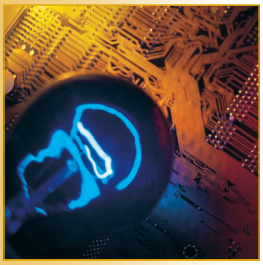

Scalable Human-Friendly Resource Names

A new uniform resource identifier proposes to improve both scalability and usability in naming replicated resources on the Web.

Gerco Ballintijn, Maarten van Steen, and Andrew S. Tanenbaum
Vrije Universiteit Amsterdam

To name resources, the World Wide Web uses uniform resource identifiers (URIs), the most common of which is the uniform resource locator. A URL serves two distinct purposes: to identify a resource and to access it. Unfortunately, combining these functions creates scalability problems because resource identification and access have different requirements.

Consider, for example, a popular Web page that we want to replicate to improve its availability. This currently requires using multiple URLs to identify each replica. To make replication transparent to users, we need a name that refers not to a specific replica but to the set of replicas as a whole. Uniform resource names (URNs), another URI type, provide a solution to this scalability problem. A URN differs from a URL in that it only *identifies* a Web resource; it does not indicate a resource's location or contain other information that might change.

In addition to scalability, we need to address usability: People need a way to name Web resources with identifiers that are easy to share and remember. To fill the gap between what URNs provide and what humans need, we propose a new kind of URI called human-friendly names (HFNs). In this article, we present the design for a scalable HFN-to-URL resolution mechanism that makes use of the Domain Name System (DNS) and the Globe location service to name and locate resources.

Naming Replicated Resources

Using URNs to identify resources and URLs to access them lets end users and Web developers use one URN to refer (indirectly) to copies at multiple locations. To access the resource identified by a URN, we need a way to resolve that URN into access information, such as a URL.

Figure 1a shows the current "one page, one URL" scheme; compare this to Figure

 1089-7801/01/$10.00©2001 IEEE

1b, which shows how URNs permit the transparent replication of Web resources. Because a URN refers to a resource rather than its location — much as an ISBN number identifies a book rather than its copies — we can move the resource around without changing its URN. A URN can thus support mobile resources by referring indirectly to a set of URLs that changes over time.

Because URNs identify resources to machines, they need not be human-friendly. RFC 1737 states only that URNs must be human transcribable — again like ISBN numbers, which people can write down but not remember easily.[1] RFC 2276 advocates high-level names that people find easy to use and that offer location-independent Web resource identification.[2] Unlike URNs, HFNs explicitly allow the use of descriptive, highly usable names.

Like a URN, an HFN must be resolved to one or more URLs when the user needs to access the named resource. We propose a two-step resolution process that binds the HFN to a URN and the URN to multiple URLs. The process, illustrated in Figure 1c, first requires resolving the HFN to its associated URN and then resolving the URN to its associated URLs.

This two-step approach offers many advantages. Replicating or moving a resource will not affect its name, for example, and a user can freely change the HFN without affecting replica placement. As in the use of symbolic links in file systems, a user might even decide to use several names to refer to a single resource.

Our HFN-to-URL resolution mechanism pays specific attention to two scalability issues: supporting a large number of resources and supporting 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.

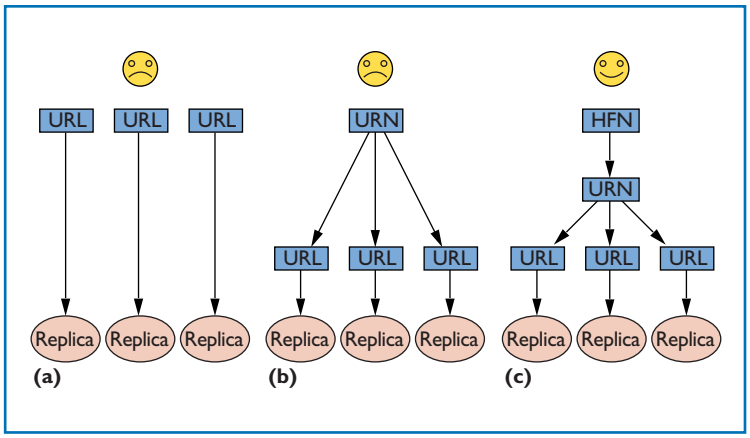

Figure 1. Three schemes for naming a replicated resource: (a) using multiple URLs, (b) using a single URN, and (c) using an HFN combined with a URN.

