Design and Scalability of NLS, a Scalable Naming and Location Service

Y. Charlie Hu, Daniel A. Rodney and Peter Druschel
Computer Science Department
Rice University
Houston, TX 77005
{ychu@cs.rice.edu, drodney@rice.edu, druschel@cs.rice.edu}

PRELIMINARY DRAFT

Abstract

This paper sketches the design, and presents a scalability analysis and evaluation of NLS, a scalable naming and location service. NLS resolves textual names to the nearest of a set of replicated objects associated with that name, and is designed to scale to the dimensions of a world-wide service. Applications include resolving Web URIs to the nearest cached or replicated object that provides the associated content. The key design goals of NLS are scalability, performance, availability and ease of administration. NLS is based on a dynamically configured, distributed search tree, with a fat-tree based topology at the global level and spanning trees at the local level. Analysis and preliminary empirical results obtained with a prototype implementation indicate that the system scales as expected.

1 Introduction

In order to improve the scalability, performance, and availability of the World Wide Web, the strict client-server model of the first generation Web is gradually being abandoned in favor of a model where Web content and services are replicated in different geographic locations throughout the Internet. This trend can be seen in the proliferation of caching proxies [7, 24, 16], content delivery services like Akamai and Digital Island [2, 9], and the recent interest in active proxy technology [6, 10].

An increasingly important problem in this context is the ability to locate the "nearest" replica of a Web resource in response to a client request for that resource. We have designed and implemented a distributed naming and location service (NLS) that is able to resolve an object name (for instance, a URI) to the address of the nearest object that provides the associated resource, while scaling to the dimensions of the entire WWW. The focus of this paper is on the analytical and experimental evaluation of NLS's scalability.

Our service can be used by a caching proxy to locate the nearest replica of a resource (stored at another proxy or server) that was requested by a client but not found in the cache of the client's local proxy. Alternatively, the service could be used directly by Web browsers to locate the nearest replica of a resource requested by the user.

NLS is based on distributed search in a tree of hierarchical domains, as originally proposed in the GLOBE location service [22]. NLS extends this prior work in several directions, improving the scalability, flexibility, and administrative ease of the previously proposed system. Moreover, this paper presents the first analytical and experimental evaluation of the scalability of this approach.

NLS differs from the GLOBE location service in several ways:

- NLS integrates the naming and location functions, which enables the aggregation of location information in the search tree for co-located objects whose names have a common prefix, thus increasing the scalability.
- NLS uses hashcodes to store names of bound objects in the interior nodes of the search tree, thereby reducing the storage requirements per object and thus further improving scalability.
- The NLS search tree dynamically configures itself to reflect the measured proximity of the NLS nodes in the network, based on a given proximity metric (e.g., latency, bandwidth, hop count or a combination thereof). This property allows NLS to track topology changes in the network and thus greatly eases administration.
- NLS uses a two-level tree-based naming architecture with a fat tree [15] topology at the global level, and a forest of spanning trees at the local level. The fat-tree topology allows the systematic use of replication (for availability) and partitioning (for scalability) in the upper portions of the search tree. The self-configuring spanning tree topology at the local level is able to track frequent changes in network topology, node availability and configuration at the fringes of the network.

This paper sketches the design, presents a scalability analysis, and presents preliminary empirical results with a prototype implementation of NLS. Based on small to medium scale experiments, we conclude that NLS should scale to a world-wide service able to maintain in excess of 1 Billion bindings. However, larger scale simulations will be necessary to confirm this conjecture.

The rest of this paper is organized as follows. Section 2 presents some background on naming and caching in the Web, and on DNS. The design of NLS is presented in Section 3. A scalability analysis is presented in Section 4. Section 5 presents preliminary experimental results obtained with a prototype implementation of NLS. Related work is covered in Section 6 and Section 7 offers some conclusions.

2 Background

In this section, we cover some technical background on naming and caching in the Web, and on DNS.

The domain name system (DNS) [17] is used in the Internet to map domain names to IP addresses. The service is implemented by a hierarchy of name servers that reflects the structure of the DNS name space. The name space is partitioned into different administrative zones representing all names with a given suffix, and an organization in charge of a zone provides one or more DNS servers, which perform name translation for all names with the corresponding suffix.

For instance, Anon University operates a set of DNS servers that are in charge of mapping all domain names that end in `anon.edu` to the associated IP address. These servers are the children of a DNS node in charge of all names with suffix `.edu`. The servers internally rely on child DNS servers in charge of names ending in `cs.anon.edu, ece.anon.edu`, etc. For high availability, DNS servers are replicated. To reduce query load, DNS leaf servers cache previously resolved bindings. The binding are timed out to prevent persistent stale mappings. Since DNS mapping are not assumed to change frequently, a typically timeout period is 15 minutes.

A domain name can be mapped to multiple IP addresses, and DNS will resolve queries for the associated domain to one of the IP addresses in a round-robin fashion. This affords a primitive form of load balancing among multiple machines that implement a given server.

Although not standard behavior, a DNS server implementation is able to take into account the network location from which a query originates in choosing an IP address used in answering the query. Using this trick, DNS can be used to map a domain name to the IP address of a "nearby" (i.e., close to the client) server or proxy. While technical details are undisclosed, it is believed that networked proxy operators like Akamai [2] and Digital Island [9] use this approach.

Once a client request arrives at a proxy, and that proxy does not hold the requested resource, the proxy needs to locate the closest proxy or server that does have the requested resource. In a large-scale system, this requires a distributed location service like NLS.

Currently, naming and location in the WWW is based on uniform resource locators (URLs) [4]. A URL contains the DNS domain name of the server that hosts the associated resource. During a lookup, the Web browser extracts the domain component from the URL, obtains the server's IP address using DNS, contacts the Web server application using the IP address and a well-known or specified (in the URL) port number, and request the document identified by the remainder of the URL relative to the Web server's local document root.

Caching proxies attempt to improve response times, reduce server load and network traffic by keeping copies of Web content close to a client community. Client requests can be intercepted by proxies in one of three ways:

Conventional proxies require that each user's browser be configured to send all client requests to a given proxy. The proxy either responds to the request using a cached copy of the requested document, or else forwards the request to the server.

Transparent proxies transparently intercept outgoing request packets from client browsers in a given subnet. Like conventional proxies, they either respond with a cached copy or forward the request to the server.

Networked proxies cache content, streaming media and, in the future, dynamic Web applications from content providers that contract with the proxy network operator (e.g., Akamai or Digital Island). The content providers specify URLs that contain the proxy network operator's domain name for all content that they wish to be served from the proxy network. Name translation proceeds as follows. The DNS server for the proxy network operator's domain returns an IP address that depends on the subnet address from which the DNS query originates (i.e., the client's location in the network). It does so by looking up the IP address of the proxy closest to the client's location. As a result, the client's HTTP request is directed to a "nearby" proxy. If that proxy caches the requested content, the latter is returned to the client. Else, the proxy attempts to locate the "closest" copy of the requested resource at another proxy in the network, fetches that content, and returns it to the client. This requires a location service.

Among the three types of proxies, the third is most powerful, as it automatically locates a copy of the requested resource closest to the client. Furthermore, networked proxies are in principle able to serve not only static Web content and streaming media, but also Web applications that can supply dynamically generated content.

Networked proxies require a location service that allows them to locate the closest copy of a requested resource within the proxy network. Today, commercial operators use proprietary location services for this purpose, with a scope that is limited to just that operator's proxies. The NLS service is intended as a general solution to the location problem, and is designed to scale to the total number of objects in today's Web and beyond ($> 10^9$ objects).

3 Design

In this section, we sketch the design of NLS, our scalable naming and location service. Due to space constraints, we focus on those aspects of the design relevant to scalability.

NLS maps hierarchical, textual names to a set of object addresses that provide the associated resource. Names in our service consist of a variable number of textual name components, separated by the slash "/" character. As a result, URLs are legal names in NLS, which is convenient for backward compatibility. However, the service makes no assumptions about the meaning of pathname components; in particular, no part of a name is assumed to imply the location of the associated resource. The object addresses are opaque to NLS; in practice, they could represent $< IPaddress, port, pathname >$ triples, CORBA/IIOP handles, or any other form of object address.

There are several key requirements that together differentiate our naming and location service from a pure naming service such as DNS:

Replication and migration of objects Our naming and location service is designed for environments where Web resources (content and services) can migrate between nodes (e.g., servers and proxies) in the Internet. Consequently, a key requirement for the naming and location service is support for multiple bindings of a name to objects at different locations in the Internet. This implies that the location service must not rely on object names to contain location information.

