Engineering a Global Resolution Service

by

Edward C. Slottow

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and
Computer Science and Master of Engineering in Electrical
Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1997

MIT-LCS-TR-712

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Engineering a Global Resolution Service

by

Edward C. Slottow

Submitted to the Department of Electrical Engineering
and Computer Science

June 1997

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer
Science and Master of Engineering in Electrical Engineering and
Computer Science

Abstract

As the World Wide Web continues to balloon in size the issue of a robust information infrastructure has become increasingly important. Currently, Web links are based on fragile names that have limited life due to semantic content. Uniform Resource Names (URNs), globally unique, persistent identifiers, are one way to solve this problem. Resolvers map URNs to the resources they identify. We describe a global Resolver Discovery Service (RDS) that determines the appropriate resolver for a URN devoid of location information. The RDS uses a high degree of distribution and replication to achieve scalability, both in the number of URNs and users. We have implemented a subset of the RDS as a proof-of-concept.

Thesis Supervisor: Dr. Karen R. Sollins
Title: Research Scientist, Laboratory for Computer Science

Acknowledgments

I would like to thank my thesis advisor, Dr. Karen Sollins, for supporting and guiding this work. She was a tremendous help by explaining naming issues, especially those surrounding long-lived identifiers, and unraveling the mystery of Massachusetts automobile insurance.

The architecture described here is based on ideas of Lewis Girod. He was invaluable for his insight into system design and also for bouncing ideas off of.

My thanks also go to the rest of the Advanced Network Architecture Group at LCS, including Andrew Parker for working on a large-scale simulation of my ideas, Nimisha Mehta, who shared my office, for trying to complete her thesis at the same time, and Garrett Wollman for restoring files from backup when I accidentally deleted them.

I would especially like to thank Tina Pinto, who proofread my thesis. She alternately motivated me to finish and distracted me from my work.

Thanks to Jason Sachs, who acted as a friend and role-model. He insistently urged me to finish even before I started.

Thanks go to the rest of my friends who occasionally asked how things were going and provided great conversation when I should have been writing.

Finally I would like to acknowledge my parents, Jeff and Joan Slottow, for providing a supportive environment in which to grow up and sending me far away to MIT. That, and bringing home my first computer, an Apple II+, when I was in elementary school.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. DABT63-94-C-0072.

Table of Contents

List Of Figures

Chapter 1
Introduction

Names typically serve two functions: identification and location. As an identifier a name refers to an entity. As an address it indicates how to find that entity. A name also has several properties. Its type defines its syntax and the kind of entity to which it refers. The space of all possible names of a given type is called a *namespace*. A name has a context in which it is meaningful and a lifetime during which it is valid. For example, a filename is only meaningful in the context of a specific directory in a specific file system for as long as the file exists. Finally, a name may be a mnemonic so that it is meaningful to humans and easy to remember. For example, files are often given names that reflect their contents. On the other hand, in general phone numbers give no indication to whom they correspond.

Names are a vital component for expressing relationships among objects in emerging information infrastructures. This linking requires names with infinite life.

One type of name, Uniform Resource Names (URNs), have the needed persistence but require a resolution system to associate them with the named resources.

1.1 Information Infrastructures

Global communications networks such as the Internet provide the means for global information services. The elements of these services, including the file systems, databases, networks, protocols, and linking models, form the underlying infrastructure that determines the functionality and quality of the services rendered by these systems. The linking model, which relies on naming systems, controls how information is related.

One architecture for a global information system, the Information Mesh, has the goal of creating a robust, reliable, long-lasting infrastructure upon which disparate applications may build. Over the last several years, another information system, the World Wide Web, has gained tremendous popularity. The Web is an amalgamation of new and existing network services linked together by hypertext pages.

1.1.1 Information Mesh

The Information Mesh [19] recognizes that applications which are primarily concerned with the distribution and exchange of information are often tied to specific network protocols and storage models. Since the lifetime of information is much longer than that of protocols and storage models, these applications should be independent of their access methods. A global information infrastructure is needed to decouple information from access methods so that the latter may evolve independently from the former. Four distinguishing goals make the Information Mesh well suited as an information infrastructure:

- **Longevity:** The infrastructure and its identifiers should be able to exist indefinitely as the lifetime of information is often measured in terms of generations of humans. The practice of mixing location and mnemonics, relatively short-lived traits, into a name limits its lifetime.

- **Mobility:** Information tends to move over its lifetime due to changes in access methods and movement of humans and institutions.

- **Evolvability:** Since the lifetime of a computer system can be measured in years, there must be a clear path for the infrastructure to evolve with advances in technology. Any part of the infrastructure may need to be replaced or enhanced as new protocols, security models, and computational capacity become available.

- **Resiliency:** A global infrastructure requires the interaction of many components, any of which may fail, as both servers and networks are subject to outages. An infrastructure will become part of the fabric of society and users will depend on it working without knowledge of its existence. The infrastructure needs to handle failure with a minimum amount of disruption in service.

One component that the Information Mesh needs to satisfy these goals is persistent identifiers used for naming objects and linking them together. While the location of an object is mutable, a pure identifier is static. Objects named by pure identifiers can move and computer systems can change without their identifiers becoming invalid.

The Information Mesh uses object identifiers, or *oids*, to name objects. An oid identifies an object without indicating how to find it, so hints are used to resolve an oid into one or more locations. Uniform Resource Names (URNs), persistent, globally unique identifiers, are a type of name that match the Information Mesh requirements for oids well.

1.1.2 The World Wide Web

The World Wide Web is a global information system consisting of multimedia-rich hypertext pages written in the Hypertext Markup Language (HTML) [1]. A link from an HTML page to a network resource is specified by a Uniform Resource Locator (URL). The Web link is a one-way relationship, thus creating a directed graph of Web resources.

The syntax of a URL [2] starts with a scheme that often doubles as a protocol. Following the scheme is a location accessible within that scheme. The location, especially for a resource that is the entry point to a Web site, or home page, is often highly mnemonic. URLs display poor longevity since they contain the location of information, causing a major weakness in the Web as an information infrastructure. URNs are being developed as an alternative to URLs since URNs are persistent even as resources move. URNs are designed to encompass existing namespaces as well as new ones with the needed semantics.

1.2 URL Deficiencies

Assumptions that were once valid have begun to erode, as the Web relies on many components that were not intended to be used as they are now. These assumptions need to be reexamined and components replaced with ones that better satisfy the needs of a global infrastructure. The strain in the area of linking can be separated into two type: names being extended beyond their abilities and links with limited functionality.

Until now commonly used naming schemes have relied on the simplifying assumption of limited scope, either in space (e.g., the name is only valid for a computer or site) or time (i.e., the lifetime of the name is longer than the resource it identifies). The result is that a name often functions as an identifier, a locator, and a mnemonic. In many cases compromises made by giving a name multiple purposes, especially location, justify the reduced complexity of the supporting name system. A global information infrastructure such as the Web cannot make the same simplifying assumptions made by small scale systems.

Usernames are one example of a namespace with limited scope. A username is usually valid only for a single computer or a site. When a username is based on the user's actual name it acts as an identifier and a mnemonic. Email addresses, which incorporate usernames, are an example of a namespace with a limited lifetime. Since an email address specifies the location of a particular inbox, it is valid only as long as the user does not change inboxes. As email becomes commonplace people are likely to change addresses more frequently as they move between schools, jobs, and Internet service providers. Global network services require names that are infinite in time and space, such as email names that are valid everywhere for the life of the people they serve.

URLs join many disparate types of names into one naming system. The `ftp` and `http` schemes, the primary types of URLs, require a server and a locator, which is often the path to a file on the server. The `mailto` scheme indicates that the URL is an email address and the `news` scheme that the URL is a Network News group or article. URLs incorporate names, such as email addresses, host names, and path-names, that have limited scope. The names display poor longevity as they identify a location instead of a resource, causing URL links to be fragile. Web links need to be based on names that have an infinite lifetime. So far, the problem of dangling links has been solved with server redirection. A forwarding pointer is left at the old location pointing to the new one. However forwarding pointers cannot be maintained indefinitely.

Besides links that become invalid over time, the Web's reliance on URLs also leads to poor load distribution. The `http` and `ftp` schemes fail to take into account heavy usage, distribution, and mirroring of sites since their URLs specify the server upon which resources reside. Contrast this with the `news` scheme, which omits the name of the news server. Since this location information is left out of the URL, any news server can be used instead of a specific one. A global information service needs a more flexible way of finding a resource than specifying its exact location to allow for various load balancing techniques.

Host specification also causes problems when an information service needs several Web servers to handle all of the requests. There are a number of ways to distribute load under the current system based on URLs. One is to have a host name resolve into several IP addresses. Another is to put HTML pages on one server and pictures on another. In both cases URNs would be useful as they provide the proper abstraction. Since a URN identifies a resource instead of a location, it can resolve not only to multiple URLs but also to meta-information, such as geographic location. Thus a URN can be used to find the location of the most accessible copy of a resource.

While URLs are excellent at specifying location and acting as a low-level linking mechanism, the long-lasting, global network services that are still in their infancy require a more robust linking model[1]. URNs fulfill the higher level naming needs of these services.

1.3 URN Resolution

Unlike URLs, URNs are designed to be location-independent. To find the location of a resource named by a URN, the URN needs to be *resolved* by a service that knows the mapping between URNs and locations. This service is essentially a global lookup table. By changing a mapping in the table, a resource can move without affecting its URN. While some types of URNs may indicate how to find the appropriate resolution service, Web objects should be named by URNs that are pure identifiers, that is, names with an extremely general syntax giving no indication of how to find resolvers. The less information a name contains the longer it will last and the broader its applicability.

The global scope of URNs requires a global resolution system to enable pure identifiers to be usable. The global resolution system can be divided into two major parts. The first, the resolvers, is a highly distributed set of servers that can resolve URNs. The other part is the Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN. Because the global resolution system will be an integral, highly utilized part of any information infrastructure, it will need to scale well with Internet growth while remaining responsive and available. The ability for it to scale directly correlates with the scalability of the RDS. We have designed one possible RDS and implemented its major features.

1. See [22] for one such linking architecture used in the Information Mesh.

1.4 Summary

Global information infrastructures should require as little maintenance as possible while providing the highest level of service. To this end, the names they use to implement linking should last indefinitely. Names in use now, such as URLs, are short-lived compared to the lifetime of the information they identify due to the fact that they are primarily addresses. Long-lived names should contain as little information as possible. URNs are global, long-lived identifiers that address the Web's linking dilemma. Since URNs may not contain any information, a global resolution system is needed to find a resource associated with a given URN.

1.5 Coming Attractions

While a URN resolution system may be the largest name resolution system ever attempted, it is certainly not the first. Previous people have tackled the issues surrounding naming and scale. Chapter 2 reviews several existing resolution systems along with Harvest, an information discovery system that deals with scaling issues. Chapter 3 explorers URNs in-depth, including a discussion of their functional requirements and syntax. It also gives a framework for the resolution process with special attention to the RDS. The optimal architecture for an RDS will scale well and provide highly reliable service. The architecture of one RDS meant for pure identifiers is presented in Chapter 4 followed by the specifics of its design in Chapter 5. We have implemented a subset of the design to develop algorithms and demonstrate feasibility. This prototype is described in Chapter 6. Finally, Chapter 7 concludes with a discussion of the current design and the direction it should take in the future.

Chapter 2

Related Work

Previous name resolution systems have provided valuable insight into the creation of a global service, including necessary requirements, design features, and operational experience. Grapevine and its successor, Clearinghouse, served primarily as messaging systems. Email addresses were resolved into inbox locations, allowing users to move and have multiple inboxes. Lampson highlighted the differences in requirements between a name service and a more general database in the description of his global name service. The Domain Name System serves as an example of a large, distributed service with extensive operational experience. Finally, Harvest, while not a naming system, deals with scaling issues and information management relevant to modern Internet services.

2.1 Grapevine

Grapevine [3] was a project undertaken at Xerox's Palo Alto Research Center in the early 1980's to explore distributed systems and provide electronic mail service. Along with messaging, it supported the naming of various entities, such as users, computers, and printers; authentication; and resource location. Grapevine had many admirable design features. It was both distributed and replicated as a way to handle scaling, load, and failure. It was also designed to have a decentralized administration.

Grapevine's name service was structured as independently administered and distributed *registries* acting as a unified *registration database*. A registry contained an arbitrary collection of names, usually organized by location or function. For example, a registry could contain the names of all the people at a site or all the names related to a specific service, such as mail delivery. The organization of the registration database into registries naturally led to a namespace with two levels: a registry and names within the registry.

Each registry was replicated on several registration servers, with each server containing multiple registries. This use of replication masked the failure of individual servers while the distribution insured that information was stored on servers close to the clients that needed it.

Due to its distributed nature, Grapevine used a weak consistency model of updating registry replicas. In this model, each replica would eventually receive an update, although there was no guarantee when. With weak consistency the registration database did not need to be globally locked while an update was occurring, so it was still available to users. Consequently a client could receive inconsistent responses during an update by querying different servers. Weak consistency was preferable to a stricter version because an update may have taken several minutes or more to propagate, during which the name service should have remained available.

To perform an update to a registry, a user contacted one of the servers that contained it. The server then sent the update to all other servers replicating the registry through Grapevine's messaging service. It could do this because every server had a registry that contained the structure of the entire database, essentially a registry of registries. If a recipient of an update was temporarily unavailable, the update remained in its inbox until it became active again.

The designers of Grapevine recognized that the registration database allowed for the separation of naming from location, an important feature in a large information system. The destination of an email message was specified by the recipient's name and registry instead of a message server. The registration database was used to find the recipient's primary message server and also allowed for secondary ones in case the primary one was down. This distinction between identification and location allowed people to change inboxes as long of they stayed within a registry.

Grapevine was widely used by Xerox's research community for a number of years, from which real world experience can be gained. In 1983 there were 17 servers containing 5900 names [18]. Scaling of the system was limited by the fact that every server contained the structure of the entire database. A larger system would have needed more space on each server and would have led to a higher frequency of structural changes. Operational practice was to create more registries as the number of names increased instead of adding to the existing ones. Increasing the size of the registries would have required larger servers, which were quite small by today's standards. Creating more increased the possibility that an individual would move between registries. Clearinghouse [17], the commercial successor to Grapevine, had better scaling characteristics at the expense of adding a level to the naming hierarchy. This creeping partitioning of the namespace reduces the longevity of the names.

2.2 Lampson's Global Name Service

Lampson used Grapevine and Clearinghouse as the basis for the design of another global name service [12]. This service was meant to be large enough to encompass all computers and users in the world. The idea behind it was to organize all names into a global file system-like structure. A name's properties, such as passwords and mailboxes, could be determined by looking in its directory. While the client interface appeared similar to that of a file system, Lampson emphasized that the two were different. A file system needs faster access times and is smaller than a name service. A name service also differs from a general database because the set of names and their properties changes slowly. The requirements for Lampson's service form a good starting point for any widely deployed name service: arbitrarily large size, long life, high availability, fault isolation, and tolerance of mistrust.

Lampson satisfied these requirements with a hierarchical namespace similar to a file system. The value of a path to a node was its child or children. The leaves were the values of name properties. For example, the value of the (fictitious) directory ANSI/MIT/LCS/Slottow/Mailboxes is the child or children of this node, the mailboxes that receive Slottow's email. ANSI/MIT/LCS/Slottow/ names all properties belonging to Slottow at MIT's Laboratory for Computer Science.

Like Grapevine, Lampson's name service was distributed and replicated. Replication occurred at the subtree level, a finer granularity than a registry. For example, LCS could allocate multiple servers to hold directory copies of the ANSI/MIT/LCS subtree. Updates originated at directory copies and were propagated by *sweeps*.