The HFN Naming Model

Human-friendly naming takes many different forms, the two major types being the well-known "yellow pages" and "white pages" services. The yellow-pages approach uses directory services such as those based on LDAP.[3] Users search for a resource based on attribute values that content providers have assigned to that resource. The main drawback to directory services is their limited scalability. In practice, only implementations based on local-area networks offer acceptable performance; large-scale, worldwide directory services have yet to be developed. At best, current implementations consist of federations of local directory services in which searches can span multiple sites only when severely restricted.

The white-pages approach employs a (possibly hierarchical) naming graph, such as that used in the DNS. Although they offer less advanced facilities than directory services, naming services have proven scalable to worldwide networks with millions of users. Taking advantage of this fact, our HFNs use a DNS-based hierarchical namespace, which gives users a convenient and well-known way to name resources.

For our naming system, we restrict ourselves to only highly popular and replicated Web resources. We have not yet incorporated support for other resource types, such as personal Web pages or highly mobile resources. We also assume that changes to a particular part of the namespace always originate from the same geographical area. We chose this restricted resource model to make efficient use of the existing DNS infrastructure. Given these restrictions, however, our HFN scheme is currently not appropriate as a general replacement for URLs.

Because we implement our HFNs using DNS, their syntax closely follows the structure of domain names. As an example, consider an HFN that refers to the source code of the current stable Linux kernel, `hfn:stable.src.linux.org`. The `hfn:` prefix identifies our URI scheme; the rest identifies the resource by name. Our security policy is designed just to prevent unauthorized changes to the HFN-to-URL mapping. Thus, the mapping is not kept confidential because we assume that HFNs will be shared as openly as URLs are today.

Because scalability depends heavily on locality, we want the HFN resolution service and its components to use nearby resources when possible.

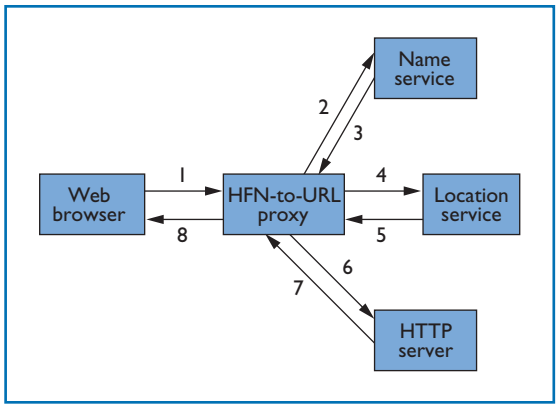

Figure 2. Retrieving Web resources named by HFNs. The HFN-to-URL proxy intercepts the HFN request, uses the name and location services to resolve the HFN, and retrieves the named resource from the HTTP server.

Resource name resolution should use locality in two ways:

- To permit scalability, the resolution service should give users direct access to the nearest replica.
- The name resolution process itself should use nearby resources whenever possible.

For example, assume a user located in San Francisco needs the DNS name `www.vu.nl` resolved. In the current DNS, name resolution would proceed through a root server — the `nl` domain name server (located in the Netherlands) — and the Vrije Universiteit name server (located in Amsterdam). If a replica of the resource happens to already exist in San Francisco, the lookup request will have traveled across the world to return a local address. The process would be far more efficient if the name resolution process used only name servers in the user's proximity.

HFN Architecture

In its general form, the HFN-to-URL mapping is an *N*-to-*M* relation where multiple HFNs can refer to the same set of URLs. This mapping might change regularly when, for example, a resource is given an extra name or a replica is added or moved. To efficiently store, retrieve, and update the HFN-to-URL mapping, we split it into two separate steps: HFN-to-URN and URN-to-URL. URNs provide stable, globally unique names for every resource. By splitting the HFN-to-URL mapping, we have an *N*-to-1 relation and a 1-to-*M* relation, each far easier to maintain than a single *N*-to-*M* relation.

To implement 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*. The HFN-to-URL proxy operates as a front end to the two services. It must recognize HFNs and then resolve them by querying the name service — which maintains the HFN-to-URN mapping for each unique resource — and the location service, which maintains the URN-to-URL mapping and determines the type of URN used in our naming scheme. The proxy then uses the URL obtained from the location service to access the named resource. In our design, the proxy is a separate process that can interact with any standard Web browser, although a plug-in module could introduce the same functionality directly into a browser.