Locating the nearest replica A fundamental requirement in the design of NLS is the ability to resolve a name (e.g., a URL) to the "nearest" of a set of replicated objects that can provide the requested content or service associated with the name. Proximity is defined here in terms of the characteristics (e.g., latency, bandwidth, hop count, loss rate, cost, security) of the network connection between the parties.

Scalability NLS should be able to support a very large number of names (more than one billion), with associated objects that can potentially be located anywhere in the world.

Availability The NLS service needs to remain available in the event of NLS server crashes, and minimize the number of clients for which the service is disrupted. In addition, after recovery, the crashed name server needs to be able to restore the lost location information to be consistent with the other servers.

Ease of administration Lastly, the NLS service needs to be easy to manage, especially when the network topology changes, when adding/removing name resolvers, and during failure and recovery of name resolvers.

The existing Internet DNS service does not meet the first requirement, for several reasons. First, in typical use, it maps domain names to IP address(es), as opposed to full object names to the nearest object address. Second, even though DNS can be manipulated into mapping a domain name to the address of a "nearby" server, this does not solve the problem of locating the closest server that provides the requested resource. In other word, the DNS server that needs to resolve a domain name query to a nearby server address needs to rely on a secondary location service for this purpose.

In the following, we describe the design of NLS, a service that satisfies all the requirements. NLS is based on a distributed search tree structure of name resolvers [21], which allows multiple bindings and migration of objects and locates the nearest replica. The scalability of the simple search tree is then significantly enhanced with the following techniques: (1) NLS uses a fat-tree based organization of resolvers in the upper level of the search tree to enable systematic replication (for availability) and partitioning (for scalability) of location information among multiple resolvers; (2) NLS uses prefixes of object names to aggregate location information, thus reducing the amount of data stored and consequently improving the scalability and performance at upper levels of the search tree; (3) NLS stores object names in the interior nodes of the search tree in the form of 64-bit hashcodes. (4) NLS caches name bindings at leaf resolvers to reduce query load on interior resolvers.

High availability is achieved by systematically replicating location information among fat-tree nodes at the upper level of the search tree, and by using automatically (re-)configuring spanning trees of resolvers at the lower level. Ease of administration and dynamic adaptation to a changing network topology is achieved via the ability to dynamically reconfigure the fat-tree at upper levels and via the use of the automatically (re-)configuring spanning tree of resolvers at lower levels.

3.1 A Distributed Search Tree Based Approach

The NLS name resolvers form a distributed tree, with the property that leaf resolvers attached to the same subtree are "closer" to each other (according to the proximity metric) than leaf resolvers that are attached to different subtrees of the same height. Objects bind themselves to a name at the NLS leaf node closest to the object's location.

The name-to-address binding of an object is stored only at the leaf resolver that represents the region in which the object's repository resides. For each new binding, it constructs a path of *forwarding pointers* from the root to the leaf resolver closest to the object's repository (that is, the leaf resolver where the object was bound), shown in Figure 1. A forwarding pointer in

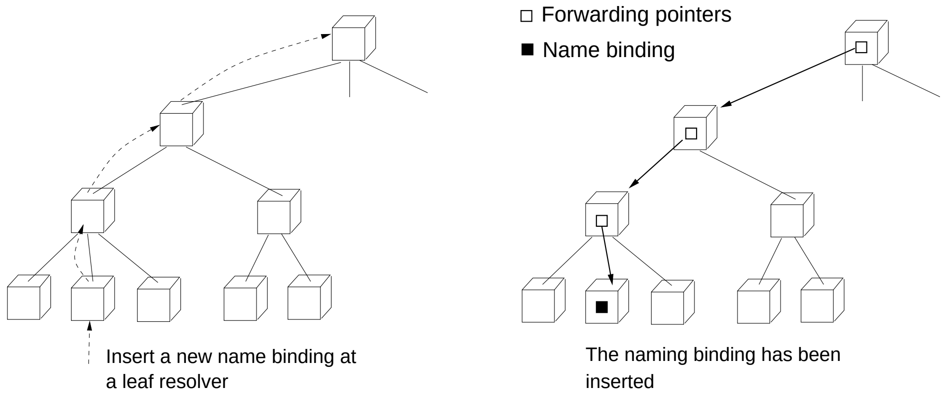
□ Forwarding pointers
■ Name binding
Insert a new name binding at a leaf resolver
The naming binding has been inserted

Figure 1: Inserting a new object in a distributed search tree.

a resolver maps an object name to a (set of) child resolvers that is (are) one hop closer to the leaf resolver(s) that has (have) bindings for the associated object(s).

This design ensures that in the worse case, the address of an object can always be found by following a chain from the root resolver. In general, the locality inherent in the search tree structure implies that looking up a name that has a nearby replica of the associated object never has to go up beyond the lowest common ancestor in the tree. The basic operations provided by NLS are implemented as follows:

Insertion An insertion of an object binding at a leaf resolver results in the storage of the name-to-address binding in the leaf resolver, followed by an upward traversal from that leaf resolver toward the root, inserting *forwarding pointers* along the way. If at some resolver *A* along the path, its parent resolver *B* already contains a forwarding pointer to a different child *C*, a new forwarding pointer is inserted at the parent resolver to point to *A*, and the upward propagation is stopped since all the resolvers along the path between the parent resolver and the root already have forwarding pointers.

Deletion A deletion of an object from a leaf resolver where it was previously bound causes an upward traversal towards the root. It removes forwarding pointers along the way, and stops at the first internal resolver that has at least one other forwarding pointer for the same object.

Lookup A lookup of an object at a leaf resolver causes an upward traversal, but stops at the first node that contains a forwarding pointer for the object. It then follows the forwarding pointers until it reaches the leaf node containing the location information of the object being looked up. A look-up operation is guaranteed to find the first forwarding pointer in the lowest common ancestor of the resolver from which the look-up originates and the nearest leaf resolver that contains a binding of the object being looked up. In other words, the look-up operation is guaranteed to return the nearest replica of the object.

NLS decentralizes the task of resolving names and locations among a group of hierarchically organized resolvers corresponding to the topological, hierarchical decomposition of the physical domain of objects. Thus it has the potential to scale well to wide-area networks by taking advantage of the inherent locality in the hierarchical organization of location information. The use of forwarding pointers is what enables an object to have multiple bindings at different leaf resolvers, and the inherent locality in the tree structure allows locating the nearest replica.

The simple search tree structure described so far, however, can suffer from both poor scalability and poor availability. First, the single root of the search tree needs to know about every object in the universe, which severely limits its scalability, both in terms of storage and query load. Second, the single parent of each node in the search tree is vulnerable to single-point failure, which results in disconnection of its entire subtree. The design of NLS significantly enhances the scalability and availability of the simple search tree, as described in the following sections.

3.2 Improving Scalability

We begin by describing several techniques used in NLS to improve the scalability of the basic distributed search tree based approach.

Using Prefixes to Aggregate Location Information NLS provides name transparency, that is, NLS makes no assumptions about the meaning of any component of a name. However, even when using names that do not reflect the location of an object, convention often dictates that names contain the name of the organization that created the associated object. This is to ensure unique name assignment without requiring centralized control of the name space.

NLS is able to take advantage of this convention as follows. When names contain the name of the organization that creates the associated object, the result is a form of locality in the name space. For instance, all URLs with the domain name `www.anon.edu` are likely to be bound at the name resolver on the Anon University campus. In this case, the Anon leaf resolver is able to create a path of forwarding pointers for *all* objects with the prefix `www.anon.edu` by inserting only this prefix. To deal with prefixes, the lookup procedure follows the forwarding pointers associated with the longest prefix match.

Note that when objects are replicated at locations other than their primary (home) location, individual name bindings for these replicated objects must be propagated up the tree. However, prefixes can still dramatically reduce the number of bindings maintained in the upper portions of the search tree by allowing all the primary (i.e., home) bindings of a name to be represented jointly with a single prefix.

Using Hashcodes to Compress Name Information Instead of storing full names associated with the forwarding pointers in the interior nodes of the search tree, NLS stores only a 64-bit hashcode for each name, thus substantially reducing the storage costs in the interior nodes. However, storing hashcodes instead of full names affects the lookup and bind procedure, due to the possibility of hash collisions. During the lookup of a name, it is possible that a chain of forwarding pointers leads to a leaf node that does not actually have a binding for the desired name, but instead has a binding for a name with the same hashcode.