A sweep collected the changes made to each directory copy and then distributed the combined changes back to each of them. The directory copies were organized in a ring topology, and if one failed during a sweep the update could be lost. After a failure the copies needed to be reorganized into a new ring manually.

This name service used hierarchy as its chief method of dealing with growth. The deeper the hierarchy in a naming scheme the less likely names will remain constant as the people will cross partition boundaries more often. The names in this system were aligned with the topology of the servers. This system may be good for routing mail or authenticating users but failed to prove its ability to meet the combined goals of arbitrarily large size and long-lived names, both important requirements for a URN system.

2.3 The Domain Name System

The Domain Name System (DNS) [14], the standard way for resolving the name of any host connected to the Internet, is the most widely deployed resolution system. Due to its extensive field testing and numerous implementations, DNS has grown to be a mature, successful resolution system that has demonstrated what does and does not work. The design goals for DNS differ from those of Grapevine and Lampson's service, perhaps due to DNS's application and the experience its creators had with existing networks and host naming. Among the goals are:

- **Consistency:** The namespace should be consistent and free from network specifics.
- **Size:** DNS replaced a previous resolution system that could not handle the growing number of Internet hosts. Given the frequency of updates, the host name needed to be stored and maintained in a distributed manner.

- **Generality:** Implementing a global service is costly so DNS is not restricted to one type of data, such as a host's IP address.
- **Protocol Independence:** While the Internet protocols are most prevalent, a disjoint set of information for each class of network should exist for a name so that it can be resolved to network specific information.

The Domain Name System is a generalized resolution system for hierarchical names primarily used to identify Internet hosts. The DNS name servers are distributed and maintained along administrative boundaries, so names typically reflect organizational structure. Coupling the topology of the name servers to the structure of the namespace allows for a simple delegation of naming authority and maintenance responsibility. DNS is composed of a tree of name servers. Each name server has authority over a *zone*, a logical portion of the namespace replicated on several servers. Each zone contains a list of names and their attributes. If the name is a host, then it will have an address attribute. If it is another zone, then its attribute will be that zone's name server.

At the top of the tree is the set of servers that contain the root zone. The names in the root zone, such as EDU, COM, and GOV, are commonly known as the top-level domains. A name server at the next level down, say for the EDU domain, authoritatively knows all of the name servers that serve the EDU zone. It also has a list of zones in the EDU domain and their name servers. For example, the root zone contains the EDU domain name, which contains the MIT domain name, and so on. The name servers for the MIT domain contain addresses of MIT's hosts, such as Web, the MIT Web server, and information about subzones, such as the LCS domain. A name is the concatenation of delegation, so the name of MIT's Web server is web.mit.edu.

The method of replication employed by DNS is simpler than that of Grapevine and Lampson's global name service. Those services were administered remotely by a few trained people. With widely distributed replicas, this required more complex update protocols and structural overhead. In the DNS model, each server is administered locally by its maintainer. An update to the names in a zone can only be performed on a master server, which reads the name information out of a file. The other zone replicas are secondary servers that either read their information from the same file or use a simple protocol to poll the master server for updates.

To discover the attributes of a name, an application queries a resolver. Often the resolver resides on the same computer, but it need not. It knows the address of at least one name server that it can query, possibly receiving referrals to other name servers. Through a repetition of queries the resolver will encounter a name server that can resolve the name.

Like Lampson's global name service, DNS names are strictly partitioned into a hierarchy. However, a computer moves less frequently than information, and when it does the disruption is not as great.

2.4 Harvest

The Domain Name System achieves a high degree of scalability through massive distribution and localized structural data. No DNS server knows about the entire tree structure, nor does it need to. In contrast, Grapevine did not scale well since every server needed to know the structure of the entire name system. The Internet's rapid growth, most recently fueled by the popularity of the Web, requires new ways of coping with the volume of data and users. While not a name resolution service, Harvest [4] was designed for the Internet's current size. Its main purpose is the gathering and indexing of the vast quantities of available information so that topics can be searched for quickly.

While Harvest employs new techniques for managing large amounts of information, it also recognizes a large user base to whom it offers service. A single, or even several, servers is no longer sufficient to quickly handle user requests. Harvest has four main components that index the information and make it available in a scalable fashion:

- **Gatherer:** Indexes information from content providers.
- **Broker:** Provides a query interface to an index generated from one or more Gatherers.
- **Replicator:** Replicates Brokers to handle user demand.
- **Object Cache:** Caches network objects to reduce network and server loads while increasing responsiveness.

The Replicator [8] is the most relevant aspect of Harvest for global resolution systems. Both the Replicator and a URN resolution system have similar needs, namely a large amount of data accessed frequently by a large user base.

The Replicator maintains a weakly consistent replicated directory tree of files. Replicas are divided into groups in which files are flooded by the *flood-d* program. The flooding pattern is determined by the logical network topology, that is the bandwidth between members of each group. Every so often the network topology is rediscovered and the flooding paths reconfigured automatically, requiring no human intervention when servers fail.

A group maintains weak consistency using an anti-entropy flooding algorithm. Occasionally, a replica starts a session to synchronize its information with that of a logical neighbor, the result being that eventually every replica will have the same information. This update method does not scale to many replicas, so they are divided up into groups of limited size. The groups are connected to each other and exchange updates.

2.5 Summary

Grapevine and Lampson's global name service articulated the needs of a global resolution system and contributed several ideas, primarily the importance of replication and distribution, but also viable consistency and update methods. All of the name services mentioned achieved scaling with a hierarchical namespace reflecting the system's structure. This sort of design inhibits the longevity of names, a major impediment to using any of these as a URN resolution system.

The Domain Name System serves as an example of a highly successful name resolution system. While it demonstrates proven design choices, it also benefits from extensive operational experience in areas not covered by its technical details. Among these are the administration of a global name database, legal issues surrounding name assignment, and demands by those wishing to own names.

Harvest's Replicator provides a means of dealing with user demands. As a core component of an information infrastructure the URN resolution system will be one of the most utilized Internet services. The Replicator demonstrates how to handle this responsibility.

Chapter 3

Uniform Resource Names

While much has been said about the advantages of URNs, little detail has been given about them. The functional requirements placed upon URNs make them appropriate for an information infrastructure. These requirements would be meaningless without a practical resolution system that can fulfill them. It is expected that there will be different types of URNs, differentiated by their syntax. Clients should be able to resolve existing and future types of URNs without modification. The resolution framework lays out a system that has the potential to meet the URN requirements while isolating the client from the specifics of a URN scheme.

3.1 Functional Requirements

[20] defines the function of URNs:

> The purpose or function of a URN is to provide a globally unique, persistent identifier used for recognition, for access to characteristics of the resource or for access to the resource itself.

The functional requirements for URNs make them appropriate for use in a global information architecture:

- **Global scope and uniqueness:** A URN has the same meaning everywhere so it must be unique and cannot be assigned to multiple resources.
- **Persistence:** A URN will last indefinitely even if the resource it names does not. Therefore a URN should never be reused.
- **Scalability:** URNs need to be appropriate for naming any resource now and in the future.
- **Legacy support:** URNs should accommodate existing namespaces.

While not all URN schemes will satisfy all of these requirements, the more they meet the more effective they will be.

3.2 Resolution Requirements

Ultimately a URN needs to be translated by a resolution system that will determine how well it meets its purpose. The requirements for a resolution system are based on three assumptions [21]:

- **Longevity:** While URNs are persistent identifiers, the resources they name will change network location. Furthermore, the infrastructure on which URNs rely, such as network layers, protocols, file systems, and even the resolution system itself, will evolve.
- **Delegation:** Delegation is an important method of distributing the responsibility for assigning URNs to those most interested. Both naming authority and resolution responsibility will be delegated at multiple levels.

- **Isolation:** An authority has control over name assignment, resolution, and policy within its namespace independent of others. Furthermore, they are free to choose resolution services.

The three major requirements for a resolution system, evolution, usability, and security, support these assumptions.

3.2.1 Evolution

The first requirement, evolution, ensures that the resolution system will promote the persistence of URNs. While it is hard to predict how the Internet will change, it is certain that it will. A URN resolution system needs to be long-lived, so it will have to evolve with the changes. Resolution will likely evolve in a number of ways. One is that various types of URNs will have varying resolution mechanisms. The system will need to be able to handle the addition of new URN schemes, either through generality or by creating specific components responsible for the new types. Existing resolution systems will also evolve in multiple ways, for example by changing the resolution algorithms or the communication protocols. In fact, any part may need to evolve with changes in demand and from operational experience. Lastly, usage of URNs will evolve from narrowly defined types based on existing namespaces to pure identifiers.

3.2.2 Usability

Publishers, users, and maintainers make different demands on the resolution system. Publishers, the providers of content, need URNs to be resolved quickly and, more importantly, correctly. They need to create and manage URNs cheaply and easily while making them globally available. Publishers will also want resolution to consider user characteristics, such as geographic location and network bandwidth.

While they will provide long-term information about their resources, publishers will also need to provide short-term information to cover short-lived changes in service and delays in changing the long-lived information.

The priority for the users is that URNs be resolved quickly and that the interface be simple. A user may need to evaluate resolution information to determine which part is best suited to her needs, for example by picking from resolutions based on authenticity. What these needs are and how to express them must be easy to understand. The user will want to interact with a friendlier, albeit shorter-lived, name that is easier to type and remember than a URN may be.

Finally, management of the resolution system, should require as few network resources as possible. Pricing will support the system and prevent abuses to it by putting negative pressure on the registration of extraneous namespaces. Configuration needs to be simple enough so that it is done properly.

3.2.3 Security

Since URNs will be a basic part of the information infrastructure, users depending on the integrity of the resolution system will have little tolerance for security violations. There are several ways that the system could be attacked, the simplest being denial-of-service. More subtle attacks could be made by servers that are part of the system. These servers could potentially provide incorrect information either to clients or to other servers. Thus, there must be a verifiable, authoritative version of resolution information. Security violations can be combatted through widespread distribution and replication, minimizing the reliance on a single server.

3.3 Resolution Framework

Different URN schemes will need different methods of resolution that all fit within the above requirements. However, client software cannot be modified every time a new URN scheme is created nor should it be required to know specific resolution methods. A general framework provides an abstraction for the resolution process[1]. The framework uses *hints* to indicate how to resolve a URN. Although a hint will usually be correct, it is not guaranteed to be. Section 5.3 discusses hints in more detail. The act of resolution is simply trying to find the correct hint, progressing from fast, unreliable sources to slow, reliable ones.

There are a number of different components essential to a robust URN resolution system. Publishers need computers that serve Web pages with URNs and hints. Clients resolve URNs through local proxies to abstract the specifics of the resolution process. The proxy will likely consult a Resolver Discovery Service which will find the appropriate resolver. Naming authorities maintain the resolvers that map URNs into other information, such as URLs.

Figure 1 shows the steps for resolving URNs within the framework. The user begins by obtaining a URN from some source. In the figure Web Server A returns a page with two links specified as URNs: URN1 and URN2. Since the server has a hint for URN1 it can either include it with the page or the client can request it. Consulting the source is the primary way to resolve a URN. Sources will want to cache the hints to provide the best performance for their users.

If the source has no hints or stale ones for a URN, then the client has to use the fall back resolution method, as in the case when the user wants to view URN2.

---

1. This framework is more extensive than the one in [21], which only considers RDS's and resolvers.

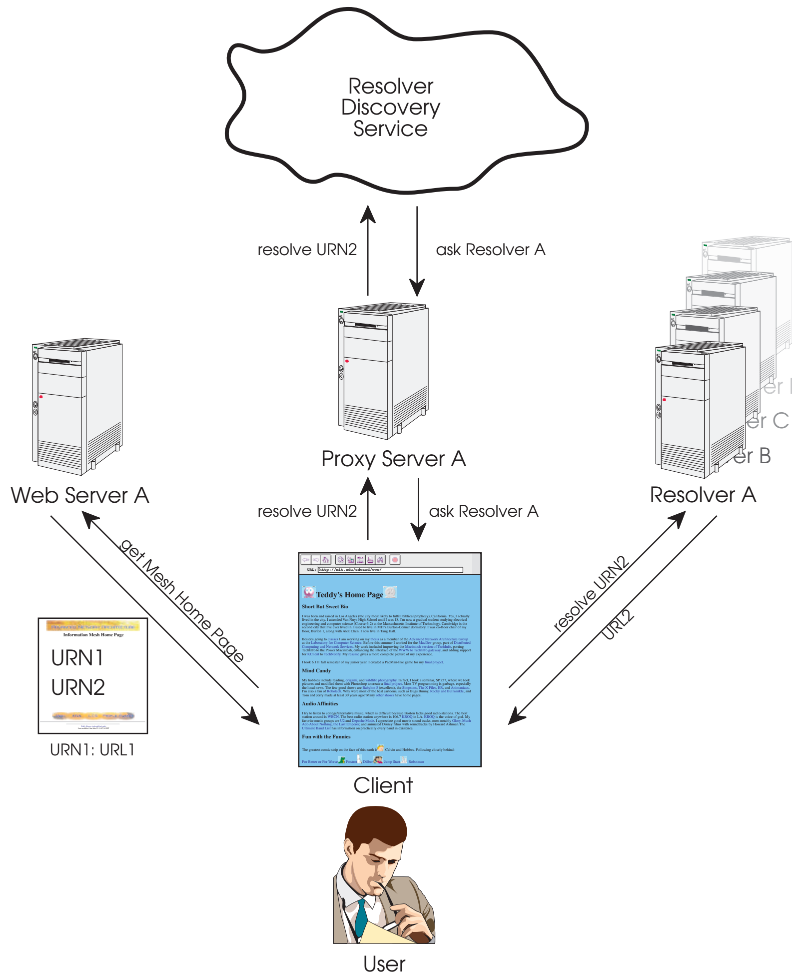


Figure 1. URN Resolution Steps

1. The source (Web Server A) of the URN is consulted.
2. If source cannot provide any hints , the c lient asks a proxy server for hints .
3. The proxy consults a Resolver Discovery Service .
4. The c lient consults one or more resolvers .

The c lient needs to contact its authoritative resolver . Publishers of URNs maintain the resources needed to resolve their names , allowing them maximum control over name assignment, quality of service , and cost. Since a URN ma y not give any indication of how to find its resolver , the global Resolver Discovery Service (RDS) contains enough information to map a URN to one or more resolvers . Different types of URNs ha ve different needs and w ays of mapping to resolvers . URNs based on arbitrary strings ma y need an RDS that is one global lookup table . However , the RDS for a strictly hierarchical URN scheme that can parse the URN may only need the mapping between delegation and resolvers.

A proxy server isolates clients from the RDS. There may be many RDS's that implement the semantics of the URN schemes. While we cannot expect client software to be updated for each new RDS, system administrators can keep the proxy servers updated. The proxies are extendable by adding resolution code made available for each RDS. In the figure, the client asks the local proxy server to resolve URN2. The proxy asks the appropriate RDS, which returns the hint that Resolver A can authoritatively resolve URN2. Resolver A resolver the URN to URL2. Alternatively the owner of Resolver A could have delegated authority over URN2 to another resolver such as Resolver B. Then Resolver A would return a hint which could be the name of another resolver (in this case Resolver B) or even another URN.

An RDS and a resolver serve similar functions; they both return hints for URNs. An RDS is a large scale, general service meant to redirect a resolution to a resolver. A resolver is much smaller and often can resolver a URN to a location. A resolver can also be more specialized to suite an individual publisher's resolution needs. Either one can return hints that redirect or resolve a URN.

The proxy server performs several secondary tasks. One is to provide a mechanism for site-wide hints. A site may choose to mirror a popular resource locally. The proxy serves site specific hints for that resource along with those from the RDS. By serving site-wide hints, the proxy can also be used to create site-wide URNs and even entire schemes without client configuration. For example, a company may wish to assign a name for an internal document that should not be exported outside its intranet. The proxy also acts as a hint cache. Caching as many URNs as possible will reduce the resolution time, especially for popular URNs.

