The Use of the Domain Name System for Dynamic References in an Online Library

by

Ali Alavi

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

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author ....
Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by ....
Jerome H. Saltzer
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

The Use of the Domain Name System for Dynamic References in an Online Library

by

Ali Alavi

Submitted to the Department of Electrical Engineering and
Computer Science

May 16, 1994

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

Abstract

Persistent, dynamic references (or links) to remote documents are an essential part of an online library. This thesis examines two distributed database systems, X.500 and the Domain Name System (DNS), upon which to build dynamic references. DNS was chosen and was used to design a model and build a sample dynamic reference system. This system seems to exhibit the scalability, robustness, usability, and efficiency necessary for building global distributed online libraries.

Thesis Supervisor: Jerome H. Saltzer
Title: Professor, Department of Electrical Engineering and Computer Science

List of Figures

Table of Contents

Acknowledgements

I would like to thank:

Jerry Saltzer, my thesis advisor, who gave me feedback in sickness and in health, and without whose guidance I would not have been able to finish.

Mitchell Charity, Mr. All-around Guru, who helped generate many of the ideas in this thesis. He deserves a lot of credit for this thesis.

Susan, *Trevor*, *Jeff*, *Nicole*, *Sahana*, *Damon*, *Jahanvi*, *Ish*, *Melissa*, *Yuk*, and *Upendra*, who helped me stay sane by helping me procrastinate.

and of course, *mom* and *dad* for footing most of the tremendous bill!

This work was supported in part by the Corporation for National Research Initiatives, using funds from the Advanced Research Projects Agency of the United States Department of Defense under CNRI's grant MDA972-92-J1029.

Chapter 1

Introduction

1.1 Motivation for the Library 2000 project

With the advance of technology, computing resources are becoming more and more affordable everyday. Magnetic storage devices such as hard disks are becoming very cheap and reliable. CPU speeds are increasing even faster than the speed at which their prices decrease. High network bandwidths are becoming more commonplace. High resolution displays are becoming reliable as well as affordable. As these trends continue, it will soon be economically feasible to place entire libraries online, accessible from workstations located anywhere. A more complete discussion of these trends can be found in a paper by Saltzer [5].

In one component of the Library 2000 project at the MIT Laboratory for Computer Science, five universities (Cornell, CMU, Berkeley, Stanford, and MIT) are currently exploring the issues involved with placing such a large volume of data online in a reliable, accessible manner. The issues range from proper replication and redundancy to scan and display issues, from copyrights to search and lookup technologies. I have chosen to explore a small subset of these issues, dynamic references, which will become evident in the following section.

1.2 Motivation for my work

Imagine that you are sitting at your computer. You have spent the past hour searching various databases with several combinations of key words. Finally, you find a reference that looks promising. As you read the plain text abstract, you realize that this journal article is exactly what you have been looking for. You happily use your mouse to click on a button instructing the computer to fetch the article. Your anticipation mounts as the pointer turns into a wristwatch icon. In a few moments the pointer returns to normal. Suddenly, instead of your article, a window pops up saying “error: cannot find article.” You swear under your breath wishing the computer had a neck you could strangle...

1.2.1 Static links are easy, but fragile

Unfortunately, the above scenario is all too common. Today, there are a variety of hypertext languages and applications which allow one to create links in and among documents. Perhaps one of the most popular of such programs is NCSA’s Mosaic, a World Wide Web client that allows users to create documents with links to others around the world. The links are very similar in nature to the references we would need in an online library. In an online library, we would have sets of information, such as cards in an electronic card catalog or citations at the end of a journal article, which refer to a document. The "reference" is the part, or subset, of that information containing all that is necessary for locating the document[1]. In Mosaic, link specifies the mechanism for locating the document.

Despite their similarities, however, there is one major difference. Currently, links in Mosaic are static. When the document is created, a part of it is linked to a specific URL (Uniform Resource Locator) which points to a very specific filename in a specific format through a specific protocol on a specific server on the internet. If any of those specifics changes even slightly, the link stops working, and once again we get the urge to strangle the computer. Such urges are tolerable when we are only playing games with Mosaic, but they are unbearable when trying to do real research in real time at a library.

1.2.2 Changes will be necessary and will happen often

Why do any of the specifics need to change and just how much of a problem is it? Today, technology is evolving quite rapidly. The file format and protocol, both of which may have been designed to last many years, may be swapped for newer ones. Even though we do not expect this to happen very often, we must be prepared to adapt to such changes. On the other hand, the computer and physical storage media will become obsolete and need to be replaced or updated every two to four years. Also, the maintainers of documents are likely to change themselves as the demand for such services grows, and new companies appear and disappear. Once again, we must be prepared to handle such changes.

---

1. The concept of a "reference" is explained further in section 3.1.

Therefore, it is likely that all references will need to be updated *at least* every five years. As the number of online documents--in addition to other digital objects--increases exponentially, so will the number of changes which must be propagated.

1.2.3 Propagation is not easy

How would we actually go about propagating these updates? In order to actually update a reference, document maintainers need to actually keep track of all the references to their documents *and* have a method for updating them. This adds quite a bit of administrative overhead. A simple way to implement this would be to require libraries to register with document maintainers and provide an email address for changes. The document maintainers would then maintain a list of subscribers on a per document basis, and automatically generate email when references change. Once a library receives such a message, its staff can update the reference in their local system. Since different libraries use different methods of maintaining references, automating this process will require a substantial amount of work at most libraries. Also, the administrative overhead at the document server side could be quite large.

1.2.4 The scale is too big to propagate changes

Even if a reference could be updated efficiently without significant overhead, the number of updates required is prohibitively large. "The ArticleFirst online database consists of nearly two million bibliographic records representing virtually every item [in the past five years] from the table of contents of over 8,500 journal titles."[1] Using these figures, we can approximate MIT's collection of 21,000 journals at about 1.15 million references per year. If we need to update 30 years worth of MIT's journals in five years,

1. From information on FirstSearch.

that averages to an update every five seconds. It is easy to see that updating all the references to all the online documents in the world would be impractical if not impossible.