To handle this case, the lookup procedure is augmented as follows. In the event of a failure to locate the desired name at a leaf node, the search backtracks to the lowest interior node that has another, not yet explored, forwarding pointer, and it follows that pointer. It is easy to see that such a node must exist if a binding for the given name exists. With a 64-bit hashcode and a suitable hash function, the expected number of collisions is low, even when the number of bound names goes into the billions. Therefore, the impact of backtracking on the average lookup performance is expected to be marginal.

Caching Name Bindings in the Leaf Resolvers After a leaf resolver has obtained a binding for a name in response to a client query, it inserts the binding into a cache. Subsequent client queries for the same name are then resolved from the cache, thus reducing query load on the interior nodes.

To control stale bindings in the cache, two measures are taken. First, cached binding are evicted after a specified timeout. This is to ensure that a leaf resolver does not continue to return bindings for a far away object, even though a nearby copy of the object has become available. Second, a client that finds that it cannot contact an object at the address provided by NLS may re-issue the lookup with a special flag that causes NLS to invalidate any cached bindings for the name.

3.3 A Two-Level Tree-Based Naming Architecture for Scalability and Availability

The distributed search tree structure of name resolvers can be logically divided into *the global level*, consisting of upper levels of the tree, and *the local level*, consisting of lower levels of the tree. The two levels have different performance and configuration requirements. At the global level, the scalability and availability are the main concerns since there are fewer and fewer resolvers compared to the local level. At the local level, the scalability is less of a concern. However, topology changes and reconfigurations are much more frequent at this level.

Fat-tree for the Global Level The pressure on scalability at the global level of the naming and location service necessitates the partitioning of location information among multiple resolvers at each level. At the same time, it is desirable to maintain the overall hierarchical organization of resolvers because of its inherent locality property. Therefore, we employ a fat-tree [15] to organize the resolvers at the global level of NLS. Figure 2 shows one example of a fan-in-3/fan-out-2 depth-two fat-tree, where every non-root node has two parents and every non-leaf node has three children.

In the context of NLS, the multiple ancestors in the fat-tree can be used for scalability or availability. First, the set of name bindings that need to be maintained in each logical node can be *partitioned* among several physical nodes, reducing the storage requirements and query processing load on the physical nodes, thus improving scalability. Figure 3 shows an example where the location information at a leaf node is recursively partitioned amongst its ancestor nodes.

In addition, each partition of the set of name bindings in a logical node can be *replicated* in multiple physical nodes to improve availability. Availability is improved because, upon a node failure, a replicated node in the same partition as the failed node can be contacted. Figure 4 shows an example where the location information at a leaf node is replicated among its parent nodes at level 1, which in turn partition the information among their parent nodes at level 2. Of course, partitioning and replication can be applied to every logical node in the fat-tree.

Routing in the fat-tree We next sketch a routing algorithm for lookup and bind requests in a fat-tree that uses both partitioning and replication at each logical node. The algorithm guarantees that query traversals are routed upward in the tree in such a way that a matching name is found at the lowest common ancestor of the leaf resolver from which the query originates and the leaf resolver where the closest object associated with the name is bound, regardless of the leaf node from which the query originates.

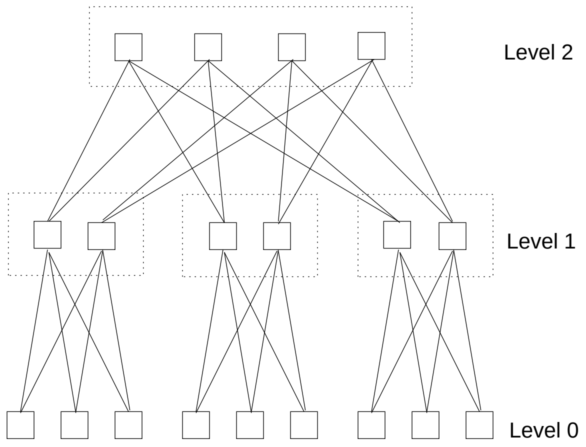

Figure 2: A depth-2 fan-in-3/fan-out-2 fat-tree. Dashed boxes denote logical nodes in the original simple search tree and solid boxes within each dashed boxes denote physical nodes that represent the logical node.

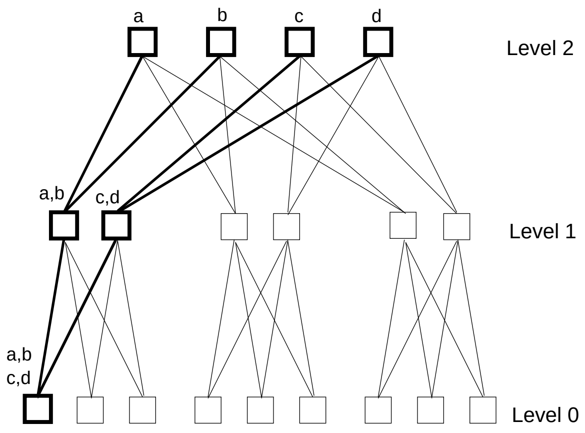

Figure 3: Partitioning location information among multiple ancestors in a fat-tree for scalability. The location information at a leaf node is partitioned evenly among its two parent nodes at level 1 and four grandparent nodes at level 2.

The routing during upward traversals in the fat-tree is performed based on the hashcode of the object's name, divided into h segments, where h is the depth of the fat-tree. For simplicity, let us assume a fat-tree with full partitioning, where each object is stored in exactly one resolver of a logical node. Here, the ith segment is used to uniquely determine which parent node at level $(i + 1)$ to route the operation to, in moving up from a node at level i. It can be proved that such a hashing-based scheme always finds the nearest replica for a lookup operation, just as in the original simple search tree.

When replication is used in addition to partitioning, the ith segment is used to select the partition at level $(i + 1)$, and then one of the replicated resolvers in that partition is selected to send the query to. The replicated resolvers in each parent partition are maintained in the order of their topological proximity to the child node. Normally, queries are directed to the closest node, unless that node failed or is overloaded, in which case the query is forwarded to the next closest replicated resolver.

Dynamic Spanning Trees for the Local Level At the local level, scalability is less of a concern, but the ability to automatically reconfigure and maintain the tree structure is more important since the resolvers are much more likely to be added or deleted dynamically. In addition, high availability of NLS remains a necessity. Our approach to address both requirements is to automatically and dynamically maintain a spanning tree of location resolvers. Due to space considerations, we omit the details.

3.4 Configuration and Adaptation

As mentioned above, the spanning tree of resolvers at the lower level of NLS is fully self-configuring.

The initial configuration of the fat-tree in the upper level of NLS is computed based on the measured or estimated density and distribution of objects and names throughout the network. The configuration should balance lookup and storage loads and can be generated in two steps. First, we form a graph with all the spanning tree roots as the graph nodes, and with measured proximity as the weighted edges between the nodes. We can then recursively partition such a weighted-edge graph using well-known heuristics such as the Kernighan-Lin algorithm [13]. Such a recursively partitioning of the graph effectively gives a simple tree of recursively decomposed subdomains of spanning tree roots. In the second step, we simply choose a fan-out, and construct a fat tree out of the simple tree.

In a dynamic network environment, the proximity measures between different subdomains represented by the fat-tree nodes can change over time, although the rate of change can be expected to be lower than at the fringes of the network. Thus, the fat-tree configuration must be able to adapt to the changing network topology and workloads. One key observation about the fat-tree is that even though there are rich connections among its nodes, there exists a local structure between two adjacent levels that can be expanded without affecting the rest of the fat-tree structure. Specifically, in a fan-in-m/fan-out-n fat-tree, every n consecutive nodes at a non-leaf level form a *parent group* – they connect to the same m child nodes at the level below (see Figure 2 for an example.) We can exploit this, for instance, if a node in a parent group is becoming overloaded. A new node can be added to the parent group in four steps, as shown in Figure 5. Space limitations prevent us from a complete discussion of the ways in which the fat-tree can be dynamically reconfigured and adapted.

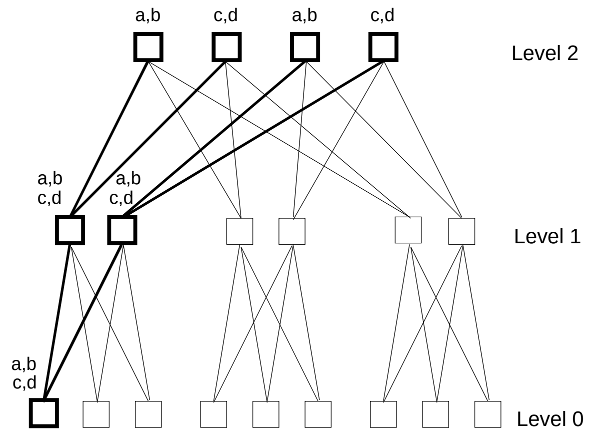

Figure 4: Partitioning and replicating location information among multiple ancestors in a fat-tree for scalability and availability. The location information at a leaf node is replicated among its three parent nodes at level 1, each of which then partitions the location information among its own three parent nodes at level 2.