This framework allows for resolution systems to potentially meet the given requirements. It supports evolution by accommodating multiple URN types without client knowledge. Since all interaction with the RDS is through a proxy, the RDS can evolve or even be replaced. By using the same protocol to communicate with all potential sources of hints, including proxies and resolvers, the resolution steps can be very general. In terms of usability, most URNs can be resolved quickly by caching at the source. Providers can make them globally available by registering with an RDS and running a resolver. This framework omits discussion of security. Individual system designers are responsible for meeting the security and other requirements.

3.4 URN Syntax

To accommodate new and existing namespaces a URN has an extremely general syntax [13]:

`urn:`*NID*`:`*NSS*

The prefix "`urn:`" identifies a string as a URN. Following the prefix is the case-insensitive NID (Namespace Identifier), which defines the type of URN (its scheme). The NSS (Namespace Specific String) is a globally unique name within an NID namespace. Each NID has a published syntax, resolution mechanism, and definition of lexical equivalence. In this manner, different types of URNs can be created that support many existing namespaces, and new namespaces can be created for various applications.

For example, a URN scheme based on ISBN numbers could have the NID "`ISBN`" while another based on pure identifiers for resources could have the hypothetical NID "`PRI`". In the ISBN namespace, the NSS's would be actual ISBN numbers which have a well published, hierarchical syntax. In the PRI namespace, the structure, if any, may be kept hidden to maintain the illusion of a flat space.

URNs have a canonical form that must be used for all interchange and transport. Two URNs are considered lexically equivalent if, after normalizing their cases, they begin with "`urn:`", have the same NID, and have the same NSS. Namespaces can have a more restrictive definition of lexical equivalence for the NSS. More details about the syntax, including a more precise definition of the syntax, %-encoding, reserved and excluded characters, and determining lexical equivalence can be found in [13].

3.5 Summary

URNs have properties that make them fit well within a global information infrastructure. The most important way in which URNs are suitable for this is through the ability to separate identification from location while still allowing for existing namespaces. The resolution framework provides a model for different types of URNs to be location independent. Location independence will promote the creation of long-lived identifiers ideally suited for a wide range of services, including persistent names for Web resources, lifelong email addresses, and public key systems.

Chapter 4

Resolver Discovery Service

One type of URN that has the potential to be long-lived by exhibiting a high degree of mobility is the pure resource identifier (PRI). Resolving one within the framework requires a Resolver Discovery Service that is scalable, responsive, and reliable. The RDS needs to separate the delegation of naming authority from resolution responsibility so that the naming authorities are independent and the names mobile.

4.1 Pure Resource Identifiers

Pure resource identifiers are names specifically created to last indefinitely by omitting location information, meaningful words, and anything else that may change over time. Thus, they act solely as identifiers. PRIs are suited to any application that would benefit from globally unique, persistent names. Possible uses include naming Web resources, creating permanent email names, and serving as identifiers in a public key distribution mechanism. The lack of mnemonics in PRIs frees them from a number of problems. First, by omitting meaningful words they do not have to change when a resource moves, which would have invalidated the meaning. Omitting mnemonics also avoids legal problems with conflicts over trademarks. Finally, character set and language issues that afflict other namespaces become moot.

Another factor that limits mobility is the partitioning of a namespace by the use of hierarchy that is coupled with its management and resolution, as once a name is created it is often locked into a specific delegation of authority and resolution, delineated by its syntax. This problem plagues Grapevine, Clearinghouse, Lampson's global name service, and DNS. Over time resources move, and their owners will need the ability to change the entity responsible for resolving their names.

PRIs utilize hierarchy to distribute naming authority and ensure global uniqueness without limiting their mobility. This can be accomplished with a very general syntax for the NSS:

<*subspace name*><*resource name*>

A PRI has a *subspace* name followed by the name of a resource within that subspace, both being zero or more characters. A subspace is a subset of names that share a common administrative authority denoted by a common prefix. Since there are no separator characters between components of the name, this syntax allows for a fluid delegation hierarchy. The subspace name may be a concatenation of administrative prefixes. Likewise, the resource name may begin with another concatenation of prefixes, so the entire name can be deconstructed into a hierarchy of delegation authority. Where the subspace name ends and the resource name begins can change over time by adjusting which prefixes form the subspace, giving PRIs greater mobility.

With this syntax there is no way to tell the structure of a PRI without resolving it. For example, say MIT owns the subspace "abc" and gives a portion, "def", to

LCS, which creates the name “1”. The full name of the resource is “abcdef1”. While not evident from the syntax, this name’s subspace is “abc” or “abcdef”. The latter example is a concatenation of delegation that forms a new, more restricted subspace. In this form MIT has lost control over then names with this prefix.

The task of the RDS is to determine which part of a PRI specifies the subspace and return the corresponding hints. This strategy will work for any type of name that goes from more general to more specific, so it will work for any URN that can be put in this canonical form. If resolution proceeds to a resolver or another RDS, one of them may apply a transformation first to put the URN in a form it needs, or it may know the syntax of the URN.

4.2 RDS Requirements

The RDS has a number of requirements that, unsurprisingly, are similar to those for a resolution system.

- **Compliance:** The RDS should fit with the requirements for a resolution system and also realize the functional requirements for URNs.
- **Scalability:** The Internet has been growing at an exponential rate. The target number of PRIs is on the order of 10^9 and the number of subspaces 10^7. Along with the number of subspaces, the RDS needs to scale with the number of users. There are about 40 million Web users over the age of 16 in the U.S. and Canada now [16] but that number is increasing as the Internet becomes ubiquitous. While actual numbers are impossible to determine, 10^9 worldwide resolution requests per day is not unreasonable. While most of the requests will be handled by caching either locally or at the source, the demands on the RDS will still be high and growing. The

service may need uncommon scaling techniques such as replicating based on usage and taking into account network bandwidth.

- **Evolution:** The RDS needs to provide an evolutionary path with little disruption to the user. Any part of the RDS may need to change, including hint content, security methods, and internal protocols.

- **Automation:** Administration should be minimal. A heavily used system needs to have the right resources available in the right places. The RDS will be large enough to make excessive oversight unmanageable. Instead the entire system should be largely self-maintaining.

- **Robustness:** The service needs a high degree of reliability because people will take for granted that it works. Not only should it mask failure, it should also be able to recover automatically when possible.

- **Speed:** While the RDS is the fall back method, it still needs to be quick. Name resolution adds an extra layer of communication that does not exist today, and users will not accept it if it adds a noticeable amount of time to accessing a resource.

- **Delegation:** Authority and resolution should be easy to delegate so that a publisher has a wide range of choices of service providers. These providers would offer to give a publisher a subspace and manage administrative duties. However, once the publisher has authority over a space, it should be free to leave its provider.

- **Access:** Ownership of a subspace should be accessible to everyone, from individuals to large organizations. The requirements to become a naming authority should be minimal to allow for the greatest opportunity.

4.3 Partitioning and Delegation

There are two extreme forms that the RDS can take based on the chain of resolution responsibility: deep, which has a high degree of partitioning, and flat, which has none. Deep is similar to other naming systems, such as DNS, where delegation of resolution responsibility mirrors the structure of the name. A flat model does not leverage this structure, instead being a single collection of names.

In the deep model, top level subspaces are registered with the root authority. The top level subspaces delegate parts of their spaces to second level authorities, and so on. Not only do they delegate naming authority, they also pass off resolution responsibility. For example, MIT gets a subspace from the root authority, and LCS gets space from MIT. Whenever the root RDS gets a query for one of LCS's PRIs, it finds that the PRI is in MIT's subspace and returns hints pointing to MIT's resolver. When MIT receives a query for the PRI, it determines that it belongs to the subspace delegated to LCS and returns hints for LCS's resolvers.

The main disadvantage of this model is that the hierarchy forces a strict partitioning of the namespace, restricting the mobility of the names. This problem exists as long as the resolution is tied to the delegation of authority. However, PRIs do not express their hierarchy so the partitioning is not fixed, regardless of how they were created. Over time the majority of subspaces may flatten as they move, leading to a flat RDS.

The opposite extreme from a deep RDS is a flat one in which a PRI is treated as a single subspace name followed by a resource name. Treating the naming scheme as flat promotes mobility by minimizing partitioning even if it is not flat. Authority may still be delegated, but all resolution responsibility remains with a central entity. For example, MIT's subspace would be registered with the central entity. MIT has the ability to give LCS a subspace, but once it does so, it loses control of the subspace, as it too is registered with the central entity.

It is probable that the RDS will be a combination of the flat and deep model. When the structure of the name can be exploited, some may prefer to take advantage of it while others will prefer to rely on the root. The strength of a PRI is that it allows flexibility, letting publishers choose the resolution model and move between the two without changing names. The flat model is the more general, and the more difficult of the two to implement. We expect it to be more prevalent, so it is the one that the RDS must be prepared to accommodate.

4.4 Possible Architectures

The architecture of the RDS needs to be suitable for an extremely large, flat namespace. One, known as NAPTR [7], that was proposed to the URN Working Group cannot meet this need. However, our RDS was created to meet it and the requirements.

4.4.1 NAPTR

Since DNS is a mature, widely used resolution system, extending it to handle URNs is easier than creating a new RDS from scratch. NAPTR proposes a way to use DNS to resolve new URNs while grandfathering in existing ones through a series of queries and rewrite rules. It operates by extracting a part of a URN and looking up its DNS NAPTR record, which contains two transformation fields. One is a new string to lookup. The other is a regular expression that when applied to the original URN will extract some part to lookup. To resolve a URN, a client performs a series of transformations and queries until a failure occurs or a terminal NAPTR record is encountered pointing to a resolution service.

NAPTR works best for URNs with a clearly delineated structure so that regular expressions can extract pieces from it. Unfortunately, regular expressions are hard to write and are confusing, which could lead to problems if they are incorrect. Furthermore, regular expressions are not powerful enough to express transformations that may be needed for grandfathered namespaces, such as reversing the elements of a name with an arbitrary number of divisions, nor are they capable of deconstructing PRIs. By relying on DNS, NAPTR inherits its problems, including names with limited mobility. NAPTR is a quick solution to a problem that requires careful thought. Any popular RDS will exist for a long time and should be well suited for its task.

4.4.2 Our Architecture

We created an RDS designed to support the longevity of URNs, especially PRIs, in a scalable, reliable manner. The basic architecture (see Figure 2) consists of two main components, the central and distributed databases. The central database contains the authoritative version of all subspace hints. It has no interaction with clients that wish to resolve URNS, just those wishing to manage their subspaces, so the demands on it are less stringent than on the RDS as a whole. It needs this isolation because, as the official source of all hints, it has to be protected from attack. Also, its availability does not need to be as high, nor does it have the ability to handle queries from all URN users. Therefore the central database can be created from a fairly conventional solution, possibly even shrink-wrapped database software.

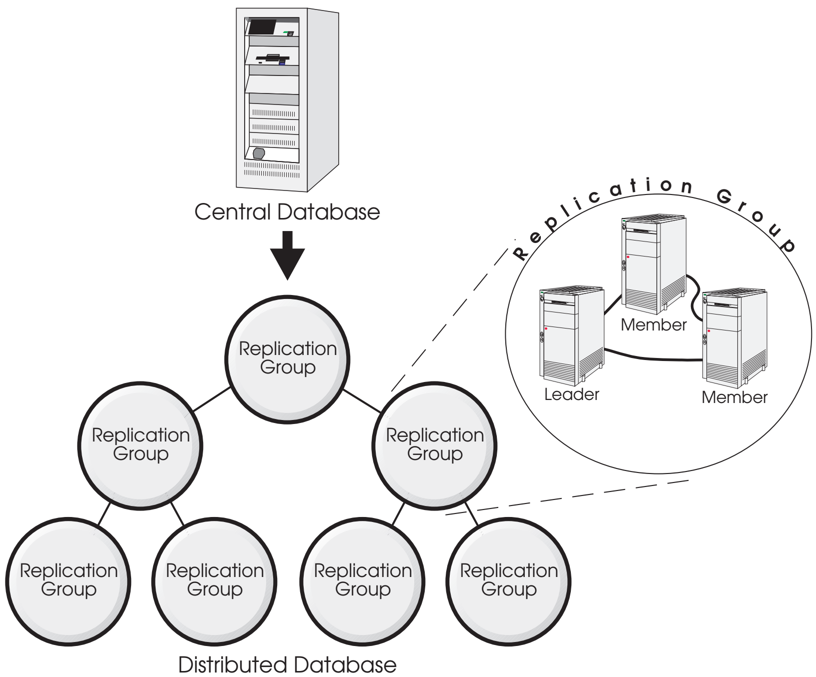


Figure 2. Our RDS Architecture
The RDS is divided into centralized and distributed parts. The hints in the distributed database are partitioned and replicated by replication groups, collections of members controlled by a leader that store the same sets of hints.

Naming authorities will either directly or indirectly request the creation of and modifications to hints stored in the central database. The direct way is by communicating with the central database administrator through some means, such as email or a Web-based interface. The other, the recommended method, is to contract with a subspace provider. Subspace providers can compete through lower prices and quality of service, for instance, with custom software for managing subspaces. All transactions go to the subspace providers, which pass them to the central database or to other providers. The effect of this is to promote competition in the registration process and to reduce the number of entities with which the central database must communicate.

The distributed database, with less protection and higher availability than the central one, handles the large magnitude of resolutions. It employs a high degree of distribution and replication to achieve scaling in the face of failure. The database is composed of *replication groups* arranged in a hierarchy, although, unlike the DNS, this hierarchy is not based on its contents. Each replication group is a collection of *members* that maintain the same set of hints. One member in each group serves as the *leader* and acts for the group as a whole. Replication groups will be discussed in more detail in Section 5.2. The central database sends updates to the root replication group, which in turn filter down to the appropriate places in the tree. URN resolution can be initiated at any member. Members route queries through the tree of groups to the location of the needed hints.

4.5 Summary

PRIs are a kind of URN specifically designed for mobility and longevity. Our RDS is tailored for extracting subspaces from these names and finding their hints, although it is general enough for other types of URNs. Given the needs of and demands on an RDS, a freshly designed solution such as ours is justified. This system has a widely distributed and replicated component to serve the needs of the URN users and a centralized and more conventional part for publishers to register hints. The distributed database is described in the next chapter.

Chapter 5
The Distributed Database

The distributed database component of our RDS uses a B-tree structure to achieve scalability independent of its contents. The algorithms for manipulating a B-tree require several modifications to make them appropriate for a distributed and replicated system. While they treat the nodes in the B-tree as single entities, the nodes are actually groups of exact replicas. Each group has a leader that is responsible for membership and other group-wide tasks. A group maintains weak consistency through the use of an epidemic flooding algorithm.

Attackers can pose a serious threat if they can alter hints undetected. Replication shields the database from attack by providing redundancy. The hints themselves are digitally signed to prevent falsification. Auxiliary hints can be added by anyone to correct and augment the ones inserted by the central database. All hints share a number of properties. They indicate the next step in resolving a URN, and possibly meta-information to aid in choosing which resolution step to follow if multiple ones are given.

The central authority runs the ADMIN system alongside the central database. ADMIN provides services to the replication groups and monitors the database as a whole, ensuring it is running efficiently, fixing and reporting problems, and collecting statistics.

5.1 Tree Structure

The distributed database is arranged in a tree of replication groups that is unrelated to the delegation structure of the names it contains. The tree must optimize search time, so we want to minimize its height, hopefully to about four so only a small number of servers need to be contacted during a search. Consequently, given the target number of subspaces, the tree will have a high branching factor. The distributed database may operate for over a decade with little change. In that time the tree should not become overly unbalanced because the database cannot be shut down for maintenance.