Figure 2 shows our proposed method for retrieving 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 HFN-designated Web resource (step 1). The proxy recognizes the HFN and contacts the name service (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 (step 4), which resolves the URN to a URL and returns it to the proxy (step 5). The proxy contacts the HTTP server storing the named resource (step 6). The server returns an HTML page (step 7), which the proxy then passes to the Web browser (step 8).

Name Service

To store the HFN-to-URN mapping, we use the DNS, which currently serves primarily to name Internet hosts and e-mail destinations. With only minimal changes, we can reuse the existing DNS infrastructure for HFNs.

The Domain Name System. DNS provides an extensible hierarchical namespace in which general naming authorities delegate responsibility for parts of their namespace (subdomains) to more specific naming authorities. The naming authority for the `.com` domain, for example, delegates the responsibility for the `intel.com` domain to Intel. A naming authority provides the resources necessary for storing and querying a DNS name, and can decide which names to store in its subdomain. Intel can thus create whatever host name or e-mail destination it wants in its subdomain.

Conceptually, resolving a host name in DNS involves contacting a sequence of name servers that store increasingly specific domains. To resolve the host name `www.intel.com`, the reso-

Related Work in Wide-Area Naming

Most URI development occurs within Internet Engineering Task Force (IETF) working groups. The uniform resource name working group, for example, has defined the overall URN namespace (RFC 2141),[1] provided an example URN namespace for IETF documents (RFC 2648),[2] and outlined a general architecture for resolving URNs (RFC 2276).[3] In this architecture, the URN namespace actually consists of several independent URN namespaces, and every URN namespace has (potentially) its own specific URN resolver. Resolving a URN thus requires selecting the appropriate URN resolver, which is accomplished by a resolver discovery service (RDS).

In RFC 2168, Daniel and Mealling propose building an RDS that uses DNS to contain resource records specifying rewrite rules.[4] A DNS name server applies these rules to find the appropriate URN resolver and possibly even the resource itself. Our research does not include the RDS because we focus on one specific URN namespace, the object handle space.

The relatively new IETF common name resolution protocol (CNRP) working group approaches human-friendly naming through *common names*[5] such as trade names, company names, or book titles. The working group seeks to create a lightweight search protocol that lets users further refine searches by providing parameters in addition to the common name. Resolving common names at different information providers permits acquiring different types of information. The working group's scope does not include implementation of a scalable common name resolution *service*, however.

The digital object identifier, a project of the International DOI Foundation (http://www.doi.org) initiated by the U.S. publishing community, identifies intellectual property in the digital environment. As its location service, the current DOI implementation uses the Handle system, which maps a DOI (known as a handle) to access information, for instance, a URL. The handle's prefix specifies a naming authority; the suffix specifies a name under that naming authority. Resolving a handle requires a user to contact a *global handle registry* to find a *local handle registry*, where the handle can be fully resolved. The Handle system supports scalability by allowing replication of both the global and local handle registries. It does not ensure, however, that the access information it provides refers to resources local to the user, nor does the handle resolution process use local resources when possible.

The Location Data System (LDS),[6] a location service based solely on DNS, maps URLs to IP addresses, whereas our approach maps HFNs to URLs. LDS stores server IP addresses directly in DNS, while we store a URN in DNS and use a separate service to provide a set of URLs for the named resource. The LDS scheme requires updating the DNS server every time a replica is added or removed; this makes the system more dynamic and caching less effective. Unlike LDS, our system can efficiently provide the URL nearest to the user.

Akamai and Sandpiper base their commercial content delivery systems on the locations of the servers storing replicated Web resources. Both systems require the content provider to change the replicated resource's original URL to point to the delivery system servers. Akamai uses a modified Web server to redirect clients to servers, whereas Sandpiper uses a DNS-based solution. Both systems reportedly consider both client location and current network conditions when providing the client with a Web server. While both provide local access to the Web resources they support, however, their naming systems are not local.