1.3 Proposed Solution

Since propagation of updates is impractical, we propose a different approach. Instead of propagating changes to all references to a document, we will make the references dynamic such that they can find the current location of the document at lookup time. This will be accomplished by querying a global database of references for the current location of the file. Using this approach, we need only to update the database once and all references automatically change. In exploring this approach, I examined two distributed database systems, DNS and X.500, to see if either would be suitable for building our system upon. DNS proved to closely match our design requirements. Therefore, I used DNS to implement dynamic references to scanned Computer Science Technical Reports.

The remainder of this work is divided into five chapters. Chapter two describes the two systems, DNS and X.500, which were investigated as possible foundations for our system. It also examines the design criteria as well as the reasons why DNS was chosen. Chapter three describes the model for our dynamic references as well as the design of the semantics of the system. Chapter four describes our implementation. Chapter five analyzes our use of DNS. And finally, chapter six concludes and lays the foundation for future work.

Chapter 2

Background

This chapter describes the two systems, DNS and X.500, which I have explored in search of a suitable foundation upon which to build our dynamic references. Then it describes our design goals which led to choosing DNS.

2.1 The Domain Name System

The Domain Name System, or DNS, is a distributed database used by almost all machines on the Internet to map machine names to IP addresses (among other things). In other words, DNS allows users to use a human-friendly name to refer to a certain machine. This machine could be moved to a different IP address, but as long as the entries in the machine's local name server were updated, no other propagation of the change would be necessary. Everybody would still be able to use the same human-friendly name to contact the same machine, even though the address had changed. In other words, DNS localizes the scope of control to segments of the database, while allowing access to the entire database.

2.1.1 The purpose of DNS

In order to better understand DNS, it is important to know why it was created. Back in the 1970s, the ARPANET (the precursor to today's Internet) was a small community of a few dozen hosts. There was no DNS, so a single file, widely known as HOSTS, was used as a host table. Whenever a machine was added, everybody's HOSTS file had to be updated. It was maintained by SRI's Network Information Center and distributed by a single host. Network administrators would typically email updates to the NIC and would periodically ftp an updated HOSTS file.

As the population of the network grew, so did the problems with HOSTS. The first problem was that the network traffic and processor load on the SRI-NIC host grew to unmanageable proportions. The second problem was that since it was one flat file, name collisions became a problem as people added more and more hosts to the network. And finally, there was a consistency problem. By the time an updated version of HOSTS propagated to the ends of the network, a newer version was already on the SRI-NIC host.

The basic problem was that the HOSTS scheme did not scale well. That is why DNS was developed. It is distributed in order to relieve the network and processor loads. It is hierarchical to solve the name collision problem. It is locally administered and managed to solve the consistency problem. Let's discuss these features, one at a time.

2.1.2 How DNS works

The key to DNS is its hierarchy. The Domain Name System is structured in the form of a tree, similar to the Unix filesystem. Each node of the tree represents a partition of the entire database, or a domain. Each domain can be further partitioned into subdomains (like directories and subdirectories). In DNS, each domain can be administered by a different organization, and within each domain, subdomains can--but do not need to--be delegated to other organizations (see figure 2.1).

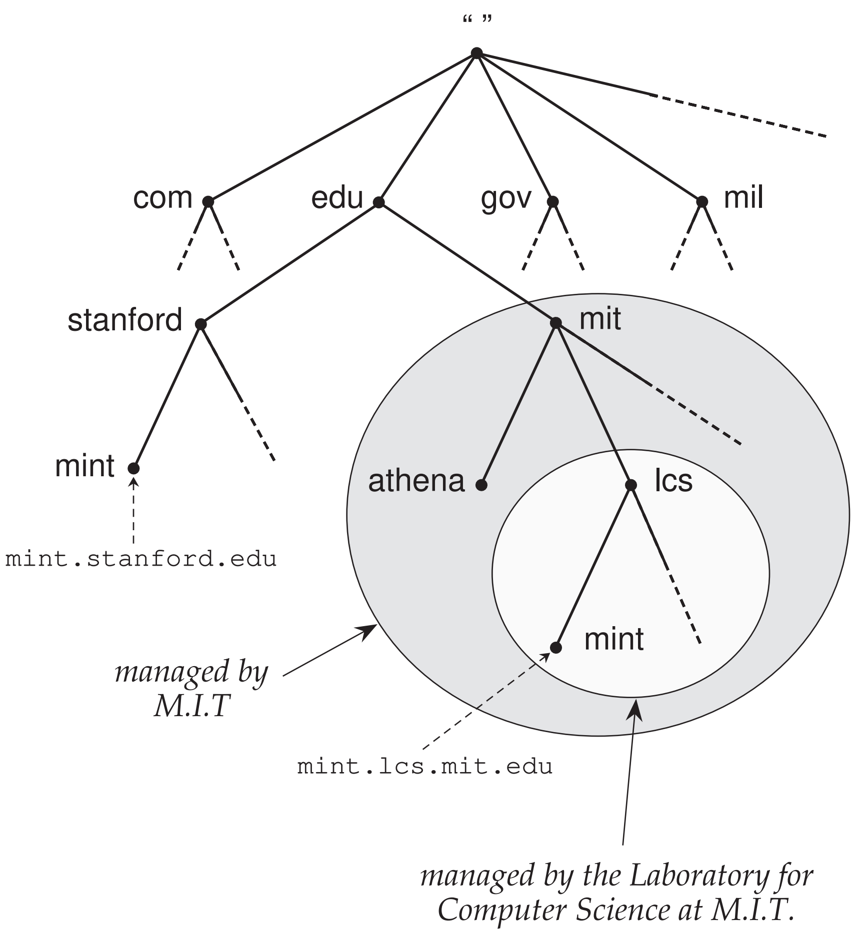


Figure 2.1: Hierarchy in the Domain Name System[1]

For example, the Internet's Network Information Center runs the `edu` (education) domain, but delegates `mit.edu` to M.I.T. Then M.I.T. delegates `athena.mit.edu`

1. Adapted from figures 1.3 and 2.5 of Albitz and Liu [4].

and `lcs.mit.edu` to different organizations within M.I.T. Therefore, when `mint.lcs.mit.edu` is replaced by a new machine and given a new IP address, only the `lcs.mit.edu` server needs to be updated by its administrator.