A B-tree [6] has an arbitrary branching degree and remains balanced with a worst-case height of $O(\log n)$, where n is the number of keys in the tree. Search and insertion times are proportional to the height of the tree. The traditional algorithms for these operations require modifications to form the basis of a distributed system. Furthermore, the searching algorithm requires additional development because the distributed database needs to find the subspace of a URN instead of an exact match.

5.1.1 Traditional B-tree Algorithm

A B-tree is a generalized version of a balanced binary search tree that can have an arbitrary branching factor. Its general properties are [1]:

- Each node x has $x.n$ keys, $x.k[1]\ldots x.k[n]$, stored in non-decreasing order.
- Each internal node x has $x.n+1$ children nodes, $x.c[1]\ldots x.c[x.n+1]$. The keys fall between those of its children such that $x.c[i].k[x.c[i].n] \le x.k[i] \le x.c[i+1].k[1]$.
- All leaf nodes have the same depth.
- For each non-root node x, $t-1 \le x.n \le 2t-1$, where t is the minimum degree. The root must have at least one key. Consequently, each internal node except for the root has at least t children. A node is full when it has $2t-1$ keys, the maximum allowed.

Inserting a key into a B-tree is more complicated than for a binary search tree since the tree must remain balanced with a uniform depth. To maintain this invariant, a non-root splits in two when it becomes full, sending its median key to its parent. Thus, a B-tree usually grows horizontally instead of vertically. Since the root node has no parent, when it becomes full a new node is created as its parent so that it can split. When the tree has become full at its current height, it grows from the top instead of from the bottom as in conventional trees. An insert operation starts at the root, which splits if full, and proceeds down to the appropriate leaf. Before descending to a child, the child is split if it is full, guaranteeing that it has room for a new key.

1. This description of the B-tree algorithm is derived from [6], pages 384-393.

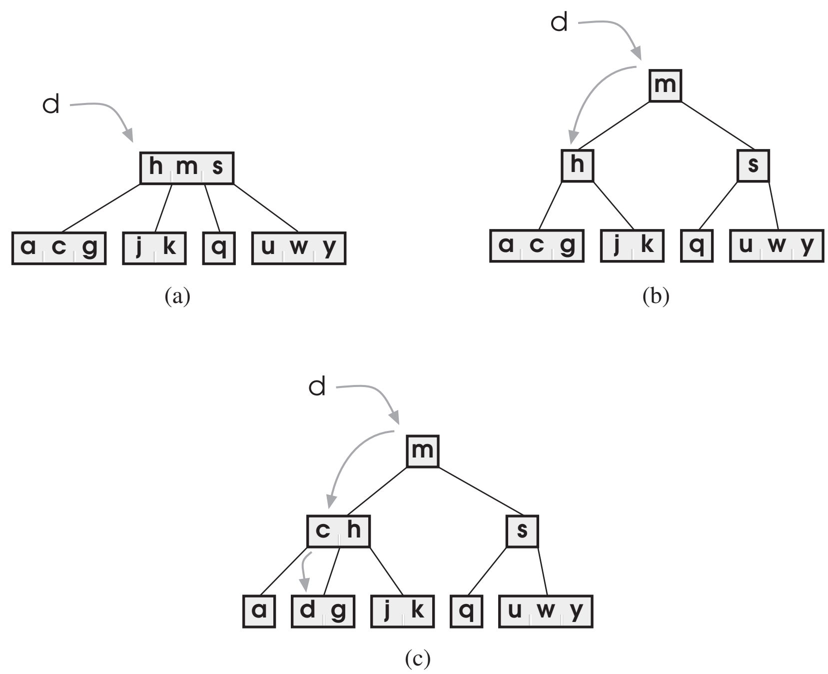


Figure 3. Single Insertion

(a) The key d is inserted into a tree with a branching degree of two.
(b) The root, which is full, splits, increasing the depth of the tree. The insert proceeds to the child.
(c) The leaf that d needs to be inserted into is full, so it splits before the insert can progress downwards and finish.

An example insertion is shown in Figure 3. The key d is being inserted into a B-tree of height one populated by single-letter keys. The tree has a branching degree of two, so each node can have between one and three keys. In part a. the key is inserted at the root. The root is full, so an empty parent node is created, and the root splits in half, pushing up its median key m. This is the only way that a B-tree can grow vertically. With the root no longer full, the insertion can progress, as shown in part b. d needs to be inserted into the first leaf, which is full, so it is split and its median key pushed up (part c.), growing the tree horizontally. With enough room in the appropriate child (the second leaf), the insert can complete.

Here is an outline of the insertion algorithm. More detailed pseudocode can be found in [6].

```
Insert(tree, key)
if tree.root is full
    create a new node x as parent of tree.root
    tree.root ← x
    Split(x, 1)
InsertNonfull(tree.root, key)
```

Insert depends on two operations: Split and InsertNonfull. Split takes a node and the index of a child and splits the child in half.

```
Split(x, i)
c ← x.c[i]
create a new node d
d.k ← c.k[t + 1 … 2t − 1]
if c is internal, d.c ← c.c[t + 1 … 2t]
key_med ← c.k[t]
delete c.k[t] … c.k[2t − 1] and, if c is internal, c.c[t + 1] … c.c[2t]
insert d at x.c[i + 1]
insert key_med at x.k[i]
```

Splitting a child node guarantees that it is not full. A key can be inserted into it with InsertNonfull:

```
InsertNonfull(x, key)
if x is a leaf
    insert key into x.k, maintaining the order
else
    find the child c to insert key into
    if c is full
        Split(x, index of c)
    find the child d to insert key into
    InsertNonfull(d, key)
```

To find a key, the tree is traversed down from the root until either it is found or the search reaches a leaf that does not contain the key, in which case the key is not in the tree. The search returns the node in which the key was found and its index:

```
Search(x, key)
if key ∈ x.k
    return ⟨x, index of key⟩
if node is a leaf
    return ⟨nil, 0⟩
else
    find the child c to search
    return Search(c, key)
```

This algorithm has some notable advantages. Insertion requires a single downward traversal of the tree, and splitting is a localized operation that takes $O(1)$ time. Furthermore, storing keys in all nodes of the tree is beneficial for resource utilization and load balancing.

The B-tree has some shortcomings when applied to a distributed system. One is that while the insertion algorithm only inserts one key at a time while the central database will need to insert many at once. Also, because a node is the abstract representation of a replication group, splitting a node is a fairly expensive operation that requires the cooperation of some number of networked computers.

5.1.2 Insertion Enhancements

The insert algorithm needs to be enhanced if a B-tree is to be used as the database structure, with each node of the tree being a replication group and the keys the URNs or subspaces of hints.

5.1.2.1 Root Splitting

In the traditional B-tree, when the root is full a new one is created. In a distributed network B-tree the root cannot change because others depend on it being well-known. The central database needs to communicate with the root, as does ADMIN, the central management and monitoring program. To keep the root constant, Insert relies on SplitRoot. Changes are given in bold:

```
Insert(tree, key)
if tree.root is full
    SplitRoot(tree.root)
InsertNonfull(tree.root, key)
```

SplitRoot creates two new nodes that become the children of the root and seeds them with the root's data:

SplitRoot(*root*)
create two new nodes, x and y
$x.k \leftarrow root.k[1 \ldots t-1]$
if internal, $x.c \leftarrow root.c[1 \ldots t]$
$y.k \leftarrow root.k[t+1 \ldots 2t-1]$
if internal, $y.c \leftarrow root.c[t+1 \ldots 2t]$
$root.k \leftarrow root.k[t]$, $root.c \leftarrow \{x, y\}$

5.1.2.2 Node Perspective

Another difference is that the data structure cannot be manipulated as easily when it is implemented with networked computers. Creating a new node with half the keys and children of an existing node is not just an array copy; it involves multiple computers cooperating over the network. The distributed algorithm considers what each node must do and how it communicates with others. We take an object oriented approach, so algorithms are applied on a node that refers to itself as *this*. Insert is now the only entry point for insertion into all nodes so each node needs a new field, *isRoot*, which is true if the node is the root. Before calling Insert on a node there is no way to know if it is full. While this could be verified, the check would require an extra round trip communication. Here is the object-oriented outline for the new Insert:

Insert(*key*)
```
if this is full
    if isRoot
        SplitRoot()
        InsertNonfull(key)
        return true
    else
        SplitNonroot()
        return false
else
    InsertNonfull(key)
    return true
```

If this node is the root and it is full, it splits itself. Since it is now guaranteed to not be full, InsertNonfull is called on it to carry out the insertion. Again, when Insert is called on a node that is not full, InsertNonfull is called. This algorithm diverges from the original in that an Insert can be called on a node that is full. If it is, the node has to split itself and then have the parent try again, which is indicated by Insert returning false. A node also needs a new field, *p*, that is a pointer to its parent. When an internal node splits, half of its children must be told of their new parent. SplitNonroot splits a non-root node in two:

```
SplitNonroot()
create a new node x to be the new sibling
x.p ← p, x.k ← k[t + 1…2t – 1]
if this is not a leaf
    x.c ← c[t + 1…2t]
    for i ← t + 1 to 2t
        c[i].p ← x
tell p about split, k[t], and x
delete k[t]..k[2t – 1]
if internal, delete c[t + 1]..c[2t]
```

A new node is created by forming a replication group from unused members. It is assumed that the minimum amount of resources are available. If they are not, then the central maintainers can temporarily make up the deficit until third parties contribute more members.

Here is the new, node oriented version of InsertNonfull:

```
InsertNonfull(key)
if key ∈ k
    replace old key with key
else
    if this is a leaf
        insert key into k, maintaining the order
    else
        find the appropriate child d to insert key into
        if !d.Insert(key)
            InsertNonfull(key)
```

InsertNonfull replaces the key if it exists and inserts it if it does not because a key is a subspace of a URN with an attached hint. In line 8, the key is inserted into a potentially full child, in which case the child will split and be unable to finish the insert. If this happens, InsertNonfull is called again. This time the child will not be full so the insert will succeed.

It should be noted that only one insertion can be performed at a time. During an insert operation the tree may be internally inconsistent and cannot be guaranteed consistent until the operation completes. Atomic operations and synchronization between nodes needs to be minimized so concurrent inserts are not allowed.

5.1.2.3 Augmentations for Searching

Searches may be initiated at any node in the tree instead of originating just at the root, unlike the traditional algorithm; a search may ascend as well as descend the tree. We have already augmented the node data with the parent so the search can ascend the tree. The enhanced search algorithm in Section 5.1.3 also needs each internal node to keep track of the range of keys in the subtrees rooted at each of its children. From this information it can determine the direction to search and refrain from descending unnecessarily.

Since all inserts proceed through the parent, it receives enough information to track child ranges. Each internal node x has $x.n + 1$ ranges, $x.r[1] \ldots x.r[x.n + 1]$. Each range has two fields, f and l, that denote the first and last keys in it. Here is modified pseudocode to maintain r, with changes in bold:

SplitRoot(*root*)
create two new nodes, x and y, to be the children
$x.p \leftarrow this$, $x.k \leftarrow k[1 \ldots t-1]$
if *this* is not a leaf
    $x.c \leftarrow c[1 \ldots t]$ and $\boldsymbol{x.r \leftarrow r[1 \ldots t]}$
    $c[1 \ldots t].p \leftarrow x$
$y.p \leftarrow this$, $y.k \leftarrow k[t+1 \ldots 2t-1]$

```
if this is not a leaf
    y.c ← c[t + 1…2t] and y.r ← r[t + 1…2t]
    c[t + 1…2t].p ← y
r ← {⟨k[1], k[t − 1]⟩, ⟨k[t + 1], k[2t − 1]⟩}
k ← k[t], c ← {x, y}
```

SplitNonroot()

```
create a new node x to be the new sibling
x.p ← p, x.k ← k[t + 1…2t − 1]
if this is not a leaf
    x.c ← c[t + 1…2t] and x.r ← r[t + 1…2t]
    c[t + 1…2t].p ← x
tell p about split, k[t], x, and x's range
delete k[t]..k[2t − 1]
if internal, delete c[t + 1]..c[2t] and r[t + 1]…r[2t]
```

InsertNonfull(*key*)

```
if key ∈ k
    replace old key with key
else
    if this is a leaf
        insert key into k, maintaining the order
    else
        find the appropriate child d to insert key into and its range w
        w ← w ∪ key
        if !d.Insert(key)
            InsertNonfull(key)
```

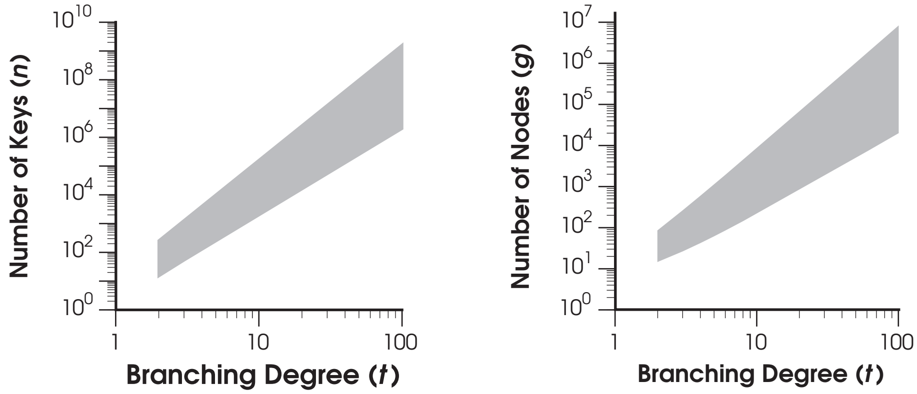
5.1.2.4 Key-to-child Ratio

Another way that the algorithm fails to consider physical resources is in the number of hints per node. Each internal node has one more child than keys, so a significant percentage of its data is composed of structural overhead. Also, the number of keys in the tree is a function of its height and branching degree:

$$2t^h - 1 \le n \le (2t)^{h+1} - 1$$

where h is the height of the tree and n is the total number of keys. The number of nodes g that contain these keys is:

$$2\frac{t^h - 1}{t - 1} + 1 \le g \le \frac{(2t)^{h+1} - 1}{2t - 1}.$$



(a) (b)

Figure 4. Number of Keys and Nodes for a B-tree of Height Three
(a) The range of keys possible in a B-tree of height three for various branching degrees.
(b) The corresponding number of nodes needed to store those keys.

Figure 4 shows the number of keys and nodes in a B-tree of height three for various branching degrees. It shows that a tree with a branching degree of 50 can hold up to 10^8 keys, but this would require up to 10^6 nodes, with each node having just 100 keys. The number of keys per node and the branching degree need to be independently controlled because a replication group should have many more hints than children. The optimal ratio between keys and children can only be learned from experience; nonetheless, the algorithm needs to be modified to allow greater control over the composition of the B-tree.

The ratio of keys to children can be adjusted by adding a new field, s. Instead of one key separating two children, s keys do. An internal node has $\frac{n}{s} + 1$ children. At its fullest any node will have $(2t - 1)s$ keys. To achieve this ratio, when a node splits it pushes up s continuous keys to its parent instead of just one. Thus the number of keys in an internal node will always be a multiple of s.