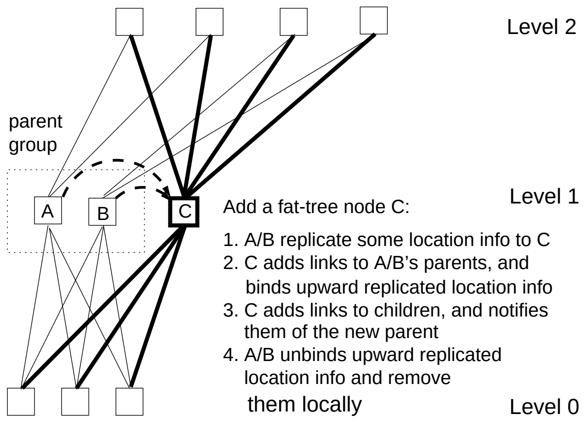
Level 2
parent
group
A
B
C
Level 1
Add a fat-tree node C:
1. A/B replicate some location info to C
2. C adds links to A/B's parents, and binds upward replicated location info
3. C adds links to children, and notifies them of the new parent
4. A/B unbinds upward replicated location info and remove them locally
Level 0

Figure 5: Adding a fat-tree node C to offload node B. The addition effectively offloads all the nodes in the same parent group since the hashing-based routing scheme balances the load amongst the nodes in a parent group.

3.5 Discussion

We next turn to a qualitative discussion of the design of NLS with regard to scalability, performance and availability. An analytical study of the scalability of NLS and a preliminary quantitative evaluation of a prototype implementation of NLS will be presented in the following sections.

We begin with a discussion of NLS's scalability. Our goal is for NLS to scale to a world-wide system that can map on the order of 1 Billion names. Let us first consider the scalability of the root with respect to storage requirements. In the worst case, every name bound in NLS requires a mapping in the logical root of the search tree. Every name binding is represented as a 64-bit hashcode, a bitmap reflecting the forwarding pointers plus some data structure overhead, for a total of no more than 16 bytes. This implies a total storage requirement of 16 GBytes for the logical root, which can be implemented by 16 partitioned resolvers with 1+ GBytes of main memory each. This figure must be multiplied by the replication factor needed for high availability and load distribution. Assuming a minimal replication of four, a minimum of 64 resolvers are needed for the root, which is entirely feasible.

Also, due to NLS's use of prefixes, this figure is likely to be a loose upper bound on the actual storage requirements. Let us assume that each object has a home location (e.g. the primary server of a Web resource) and that the names of all such objects at a given home location have a common prefix, due to naming conventions. Under this assumption, for an individual name to propagate to the logical root requires that the name have associated objects bound in at least two of the subtrees of the logical root. These subtrees represent large areas like large countries or subcontinents. Since a large fraction of Web content is of only local interest, such wide distribution is unlikely.

A second consideration is the scalability of NLS with respect to query load. There are three aspects to query load: the number of resolvers involved in a query, the resulting query processing load on any given node, and the message load on the network. In general, the number of levels that a query ascends upward in the search tree is related to the distance between the originator of the query and the closest replica of the object associated with the name. As a special case, object queries that do not result in a match within the subtree of the root from which the query originated (including queries for non-existent objects) must reach a resolver in the logical root. Due to spatial locality of Web objects and interested clients, many queries can be expected to terminate at low levels of the tree. Moreover, the use of binding caching in the NLS resolvers can be expected to reduce the load on interior nodes of the search tree considerably, since name lookups tend to exhibit spatial and temporal locality.

Finally, we consider availability in the NLS system. Nodes in the fat tree at the global level are normally replicated. When a node fails, the peer resolver automatically contacts one of the replicated resolvers in the same partition. This process is completely transparent to users. Once the failed resolver comes back on-line, it recovers its state from another resolver in the same partition. When a resolver in a spanning tree at the local level fails, its subtree becomes temporarily disconnected. The children of the failed node enter a configuration phase during which they re-attach to the spanning tree. Service for clients and objects in the disconnected subtree is typically disrupted for up to a small number of seconds during this process.

4 Scalability Analysis of Fat-Tree-Based NLS

In this section, we analyze the scalability of the fat-tree-based NLS in terms of storage requirements when objects are replicated. The analysis will then be used to guide the scalability experiments in the next section.

4.1 Scenario

Throughout this analysis, we consider the following scenario. We assume that each leaf resolver of a fat-tree-based NLS has a fixed set of objects permanently bound locally; thus, initially only one prefix per leaf node needs to be bound in the fat-tree. An (initially empty) object cache is co-located with each leaf node, which holds replicated objects. Since the storage requirement for the prefixes is insignificant, the analysis is only concerned with the bindings of objects that are cached away from their home locations.

When the NLS is in operation, there will be a continuous sequence of names is being looked up at each leaf resolver. For each name, if the object found by NLS is neither a local object, nor an object cached in the local object cache, then the object is fetched from its nearest replica and added to the local object cache, and its name bound at the local leaf node and up the fat-tree. The local object caches implement the LRU replacement policy. If an object is replaced, its binding is unbound from the leaf node and up the fat-tree. Therefore, the total number of names bound up from each leaf resolver is bound by the local object cache size. In addition, the upward insertion of a new binding will stop at an internal resolver which already has an forwarding pointer for the same name, as described in Section 3.1.

We assume all resolvers have the same amount of memory. Therefore, if the number of bindings per resolver during the operation of NLS remain the same for all resolvers at all levels of the fat-tree, we say the NLS is scalable in terms of storage requirement.

Intuitively, if there is no overlap among the names of replicated objects bound at different leaf resolvers, then the scalability of a fat-tree based NLS is trivially achieved if the fan-in/fan-out ratio at each node matches the *bindup ratio*, defined as the total number of bindings reaching a node, divided by the number of bindings that need to be propagated further up the tree. Under these conditions, the total number of bindings reaching the next level will diminish by the same factor as the number of resolvers, and thus the number of binds at each resolver remains constant.

In practice, the names of remote objects replicated at different leaf resolvers can overlap, and the number of unique names bound at a certain level of the fat-tree can be smaller than the sum of the bindings bound up from all the sub-fat-trees below that level. The interesting question is then whether there exists a fat-tree with a fan-in/fan-out ratio smaller than the bindup ratio that would still be scalable.

The analysis in this section shows that such a fat-tree does not exist, and that the matching between the bindup ratio and the fan-in/fan-out ratio is required for the fat-tree to be scalable. In addition, we give a simple extension to the binding operation that would guarantee the scalability of the fat-tree when the bindup ratio is higher than the fan-in/fan-out ratio, i.e., when there is not enough spatial locality in the names of replicated objects.

The analysis below assumes a balanced fat-tree topology, as this greatly simplifies the analysis. Due to page limitation, the proofs of the analytical results were omitted.

4.2 Terminologies

We first introduce some useful terminologies about fat-trees. The shape of a fat-tree is uniquely determined by the fan-in and fan-out at each node. Figure 6 shows one step of the recursive construction of a fat-tree. Assume we have M depth-h fat-trees, each with M^h leaves and N^h roots. To construct a depth-$(h+1)$ fat-tree, we add $N * N^h = N^{h+1}$ new roots, grouped into N^h groups of N roots each. Next, we connect the ith root of each of the M depth-h fat-tree to all N roots in the ith group of the new roots. The resulting tree structure is a depth-$(h+1)$ fan-in-M/fan-out-N fat-tree, with M^{h+1} leaves and N^{h+1} roots.

Lemma 1 *A fan-in-M/fan-out-N Fat-tree with M^h leaf nodes has a depth of h, with N^h root nodes. Thus the ratio of the number of roots over the number of leaves is $(\frac{N}{M})^h$.*

Next, we define the following notations in the operaton of a fat-tree-based NLS. X_{local} denotes the number of objects permanently bound at each leaf node. The leaf node where an object is permanently bound is called its *home leaf node*. X_{cache} denotes the size of the object caches attached to each leaf node. X_{lookup} denotes the number of names looked up at each leaf node.