This hierarchy solves not only the network and processor load problem with updates, but it also solves the name collision problem. Administrators for each subdomain need only prevent name collisions in their own subdomains, and the hierarchy ensures uniqueness. Therefore, many machines can be named `mint`, but `mint.lcs.mit.edu` is different from `mint.stanford.edu` (see figure 2.1).

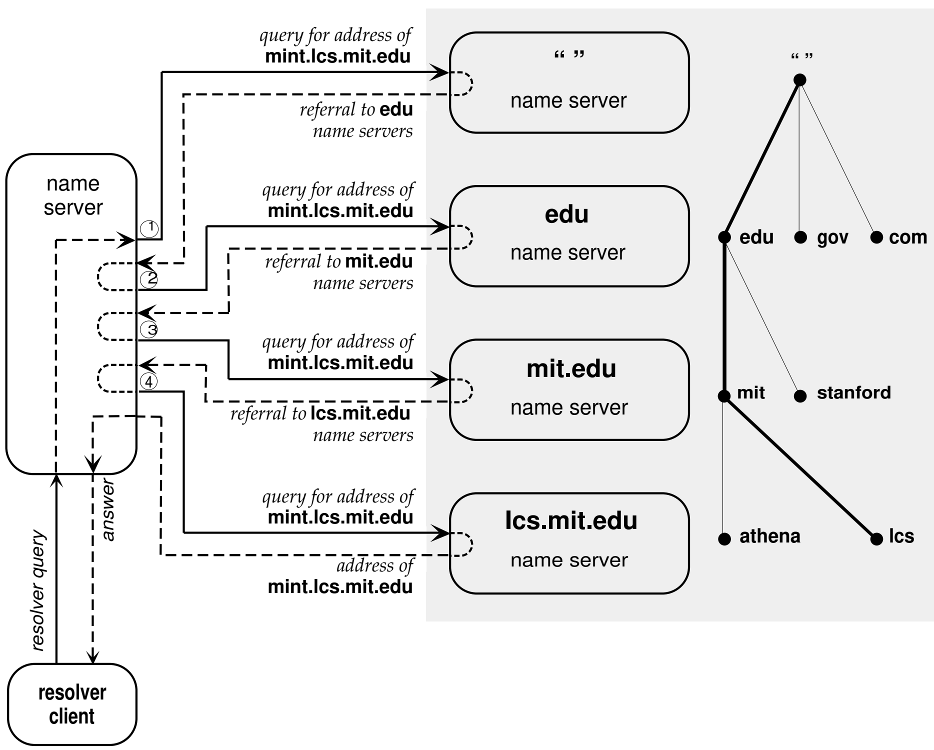


Figure 2.2: Recursive queries[1]

Possible inconsistencies are eliminated by referring all queries for a given DNS name to the same server.[2] In other words, regardless of which part of the world queries are

1. This figure was adapted from figure 2.10 of Albitz and Liu [4].
2. Except in caches. But this is an implementation issue which is discussed later.

initiated from, the responses will always be consistent. When a user makes a query to DNS, a resolver (the DNS clients) will query the local name server which in turn recursively queries name servers in the hierarchy in order, beginning in principle at the root[1] (see figure 2.2).

For example (see figure 2.2), when trying to find `mint.lcs.mit.edu`, the local name server queries the root server (denoted by " ") first (1). The root server returns the address of the `edu` server. The local name server then queries the `edu` server and gets back the address of the `mit.edu` server (2). The local name server then queries the `mit.edu` server and gets the address of the `lcs.mit.edu` server (3). Finally, when the local name server queries the `lcs.mit.edu` server, it receives the address of `mint.lcs.mit.edu` (4), which it then returns to the resolver. Using this method, at any given time, all resolvers in the network receive a consistent result, because they all get their information from the same server (in this case, from `lcs.mit.edu`).

2.1.3 DNS reliability and performance

In addition to this basic model, two additional features make DNS more robust and efficient. In the model described so far, if a certain name server dies, then no one from the outside world can access any hosts in the server's domain nor any of the subdomains under that server. In other words, if *any* of the nodes in the tree fail, so will the entire subtree under them. To remedy this problem, DNS uses replication. Each name server has at least one replica. Therefore, if one server goes down, there is at least one other that would still be operational. This replication also helps spread the load on a given server. There are, for example, seven replicas of the root server.

1. Most implementations use caching and hints to avoid a complete hierarchical search on every lookup, but the basic concept remains the same.

The main method DNS uses to improve performance, however, is caching. Each server has a cache to store the values it receives from other servers. For example, when looking up `mint.lcs.mit.edu`, the server caches the addresses of `edu`, `mit.edu`, `lcs.mit.edu`, and `mint.lcs.mit.edu`. Thereafter, if we query the DNS server for `rr3.lcs.mit.edu`, the server already has the address of `lcs.mit.edu` and asks that server for the address of `rr3`. And even more importantly, if we need to access `mint` again, the local server has the address cached, obviating the need to make four separate server queries.

2.1.4 Limitations

As always, there some limitations are introduced when going from theory to implementation. One limitation of DNS is that it only allows a very limited set of record types. Since it was designed to allow host lookups and email routing, DNS is a rather minimal system. It supports less than a dozen (varies from one implementation and version to another) record types, and the generic TXT record was only added in the past 5 years. Also, DNS is simply a key-value mapping; it has no real search capabilities.