During an insert to an internal node, the new key may fall between two keys not separated by a child. The number of keys between two children cannot be

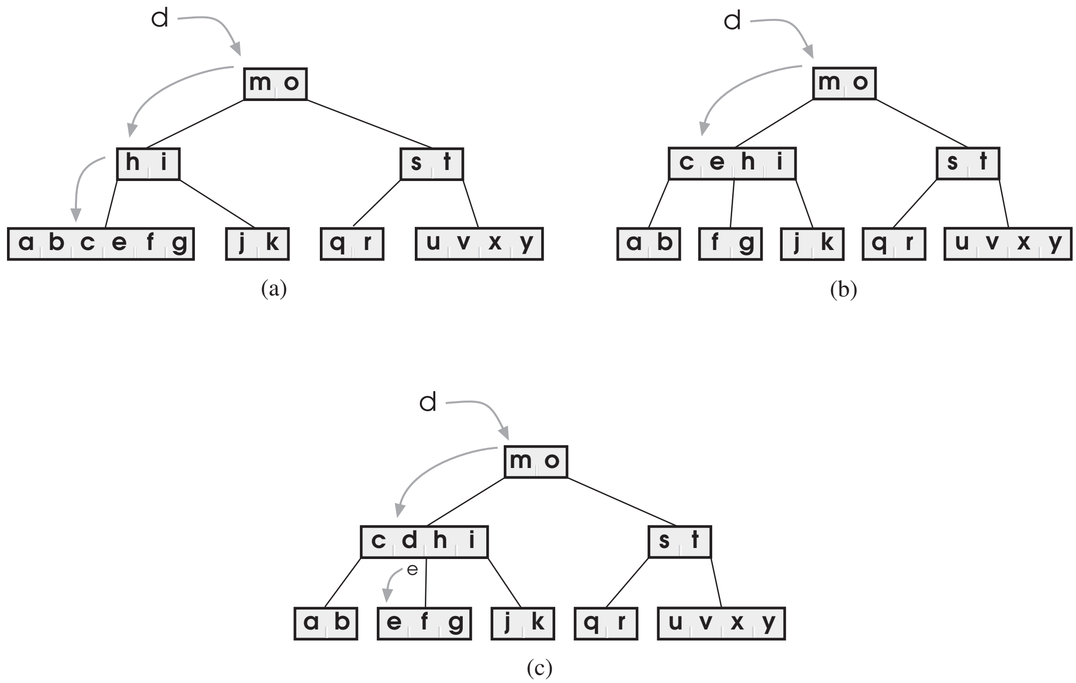


Figure 5. Single Insertion into a B-tree with $s = 2$

(a) The key d is inserted into the tree. The search progresses to the first leaf.
(b) The leaf is full so it splits, pushing up two keys, c and e. The insertion backs up to the leaf's parent.
(c) d falls between c and e, so it cannot be inserted into a child. d is inserted into the node and e is pushed out.

reduced by splitting because the tree must stay balanced. In this case a key that is next to a child is inserted into the child, making room for the new key.

Figure 5 shows how insertion is modified for a tree with multiple keys per child. The example has a ratio of two ($s = 2$), so internal nodes have two keys between each child. In part a. the insertion proceeds as usual. The first leaf is full and splits when it receives the new key d (part b.). Instead of pushing one key up to the parent it pushes two. In general, every split pushes s keys up, so internal nodes always have a multiple of s keys. The insertion backs up to the parent because it may, as in this case, proceed to the new child. Since d falls between keys c and e, there is no child into which it can be inserted. To make room for d, e is inserted into its adjacent child, as shown in part c.

Here is the new version of the splitting algorithm to support multiple keys per child. The two main modifications are to push up multiple keys and push end keys into their children as shown in part c.

SplitRoot(*root*)

create two new nodes, x and y, to be the children
$x.p \leftarrow this$, $x.k \leftarrow k[\mathbf{1} \ldots (\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]$
if *this* is not a leaf
    $x.c \leftarrow c[1 \ldots t]$ and $x.r \leftarrow r[1 \ldots t]$
    $c[1 \ldots t].p \leftarrow x$
$y.p \leftarrow this$, $y.k \leftarrow k[\boldsymbol{st}+\mathbf{1} \ldots (\mathbf{2}\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]$
if *this* is not a leaf
    $y.c \leftarrow c[t+1 \ldots 2t]$ and $y.r \leftarrow r[t+1 \ldots 2t]$
    $c[t+1 \ldots 2t].p \leftarrow y$
$r \leftarrow \{\langle k[1], k[(\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]\rangle, \langle k[\boldsymbol{st}+\mathbf{1}], k[(\mathbf{2}\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]\rangle\}$
$k \leftarrow k[(\boldsymbol{t}-\mathbf{1})\boldsymbol{s}+\mathbf{1} \ldots \boldsymbol{st}]$, $c \leftarrow \{x, y\}$

SplitNonroot()

create a new node x to be the new sibling
$x.p \leftarrow p$, $x.k \leftarrow k[\boldsymbol{st}+\mathbf{1} \ldots (\mathbf{2}\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]$
if *this* is not a leaf
    $x.c \leftarrow c[t+1 \ldots 2t]$ and $x.r \leftarrow r[t+1 \ldots 2t]$
    $c[t+1 \ldots 2t].p \leftarrow x$
tell p about split, $\boldsymbol{k}[(\boldsymbol{t}-\mathbf{1})\boldsymbol{s}+\mathbf{1}] \ldots \boldsymbol{k}[\boldsymbol{st}]$, x, and x's range
delete $k[(\boldsymbol{t}-\mathbf{1})\boldsymbol{s}+\mathbf{1}] .. k[(\mathbf{2}\boldsymbol{t}-\mathbf{1})\boldsymbol{s}]$
if internal, delete $c[t+1] .. c[2t]$ and $r[t+1] \ldots r[2t]$

InsertNonfull has the largest change. If the key to be inserted falls between two keys that are not separated by a child, the key is inserted and a key next to a child is inserted into it to make room.

InsertNonfull(*key*)

if $key \in k$
    replace old key with *key*
else
    if *this* is a leaf
    insert *key* into k, maintaining the non-decreasing order
    else
        $\boldsymbol{i} \leftarrow$ index where *key* should be inserted to maintain order
        if $\boldsymbol{i}$ mod $\boldsymbol{s} \neq \mathbf{1}$
            insert *key* at position $\boldsymbol{i}$
            $\boldsymbol{j} \leftarrow \boldsymbol{i}+\boldsymbol{s}-((\boldsymbol{i}-\mathbf{1}) \textbf{ mod } \boldsymbol{s})$
            $\boldsymbol{key} \leftarrow \boldsymbol{k}[\boldsymbol{j}]$
            delete $\boldsymbol{k}[\boldsymbol{j}]$
            $\boldsymbol{i} \leftarrow \boldsymbol{j}$
            $\boldsymbol{i} \leftarrow (\boldsymbol{i}-\mathbf{1})/\boldsymbol{s}+\mathbf{1}$

```
        r[i] ← r[i] ∪ key
        if !c[i].Insert(key)
            InsertNonfull(key)
```

5.1.2.5 Bulk Insert

The insert algorithm only inserts one key at a time. However, the central database will need to insert many hints at once, necessitating a bulk insert method. Bulk insertion is significantly more complicated than the single version. The single insert algorithm splits full nodes in half on the way down to make room for one key to be pushed up. In a bulk insert many keys may be pushed up from multiple children, possibly more than the full limit. It is not clear how to split a node until after its children have split, so the bulk insert must filter down and then back up the tree.

The bulk insert passes all new keys down to the appropriate leaves, updating internal node ranges in the process. Each leaf that receives keys either stores them, or, if it does not have enough room, splits into two or more nodes, pushing the dividing keys up to the parent. After collecting all pushed up keys from its children, a parent splits into multiple nodes if needed, and pushes its dividing keys up. If the root receives enough keys, it may split multiple times, growing the height. If the root needs to create more than *st* children it will still be full and will split again.

The bulk insert is demonstrated in Figure 6. In part a. two keys, d and n, are inserted into a tree with a branching degree of two and a key-to-child ratio of two. The root passes the insert to two of its children. d is inserted into the first child. Since n cannot be passed to a child, it is inserted into the root node and the existing key o inserted into the third child. These are concurrently inserted by the children in part b. The first child is now full so it splits, as shown in part c. After the first and third children have finished with the insert, the root can split if it is full, which it is in this example. Part d. shows that the root has split and the bulk insert is complete.

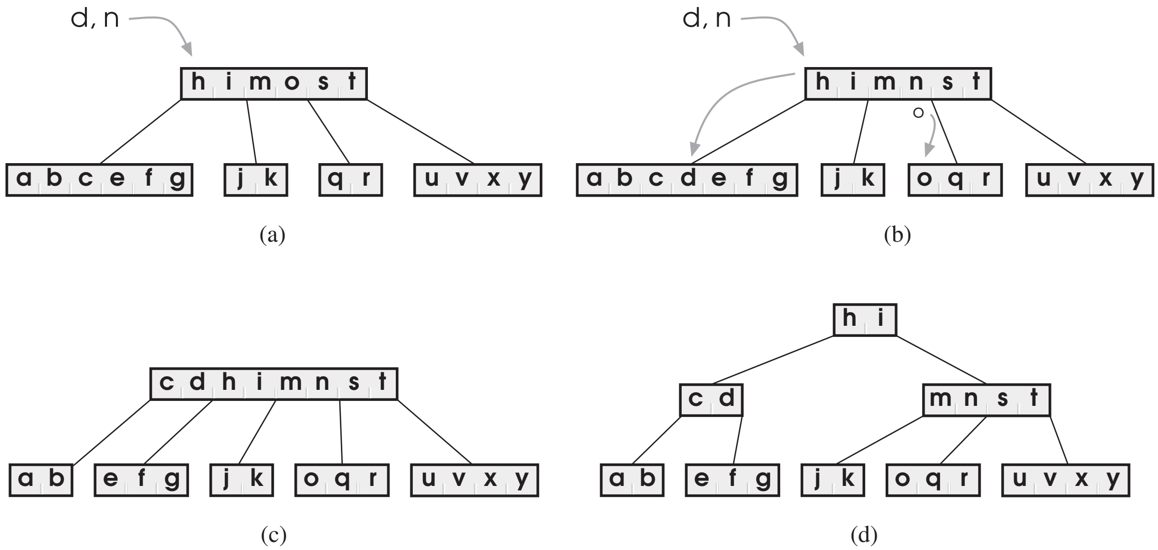


Figure 6. Bulk Insertion into a B-tree with $s = 2$

(a) Two keys, d and n, are inserted into the tree.
(b) d is inserted into the first child. Since n fits between keys m and o, o is inserted into the third child to make room for n. The first child is now full.
(c) The first child splits, pushing up two keys. Now the root is full.
(d) After all children are done with the bulk insert, the root, which is now full, splits.

BulkInsert is a node's entry point for inserting more than one key. Since internal nodes and leaves must behave differently on the way down, this algorithm simply calls on the proper behavior given the node's position.

```
BulkInsert(keys)
if this is a leaf
    LeafInsert(keys)
else
    InternalInsert(keys)
```

InternalInsert determines which keys should be inserted into which children:

```
InternalInsert(keys)
for each key in keys
    if key ∈ k
        replace old key with key and remove key from keys
    else
        if key does not fall into any child
            insert into k and push end key into keys, as in InsertNonfull
partition keys into blocks for children
for each b in blocks and its associated child index i
    r[i] ← r[i] ∪ range of b
    c[i].BulkInsert(b)
```

LeafInsert inserts keys into a leaf node , splitting it if necessary:

```
LeafInsert(keys)
merge keys with k, maintaining order
if n > (2t – 1)s
    BulkLeafSplit()
else if !isRoot
    tell p that insert fit
```

After all children received a bulk insert have responded to the parent, either by pushing up the divider keys or by telling the parent that the insert fit, the parent can finish its part, splitting if full. This is accomplished with the Finish algorithm:

```
Finish()
if n > (2t – 1)s
    BulkInternalSplit()
else if !isRoot
    tell p that insert fit
```

LeafInsert and Finish rely on BulkLeafSplit and BulkInternalSplit to split the node if it is full.

```
BulkLeafSplit()
numNodes ← ⌊(n + s)/(st)⌋
step ← (n + s)/numNodes
for i ← 1 to numNodes
    splitK[i] ← k[⌊(i – 1)step + 1⌋…⌊i · step – s⌋]
    if i ≠ numNodes
        dividers[(i – 1)s + 1…is] ← k[⌊i · step – s + 1⌋…⌊i · step⌋]
FinishSplit(numNodes, splitK, nil, nil, dividers)
```

```
BulkInternalSplit()
numNodes ← ⌊(n + s)/(st)⌋
step ← (n + s)/(s · numNodes)
for i ← 1 to numNodes
    splitK[i] ← k[⌊(i – 1)step⌋s + 1…⌊i · step – 1⌋s]
    splitC[i] ← c[⌊(i – 1)step + 1⌋…⌊i · step⌋]
    splitR[i] ← r[⌊(i – 1)step + 1⌋…⌊i · step⌋]
    if i ≠ numNodes
        dividers[(i – 1)s + 1…is] ← k[⌊i · step – 1⌋s + 1…⌊i · step⌋s]
FinishSplit(numNodes, splitK, splitC, splitR, dividers)
```

```
FinishSplit(numNodes, splitK, splitC, splitR, dividers)
if isRoot
    create numNodes new nodes, x[1]…x[numNodes]
    for i ← 1 to numNodes
        x[i].p ← this, x[i].k ← splitK[i]
```

```
      k ← dividers
      c ← x , r ← x's ranges
      if internal
            x[i].c ← splitC[i] , x[i].r ← splitR[i]
            for each sc ∈ splitC[i] , sc.p ← x[i]
      Finish()
else
      create numNodes − 1 new nodes, x[2]...x[numNodes]
      for i ← 2 to numNodes
            x[i].p ← p , x[i].k ← splitK[i]
      k ← splitK[1]
      if internal
            x[i].c ← splitC[i] , x[i].r ← splitR[i]
            for each sc ∈ splitC[i] , sc.p ← x[i]
      tell p about split, dividers, x, and x's ranges
```

Depending on the number of hints inserted, the bulk update can involve a large number of replication groups and significant coordination between a parent and its children, increasing the chance that something will fail.

5.1.3 Searching Enhancements

As in the case of insertion, searching also needs to be modified. First, as mentioned in Section 5.1.2.3, searches can be initiated anywhere in the tree, not just at the root, so the search must progress up, as well as down, the tree. Caching is used to make non-local jumps to reduce the number of nodes involved in a search and avoid a bottleneck at the root. Finally, the distributed database contains subspaces, not URNs, requiring a more sophisticated searching algorithm instead of the exact match version given above.

5.1.3.1 Search from Any Node

The first modification to the search algorithm allows searches to start anywhere. When a node receives an inquiry for a key that it does not contain, it must determine if the search request should be passed to the parent or a child. Searching is done in two stages. First it progresses up the tree until it reaches a subtree that contains the key. In the second stage it has reached the root of a subtree that may contain it so the search descends downwards as in the traditional algorithm. Here is the new, object-oriented version of the search algorithm that starts at any node. This version returns the key, which is actually a hint, instead of its index. Search simply begins the upward stage:

```
Search(key)
return SearchUp(key)
```

SearchUp searches up the tree. If the key falls within the node's subtree then the search starts downwards.

```
SearchUp(key)
if key out of this node's range
    if isRoot
        return ⟨nil, nil⟩
    else
        return p.SearchUp(key)
else
    return SearchDown(key)
```

SearchDown differs from the original Search in that the data structure has been augmented with ranges, so it can determine if a key does not exist before descending. It also must factor in the key-to-child ratio.

```
SearchDown(key)
if key out of this node's range
    return ⟨nil, nil⟩
else
    if key ∈ k
        return ⟨this, k[index of key]⟩
    if this is not a leaf and r[i].f ≤ key ≤ r[i].l for some 1 ≤ i ≤ r.n
        return c[i].SearchDown(key)
    return ⟨nil, nil⟩
```

5.1.3.2 Caching

A node that receives queries for URNs in the same subspace can benefit from caching. Caching information about other nodes allows searches to jump sideways and not just up and down. Trees have the weakness that there are fewer nodes at higher levels, creating a bottleneck at the top. Caching individual URNs would result in a low hit rate due to their extremely large number. Instead, nodes cache the ranges of other nodes, which they discover by recording the path a search takes. When the node that initiates a search gets the result back, it can cache each node encountered and its range. If a cache is stale then the search will jump to a node that does not have the expected range, but the search can progress from there.