References

1. R. Moats, "URN Syntax," IETF RFC 2141, May 1997; available at http://ietf.org/rfc/rfc2141.txt.
2. R. Moats, "A URN Namespace for IETF Documents," IETF RFC 2648, Aug. 1999; available at http://ietf.org/rfc/rfc2648.txt.
3. K. Sollins, "Architectural Principles of Uniform Resource Name Resolution," IETF RFC 2276, Jan. 1998; available at http://ietf.org/rfc/rfc2276.txt.
4. R. Daniel and M. Mealling, "Resolution of Uniform Resource Identifiers Using the Domain Name System," IETF RFC 2168, June 1997; available at http://ietf.org/rfc/rfc2168.txt.
5. N. Popp et al., "Context and Goals for Common Name Resolution," IETF RFC 2972, Oct. 2000.
6. J. Kangasharju, K.W. Ross, and J.W. Roberts, "Locating Copies of Objects Using the Domain Name System," *Proc. 4th Web Caching Workshop*, Cooperative Assn. for Internet Data Analysis (CAIDA), San Diego, Calif., 1999.

lution process visits, in turn, the name servers responsible for the root, .com, and .intel.com domains. The latter name server resolves the complete host name.

DNS uses caching extensively to enhance performance. When a name server is asked to resolve a DNS name recursively, it contacts the sequence of name servers itself to resolve the name. The name server can thus cache the intermediate and end results so that it won't have to repeat the process to look up the same or a similar name. For effective caching, however, 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 that fall into two categories. The first type stores user data such as the resource records for naming Internet hosts and e-mail destinations, and associates an IP address or mail server with a DNS name. DNS uses the second type, the name server resource record, internally to implement the namespace delegation. This resource record indicates another name server at which to continue name resolution.

Using DNS to store HFNs. We introduce a new type of resource record to store the association between

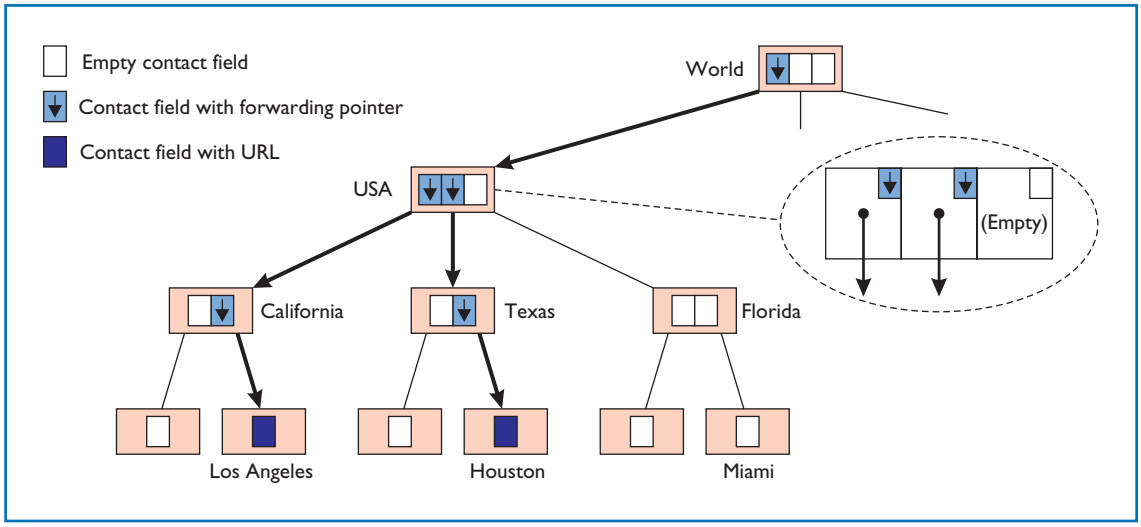

Figure 3. Search tree with contact records for a specific resource. The resource has two replicas, one located in Los Angeles and one in Houston. The contact record at the USA node has three contact fields, two containing a forwarding pointer and one that is empty.

a URN and a DNS name. When a user introduces a new HFN, he or she creates a resource record to store the associated URN. The user will subsequently insert this record into the DNS namespace. The name server responsible for the HFN's parent domain stores the record. To insert the HFN `hfn:devel.src.linux.org`, for instance, we must contact the server responsible for the `src.linux.org` domain. We can then insert the HFN at that server dynamically using the DNS `update` operation as described in RFC 2136.[4]

Location Service

To resolve URNs into URLs, we use the Globe location service.[5] A location service lets us associate a set of URLs with a single URN. To identify resources, Globe uses *object handles*, which we therefore use as URNs in our two-level HFN resolution scheme. (To simplify discussion, however, we will continue to use the term URN.) The location service also offers two update operations, `insert` and `delete`, for modifying the set of URLs associated with a URN.