The scalability of the fat-tree-based NLS clearly depends on the spatial locality of the objects being replicated. Here the spatial locality of an object is defined in terms of the distance to its home leaf node in the fat-tree topology, e.g. how many hops away, or the size of the smallest sub-fat-tree that covers the home leaf node and the node where it is replicated. We consider two criteria for the scalability of a fat-tree-based NLS:

Criterion 1 When scaling up the size of a balanced fat-tree with the total number of names in the name space, i.e. keeping X_{local} and X_{cache} constant, does there exist a fat-tree with given fan-in and fan-out that can maintain the same number of bindings per node at the root level?

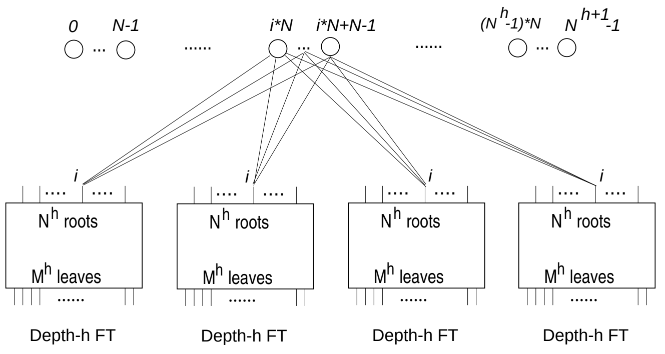

Figure 6: Construction of a depth-(h+1) fan-in-M/fan-out-N fat-tree using M depth-h fan-in-M/fan-out-N fat-trees.

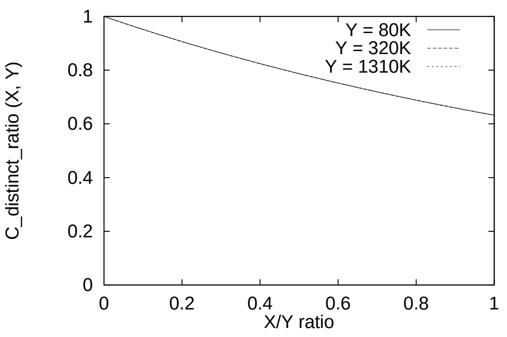

Figure 7: $C_{distinct\_rato}(X, Y)$ for various values of Y.

Criterion 2 Given a fixed number of names $X_{local} \cdot M^h$ in the name space, and a fixed total object cache size $X_{cache} \cdot M^h$, does there exist a fat-tree such that the numbers of bindings per node across all levels are roughly balanced?

Last, we describe two Lemmas that will serve as the basis of our analysis.

Lemma 2 *If we uniformly sample X objects out of a pool of Y distinct objects with replacement, meaning that the pool of source objects stay the same after each sampling, the expected number and the corresponding percentage of distinct objects within the X sampled objects are*

$$C_{distinct\_number}(X, Y) = Y - Y(1 - \frac{1}{Y})^X) \qquad C_{distinct\_ratio}(X, Y) = (Y - Y(1 - \frac{1}{Y})^X)/X$$

Figure 7 plots the $C_{distinct\_rato}(X, Y)$ function for 3 different values of Y, 80K, 320K, and 1310K, as a function of the ratio $\frac{X}{Y}$. The figure shows that the function is only a function of $\frac{X}{Y}$, invariant of the absolute values of X and Y. Furthermore, as $\frac{X}{Y}$ approaches 0, the function value approaches 1, suggesting that there are very few overlapped objects.

The following Lemma simplifies the scalability analysis by removing the complexities involved with cache replacement.

Lemma 3 *If each leaf node in the fat-tree-based NLS looks up X_{lookup} objects with replication, and there are more distinct objects than the object cache size X_{cache}, then the state of the NLS at the end of all lookups will be the same as if each leaf node has only looked up the last X'_{lookup} objects in its original lookup sequence, where there are X_{cache} distinct remote objects in these last X'_{lookup} objects.*

Based on the above Lemma, the scalability analysis can be described in terms of X'_{local}, which is a function solely of X_{cache} and the locality in the replicated objects.

4.3 Exponential-Decay Locality

As we increase the depth of the fat-tree, the ratio of the number of roots over the number of leaves decreases exponentially. We next identify conditions for the spatial locality in the set of remote objects bound at the leaf nodes, such that the resulting number of bindings per node at each level of the tree is constant.

Definition 1 *Exponential-Decay Remote Factor E_f: we recursively define the remote factor E_f to be the percentage of objects being bound at the roots of a depth-k sub-fat-tree, before filtering out repeated ones, that are from the rest of the depth-k sub-fat-trees.*

Remote factor E_f is thus recursively defined. In a depth-h fat-tree, i.e. with M^h leaf nodes, out of the X'_{local} objects looked up at each leaf node, $E_f \cdot X'_{local}$ will be from the rest $(M^h - 1)$ leaf nodes (depth-0 subtrees), and will reach level 1 nodes. Among these $E_f \cdot X'_{local}$ objects, $E_f^2 \cdot X'_{local}$ will be from the other $(M^{h-1} - 1)$ depth-1 subtrees, and will reach level 2, and so on. At the level $(h - 1)$, $E_f^h \cdot X'_{local}$ out of the original X'_{local} objects will be from the rest $M - 1$ depth-$(h - 1)$ subtrees and will reach the root level.

Theorem 1 *In the replication scenario, if the set of objects being replicated at each leaf resolver observes exponential-decay locality with remote factor E_f, the average expected number of distinct objects bound at each root of the depth-h fat-tree is $M \cdot C_{distinct\_number}(X'_{local} \cdot E_f^h \cdot M^{h-1}, X_{local} \cdot M^{h-1})/N^h$.*

Theorem 2 *In the replication experiment, if the objects being replicated at each leaf resolver contains exponential-decay locality with remote factor E_f, the expected number of distinct objects bound at each node at nonroot level k of a depth-h fat-tree is* $M \cdot C_{distinct\_number}(X'_{local} \cdot E_f^k \cdot (1 - E_f) \cdot M^{k-1}, X_{local} \cdot M^{k-1})/N^k + C_{distinct\_number}(X'_{local} \cdot E_f^{k+1} \cdot M^k, X_{local} \cdot M^k \cdot (M^{h-k} - 1))/N^k$.

For NLS to be scalable under Criterion 1, the expected number of bindings per root $M \cdot C_{distinct\_number}(X'_{local} \cdot E_f^h \cdot M^{h-1}, X_{local} \cdot M^{h-1})/N^h = (\frac{M}{N})^h \cdot C_{distinct\_ratio}(X'_{local} \cdot E_f^h, X_{local})$ should not increase with the depth h. Since as h increases, $C_{distinct\_ratio}(X'_{local} \cdot E_f^h, X_{local})$ increases, the only value of E_f that satisfies the above condition is $E_f = \frac{N}{M}$. In fact, assigning $E_f = \frac{N}{M}$ gives a simple bound of X'_{local} to the average load per root:

$$M \cdot C_{distinct\_number}(X'_{local} \cdot E_f^h \cdot M^{h-1}, X_{local} \cdot M^{h-1})/N^h < M \cdot X'_{local} \cdot E_f^h \cdot M^{h-1}/N^h = X'_{local} \tag{1}$$

Similarly, the upper bound on E_f that would make NLS scalable under Criteria 2 is also $\frac{N}{M}$, using which, the expected number of bindings per node at any level of the fat-tree is also bound by X'_{local}.

The relationship between X_{cache} and X'_{local} is as follows:

$$X_{cache} = C_{distinct\_number}(X'_{local} \cdot E_f, X_{local} \cdot (M^h - 1)) \approx X'_{local} \cdot E_f$$

The approximation comes from the fact that the ratio of the first parameter to the second parameter of $C_{distinct\_number}$ function is very close to zero in practice, in which case the $C_{distinct\_number}$ function evaluates to a value very close to its first parameter.

4.4 Controlled Upward Bindings

In practice, we cannot guarantee that there is exponential-decay locality with remote factor bounded by $\frac{N}{M}$ in the set of replicated objects, and thus the above scalability result may not hold. To maintain the scalability of NLS under such conditions, we can simply impose an upper limit on the number of bindings that are allowed to bind upward from each resolver in the NLS. We denote this upper limit as the *bindup threshold*. When the number of bindings bound upward from a given resolver reaches the bindup threshold, new bindings inserted at that resolver (from its child nodes) that would have been bound up further under normal conditions are not bound upward. With this simple mechanism, there is a trivial upper bound on the storage requirement per node.