A node needs a new field, *cache*, that is a list of cache entries. Each entry contains a node *m*, its range *r*, and auxiliary fields for use by a cache replacement strategy. These can be the number of times an entry has been hit, the last time it was used, and its age. The new version of Search adds the nodes and ranges encountered during a search to *cache*. SearchUp and SearchDown now return a triplet of the node at which the hint was found, the hint, and the search path. (i.e. $\langle x, h, e\rangle$). The search path is the set of nodes and ranges encountered during the search

```
Search(key)
⟨x, h, e⟩ = SearchUp(key)
for each entry in e
    if entry.m ∉ {this, p} ∪ c
        remove cache entries with entry's range or node
        add entry to cache
return ⟨x, h⟩
```

SearchUp now looks in its cache for a better node to search before proceeding as normal. It additionally returns the nodes encountered and their ranges.

```
SearchUp(key)
if key is in the range of cache entry e
    if key is not in this node's range or e.r is subsumed by it
        ⟨x, h, e⟩ = e.SearchUp(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
if key out of this node's range
    if isRoot
        return ⟨nil, nil, {⟨this, range⟩}⟩
    else
        ⟨x, h, e⟩ = p.SearchUp(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else
```

```
    ⟨x, h, e⟩ = SearchDown(key)
    return ⟨x, h, ⟨this, range⟩ ∪ e⟩⟩
```

Similarly for SearchDown:

```
SearchDown(key)
if key is in the range of cache entry e
    if key is not in this node's range or e.r is subsumed by it
        ⟨x, h, e⟩ = e.SearchUp(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
if key out of this node's range
    return ⟨nil, nil, {⟨this, range⟩}⟩
else
    if key ∈ k
        return ⟨this, this.k[index of key], {⟨this, range⟩}⟩
    if this is not a leaf and r[i].f ≤ key ≤ r[i].l for some 1 ≤ i ≤ r.n
        ⟨x, h, e⟩ = c[i].SearchDown(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
    return ⟨nil, nil, {⟨this, range⟩}⟩
```

5.1.3.3 Subspace Determination

Unlike the traditional B-tree search, a successful search of the distributed database may not result in an exact match. The database contains hints lexically sorted by their keys, the subspaces and URNs. A search needs to discover the subspace to which a URN belongs. After a search for an exact match fails, a subspace for the URN is hypothesized and looked for. The hint one less than the target URN is either the URN's subspace, within the URN's subspace, or unrelated. In the last case their are no hints for the URN's subspace, but in the first two cases there may be. If the hint directly less than a URN is its prefix then it is the URN's subspace, and the hint is result of the query. If the hint and the URN share a common prefix, then they may be in the same subspace. The common prefix is searched for in the same manner until either a subspace is found or it can be determined that none exists. The search steps for finding a URN's subspace are:

1. Search for the given URN.
2. If an exact match is found, return it.

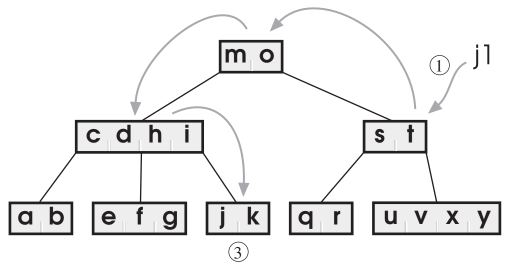


Figure 7. Subspace Determination

A user queries the node with keys s and t about the URN jl. In step 1 the tree is searched for an exact match. Step 1 ends at the node with keys j and k, where it can be determined that an exact match does not exist. The search then looks for the key less than the URN, as per step 3. The key j is found in the same node. By step 4, since it is the URN's prefix it must be its subspace.

3. Search for the hint directly less than the URN.

4. If the hint is the URN's prefix, return it; it is the subspace. If it is not, but shares a common prefix, go back to step 1 and search for the prefix, which may be the subspace.

An example search is shown in Figure 7. Here a hint for the URN jl is being looked for. The search begins with step 1, looking for an exact match. It traverses the tree until it can be determined that an exact match does not exist. In this case the search ends at a node with the keys j and k. Since an exact match does not exist, the search proceeds to step 3 to look for the subspace. In this case, the key directly less than the URN is in the node where step 1 ended. It will often be the case that step 1 takes the search to a node that is close to the key found in step 3. The key j is directly less than jl. This is the URN's prefix, so by step 4 this is the best subspace, and its hint is returned.

The new versions of SearchUp and SearchDown carry out steps 1 and 2. When either one determines that an exact match does not exist, it looks for the key directly less than the URN by calling SearchForSubspace, which is responsible for step 3.

Finally, if such a key is found, SearchForSubspace proceeds to step 4 with the ExplorePossibility algorithm.

The modified SearchUp progresses to step 3 by calling SearchForSubspace if it fails to find a subtree that contains the key, but only if the key is greater than the greatest one in the tree.

```
SearchUp(key)
if key is in the range of cache entry e
    if key is not in this node's range or e.r is subsumed by it
        ⟨x, h, e⟩ = e.SearchDown(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
if key out of this node's range
    if isRoot
        if key is greater than the top of the node's range
            ⟨x, h, e⟩ = SearchForSubspace(key)
            return ⟨x, h, ⟨this, range⟩ ∪ e⟩
        else
            return ⟨nil, nil, {⟨this, range⟩}⟩
    else
        ⟨x, h, e⟩ = p.SearchUp(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else
    ⟨x, h, e⟩ = SearchDown(key)
    return ⟨x, h, ⟨this, range⟩ ∪ e⟩⟩
```

While searching downwards, if the search key is not in the node's range, there are two possibilities. The key could be greater than the top of the range, in which case its subspace may be in the node's subtree. If it is less than the bottom of the range, the subspace is not in the subtree so the search passes to the parent. If the key is in the node's range but not in the node, either it or one of its children may contain then the subspace.

```
SearchDown(key)
if key is in the range of cache entry e
    if key is not in this node's range or e.r is subsumed by it
        ⟨x, h, e⟩ = e.SearchDown(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
if key out of this node's range
    if key is greater than the top of the node's range
        ⟨x, h, e⟩ = SearchForSubspace(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
```

```
    else
        if isRoot
            return ⟨nil, nil, {⟨this, range⟩}⟩
        else
            ⟨x, h, e⟩ = p.SearchForSubspace(key)
            return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else
    if key ∈ k
        return ⟨this, this.k[index of key], {⟨this, range⟩}⟩
    if this is not a leaf and r[i].f ≤ key ≤ r[i].l for some 1 ≤ i ≤ r.n
        ⟨x, h, e⟩ = c[i].SearchDown(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
    ⟨x, h, e⟩ = SearchForSubspace(key)
    return ⟨x, h, ⟨this, range⟩ ∪ e⟩
```

Step 3 is handled by SearchForSubspace, which searches for the key one less than the given one. It ascends the tree zero or more times until it reaches a subtree that contains the desired key. It then descends until it finds the key. There are three possibilities when SearchForSubspace is called with a given key. The key is either greater than every key in the subtree, less than all of them, or falls in the subtree. In the first case, the result is the greatest key in the subtree. In the second case the search ascends to the parent, and in the third, the search descends until it finds the key. Upon finding the desired key, SearchForSubspace calls ExplorePossibility to determine the subspace and search for it.

```
SearchForSubspace(key)
if key is greater than the top of the node's range
    if this is a leaf
        ⟨x, h, e⟩ = ExplorePossibility(key, k[n])
    else
        ⟨x, h, e⟩ = c[n/s + 1].SearchForSubspace(key)
    return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else if key is less than the top of the node's range
    if isRoot
        return ⟨nil, nil, {⟨this, range⟩}⟩
    else
        ⟨x, h, e⟩ = p.SearchForSubspace(key)
        return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else
    let i be the index of the key directly less than key
    if this is not a leaf and i mod s = 0 and key > r[(i + 1)/s + 1].f
        ⟨x, h, e⟩ = c[(i + 1)/s + 1].SearchForSubspace(key)
```

```
else
    ⟨x, h, e⟩ = ExplorePossibility(key, k[i])
return ⟨x, h, ⟨this, range⟩ ∪ e⟩
```

ExplorePossibility performs step 4 by comparing the URN and the key directly less than it. If the key is a prefix of the URN, then the search is done and the subspace hint has been found. If they have a common prefix, then the search for the prefix proceeds. If they have nothing in common, then there is no hint for the URN.

ExplorePossibility(*key, lessKey*)

```
if lessKey is a prefix of key
    return ⟨this, lessKey, ⟨this, range⟩⟩
else if lessKey shares a prefix pre with key
    ⟨x, h, e⟩ = SearchUp(pre)
    return ⟨x, h, ⟨this, range⟩ ∪ e⟩
else
    return ⟨nil, nil, {⟨this, range⟩}⟩
```

5.1.4 Deletion

Subspaces will eventually need to be deleted when their owners no longer want them. Deletion is an expensive operation requiring the coordination of several replication groups. Any operation involving the restructuring of the database is complex and can fail in unexpected ways, so instead of being deleted, hints are marked as expired. They are replaced as new hints are inserted.

5.2 Replication Groups

So far the database has mostly been described as algorithms on a data structure. As mentioned, a node is a replication group composed of *members* that all have the same storage capacity. Servers have widely varying storage capacities and speeds that tend to increase over time, but a replication group needs replicas to be the same size, so a server is divided into a number of members, each satisfying the minimum size requirements. As a server gains more storage capacity, the number of members that it contributes to the database can increase.

Each site with URN users is expected to contribute a server with some number of members to the distributed database. The members are used as needed in any replication group, for example when a node splits or a replication group needs to grow. This insures that resources are best utilized throughout the entire database and allows them to be reallocated as needed. Groups request new members from ADMIN, which keeps a list of members that are unused but available. If none are free, ADMIN removes an underutilized member from the database. As part of its management duties, ADMIN keeps track of replication group sizes and loads as discussed in Section 5.4. There are a number of incentives for a site to contribute members. The primary one is to gain access for its users. Another incentive for a site to participate is to control the quality of service. The more members it contributes, the more entry points it will have into the distributed database, and its users will have a higher chance of starting a search closer to the results of their queries. The site can also control the size of its cache to shorten the search path.

Replication serves to handle the number of requests and improve robustness, which is especially important both for high reliability and so that a subtree does not become disconnected. The latter scenario could disable a large part of the database and require manual repair. As a precaution against this the minimum replication group size is three, with at least two of the members possessing leadership ability.

Each group has a leader that acts for the group as a whole and maintains the abstraction of a node being a single entity. The leader is responsible for maintaining group membership, receiving and distributing inserts to its members and children, performing tree operations, and making sure the group does not get overloaded. Not all members may have the ability to act as a leader either because their implementations are lacking the needed functionality or because they have been configured not to. Their owners may choose to do this because the member has a low bandwidth Internet connection or the computer that serves the member does not have the extra capacity or reliability to be a leader.

Since a node is actually multiple entities, a pointer to it is a list of all of the members. When a member needs to pass a search on to a parent or child, it can communicate with any member of the destination group, distributing the load. This organization adheres to the philosophy that there should be no single point of failure, often realized by a high degree of symmetry. Any member should be able to fail without adversely affecting the database, so no single member should have any information that cannot be reconstructed by another.

5.2.1 Group Membership

Each member of a group has a copy of the group's membership table so that anyone can become the leader and flood updates within the group. The table has an entry for each member with the following fields:

- **member:** A reference to the member.

- **health:** The member can be either LIVING or DEAD. It is LIVING unless it becomes disconnected from the group. If a member discovers that another one is not responding, it contacts the leader which makes an assessment that may lead to declaring the bad member DEAD. A DEAD member may not realize its health has changed, so when it contacts the group it is informed that it died and refreshes its data.

- **weight:** The weight is a rough measure of the member's computational ability based on the load it experiences. It is used by parents and children when picking with whom to communicate during a search.
- **leadership potential:** Indicates that this member can act as a leader.

The membership table also stores the ID of the current leader, a unique ID for the group (a randomly chosen eight byte integer), a version number, and a topology that specifies how members communicate. The ID assures others that they are communicating with a member from the intended group. The topology guarantees that each member has at least two neighbors with which to communicate.

A group needs a new member when minimum membership requirements are not being met or it is getting more requests than it can handle. The leader monitors the load of each member and adjusts the weights if the load is poorly distributed. If the load is uniformly high, then the group needs a new member to help out. The leader gets a new member from ADMIN and gives it a copy of its hints other data, and adds it to the membership table. A member may leave the group unexpectedly if its owner or ADMIN removes it, or if the group loses contact with it. If a member realizes that it has been disconnected for any reason, it will reestablish contact with its group and refresh its data. If it cannot find a group member then it returns to ADMIN's pool of unused members.

Each time the table changes, the leader increments its version and floods it to the group. When the leader adds a member, it informs the parent and children, which flood the information internally so that they can immediately use it. When a member is deleted or declared DEAD the parent and children are not informed. They will discover this for themselves if they try to contact it and subsequently try another member. If a member's table for its parent or a child reaches the minimum group size then it will get a new copy from a living member of the group.

While determining if a member is DEAD is straightforward, deciding if the leader has died is more problematic. Giving a member the authority to declare the leader DEAD would invite abuse by members that try to take over the group. When a member suspects that its leader is dead, it contacts the parent leader, which makes an assessment and may appoint a new leader. If the parent leader does not respond, then any other parent member is told that its leader is dead and takes the same steps. At the root, which has no parent, any of the trusted members can become the leader.

5.2.2 Flooding

Whenever a membership table changes it is flooded within the group, as is other replicated data such as hints. Flooding is the means by which all members eventually obtain the same information, similar to the mechanism used by Harvest. While Harvest uses an anti-entropy epidemic algorithm to spread updates, replication groups require them to spread more quickly during splits. The replication groups use a modified rumor mongering algorithm [9]. When a member receives an update it asks each of its topological neighbors given in the membership table and another member chosen randomly if they also want it based on its version, and sends it to them if they do. Eventually an update will reach all members. Every time a member is added or removed the leader begins a topology calculation based on bandwidth and includes it with the membership table update.

5.2.3 Replicated Algorithms

The insertion algorithm treats a node as a single entity even though it is actually group of members. To maintain this abstraction, insertions proceed through the leaders. When a leader receives new data it floods it to the rest of the group. It includes a checksum of all hints along with new and updated ones so that the receiver can tell if it missed any updates. This way network resources are not wasted sending unneeded data. Searches can only be halted for an extremely short time while the insert is occurring so the pointers to parents and children must be changed in the proper order. When a group needs to split, the leader forms a replication group from three new members, seeds each of them with the needed data, and makes the other adjustments to the tree's structure. When all the pointers have been safely rearranged, the leader removes the hints and children that it gave to the new group and floods this to the rest of its own group. If something goes wrong during the insert it may be lost, in which case it would have to be tried again. Any errors are propagated back to the root of the tree, but the best way to determine the success of an insert is to verify it by searching for one of the newly inserted hints.

Replication does not add as much complexity to searching as it does to insertion. The worst that can happen is that a search will result in a false negative, which is tolerable as long as this happens infrequently. When a member needs to call its parent or a child, it picks a member of the appropriate replication group at random based on the weights of all the members in the callee group.