The other reality check occurs when DNS uses caches to increase efficiency. If a cached item changes at a remote location, the local machine will continue to use the cached data. Therefore, we *can* have inconsistent views of the database. Rather than trying to propagate changes to caches, each DNS server also suggests a timeout value for its data on a record by record basis. Therefore, if another host is caching this server's data, the data will expire after the timeout. Of course, setting the timeout too low would defeat the purpose of having a cache. Therefore, we have a performance-consistency tradeoff that all maintainers of subdomains must calibrate to their own specific needs.

This discussion of DNS was purposefully kept short. For a more comprehensive and detailed discussion of DNS see Albitz and Liu [4].

2.2 X.500[1]

X.500 is a CCITT protocol designed for building a distributed, global directory. As such, it has some similarities, as well as some major differences, with DNS. Like DNS, X.500 is distributed and locally maintained. Unlike DNS, X.500 has powerful searching capabilities built-in. To better understand X.500, once again, we must examine why it was created in the first place.

2.2.1 Motivation for X.500

With the acceleration of industrial, scientific, and technological development, it has become increasingly likely that two or more geographically distant people could be working on the same problem. The growth of communication technologies has alleviated the problem of being able to collaborate over great distances, *provided that the parties involved already know how to reach one another*. Hence the development of directory services. There are various models of directory services, which we will now discuss to show some of their deficiencies and some solutions provided by the X.500 standard.

The first model is, of course, is the local telephone company's directory services. They maintain a database of people with phone service with their names, phone numbers, and addresses. Though invaluable, it has some limitations. For example, you can only find someone's number if you know their name *and* city. If there is more than one John Smith in the same part of a city, there is no further information upon which to select the correct number. Also, the phone book can be up to one year out of date. And finally, you have to

1. Much of the material in this section was derived from RFC's 1279 [3], 1308 [7], and 1309 [8].

call Directory Assistance in a given area code to get information for that area; you cannot call a single number consistently.

There are also a variety of directory services available on the internet. First, there is the finger protocol, which has been implemented for UNIX systems and a handful of other machines. This protocol allows you to "finger" a specific person or username at a specific host running the finger protocol. This returns a certain set of information which usually includes the user's name and phone number, in addition to other information the user wants to broadcast. Once again, there are some limitations. First, you must already know on which machine to finger the person. Second, finger is disabled on some systems for security purposes. Third, people not running UNIX (a majority of computer users) will most likely not be listed. And, finally, there are no search capabilities. There is, for example, no way to search stanford.edu for all people working on X.500 (even if the people provided this information).

The next directory service is the whois utility. Whois is available on a variety of systems and works by querying a large central database. It has all the problems of a large, centralized database. Just like HOSTS, the one-machine bottleneck is a problem as is the updating of information (changes need to be emailed to a "hostmaster" and manually reentered into the database).

Finally, DNS is also a flavor of directory services. Built mainly for fast host name to IP address mapping, it therefore has very limited search capabilities. Though effective for the purpose it was designed for, its lack of search capabilities prevents DNS from becoming a rich Directory Service.

2.2.2 Features

X.500 was designed to address some of these problems. These capabilities include:

- Decentralized Maintenance: Each site running X.500 is only responsible for its local part of the database; therefore, updates are fast and collisions are avoided.
- Powerful Searching Capabilities: X.500 allows users to construct arbitrarily complex queries.
- Single Global Namespace: Similar to DNS, but the X.500 namespace is more flexible and expandable.
- Standards-Based Directory: Since X.500 is a well-defined standard, other applications (such as email and specialized directory tools) can have access to the entire planet's worth of information, regardless of specific location.
- Security Issues: X.500 defines both Simple Authentication (using passwords) and Strong Authentication (using cryptographic keys) for accessing data. It also supports Access Control Lists on a per-attribute basis.
- Replication: X.500 has partial specifications for replication of data, with a simple implementation currently in place.

2.2.3 limitations and problems

X.500 is not a very tight standard. On the contrary, it is trying to define a standard that would be enough things to enough people so that most people around the world will use it to build directory services. Although this standard is useful, it also makes the structure somewhat amorphous. For good directory services, this flexibility to shape information into any form is important, if not essential. However, this flexibility also causes a decrease in performance. The complex search functions give us access to information that we never would have been able to find before; but if we are looking for the same simple piece of information, the process can be significantly slower.

2.3 Design Goals

Now that we have described the two candidate systems, we must define the decision criteria. In order to do this, we must examine the design goals of our dynamic reference system.

2.3.1 References must be persistent and dynamic

The basic goal of this project is to design an architecture for references which would, at the time of invocation, dynamically find the object to which they refer at lookup time. Such a reference would theoretically never be out of date and would stay valid as long as the object to which it refers remains valid.

Such persistence of references is a significant concern in building online libraries. Libraries have large collections of materials, often from many different geographical locations. In an online library, such materials would remain in their faraway locations. When users want to look up specific documents, they would be able to simply ask for them, and the software will fetch the image of the documents by following the references held at the local library. If these references were to fail often, the library would become almost useless.

With the passage of time, these documents will be moved onto newer storage devices, new locations, and even acquire new names, thereby requiring updates for their references. As the number of online documents and online libraries grows, it becomes increasingly important for the references to be dynamic.

2.3.2 Scalability

In designing this system, we need it to be more than just dynamic references. It must also be inherently scalable. With the exponential proliferation of online documents, if our system were not scalable, it would soon go the way of HOSTS.

The first part affected by scaling is the update process. It would certainly be very inefficient, and soon completely impractical, to update all the references to a given document in many libraries all over the world when a change occurs. Therefore, we need a localized system which allows updates to be done locally, while affecting all of the references to that specific document. Such a system would not be restricted by scale because it would keep the scope of updates at a constant, manageable scale.

Such a system would also alleviate propagation delay and reference inconsistency problems. As we saw in the case of the HOSTS file, as the size of a system grows, the propagation of changes slows to the point that different parts of the system will present inconsistent views. If we could eliminate the need for the physical propagation of changes, we would also eliminate these inconsistencies.