Lemma 4 *In the replication experiment, if we set the the bindup threshold to be L_{bindup}, then the average expected number of distinct bindings bound at any interior node or any root of a depth-h fat-tree is bound by $\frac{M}{N} \cdot L_{bindup}$.*

Clearly, when the NLS is operated with controlled upward bindings and with insufficient spatial locality in the bound objects, the objects that were not bound all the way up are only visible in a limited topological scope. As a consequence, some lookups from nearby regions that otherwise would have been serviced by these "nearby" bindings may now have to travel into higher levels of the fat-tree to find a match of a forwarding pointer. Thus, these lookups are paying a higher lookup cost and may produce a reference to an object that is not strictly the closest to the client. In essence, controlled upward binding allows NLS to gracefully degrade in the event of insufficient spatial locality in the set of bound objects; insufficient locality merely causes a slight increase in lookup costs and a decline on the location precision, but does not affect NLS's scalability. We will quantify this effect experimentally in Section 5.

5 Experimental Results

We have implemented a prototype of NLS in Java. In this section, we experimentally evaluate the scalability of NLS by measuring its performance for binding, lookup, and migration/replication operations over a name space consisting of up to 1.3 million of URLs collected from 453 organizations in eight universities in the U.S.

5.1 Methodology

We evaluate the scalability of our NLS in terms of storage requirement and lookup cost. To measure the scalability, we run NLS in three configurations, a depth-1 fat-tree with four leaf nodes, a depth-2 fat-tree with 16 leaf nodes, and a depth-3 fat-tree with 64 leaf nodes, respectively, all with fan-in-4 and fan-out-3. For each configuration, there are about 20,000 URLs permanently bound at each leaf node, with the largest configuration consisting of about 1.3 million URLs total.

We then measure the performance of NLS in the following three scenarios:

Bindings only Each leaf resolver binds up the prefixes of local URLs. We measure the storage requirement of NLS when using prefixes.

Lookups After initial bindings of prefixes above, each leaf resolver performs lookups of a pool of URLs. By comparing the lookup costs in the three configurations, we evaluate the scalability of the NLS in terms of lookup costs.

University	# of Org.	Total URLs
MIT	63	163,276
Berkeley	90	165,257
Rice	35	163,498
U. Washington	35	165,770
Harvard	96	162,344
Stanford	83	162,512
U. Texas	24	165,750
UCSD	27	162,509

Table 1: The number of URLs collected and the number of organizations from each of the eight universities.

NLS	Fat-Tree		URL	
Config.	# of Leafs	Depth	Source Universities	Total URLs
Fat-4	4	1	half of MIT	83,627
Fat-16	16	2	MIT and Berkeley	328,533
Fat-64	64	3	All of 8	1310,916

Table 2: Statistics of NLS configurations used in the experiments.

Replication After initial bindinds of prefixes, each leaf resolver performs lookups of a pool of URLs with its object cache turned on. The caches implement the LRU replacement policy. We evaluate the storage requirement and lookups in the presence of caching of replicated objects at each leaf resolver. A lookup of a remote URL at a leaf node, if not found in the cache, results in the fetching and local binding of the associated object. In addition, any object that is evicted from the cache is unbound up the tree.

We perform the above lookup and replication experiments using two sets of URLs with different locality:

Exponential-decay locality The pool of URLs being looked up/replicated have the exponential-decay locality, as define in Section 4.3. NLS is running with no limit on the number of URLs bound up at any node.

Flat locality The pool of URLs being looked up/replicated have the flat locality, i.e. each pool of URLs are sampled from the URLs with their home binding at all the leaf resolvers. The NLS is running with controlled number of URLs bound up at each node.

In summary, with the above measurements, we show that on one hand, if the URLs being looked up/replicated at each leaf node have sufficient spatial locality, i.e., the ratio of the number of URLs bound at each resolver and the number of the URLs that need to be bound up further is about $\frac{M}{N}$, then the NLS scales automatically without having to explicitly control the number of URLs bound up at any node. On the other hand, if there is poor locality, with the flat locality being the worst case, we show that the NLS still scales in terms of storage requirement if it limits the number of URLs being bound upward at each resolver. The tradeoff here is the reduced visibility of the replicated URLs. The visibility is measured as the average number of remote lookups per replicated URL when looked up at every other leaf node in the fat-tree.

All the measurements reported below assume there is no caching of name-to-location bindings in the leaf resolvers. This assumption makes some of the measurements such as the average number of remote queries per lookup the worst case. We will present evaluation results with such a caching mechanism in the final version of the paper.

5.2 URL Traces

To drive our experiments, we use names collected by crawling the Web sites of eight US universities and collecting the URLs published by these organizations. Table 1 shows some statistics on the URLs we collected. The column labeled "Number of Organizations" corresponds to the server domain names that appear in associated URLs from each university. These domain names correspond to the prefixes used when binding the URLs in NLS. Table 2 shows the fat-tree configurations of the NLS name resolvers and the set of names used for each configuration.

5.3 Experimental Setup

Our experimental platform consists of a dual-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 1 GByte of main memory, running True64 UNIX, version 4.0F. Our prototype NLS resolver was implemented in Java, using Compaq's port of the SUN Java 2 SDK, version 1.2.1.

The NLS name resolver instances use Java remote object invocation (RMI) to communicate with each other. However, to allow experimentation with significantly large NLS configurations (up to 175 resolvers), unless otherwise stated, all NLS resolvers were configured to run in a single Java VM. This is largely transparent to the NLS resolver implementation — the Java runtime system automatically reduces communication among the NLS resolvers to local object invocations.

5.4 Performance Results for the Exponential-Decay Locality Case

In this section, we report performance results for the lookup and replication experiments using a pool of URLs with the exponential-decay locality. For each leaf resolver, we generate a pool of 10k URLs with spatial locality that matches the fan-in/fan-out ratio of the fat-tree. Specifically, since the fan-in/fan-out ratio of fat-tree nodes in all three configurations is $\frac{4}{3}$,

Configuration		Home	Cache entries	Avg. number of entries per internal node		
Fat-Tree	Cache size ($\times \frac{M}{N}$)	bindings	per leaf node	Level 1	Level 2	Level 3
Fat-4	bind	19906	0	18/0	n/a	n/a
	1K (1333)	19906	1000	18/1312	n/a	n/a
	2K (2667)	19906	2000	18/2580	n/a	n/a
	4K (5333)	19906	4000	18/4970	n/a	n/a
	8K (10667)	19906	7049	18/8324	n/a	n/a
Fat-16	bind	20521	0	39/0	153/0	n/a
	1K (1333)	20521	1000	39/1320	153/1314	n/a
	2K (2667)	20521	2000	39/2629	153/2575	n/a
	4K (5333)	20521	4000	39/5209	153/4989	n/a
	8K (10667)	20521	7402	39/9497	153/8624	n/a
Fat-64	bind	20473	0	29/0	113/0	453/0
	1K (1333)	20473	1000	29/1279	113/1327	453/1360
	2K (2667)	20473	2000	29/2554	113/2633	453/2633
	4K (5333)	20473	4000	29/5094	113/5239	453/5157
	8K (10667)	20473	7446	29/9418	113/9491	453/9026

Table 3: Number of bindings in the resolvers at various levels of the fat-trees with fan-in-4 and fan-out-3, exponential-decay locality, unlimited bind-up. "m/n" denotes there are m prefixes and n URLs. 10K URLs are looked up or replicated.

Configuration		Avg. remote lookups per node				Avg. remote lookups
Fat-Tree	Cache size	Level 0	Level 1	Level 2	Level 3	per URL
Fat-4	lookup	7500	10000	n/a	n/a	1.50
	1K	7378	9840	n/a	n/a	1.48
	2K	7279	9711	n/a	n/a	1.46
	4K	7132	9515	n/a	n/a	1.43
	8K	7049	9404	n/a	n/a	1.41
Fat-16	lookup	7494	17994	10250	n/a	2.68
	1K	7471	17862	10126	n/a	2.66
	2K	7450	17745	10020	n/a	2.64
	4K	7421	17582	9869	n/a	2.62
	8K	7402	17477	9772	n/a	2.60
Fa-64	lookup	7488	21437	19043	10000	3.57
	1K	7478	21300	18906	9913	3.53
	2K	7469	21192	18802	9841	3.46
	4K	7456	21018	18639	9733	3.44
	8K	7446	20906	18530	9661	3.43

Table 4: The average number of resolver queries per object being looked-up at a leaf node with fan-in-4 and fan-out-3, exponential-decay locality, unlimited bind-up. 10K URLs are looked up or replicated at each leaf resolver.

we sample the URLs from all URLs in the universe by first selecting $1/4$ from the local resolver (these URLs are not bound up by the leaf resolver). Among the remaining $3/4$ URLs to be sampled, we sample $1/4$ from the other 3 leaf resolvers that are in the same depth-1 sub-fat-tree. These URLs are only bound up to the roots of the level-1 sub-fat-tree. This recursive sampling process continues until reaching the root level, when $(3/4)^h$ percentage of URLs will be uniformly sampled from the leaf resolvers of the other 3 depth-$(h-1)$ sub-fat-trees. The NLS is operated without limiting the number of URLs bound upward at any resolver.