Architecture. To permit efficient URL updates and lookups, we organize the underlying wide-area network (the Internet, for example) into a hierarchy of domains. These domains resemble those used in DNS except that we have tailored them to the location service only, where they represent geographical, administrative, or network topology regions. A lowest-level domain might represent a university campus-wide network, for example, and the next higher domain could be the city in which that campus is located. Another important difference is that the domain hierarchy in the location service is a completely internal structure that, unlike DNS, is invisible to users.

The location service maintains directory nodes representing each domain, and together these nodes form a worldwide search tree. Each directory node has a contact record, divided into contact fields for all child nodes, for every (registered) resource in its domain. A directory node stores either a forwarding pointer or the actual URLs in the contact field. A forwarding pointer indicates at which child node a lookup operation can find URLs. Because leaf nodes do not have any child nodes, their contact records differ from those at directory records in the rest of the tree in that 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 that leads to it from the root down. We can always locate a URL by starting at the root node and following this path. URLs are normally stored in leaf nodes, but storing them at intermediate nodes could lead to considerably more efficient lookup operations for highly mobile resources. Our current model excludes (highly) mobile resources, however, and assumes that all URLs are stored in leaf nodes.

Figure 3 shows an example of the contact records for one URN. The root node has one forwarding pointer for the URN, indicating that there are URLs in its left subtree, which is rooted at the USA node. The USA node, in turn, has forwarding pointers to the California and Texas nodes, and

both nodes have forwarding pointers to a leaf node that actually stores the URLs.

Operations. To find a resource's URL, a user initiates a lookup operation at the leaf node of the domain in which he or she resides. The user provides the resource's URN as a parameter, and the lookup operation checks to see whether the leaf node has a contact record for it. If so, 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, it follows the path of forwarding pointers from this node down to a leaf node where a URL is found. If the lookup finds no contact record at any node between the leaf node and the root, the location service considers the URN as unknown.

Consider a user located near the Miami leaf node, for example, as shown in Figure 3. When the user contacts the leaf node with a URL request, the node forwards the request to its parent, the Florida node, because it does not contain a contact record. The Florida node also has no record for the URN and forwards the request to its parent, the USA node. The USA node, storing a contact record for the resource, sends the request to one of its children based on the forwarding pointers in the contact record. The lookup operation then follows the path of forwarding pointers to a leaf node such as Houston. By going higher in the search tree, the lookup operation effectively broadens the search area for a URL, thus resembling search algorithms based on expanding rings.

The insert operation stores a URL at a leaf node and creates a path of forwarding pointers leading to that node. When a resource has a new replica in a leaf domain, the resource's owner inserts the replica's URL at the leaf domain's node. The insert operation starts by entering the URL in the leaf node's contact record, then recursively requesting the parent node, grandparent node, and so on, to install a forwarding pointer. The recursion stops when the insert operation finds a node that already contains a forwarding pointer; otherwise, it stops at the root. The delete operation removes the URLs and paths of forwarding pointers in a manner analogous to the insert operation. Further technical details are available elsewhere.[6]

Improvements. Obviously, the basic search tree described so far does not yet scale. Higher-level directory nodes, such as the root, pose serious problems because they must store numerous contact records and handle many requests. Our solution is to partition overloaded directory nodes into multiple subnodes, each of which manages only a subset of the contact records. We employ a hashing technique that uses the URN to decide which subnode should store each contact record.

We can also use caches to alleviate the load on higher-level nodes. A scheme that caches URLs won't be effective because these can easily change as resources become mobile. We therefore devised a caching scheme, detailed elsewhere, called *pointer caches*.[7] If a resource changes its URL but rarely moves outside a domain *D*, we can let the directory node for *D* store the URL. Other nodes can cache pointers to the directory node. Because the resource will typically remain within *D*, cached pointers will remain valid even if the resource's URL changes regularly.

We can also use caches to alleviate the load on higher level nodes.

In this approach, whenever a lookup operation finds a URL at node *N*, it returns the URL and a pointer to *N*. All nodes visited during the lookup will subsequently store the pointer to *N* in their local pointer cache. The next time a lookup operation visits any of these nodes, it will immediately be directed to *N*, bypassing higher-level nodes.