2.3.3 Usability

In addition to being scalable, this system must also be usable. Without this attribute, building the system would be pointless. For it to be useful, people must use it. People will not want to use it if it is too much of a hassle to set up and use. We do not want to try to convince users to change everything just so that they can have dynamic references. Therefore, it would be highly desirable to make maximum use of what is currently widely installed, thereby minimizing additions.

Not only should the setup be minimal, it is also important for the system to be easy to use. There are three sets of users of our system. First, we have the library patron who simply wants to find a document quickly. It goes without saying that these users should not need any special knowledge to follow references to a document. The point of putting a library online is to make it easier to find documents, not to find a new way to make it difficult.

The second set of users is the maintainers of the documents who need to actually make the updates. These updates must be easy to perform. The user should not be forced to check with several other “authorities” to make an update. Also, it should not take an inordinate amount of time. In other words, it should not be harder nor more time-consuming than it is in a nondigital library.

The ease of use for these two sets of users is only partially dependent on the underlying system. The greater part is actually dependent on the third set of users, the programmers of the library search and update systems. If the dynamic reference system is well-designed semantically, the programmers should be able to implement their library in such a way as to make it relatively easy for the first two sets of users to use. However, it may not be an easy task for them. Therefore, a goal of our system is to be based on something that programmers already know or use, thereby minimizing the time and anguish in learning new systems.

2.3.4 Reliability

Another important design goal is reliability. We would not be successful in convincing users to adopt a system that works only part of the time. Therefore, robustness of our implementation is a crucial design consideration.

In a system built from scratch, it is easier to guarantee robustness--at least we would not be able to blame anyone but ourselves. However, we would like to not reinvent the wheel but build upon a system already implemented. If feasible, this would allow us to concentrate more on the higher level design issues and less on the lower level implementation details. Unfortunately, it also means that if the system we were basing ours upon is not robust, our system would also be prone to failure. Therefore, we must choose a system which has a robust implementation.

2.3.5 Speed

Finally, our system needs to be fast. We have already discussed the speed and efficiency of updates, but there is still another concern we have not addressed: the speed and efficiency of use. When a library patron performs a search, it should not take hours for the result of the search to appear--at least we do not want it to be our fault. We would, therefore, like the reference lookup speed to be as fast as possible. Since we are building our system on top of an already existing one, we need to choose an underlying system which is fast. Speed often comes from minimizing overhead. Therefore, we would like both our system and the system we build upon to require minimal overhead.

2.4 DNS Chosen

Given those design goals, we chose the Domain Name System as the underlying architecture for our dynamic references. This decision was based on several factors. Both DNS and X.500 had the hierarchical, distributed, and localized nature necessary for the implementation of our references. However, two main factors made DNS the correct base for our architecture.

First, semantically, DNS is much simpler than X.500. Since it is more flexible and powerful, X.500 also becomes more complex and slow. X.500 has flexible search capabilities which, while great for directory services, are not necessary for our system. We wanted our system to be as small and fast as possible. Therefore, we chose to include only the essential functionality, allowing others to build their individual complex capabilities on top of our system. In this way, we are allowing other designers maximum flexibility and efficiency.

The main reason for the decision, however, was that DNS is already widely used and installed all over the world. This is important for several reasons: first, we were not forced to implement a name server and try to make it robust. Other people are already actively working on that task. Second, we did not have to distribute it and convince people to use it--DNS is already on most of the machines on the Internet. Finally, we did not have to provide the necessary infrastructure to make our system work. The NIC is taking care of the root servers, and the rest of the world is taking care of their local domains. In other words, everything else we need is already in place.

This practical ubiquity, in addition to seeming to possess the necessary features, made the Domain Name System the obvious choice. The next chapter describes the model for our system and how DNS fits into that model.

Chapter 3

System Design

In this Chapter, I will describe the model we used for our system and examine the design of our system based on this model.

3.1 The Model

We walk into a library, walk up to the card catalog, look under subject, author, or title, and with some luck, find something that looks promising. Then we write down the Dewey Decimal number we find on the card, and go to the stacks to find the book. In most libraries nowadays, this card catalog has been computerized. Now, instead of walking up to a card catalog, we walk up to an ASCII terminal and search the online catalog much faster and more reliably than its paper ancestor. In many places, the online catalog contains references to several libraries' holdings and will even tell us if a certain book is checked out. Today, that is where the role of the computer ends. In the Library 2000 project we are expanding the role of the computer into document recovery as well. We would like the reference, or electronic "card", to also indicate whether we can access the document online, and if so, provide a means of fetching and displaying it.

What exactly do we mean by a reference? In general, a reference is anything that refers to something. In this thesis, we are interested in references which refer to documents. Examples of such references in a library consist of the cards in the card catalog and the bibliography, citations, and references in books and journals. For our model, we have simplified the notion of reference materials to the collection of information in the electronic card which refers to a document. We assume that any other form of reference material can be used to find the corresponding electronic card. In this project we are interested in the part of the electronic card which links us to the actual document. This link is called the "reference." This reference provides all the information necessary for us to access the actual document.

3.2 What do we put in DNS?

Once we decided to use DNS to implement our model, we needed to figure out how. In other words, what exactly do we store in the distributed database? Every item in DNS is a mapping between a name and a value, therefore, we needed to design those two parts.

3.2.1 The Uniform Resource Name

First, we designed the name. This is the unique name of a given document and is referred to as the Uniform Resource Name, or URN, of the document. Using this unique URN, we can find the document it maps to. We must guarantee uniqueness of these names so that there is an exactly one-to-one mapping of names to values (i.e. each URN must map to exactly one document). In order to do so, we take advantage of the hierarchical nature of DNS. Due to this hierarchy, we only need to guarantee uniqueness at the leaf nodes of a given branch (see figure 2.1). Also, since we can arbitrarily subdivide branches, we can always keep the leaf nodes of a given branch at a reasonable number, thereby making it easier to guarantee uniqueness.