Binding Performance Our first set of experiments explores the scalability of NLS, in terms of storage requirement and lookup cost, in a simple scenario where names are only bound once, namely at their home site, and then looked up from various locales. That is, NLS is used in this scenario like a conventional naming system (e.g., DNS).

The first row (labeled "bind") under each fat-tree configuration in Table 3 shows the number of bindings in each resolver at various levels of that fat-tree configuration and the associated numbers of names. The measurements reflect the state of the NLS search tree after all names are bound. The numbers show that the total number of binding in levels 1 and above are dramatically lower that those at the leaf level. This is a result of the use of prefixes – because objects are only bound at the home leaf resolver in this experiment, only prefix bindings exist at the interior nodes of the search tree.

Lookup Performance The first row (labeled "lookup") under each fat-tree configuration in Table 4 shows the average number of resolver queries per node at each level during the lookup experiment.

The numbers show that the average number of resolver lookups grows sublinearly with the depth of the fat-tree. In fact, the expected average number of remote lookups per URL looked up at each leaf node is $\sum_{i=1}^{h-1} \frac{M-N}{M} \cdot (\frac{N}{M})^i \cdot 2i + (\frac{N}{M})^h \cdot 2h$. In practice, this is an upper bound since there are repetitions of some URLs from the sampling. With fan-in 4 and fan-out 3, the above formula gives an upper bound of 1.5, 2.63, and 3.47 for a depth-1, a depth-2, and a depth-3 fat-trees, respectively. The average numbers of remote lookups shown in Table 4 conform with these bounds.

Configuration		Home	Cache entries	Avg. number of entries per internal node		
Fat-Tree	Cache size ($\times \frac{M}{N}$)	bindings	per leaf node	Level 1	Level 2	Level 3
Fat-4	bind	19906	0	18/0	n/a	n/a
	1K (1333)	19906	1000	18/1309	n/a	n/a
	2K (2667)	19906	2000	18/2578	n/a	n/a
	4K (5333)	19906	4000	18/4984	n/a	n/a
	8K (10667)	19906	7040	18/8320	n/a	n/a
Fat-16	bind	20521	0	39/0	153/0	n/a
	1K (1333)	20521	1000	39/1318	153/1303	n/a
	2K (2667)	20521	2000	39/2627	153/2575	n/a
	4K (5333)	20521	4000	39/5213	153/5047	n/a
	8K (10667)	20521	8000	39/10235	153/9589	n/a
Fat-64	bind	20473	0	29/0	113/0	453/0
	1K (1333)	20473	1000	29/1316	113/1318	453/1304
	2K (2667)	20473	2000	29/2628	113/2648	453/2615
	4K (5333)	20473	4000	29/5246	113/5226	453/5074
	8K (10667)	20473	8000	29/10444	113/10317	453/9810

Table 5: Number of bindings in the resolvers at various levels of the fat-trees with fan-in-4 and fan-out-3, flat locality, limiting bind-up. "m/n" denotes there are m prefixes and n URLs. 10K URLs are looked up or replicated.

Replication Performance The replication experiment is performed to measure the scalability and performance of NLS when objects are replicated. For each fat-tree configuration, we vary the object cache size per leaf resolver between 1K, 2K, 4K, and 8K. As described in Section 4.1, for each cache size, the number of bindings in the internal resolvers of the NLS is the same as if only the URLs that are left in the caches were ever bound upward.

Table 3 shows the performance results of the replication experiment using the set of URLs with exponential-decay locality. For each fat-tree configuration, each row that is labeled "10K-X" where X ranges from 1K to 8K shows the average number of prefix and URL bindings in the nodes at each level with the leaf resolver cache size set to X. The results show that for each cache size, as we scale up the size of the fat-trees along with the number of URLs in the universe, the number of entries bound at each resolver is bound by $X_{cache} \cdot \frac{M}{N}$, as predicted by Equation 1. We conclude that if the locality in the URLs being replicated coincides with the shape of the fat-tree, then the NLS is scalable in terms of storage without having to explicitly control the number of URLs bound upward from any resolver.

The corresponding number of lookup operations for the above replication experiment are shown in Table 4. For each fat-tree configuration, each row that is labeled "10K-X" where X ranges from 1K to 8K shows the average number of remote lookups at the nodes at each level, with the leaf resolver cache size set to X. The table shows that compared to the number of lookups in the pure lookup experiment, there is a gradual decrease in the number of remote lookups as we increase the cache size. This is because there is some repetition of URLs in the pool of sampled URLs being looked up at each leaf resolver, as predicted by the Sampling Lemma in Section 4.1. A repeated URL is satisfied out of the cache if it has not been evicted from the object cache.

5.5 Performance Results for the Flat Locality Case

To demonstrate the scalability of our NLS when there is insufficient locality in the URLs being looked up/replicated at each leaf resolver, we repeat the above bind/lookup/replication experiments on a set of URLs with flat locality at each leaf resolver. This test case models the worst case scenario in terms of locality in the URLs. Obviously, if we do not control the number of URLs bound upward from each resolver, the storage requirement at each root resolver will increase by a factor of M/N each time the depth of the fat-tree is increased by one. This is because the number of leaf nodes and thus URLs that need to be bound at the roots will increase by a factor of M, while the number of roots only goes up by a factor of N. Thus the NLS is operated with a controlled number of URLs bound upward at each resolver.

Binding Performance The binding results shown in Table 5 are identical to the results when using the test set with exponential-decay locality, since the URLs bound at each leaf resolver are the same.

Lookup Performance The first row (labeled "lookup") under each fat-tree configuration in Table 6 shows the average number of resolver queries per node at each level performed during the lookup experiment. The numbers show that the average number of resolvers grows superlinearly with the depth of the fat-tree. In fact, since the URLs looked up at each leaf resolver is uniformly sampled from the URLs from all leaf resolvers in the fat-tree, the expected average number of remote lookups per URL looked up at each leaf node is $((M-1) \cdot \sum_{i=1}^{h} M^{i-1} \cdot 2i)/M^h$. For fan-in 4 and fan-out 3, the above formula gives expected numbers to be 1.5, 3.38, and 5.34 for a depth-1, a depth-2, and a depth-3 fat-trees, respectively. The average numbers of remote lookups shown in Table 6 conform with these bounds. These results show that if the URLs being looked up have flat locality, the average lookup cost will increase when scaling up the fat-tree and the URL space.

Configuration		Avg. remote lookups per node				Avg. remote lookups
Fat-Tree	Cache size	Level 0	Level 1	Level 2	Level 3	per URL
Fat-4	lookup	7485	9981	n/a	n/a	1.50
	1K	7371	9828	n/a	n/a	1.47
	2K	7272	9696	n/a	n/a	1.45
	4K	7129	9505	n/a	n/a	1.43
	8K	7040	9386	n/a	n/a	1.41
Fat-16	lookup	9374	23128	13642	n/a	3.44
	1K	9346	22954	13471	n/a	3.41
	2K	9321	22796	13320	n/a	3.39
	4K	9281	22546	13076	n/a	3.35
	8K	9236	22286	12824	n/a	3.32
Fat-64	lookup	9841	31388	32589	17796	5.42
	1K	9833	31164	32317	17615	5.38
	2K	9826	30988	32090	17451	5.25
	4K	9816	30688	31704	17177	5.23
	8K	9804	30326	31236	16837	5.22

Table 6: The average number of resolver queries per object being looked-up at a leaf node with fan-in-4 and fan-out-3, flat locality, limiting bind-up. 10K URLs are looked up or replicated.

Replication Performance We repeat the replication experiment in using the sets of URLs with flat locality. The performance results are shown in Table 5. For each fat-tree configuration, each row that is labeled "10K-X" where X ranges from 1K to 8K shows the average number of prefix and URL bindings in the nodes at each level with the leaf resolver cache size set to X. The results show that for each cache size, as we scale up the size of the fat-trees along with the number of URLs in the universe, the number of entries bound at each resolver is bound by $X_{cache} \cdot \frac{M}{N}$, which is the same as in the experiments using URLs with exponential-decay locality and operating NLS without controlling the number of upward bindings. We conclude that even if there is poor locality in the URLs being replicated, the NLS can be made scalable in terms of storage by controlling the number of URLs bound upward from any resolver.