5.3 Hints

The hints mentioned so far form the primary contents of the distributed database. These are *central hints*, hints from the central authority marked by slowly changing information. The database contains two other types of hints based on their source: *publisher hints* and *unauthorized hints*. Both are injected directly into the distributed database either to supersede central hints or suggest alternative resolutions. Publisher hints are digitally signed by the owners of the central hints they override, guaranteeing their authenticity. They are meant for short-term changes, such as the temporary addition of resolvers and to mask the delay in updating central hints. Publisher hints either expire or are purged over time. Unauthorized hints originate from entities that have no authority over a URN but wish to suggest an alternate resolution. This would be useful, for example, if we were going to publish HTML versions of plain text Internet Requests for Comments independently from the Internet Engineering Task Force. Our unauthorized hints for their URNs would inform clients of our alternate versions. Unauthorized hints should be treated with caution as their legitimacy cannot be determined.

Publisher and unauthorized hints can be injected into the distributed database at any member. Publisher hints are routed to the replication group with the corresponding central hints and unauthorized hints to the groups that would contain them if they were central. Instead of travelling through the leaders, they are sent through any member and are flooded when they reach their final destinations. Members use extra capacity to store them but are under no obligation to ensure their existence; they can delete these hints at any time. When there is a choice between purging publisher and unauthorized hints, the unauthorized hints are deleted first. A successful search will return a central hint, and possibly a publisher and one or more unauthorized hints.

Hints have an extensible format to allow for evolution. A distributed database hint contains the following information:

- **Subspace or URN:** This indicates the subject of the hint and acts as the hint's key.

- **Time stamp:** The time stamp tells when the hint was created and hence its age. A publisher hint with a newer time stamp replaces a previous one with an older one, and is purged after the central hint is updated with a newer time stamp.

- **Time-to-live:** This gives an upper bound on the lifetime of a hint. Publisher and unauthorized hints have a fixed maximum lifetime.

- **Expired:** Central hints that are no longer valid are marked as expired and may be purged. If an expired hint exists it may still be more useful than no hint at all.

- **Access control list:** A URN may resolve to a number of copies of a resource maintained by different entities who all need to modify its hints. The access control list is a list of public keys of entities, including the owner of the hint, that are allowed to create associated publisher hints. A publisher hint is only valid if it is signed with a private key that corresponds to one of the keys in this list. This list is only meaningful for central hints.

- **Resolution data:** This is a list of ways to resolve the URN. Each item can be the actual resolution, that is a URL or other string to which the URN maps; a redirection to a resolver along with a protocol; or another URN. It may be both a resolver and a URN, in which case the given resolver should be queried about the given URN. Optional meta-information about the resolution data can also be included to help a client choose which item to pick. Possible information includes geographic location, network bandwidth, the resource's language, and its file type. Each can have a public key belonging to the next resolver to guarantee the authenticity of resolution.

Resolvers may need other properties. Unknown ones should be ignored by clients to allow hints to be extended, possibly with a different security model, application specific data, or with other information as hints evolve.

5.4 Tree Operations

ADMIN maintains the database, managing resources and reporting problems that it cannot resolve. As part of the process it monitors replication groups and collects statistics on how the database is being used. The entire database needs to be monitored to ensure that it is operating smoothly and resources are well utilized. It monitors utilization with a tree operation that gets the number and type of members in and load of each group. The load is averaged over the period of a day to identify usage trends instead of hourly variations. Using this information, ADMIN can determine the resources allocated to each group and rearrange them. Short term load spikes are handled by caching. The other set of operations are for gathering statistics that indicate how people are using the database and subspaces. Among the data that can be collected is the most and least accessed hints and the total number of queries.

ADMIN initiates a tree operation at the root. In response to an operation request, a replication group reports the combination of the result of the operation performed on each child with its own data. Thus, an operation traverses the tree down and back up, visiting every group. For example, to find the maximum and minimum group loads (see Figure 8), each node calculates the maximum and minimum of its load and those reported by its children. Consequently, the time an operation takes is proportional to the tree height.

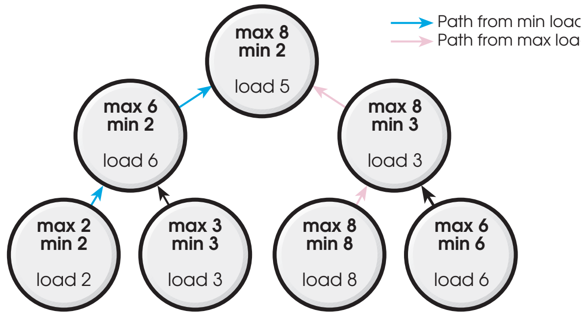


Figure 8. Computing the Maximum and Minimum Loads
The maximum and minimum loads are computed by traversing the tree of replication groups. Each computes its own load and combines it with the maximum and minimum loads of its children, if any.

5.5 Security

Since the database is distributed across computers owned by thousands of institutions, there is no way to guarantee the security and trustworthiness of replication servers. A security model needs to make sure that the authenticity of hints acquired from an untrusted server can be verified and that any malicious acts can be contained.

While the entire database cannot be trusted, the central database has to be, because it is the final authority on all information in hints and has a contractual obligation to those who register subspaces to provide the correct information. Even though the distributed database members cannot be trusted, at least some of the root members must be since they control the hints that enter the database. Therefore the root has a core set of trusted members maintained by the central authority that the central database communicates with exclusively. The root group may be larger than the trusted members, but the extra ones are not trusted by the central database.

There are two types of security attacks: external and internal. External attacks consist of outsiders trying either to disable a server or break into it. The former problem is handled through replication. The latter is a problem of outsiders trying to change the behavior of a member or modify hints. There is no danger in attackers reading them.

Attacks by members of a replication group are more insidious. They can exhibit several possible destructive behaviors, such as adding hints for fake subspaces that interfere with existing ones and changing existing hints. In this way someone could hijack a Web site's URNs to resolve to URLs for a fake site that looks like the real one to steal private information such as credit card numbers. Another type of attack is to delete hints, perhaps those belonging to a competitor. Finally, there are a host of problems related to a member simply misbehaving.

One precaution against security violations is to prevent as much ill behavior as possible. That which cannot be stopped can be isolated so its damage is minimized. Since hints are obtained from untrusted sources, each one is digitally signed by the central database, which has a well known public key, effectively making them read only and exposing tampering.

External denial-of-service attacks can be isolated through replication. If one member is disabled, the rest of its replication group will mask the attack. Similarly, the behavior of a malicious member can be isolated by recognizing its odd actions and possibly inconsistent data and removing it from the replication group. A member will have a hard time targeting specific hints because it cannot control which replication group it belongs to.

5.6 Summary

The distributed database is a physical implementation of a B-tree with modifications for distribution and replication. Replication is a technique that provides fault tolerance and protects against attacks, as well as a way to handle load. Resolution integrity can be assured through a chain of trust, starting with signed central hints that give public keys for verifying publisher hints and resolvers. The hints are extensible to allow for evolution. The tree structure of the database is independent of the hints that it contains, creating a scalable way to contain all hints without limiting their mobility.

Chapter 6

Implementation

We have created a simple implementation of the distributed database to experiment with algorithms, gain insight into how they work, and act as a proof-of-concept. It handles insertions and searches while allowing for the branching factor and caching parameters to be explored. It is not suitable for production, most notably because it is not optimized to store a large number of hints on each server or recover from failure, as all state, including hints, is kept in memory. The prototype does not deal with secondary, although no less important, features such as load balancing, digital signatures, tree operations, or optimal network topologies. However it does serve as a proof-of-concept for creating a scalable RDS solution.

The database is implemented exclusively in the Java programming language [10] to allow for quick development time. Java is an object-oriented language with notable features, such as garbage collection and a simple graphical user interface

API. Java's cross-platform nature also allows us to run the implementation on different platforms without having to recompile code.

Network communication is handled by Java's Remote Method Invocation (RMI) [11], a means of remotely executing Java object methods. RMI was chosen for the ease and flexibility it would allow while developing the functionality of the objects that implement the distributed database.

The distributed database was implemented with three major programs. The MemberPool Server keeps track of available unused members. The Member Server program runs a server that creates a given number of members and registers them with a MemberPool Server. Finally the DistDB program provides a way to interact with the distributed database.

6.1 MemberPool Server

The MemberPool Server provides a subset of the functionality of ADMIN by keeping track of unused members. Whenever a member becomes available it registers itself with its default MemberPool Server. A group leader that needs to perform a split gets unused members from its default MemberPool server.

The core of the MemberPool Server is a remote Java object, that is, one that can be called with RMI. Any member that wishes to join a member pool simply calls the object's join method. Similarly, any leader that needs an unused member calls that object's method for getting a member.

The MemberPool Server supports both local and remote monitoring of the pool. Figure 10 shows that simple text interface that allows an administrator to list the unused members.

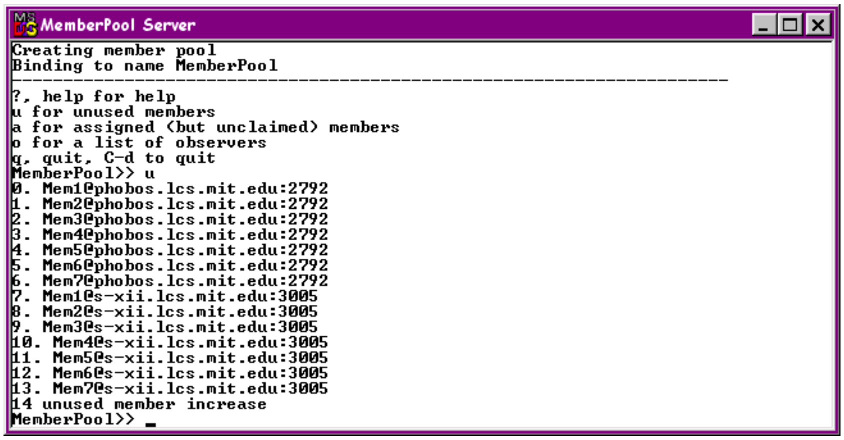


Figure 9. MemberPool Server Administrative Interface
The MemberPool Server has a menu-driven interface. The member pool shown here has fourteen unused members, seven supplied by a Member Server running on phobos.lcs.mit.edu and the other seven by s-xii.lcs.mit.edu.

MemberPool Server implements the minimum functionality needed to support the distributed database. An industrial strength implementation needs a more sophisticated way of choosing an unused member. It needs to consider the members in the destination replication group so that a group does not have two members from the same Member Server. Another criteria for picking an unused member needs to be the level of participation by each server. Finally if the MemberPool Server is out of members, it needs to retrieve underutilized ones from the database.

6.2 Member Server

The Member Server creates members and registers them with a MemberPool Server. Besides contributing members to the distributed database, it also provides a way for clients to resolve URNs.

The members themselves are instances of the remote MemberImpl class. This class has methods that implement insertion, searching, and other member behavior. The server creates members that are only suited for a small number of keys, which they store in memory. If the server crashes, the members have no means of recovery;

new members must be created. A production server would have a large number of keys stored on a permanent media and keep the most popular ones in memory. So as not to disrupt replication groups unnecessarily, after recovering from a short-term problem it should recreate each of its members, which then resynchronize themselves with their respective group leaders. A server that is unavailable for an extended period of time should create new members because the groups it participated in should have recovered from the loss.

The other function of the Member Server is to accept queries for URN resolution using a simple protocol. To resolve a URN, a client opens a connection to the server's port and sends the URN. The Member Server initiates a search at the member that is closest to the result. After searching the database the server responds with the result: an error, no match, or a central hint. The hint is just the name of a resolver. A search that results in a subspace returns the subspace as well as the member from which the hint came.

The Member Server also has a menu-driven interface for administrative tasks. An administrator can create more members, set the member pool, and, as shown in Figure 10, view each member's status. A member can be 1) unused, in which case it belongs to a MemberPool server, 2) living, meaning that it is an active part of the distributed database, or 3) dead, if it knows it has become disconnected from the database. The Member Server periodically checks the status of living members to see if they are still part of the database and changes their status to dead if they are not. It also reclaims dead members, returning them to the unused member pool.

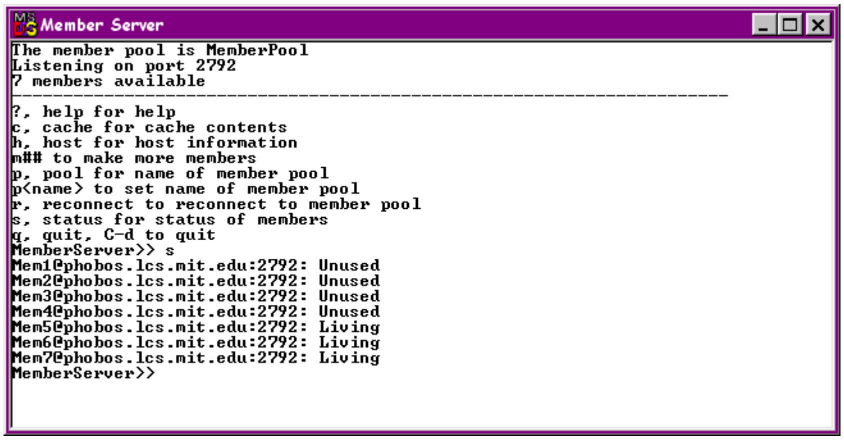


Figure 10. Member Server Administrative Interface
The Member Server has a text-based administrative interface that lets the user view the state of each member and set the MemberPool server. This Member Server has seven members, four unused and three part of a distributed database.

6.3 DistDB

The DistDB program lets the user experiment with a distributed database. The user can create a new database and insert hints singly or in bulk from a file. DistDB also allows the user to monitor the database by displaying each replication group in the tree and the subspaces it contains. This lets the user see the structure of the tree and explore the database's behavior in response to inserts. With DistDB, the user can control the branching degree and caching parameters of the entire database or of individual nodes.