So what does a URN look like? In our system it is composed of two parts: the document maintainer's name and the document name unique to that specific document maintainer. Suppose that Computer Science Technical Report number TR-93-123 is maintained at the MIT online library, which is itself part of the "library" domain in the United States. The the name could be:

$$\underbrace{\texttt{123.93.TR.CS}}_{\textit{local unique name}}\texttt{.}\underbrace{\texttt{library.mit.us}}_{\textit{maintainer's name}}$$

Figure 3.1: A Sample URN

Even though in figure 3.1 both parts look the same, making the URN look like one, long, uniform DNS name, it does not need to be that way. In fact, we chose a two part name so that document maintainers can have the flexibility of naming their documents in any way they please (as long as it is a valid DNS name), without being restricted to a specific naming convention which will be necessary for the names of the maintainers. For more information on URN's see Weider and Deutsch [6], and for more information on general naming issues, see Yeo, Ananda, and Koh [9].

3.3 The Uniform Resource Locator

The second part is the value associated with the URN. The first question is, what does this value contain? Since we are using DNS to map names of documents to the online version of the document itself, we needed the value to actually point to the document. This yields two more questions: What do we mean by “point to” and what do we mean by “document”?

We are using DNS not only to find a document, but also to tell us *how* to “get” it. It is not strictly necessary for the DNS value to include more than just *where* a document is located. However, since other people have already devised a widely used method of specifying both the where and the how, we chose to use it. This descriptor is called the Uniform Resource Locator, or URL, and is used in the rapidly expanding World Wide Web. URL’s specify what protocol to use, in addition to where to get the file (see figure 3.2). For a more comprehensive discussion of URL’s, see Berners-Lee [1].

$$\underbrace{\texttt{http:}}_{\textit{how}}\underbrace{\texttt{//www.mit.edu:8001/dilbert.html}}_{\textit{where}}$$

Figure 3.2: A sample URL

The second question is not as easy to answer. What do we mean by an online document? What specific set of bytes should our URL point to? This may be a relatively easy question if the document is relatively small and stored as a single text file. However, when a document is stored in image form, it may be composed of hundreds of images in different resolutions and formats, yielding many Megabytes. In this case, it is not very clear what the URL should point to. Since this is a difficult question, we created the abstract notion of a Document Descriptor which contains information about the pieces of the document. An example of a minimal Document Descriptor would be an interface with a list of all the pages of the documents, each with a link to the actual image. A more fully developed Document Descriptor would probably contain some of the document’s bibliographic information, as well as the image formats in which the document is available, among other things. The implementation of the Document Descriptor has been left for future theses.

3.4 How do we use DNS?

Once we know what mappings to keep in DNS, the question becomes how to use DNS. Who fills the database? How does a typical user find a document?

3.4.1 Filling DNS

The first step is to enter the URN to URL mappings into DNS. One seemingly reasonable way to accomplish this is to require the organizations that maintain online documents to also maintain DNS subdomains with the mappings to the documents which they maintain. Of course, this responsibility could also be passed onto third parties. The only important part is for the maintainers of documents to be able to easily update the DNS mappings of their documents when the information in the URL portion changes or a new mapping needs to be added.

3.4.2 Using DNS

What does the client program do? The purpose of the client program is to allow users to find the document(s) they want. We assume that it already has the capability of searching databases (local or remote) of bibliographies to find the appropriate reference. Currently these references contain information such as the title, author, and date of publication as well as many others. For documents available online, these references should also contain enough information for the client program to be able to generate the corresponding URN. The easiest way to do this would be explicitly to place the URN in the reference, thereby allowing the client program to simply copy it. It might also be possible to materialize the URN from the information found in separate fields of the reference, using an appropriate transformation. The client program can then use the URN to query DNS for the URL. Once it has the URL, it can pass it to a document display program which will provide the interface for the user to view the document. (see figure 3.3) Therefore, DNS is only used for the URN->URL mapping and not searching for, actually fetching, or displaying the document.

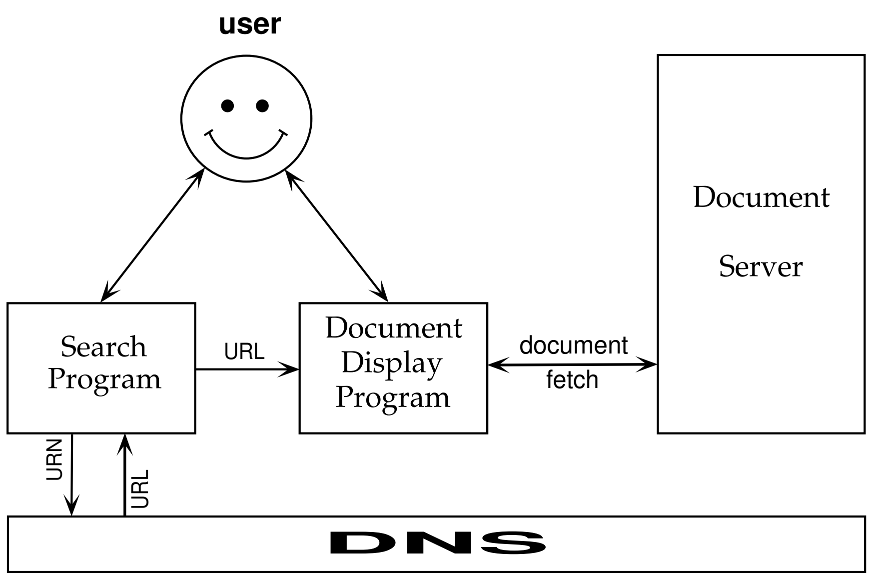


Figure 3.3: The model of our system

The following chapter describes our implementation based on this model.

Chapter 4

System Implementation

The design of the semantics is only the first step to building a system. Implementation details often cause changes or refinements of the semantics. Also, in this case, the semantics define a system much larger than can be currently built. Hence, we chose to implement a small sample online library providing access to Computer Science Technical Reports. It is our hope that lessons learned from this experiment can be applied to building a much larger global library system.