The corresponding numbers of lookup operations for the above replication experiments are shown in Table 6. The numbers are similar to the numbers in the pure lookup experiment. Again, the gradual decrease when increasing the cache size is due to some degree of repetition of URLs in the sampled pool of URLs, as predicted by the Sampling Lemma in Section 4.1.

Reduced Visibility with Controlled-Bindup To measure the reduced visibility due to controlled upward binding, we repeat the replication experiment using 2K URLs with flat locality, varying the number of URLs allowed to bind up at each internal resolver. To measure the visibility of replicated URLs, we perform a lookup of the URLs replicated on Node 0 at leaf nodes of increasing distance in the fat-tree, i.e. leaf Nodes 1-3 of distance 1 which are in the same depth-1 sub-Fat-tree, leaf Nodes 4-15 of distance 2, and leaf Nodes 16-63 of distance 3. We then measure the average number of remote lookups per such URL lookup. Note, however, that for the leaf nodes 16-63, the number of remote lookups of the URLs replicated at Node 0 are not affected by the replication: if an URL has its home binding in any of the 48 nodes, that home binding will be no further than the new binding at Node 0. This is because if an URL has its home binding in any of Nodes 0-15, the lookup from any of Nodes 16-63 has to pass a root node anyway.

Table 7 shows the number of entries bound per node at each level, varying the bindup limits. It shows that as we increase the number of bindups allowed, more URLs are propagated up and higher, until the bindup limit reaches 2.667k. This is because with the limit being 2.667k, all the replicated bindings can reach level 2. Thus the number of bindings at levels 2 and below will not increase when increasing the bindup limit further. The reason the number of bindings at the root level also stops increasing is as follows. The number of URL bindings per node at level 2 is $2k \cdot \frac{4}{3} \cdot \frac{4}{3}$. Among these, only those with home bindings in the rest depth-$(h-1)$ sub-Fat-trees need to bind up to the root level. With flat locality, that ratio is exactly $\frac{M-1}{M} = \frac{3}{4}$. Thus the number of bindings that need to go up at a level 2 node is $2k \cdot \frac{4}{3} = 2.667k$.

Table 8 shows the visibility of the URLs replicated at Node 0. It shows the average number of remote lookups for these URLs averaged over all leaf resolvers decreases as we increase the bindup limit, until reaching 2.667k.

6 Related work

The GLOBE project [21] proposed a location service based on a distributed search tree. Our NLS was inspired by, and its distributed search tree is based on the GLOBE location service. As previously explained, NLS differs from the GLOBE location service in several ways. Moreover, this paper presents a scalability analysis and an experimental evaluation of a prototype implementation of our enhanced design of such a location service.

Several prior works consider issues in replicating Web content in the Internet, and selecting the nearest replica relative to a client HTTP query [3, 11, 12]. NLS offers a general and highly scalable solution to the problem of locating the nearest copy of an object bound to a name.

Bindups	Avg. num. of entries per internal node		
allowed	Level 1	Level 2	Level 3
1.6K	29/2586	113/2122	453/2100
1.8K	29/2586	113/2384	453/2358
2K	29/2586	113/2648	453/2616
2.2K	29/2586	113/2910	453/2873
2.4K	29/2586	113/3159	453/3129
2.667K	29/2586	113/3250	453/3422
3K	29/2586	113/3250	453/3436

Table 7: Number of bindings in the resolvers at various levels of the fat-tree with fan-in-4 and fan-out-3, flat locality, controlled bind-up. Only 2K URLs are being replicated and then looked up. The number of home bindings is 20473 and the number of caches entries per leaf node is 1967.

Bindups	Avg. num. of remote lookups per URL			
allowed	Nodes 1-4	Nodes 5-16	Nodes 17-64	Overall
1.6K	1.47	4.53	5.63	5.16
1.8K	1.47	4.37	5.60	5.11
2.0K	1.47	4.20	5.57	5.05
2.2K	1.47	4.06	5.54	5.01
2.4K	1.47	3.90	5.51	4.96
2.667K	1.47	3.83	5.48	4.92
3K	1.47	3.83	5.48	4.92

Table 8: The average number of resolver queries per object being looked-up at a leaf node with fan-in-4 and fan-out-3, flat locality, controlled bind-up.

As discussed earlier, the Internet Domain Name System is a scalable naming system that was not designed as a location service. Although it can be used to allow the resolution of a domain name to the "nearest" host address, this places the burden of locating the host address closest to the client from where the DNS lookup originated on the DNS resolver for the associated domain.

Lampson's Global Naming System (GNS) [14] is another example of a scalable naming system that relies on a hierarchy of name servers that directly corresponds to the structure of the name space. Like DNS, it is a pure naming system and was not designed as a location service.

Cheriton and Mann [8] describe another scalable naming service. Like NLS, their service is based on multiple levels (three in their case), distinguished by different requirements with respect to scalability, reliability and administration. Like DNS and GNS, their service is a pure naming service and relies on a hierarchy of name resolvers that reflects the structure of the name space.

Attribute based and intentional naming systems [5, 1], as well as directory services [18, 19] resolve a set of attributes that describe the properties of an object to the address of an object instance that satisfies the given properties. Thus, these systems support far more powerful queries than NLS. However, this power comes at the expense of scalability and performance. NLS supports only simple name queries and resolves to the nearest replica of the associated object, but it is designed to scale to a worldwide service with billions of objects and a corresponding query load similar to that experienced by DNS.

LDAP [23] is an standardized access protocol for naming and directory services, but does not specify the service itself. It is therefore largely orthogonal to NLS. Active Names [20] are client and/or server provided mobile programs responsible for locating and retrieving named resources on behalf of a client. The location functionality of NLS could be achieved with active names, but NLS's focus on a single purpose affords it greater efficiency and scalability.

7 Conclusion

This paper sketches the design of NLS, a scalable naming and location service, it presents an analysis of NLS's scalability, and it presents results of a performance evaluation based on a prototype implementation in Java. NLS resolves textual names to the nearest of a set of replicated objects associated with that name, and is designed to scale to a world-wide service capable of maintaining in excess of 1 Billion names and objects.

Applications include resolving Web URIs to the nearest cached or replicated object that provides the associated content. The key design goals of NLS are scalability, performance, availability and ease of administration. NLS is based on a distributed search tree, with a fat-tree based topology at the global level and self-configuring spanning trees at the local level. Location information is aggregated for names with a common prefix, and hash codes of names are stored in the interior nodes of the search tree to improve scalability.

The scalability analysis yields bounds on the spatial locality required in the set of name binding for NLS to scale. Analysis also shows that alternatively, NLS can be made to scale independently of the locality in the name binding, at the expense of reduced visibility of bound objects when locality is insufficient. Results of experiments with the prototype implementation confirm the scalability analysis.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, Kiawah Island, SC, Dec. 1999.

[2] Akamai FreeFlow. `http://www.akamai.com`.

[3] Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proceedings of the 12th International Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.

[4] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, Aug. 1998.

[5] M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software—Practice and Experience*, 20(4):403–424, Apr. 1990.

[6] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, ??, ??, 1998.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[8] D. Cheriton and T. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[9] Digital Island. `http://www.digisle.net`.

[10] Dynamai. `http://www.dynamai.com`.

[11] J. Kangasharju, J. W. Roberts, and K. W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proceedings of the 4th Web Caching Workshop*, San Diego, CA, Mar. 1999.

[12] J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proceedings of the IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.

[13] B. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. Technical report, The Bell System Technial Journal, Feb. 1970.

[14] B. Lampson. Designing a global name service. In *Proceedings of Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, Aug. 1986.

[15] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercoming. *IEEE Transactions on Computers*, C-34(10), Oct. 1985.

[16] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[17] P. Mockapetris. Domain names—implementation and specification. Request for Comments 1035, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1987.

[18] J. Reynolds. RFC 1309: Technical overview of directory services using the x.500 protocol, Mar. 1992.

[19] M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proceedings of the 3rd International World Wide Web Conference*, 1995.

[20] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: Flexible location and transport of wide-area resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'99)*, Boulder, CO, Oct. 1999.

[21] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, Jan. 1998.

[22] 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. `http://www.cs.vu.nl/˜steen/globe/`.

[23] M. Wahl and T. Kille. Lightweight Directory Access Protocol V3. RFC 2251, Dec. 1997.

[24] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the SIGCOMM '96 Conference*, Palo Alto, CA, Aug. 1996.