Implications

For our HFN-to-URL resolution scheme to be scalable, the name and location services must support numerous resources distributed over a large geographical area. This entails several technical challenges for each service.

Name Service

Our first requirement is for the name service to handle numerous HFN-to-URN mappings. The current DNS infrastructure supports about 10^8 host names and e-mail destinations. By supporting only popular Web resources, we limit the number of HFNs stored in DNS and thereby ensure that we do not exceed its capacity.

We can handle geographically dispersed names by localizing lookup and update operations. DNS uses caches to localize lookup operations. Assuming the use of popular Web resources and a stable HFN-to-URN mapping, caching will remain effective. DNS queries will obtain the URNs from the name server cache without having to contact remote name servers, giving nearby users local

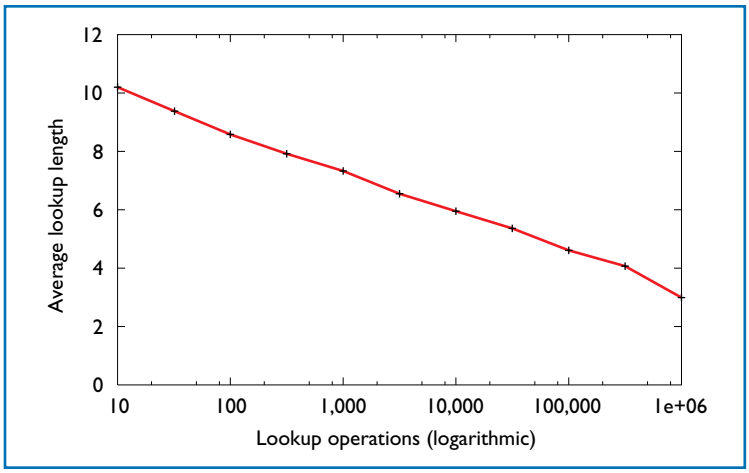

Figure 4. Average length of a lookup operation. With increasing popularity (that is, number of lookup requests), caching decreases the number of nodes visited in the search tree.

access to the HFN-to-URN mapping. Name service update operations also exploit locality. Assuming changes to a specific part (subdomain) of the namespace always originate from the same geographical area, we can place the name server responsible for that subdomain in or near the appropriate area.

In spite of its restrictions, DNS is an attractive name service with a pervasive infrastructure. Unfortunately, DNS was never designed to support the HFNs we propose; some might even argue that we are misusing it. For this reason, we impose restrictions on the resource model and avoid scalability problems in DNS that could otherwise threaten our HFN resolution mechanism. For instance, caching will prove ineffective in supporting HFNs for unpopular resources, and DNS might get overloaded. The caching mechanism might also cache mappings for mobile resources in the wrong place. Therefore, to support a more general resource model, we need to replace DNS with a more scalable name service. We have described a design for such a system elsewhere.[8]

Location Service

Storing many URN-to-URL mappings in the Globe location service forces us to face both storage and processing issues. Consider, for example, the root node, and assume that a single contact record takes up 1 Kbyte. This record should contain the URN, forwarding pointers, and local administrative information, while still leaving space for additions such as cryptographic keys. Assuming the worst-case scenario, in which our system supports 10^8 resources, the root node must store 100 Gbytes. Our partitioning scheme lets us distribute the contact records over, say, 100 subnodes, resulting in 1 Gbyte per subnode. Using this scheme, the location service's storage requirements clearly create no problem.

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

To measure our location service's scalability, we introduce the *lookup length*. This metric represents 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 generally means that nodes higher up in the tree (those more centralized) have also been visited. In essence, we would like to keep the lookup length as small as possible.

We first investigated how resource replication affected the location service. When a resource becomes more popular, replicas invariably proliferate, and more URLs will need to be stored in the location service. To provide optimal local access, we distributed the replicas widely. This also creates a search tree in which the paths of forwarding pointers from the root to the different URLs meet only in the root node. Assuming each node in the tree has a fan-out of N and that M replicas are evenly distributed across the leaf domains, we expect that M of the N children of the root node will have registered a replica in their respective domain. Consequently, M out of N lookup requests no longer need to be forwarded to the root node. Distributing replicas evenly across leaf domains decreases the load on the root node linearly with the number of replicas until $M = N$ and lookup operations no longer use the root node.