4.1 Modification to the Reading Room catalog

My system was based on an experimental catalog interface built on top of Mosaic, by Mitchell Charity. His interface allows the user to specify a search string for which to search the database. The result of the search is a list of the bibliographic records in the catalog matching the search string. I took this system and added DNS capabilities. Now, once the search engine finds the references the user was searching for, it provides a means of reaching the documents, if they are online (see Appendix A). I will now discuss my additions and the issues involved.

4.1.1 How do we know if a document is online?

Ideally, everything is online once it is published, so that if you have a reference to it, you know it is online. However, it is not certain if that will ever be the case. Therefore, we need a method of checking to see if a given document is actually online. Or do we? We could simply assume everything is online and if our document is not online, the lookup fails and we get an error message, indicating that it is not online. This could be a valid implementation if we can guarantee that such failures would be rare.

However, as we are only beginning to maintain documents online, the rare case is when a document *is* online. Therefore, as we would like to minimize failures, it would be wise to check to see if the references we present to the user are actually online. The question then becomes where do we check?

One method would be to keep that information in the Bibliographic records that the search engine checks. This approach would make it very easy to check to see if a given document is online. Unfortunately, this is not a viable solution. Actually, it is not a solution at all, because it causes problems in another part of the system. How does the Bibliographic record get updated? How would those updates get propagated? We would need a whole separate DNS-like database for that! Unfortunately any update system for the bibliographic records would either be redundant or not as current as the DNS database.

A better solution would be to simply query the DNS database for the given documents and if a URN->URL mapping exists, then we know that the document is probably online. We say “probably” because it is possible for the DNS to be slightly out of date or the document server may be down. In order to reduce this possibility for error even further, the search program could also try to fetch the actual document to see if it is in fact accessible. On the semantic side, we cannot check for all errors. Even if we check the Document Descriptor, the pages of the document may be corrupted or not available, or the Document Descriptor itself could contain erroneous information. On the implementation and practical side, it would needlessly slow down the system. Therefore, we have left this potential refinement for future research.

Therefore, in our system, we guarantee only with high probability, that if a document appears to be online, one can reach the Document Descriptor.

4.1.2 How does the user actually fetch the document?

Once users are presented with the results of the search, they should be able to have a way of reaching the actual documents. We have implemented this as a HTML link in Mosaic: when users see the word “(Online)” after a reference, they can click on it to go to the Document Descriptor via the URL returned from DNS (see Appendix A).

It is important to note here is that this link is generated automatically. When the user requests a search, the search program returns a list of matching records. For each record, it uses the data in the record to generate the corresponding URN. Then it queries DNS with the URN, and if there is a positive response (i.e. a URL), the search program generates an HTML link to the Document Descriptor (see Appendix A).

Currently, the Document Descriptor is nothing more than the interface provided by a server developed at Cornell. From this interface, we can read the abstract, bibliographic record, and usually, the actual images of the pages of the document. Though useful, this interface is not what we mean by a Document Descriptor. We would like it to be a file containing all the pertinent information about the file (i.e. formats, filenames, length, etc.), without any interface. Users should not be restricted to use any specific interface to access a document. Rather, they should be allowed to build any interface which would be most useful for their particular purpose, and simply use the Document Descriptor as a resource for attributes of the document. This will become increasingly important with the evolution of technology and proliferation of online documents and their applications. However, since the work on a true Document Descriptor is not yet complete, and as there are no other usable interfaces for viewing documents, we chose to make use of Cornell's server as a means of viewing the documents.

4.2 Generation of URN-URL mappings

In order for users to be able to use the DNS database to lookup URL's, the mappings must be somehow inserted into the DNS. Ideally, we would like each document maintainer to also maintain a DNS subdomain with mappings for the documents it maintains. However, once again, as we are setting up the only system using DNS, we need to find another way.

4.2.1 How do we fill the DNS?

There are a few different models we need to consider here. If we had no catalog and no images to start with, we could simply generate the catalog (with built-in URN's) as we scanned in documents, one reference at a time. If we had either the catalog or the documents alone, we would only need to update one as we created the other.

In our case, however, there are already several hundred Technical Reports scanned by the five Universities. Therefore, we need to do a retrospective conversion of a pre-existing catalog to pre-existing images. This brings up two new issues. First, how do we know what others have placed online, and how do we generate the URN->URL mappings?

The first problem is easy to fix. The other sites are using Cornell's document server, and it provides a list of all the documents it maintains. Therefore, it is easy to grab these lists and parse them for the requisite information. However, they contain neither the URN nor the URL. The lists contain only the actual file id, which is part of the URL. From this information we need to generate both the URN and URL. Luckily, the URL is standardized and we have established a routine method of generating URN's, which makes their generation straightforward.

There are a few standardized schemes which make this automatic generation possible. First, there is a standardized format for specifying bibliographic records for Computer Science Technical Reports. This scheme provides a standard method for numbering and naming these documents which includes the issuing organization's standard abbreviated name. The naming scheme for the actual files on the servers is also standardized to closely resemble the names in the bibliographic records.

Therefore, the only piece of information necessary is the location of the server. Once we know the server's address, we query it for its holdings. Then we generate a list of URL's by piecing together the server's address and the file ids it returned, using the server's standard file naming scheme. We then generate a corresponding set of URN's by converting the file ids to match the ids in the bibliographic records, making use of the standardization in both ids. Using this scheme, when the user's search program needs to generate a URN to query DNS with, all it needs is the id field in the bibliographic records which matched the search criteria. Therefore, both the filling and searching processes have been made easy through the use of standardization.