Figure 11 shows a small database with a branching degree of three. DistDB displays each replication group as a box with its members followed by the subspaces it contains. The last line of the box is the range of the tree rooted at that group. The redness[1] of a member indicates its cumulative load, measured as the number of searches in which the it has been involved. The lines between groups show parent/

1. Redder members appear darker in a monochrome figure.

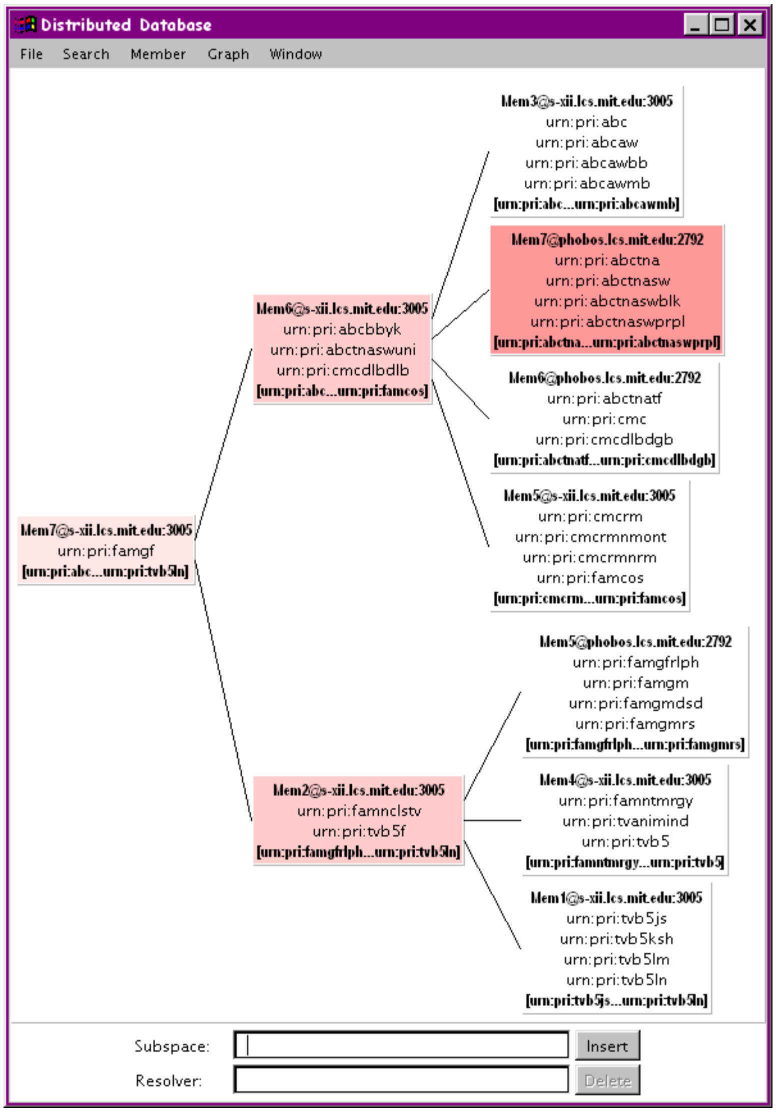
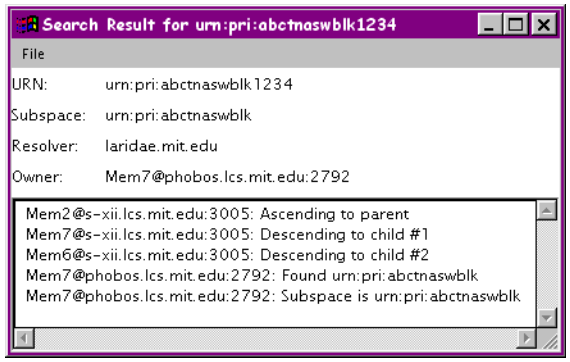


Figure 11. The Main DistDB Window

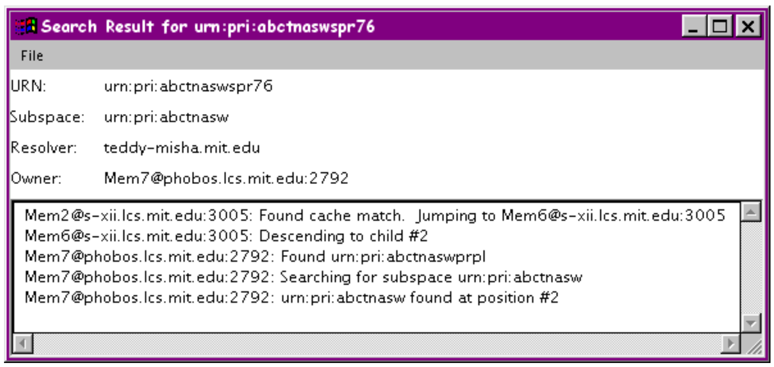
The main window shows the structure of the database and the contents of each replication group. This database has a branching degree of three and key-to-child ratio of one.

child relationships, so this database has height two with Mem7@s-xii.lcs.mit.edu:3005 at the root.

A search can be initiated at any member directly from DistDB as well as by contacting a Member Server. For example, the result of querying Member2@s-xii.lcs.mit.edu:3005 for the fictitious URN urn:pri:abctnaswblk1234 (see Figure 12, part a.) is that it falls into the subspace urn:pri:abctnaswblk over which the resolver



(a)



(b)

Figure 12. Searching For URNs

(a) The result of a search started at Mem2@s-xii.lcs.mit.edu:3005 for URN urn:pri:abctnaswblk1234 was found to be the subspace urn:pri:abctnaswblk at Mem7@phobos.lcs.mit.edu:2792. Its resolver was laridae.mit.edu.

(b) Mem2 cached the members and their ranges encountered during the search in a. A subsequent search for urn:pri:abctnaswpr76 was able to jump to Mem6@s-xii.lcs.mit.edu:3005, bypassing the root.

laridae.mit.edu has authority. The search began looking for an exact match. Since the URN did not fall within Mem2's range it progressed to the parent and from there down to Mem7@phobos.lcs.mit.edu:2792. It was not found there but the subspace directly less than it was the URN's prefix so it was the result of the search.

From this search, Mem2 learned about members Mem6@s-xii.lcs.mit.edu:3005, Mem7@phobos.lcs.mit.edu:2792, and their ranges. A subsequent search starting at Mem2 for urn:pri:abctnaswspr76, although not in the same subspace as the previous one, was able to jump to Mem6. When the search reached Member2 the exact match failed. The subspace directly less than the URN was

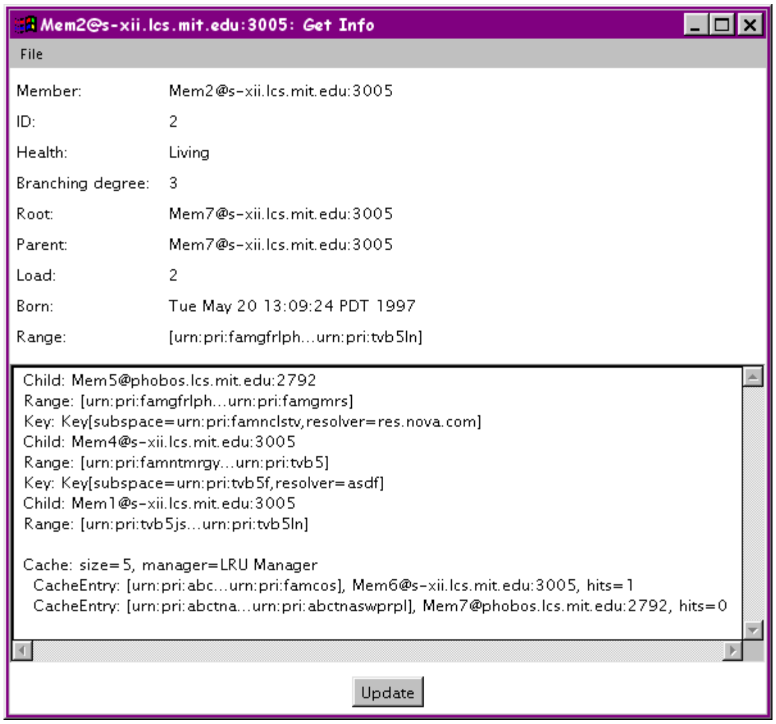


Figure 13. Information Window for Mem2@s-xii.lcs.mit.edu:3005
The information window shows the state of a member. Mem2's information window reflects the result of the two searches.

urn:pri:abctnaswprpl, not the subspace of the URN, but it shares the subspace urn:pri:abcrtnasw, which was found.

The state of any member can be examined, including its concept of its health, branching degree, root, parent, load, range, and date it joined the database. The information window (see Figure 13) also shows the member's hints, and, if it is internal, its children and their ranges. Additionally, the window shows the size and contents of the member's cache along with its replacement strategy. Besides examining a member's state, the user can alter the branching degree, cache size, and cache replacement strategy, allowing for experimentation and also fine-tuning of the database after it has been deployed.

6.4 Summary

The distributed database has been implemented as three programs. The Member-Pool Server tracks unused members provided by Member Servers. Each Member Server provides a configurable number of members and can accept URN queries from clients. The entire database can be managed with the DistDB program, which provides a graphical interface for inserting hints and searching for subspaces. It also allows the state of a member to be examined and modified.

This implementation was instrumental in the development of the insertion and searching algorithms. Using RMI as the means of communication was an easy way to extend the functionality of members, but the method invocation semantics, that of a single control thread across distributed objects, at times proved to be a hindrance, such as when a member fails, breaking the chain of execution. It also added complication to the bulk insert algorithm, which, unlike the single insert, requires concurrent operations. RMI also suffers from poor performance and resource utilization, keeping many TCP sockets open. Even though Java is supposed to be cross platform, RMI only worked reliably on computers running Microsoft Windows.

The distributed splitting algorithm is easy to get wrong. Replication will make it more resilient to failure, but it is still complex. The bulk insert is even more complicated and prone to failure. This implementation handles errors less than optimally. Problems need to be isolated and reported back to the user.

As a prototype, this implementation lacks a number of important features needed for a deployable version, such as the key-to-child ratio enhancement, replication (under development), the full hint format, and the resolution protocol. Other implementation details, such as secure member-to-member communication still remain.

Chapter 7
Conclusion

The RDS described here is sufficient for existing and future URN namespaces as long as they can be put in canonical form. Its structure and resolution mechanism promote longevity through a single, global registry. No single entity can control the distributed database, nor does the central authority have absolute control over all hints. While the computer industry changes rapidly, established components of the network, such as the DNS, have remained stable for a relatively long time. As a basic network service, this RDS is meant to be viable for many years and provide an evolutionary path both for future needs and to transition to a different system if the need arises.

The distributed database has been tested on a small scale but not on a large one. Distributed and replicated systems need to be simple because they require coordination amongst independently operating computers and thus are subject to more types of problems. Splitting a replication group is fairly complicated because it requires all pointers to change in a short amount of time, as opposed to other name services, such as DNS, where updates can proceed at a more leisurely pace. Errors need to be accurately reported so that the central authority can detect and deal with them. They are easy to detected and back out of for the single key insert but more difficult to not just report, but also isolate in the bulk insert because of the concurrency in the algorithm. While a B-tree does remain balanced, the cost may not be worth the risk. A regular high degree search tree, while not as optimal, may form the basis of a more robust, and simpler, distributed database.

7.1 Future Directions

There are a number of improvements that can, and should, be made to the distributed database to make it more appropriate for global deployment. First is that currently the minimum amount of data a member must hold is fixed. However, what this amount is cannot be determined without experience and may even change over time. Instead of splitting a server's capacity into fixed chunks, members should be of variable size. A replication group should be able to request the needed amount of capacity from ADMIN, which gets it from a server that has at least that much. This is especially important because we do not know the optimal member size, which may not be constant across replication groups. The central authority may wish to make some groups have a higher or lower branching degree or key-to-child ratio in order to deal with hot spots of activity.

Another area that needs work is the member-to-member communications protocols. RMI is not appropriate for an actual system because it limits the implementation to Java. A commercial-grade distributed database would have many implementations in many languages. The distributed database needs to use either a custom protocol or possibly one based on CORBA [5] that allows distributed objects to communicate regardless of implementation language.

Communication between members was specifically designed to avoid use of Internet multicast so that it could be immediately deployable. If worldwide multicast becomes a viable, commonplace means of communication then the structure of the replication group needs to be revisited. Much overhead, both memory and network, could be reduced through the use of multicast to keep track of group membership.

The RDS only touches on security, using replication and digitally signed hints to fight denial-of-service and ill-behaving member attacks. It addresses one way of providing some assurance that a URN is being resolved correctly, but security is an open issue and its role in the RDS needs further study. The integrity of hints needs to be assured so that URNs are resolved correctly and cannot be tampered with. How this will be achieved depends on the global authentication and authorization methods that become accepted. Like the rest of the URN resolution system, security is an evolving component and its role needs to be reevaluated periodically.

Finally, a deployable database will need better administrative tools than the rudimentary ones described in the last chapter.

7.2 We Live in Interesting Times

Many assumptions have been made based on rough numbers and ideas. There is a lot we cannot know until URNs are adopted by the public. There is little data on the number of URNs, users, and resolution requests that would result from a successful URN system. We also lack a sense of how URNs will be used and subspaces will be managed. What little data we have comes from the DNS and the Web, although neither can be measured well due to their distributed nature. After we have operational experience with URN resolution the assumptions in this thesis will need to be reexamined.

Work on URNs is ongoing in the Internet Engineering Task Force URN Working Group. While we have proposed herein one method of resolving URNs intended to be scalable and long-lasting, others have other ideas about what URNs are and how they should be managed. Only time will tell if URNs are adopted by the community at large, especially by companies that write client software and publishers that would use them.

References

[1] T. Berners-Lee and D. Connolly, *Hypertext Markup Language - 2.0*, Network Working Group RFC 1866, November 1995. Available from <ftp://ds.internic.net/rfc/rfc1866.txt>

[2] T. Berners-Lee, L. Masinter, and M. McCahill, *Uniform Resource Locators (URL)*, Network Working Group RFC 1738, December 1994. Available from <ftp://ds.internic.net/rfc/1738.txt>

[3] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, *Grapevine: An Exercise in Distributed Computing*, **Communications of the ACM**, 25(4), April 1982, pp 260-274.

[4] C. Mic Bowman, Peter B. Danzig, et al., *Harvest: A Scalable, Customizable Discovery and Access System*. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, revised March 1995. Available from <ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Jour.ps.Z>

[5] **The Common Object Request Broker: Architecture and Specification**, The Object Management Group, Inc., July 1996. Available from <http://www.omg.org/corba/corbiiop.htm>

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, **Introduction to Algorithms**, The MIT Press, 1990.

[7] R. Daniel and M. Mealling, *Resolution of Uniform Resource Identifiers using the Domain Name System*, Internet Draft, March 1997. Available from <ftp://ds.internic.net/internet-drafts/draft-ietf-urn-naptr-04.txt>

[8] P. Danzig, K. Obraczka, et al., *Massively Replicating Services in Autonomously Managed Wide-Area Internetworks*. Technical Report, University of Southern California, January 1994. Available from <ftp://catarina.usc.edu/pub/kobraczk/ToN.ps.Z>

[9] A. Demers, D. Greene, et al., *Epidemic Algorithms for Replicated Database Maintenance*, **Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing**, Vancouver, August 1987, pp 1-12.

[10] J. Gosling, B. Joy, and G. Steele, **The Java™ Language Specification**, Addison-Wesley, 1996. Available from <http://www.javasoft.com/docs/language_specification.html>

[11] **Java™ Remote Method Invocation Specification**, Sun Microsystems, Inc., February 1997. Available from <ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf>

[12] B. W. Lampson, *Designing a Global Name Service*, **Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing**, Calgary, August 1986, pp 1-10.

[13] R. Moats, *URN Syntax*, Network Working Group RFC 2141, May 1997. Available from <ftp://ds.internic.net/rfc/rfc2141.txt>

[14] P. Mockapetris, *Domain Names - Concepts And Facilities*, Network Working Group RFC 1034, November 1987. Available from <ftp://ds.internic.net/rfc/1034.txt>

[15] T. Newell, *What's in a Name?*, **InterNIC News**, 2(3), March 1997. Available from <http://rs.internic.net/nic-support/nicnews/mar97/>

[16] *Nielsen/CommerceNet Demographic Studies*, CommerceNet/Nielsen Media Research, Spring 1997. Available from <http://www.commerce.net/work/pilot/nielsen_96/>

[17] D. C. Oppen, Y. K. Dalal, *The Clearinghouse: A Decentralized Agent for Locating Named Objexts in a Distributed Environment*, Xerox OPD-T8103, October 1981.

[18] M. D. Schroeder, A. D. Birrell, and R. M. Needham, *Experience with Grapevine: The Growth of a Distributed System*, **ACM Transactions on Computer Systems**, 2(1), Febraury 1984, pp 3-23.

[19] K. R. Sollins, *Supporting Longevity in an Infromation Infrastructure Architecture*, **Proceedings of the 1996 SIGOPS European Workshop**, Connemara, Ireland, September 1996. Available from <http://www-sor.inria.fr/sigops96/papers/sollins.ps>

[20] K. R. Sollins, *Functional Requirements for Uniform Resource Names*, Network Working Group RFC 1737, February 1995. Available from <ftp://ds.internic.net/rfc/1737.txt>

[21] K. R. Sollins, *Requirements and a Framework for URN Resolution Systems*, Internet Draft, March 1997. Available from <ftp://ds.internic.net/internet-drafts/draft-ietf-urn-req-frame-01.txt>

[22] K. R. Sollins and J. R. Van Dyke, *Linking in a Global Information Architecture*, **World Wide Web Journal**, 1(1), November 1995, pp 493-508. Available from <http://www.w3.org/pub/WWW/Journal/1/e.193/paper/193.html>