To investigate the effects of our pointer cache system, we conducted a simulation experiment. We surmised 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 fan-out 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 leaves. Each operation uses pointer caches, possibly creating new entries. We computed each lookup length by

counting the number of nodes visited, then computed an average length. This average lookup length should decrease with the number of operations performed.

Figure 4 shows the result of our simulation, and confirms that lookup length decreases as the number of lookup operations increases, lightening the load on the tree's higher nodes. More importantly, the figure also shows that this effect emerges even with small numbers of lookup operations. Because 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 at the leaf node, searching 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 proves accessible using just local resources. Given our goal of supporting popular replicated Web resources, there should always be a replica nearby.

Future Work

Our location service resolves HFNs to URLs by employing two distinct mappings — one for naming resources and one for locating them. This separation aids scalability by letting us apply techniques specific to each respective service. Reusing the DNS infrastructure provides us the benefits of an existing infrastructure and experience using it. Recognizing the limitations DNS imposes, however, we use it simply as an example naming system to demonstrate the feasibility of our approach.

We have implemented our HFN resolution scheme using the Berkeley Internet Name Daemon (BIND) and software we developed as part of the Globe project. The initial setup involves four sites in Europe, one in the U.S., and one in the Middle East. We plan to use this implementation in two experimental applications, the first dealing with replicating Web documents, the second with distributing free software packages. These experiments will, we hope, allow us to substantiate our scalability and human-friendliness claims.

References

1. K. Sollins and L. Masinter, "Functional Requirements for Uniform Resource Names," Internet Engineering Task Force (IETF) RFC 1737, Dec. 1994; available at http://ietf.org/rfc/rfc1737.txt.
2. K. Sollins, "Architectural Principles of Uniform Resource Name Resolution," IETF RFC 2276, Jan. 1998; available at http://ietf.org/rfc/rfc2276.txt.
3. P. Loshin, ed., *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs,* Morgan Kaufman, San Francisco, Calif., 2000.
4. P. Vixie et al., "Dynamic Updates in the Domain Name System (DNS UPDATE)," IETF RFC 2136, Apr. 1997; available at http://ietf.org/rfc/rfc2136.txt.
5. M. van Steen et al., "Locating Objects in Wide-Area Systems," *IEEE Comm.*, vol. 36, no. 1, Jan. 1998, pp. 104–109.
6. M. van Steen et al., "Algorithmic Design of the Globe Wide-Area Location Service," *Computer J.*, vol. 41, no. 5, 1998, pp. 297–310.
7. A. Baggio et al., "Efficient Tracking of Mobile Objects in Globe," tech. report IR-481, Vrije Universiteit, Amsterdam, Nov. 2000.
8. G. Ballintijn and M. van Steen, "Scalable Naming in Global Middleware," *Proc. Sixth Int'l Conf. Parallel and Distributed Computing Systems*, Int'l Society for Computers and Their Applications (ISCA), 2000, pp. 624-631.

Gerco Ballintijn is a PhD student in the computer systems group at the Vrije Universiteit in Amsterdam. He is currently completing his research on the Globe location service. He is a student member of the IEEE and the ACM. His research interests include naming systems, middleware, computer networks, and operating systems.

Maarten van Steen is associate professor of computer science at the Vrije Universiteit, Amsterdam. He has an MS in applied mathematics from Twente University and a PhD in computer science from Leiden University. Van Steen worked at an industrial research laboratory for several years before returning to academia. His research interests include operating systems, computer networks, wide-area distributed systems, and Web-based systems. He is a member of the IEEE and the ACM.

Andrew S. Tanenbaum has a BS from the Massachusetts Institute of Technology and a PhD from the University of California, Berkeley. He is currently a professor of computer science at the Vrije Universiteit and dean of the inter-university computer science graduate school, ASCI. Tanenbaum is the principal designer of three operating systems and chief designer of the Amsterdam Compiler Kit. He is an IEEE Fellow, an ACM Fellow, and a member of the Royal Dutch Academy of Sciences. In 1994 he received the ACM Karl V. Karlstrom Outstanding Educator Award, and in 1997 he won the SIGCSE award for contributions to computer science.

Readers can contact the authors at gerco@cs.vu.nl or steen@cs.vu.nl.