP) working group. The group is relatively new, and deals with the notion of human friendly naming through so-called “Common Names” [7]. Examples of common names are trade names, company names, and book titles. The goal of the working group is to create a lightweight search protocol. In this protocol,

a user provides parameters beside the common name to further specify the information being searched. Common names can be resolved at different information providers to get different types of information. The implementation of a scalable common name resolution *service* is outside the scope of the working group.

Related to the work done by the URN working group, is work done by the International DOI foundation [4]. Its goal is to develop the Digital Object Identifier (DOI). This is a system for identifying and exchanging intellectual property in the digital environment. This work was initiated by the American publishing community. The current DOI implementation uses the Handle system as its location service. The Handle system maps a DOI (known as 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. However, it does not ensure that the access information it provides refers to resources local to the user, nor does the handle resolution process use local resources when possible.

Kangasharju et al. [5] describe a location service (called LDS) that is based solely on DNS. Their system maps URLs to IP addresses, whereas in our approach, HFNs are mapped to URLs. In LDS, IP addresses of the servers are directly stored in DNS, while we store a URN 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 is added or removed, making their system more dynamic and caching less effective. In addition, we can easily and efficiently provide the URL that is nearest to the user, which is not the case in LDS.

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 systems are said to take both the client location as the current network condition into 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

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 for 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 in-

frastructure. This provides us with benefits in the form of an existing infrastructure and experience using it. We are aware of the limitations imposed by DNS, which has never been designed to support naming as proposed by us. As such, DNS is to be seen as an example naming system that can be used for demonstrating the feasibility of our approach.

We have implemented our HFN resolution scheme using the software of the BIND project and software we developed ourselves as part of the Globe project. The implementation is currently used in an initial setup involving four European sites, one site in the USA, and one site in the Middle-East. Our future work will consist of using the implementation in two experimental applications to gain more experience. The first application deals with replicating Web documents, and the second deals with the distribution of free software packages. These experiments will allow us to substantiate our scalability and human-friendliness claims.

References

[1] A. Baggio, G. Ballintijn, M. van Steen, and A.S. Tanenbaum. Efficient Tracking of Mobile Objects in Globe. Technical Report IR-481, Nov. 2000.

[2] G. Ballintijn and M. van Steen. Scalable Naming in Global Middleware. *Proc. Sixth Int'l Conf. on Parallel and Distributed Computing Systems*, Las Vegas, NV, USA, Aug. 2000.

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

[4] The International DOI Foundation. The Digital Object Identifier. `http://www.doi.org/`.

[5] J. Kangasharju, K.W. Ross, and J.W. Roberts. Locating Copies of Objects Using the Domain Name System. *Proc. Fourth Web Caching Workshop*, San Diego, CA, USA, Apr. 1999.

[6] P. Loshin, editor. *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. Morgan Kaufman, San Mateo, CA, USA, 2000.

[7] N. Popp, M. Mealling, L. Masinter, and K. Sollins. Context and Goals for Common Name Resolution. RFC 2972, Oct. 2000.

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

[9] M. van Steen, F.J. Hauck, P. Homburg, and A.S. 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.