4.2.2 If we generate both the URN and URL, then why do we need DNS?

Under the current implementation, we are basically using DNS only to check to see if a document is online. We also precompute the URN and URL and put them into the DNS, but we could do it just as easily at search time. If nothing ever changed, there would be very little need for DNS. Unfortunately, this scheme does not scale either in size or time. Only two weeks after the first successful implementation of this system, Cornell made a slight change to their naming scheme, making our generated URL's incorrect. Also, as online documents proliferate, it will become increasingly difficult, if not impossible, to agree to a standard method of generating URL's. It will also become increasingly impractical to update software to keep in pace with newer URL generation standards.

Therefore, we propose that for real systems, the maintainers of document servers also maintain a DNS subdomain for their documents. It does not seem like it would be too difficult to automate this process, therefore, we do not expect this to create too many extra headaches for the maintainers. Also, this approach gives the maintainers more flexibility while reducing potential errors caused in our automatic generation of the URN->URL mappings. In other words, if the document maintainers generate their own DNS mappings, there is less chance of an error in the mapping.

4.3 how do we add mappings to DNS?

The data for a DNS nameserver is maintained in a single flat text file which is read at `named`[1] startup time. The only way to update the information maintained by the nameserver is to update the text file and send a signal to `named`. Very few tools exist for reliably updating the name server data. As a matter of fact, often name server administrators have problems which arise out of the complexity and difficulty of setting

1. `named` (name daemon) is the executable DNS name server program.

up and maintaining nameservers.

Ideally, we would want to have tools that would allow us to easily add new mappings to DNS, checking for errors and duplicates, modify old mappings, and delete obsolete ones. Such tools would be necessary for widespread use of our system. Unfortunately, building robust tools is not easy. There are issues of authentication, contention, and efficiency to be considered. We leave these issues for future research.

In order to prove the feasibility of our design, however, we have chosen an oversimplified method. Every night, the `dns_fill` program queries the document servers for a list of their current contents. It then uses this information to generate the URN->URL mappings and writes them to a text file in the right format. It then replaces the DNS data file with this new file, and sends a signal to `named`, causing it to reread the data file. Therefore, every night, the entire DNS database is created anew.

Of course, this is not very efficient, but since we currently only have about 1600 entries, it takes less than a minute. Even with ten times as many entries, it would take no longer than 10 minutes to gather the data and mere seconds to update `named`. Even if at some scale it became too slow, we could simply add a subdomain and another server to balance the load.

Instead of recreating the database every night, we could do incremental changes. We could keep a copy of the old contents of each server and compare them with the new, and generate a change list instead of the entire database. Unfortunately, once we move from bulk processing to incremental updates, updates become a complicated process. In addition to authenticating, we need to be able to handle several simultaneous users. We would also need to limit how often we update `named` in relation to the size and load of the server. We could also check for errors in the servers' output. For example, if the list from a server contains significantly fewer documents from one night to the next, there is a high probability that there was a problem somewhere. There are many ways to deal with these issues and we leave them for future research.

The following chapter analyzes the performance and feasibility of our system.

Chapter 5

Analysis

Now we need to analyze our system to see if it met our expectations, where it failed, and where it could be improved

5.1 DNS fill

In order to supply DNS with the URN->URL mappings, our system queries the few document servers for a listing of their documents and generates the mappings. There are several problems with this method. First, even though the documents are still available online, the servers are not always operational. If the server fails to return a list of its holdings, large portions of the database will be missing. One method of preventing these gaps would be to make the query process separate from the fill process. The query process could then query the servers and retry a few times if it gets no response, keeping the last valid response from each server until it receives a new valid response. If the server is really down, the query process can return the old list. This approach is generally valid since the server list is usually not much different from one day to the next.

Second, servers change addresses (i.e. move) which once again makes portions of the database invalid. Even in the short timespan of this thesis, one of the four servers changed its address. This problem fundamentally should be attacked by using DNS, since it is once again a name mapping problem. One suggestion is to have CNRI keep track of the document servers at a central server. This would add the control and stability necessary for a real system.

The main problem, however, is that this scheme does not lend itself well to larger scales. As the number of servers grows, the query process will need to be continually updated. Even if the current method of generating mappings for CSTRs is adopted globally, as the number of different kinds of online documents grows, it will become increasingly difficult, if not impossible, to generate URN->URL mappings automatically. The main reason we have been able to do this so far has been due to the standards in place. As different kinds of digital objects are placed online, new organizational and naming standards become necessary for the new types of URN's and URL'sURN's. Creating these standards requires time, and updating all DNS filling systems takes even longer.

Besides, the generation of these mappings defeats the purpose of using DNS in the first place. We are using DNS because we cannot *always* simply generate URN->URL mappings! Therefore, we hope that now that the feasibility of using DNS has been established, document maintainers will maintain their own DNS subdomains.

5.2 DNS lookup

When users do a search, they are presented with a list, complete with pointers to the documents, if they are online. We found this part to work rather effectively. Even though the DNS lookup made the search slower, the total time to search and make DNS requests was quite small (about 50 milliseconds per request). Of course, the name server was on the local machine which made it faster. Regardless of what system we use, network latency will be a problem when dealing with remote servers (about 100 milliseconds per request to California and 4 seconds per request to Australia). Once we factor out the network, we see that it is pretty fast. Also, network latency can be masked by prefetching, caching, and incremental fetching which continues to query DNS in the background while showing the user the results so far.

The real problem however, was with our replacement for document descriptors. Currently, the URL simply points to the server interface for a given document. This forces the users to learn and rely on other people's interfaces. Even though these interfaces were supposedly all the same, they were different versions and certain parts were missing from certain documents, making the interface inconsistent. It would be better for the user to be able to use a single interface to access everybody's documents. Therefore, the Document Descriptor must be completely separate from the interface.

5.3 BIND

We based our entire system on the BIND 4.9.2 implementation of DNS. DNS uses typed records, and we chose the TXT record type. It seemed to work very well. We did learn, however, that TXT records were not implemented before version 4.8.3 of BIND. Although a significant portion of machines are still running old versions, we believe that as hardware and software gets upgraded, this problem will disappear. However, since one of our design goals was not to force people to upgrade their system, we could easily use the older plain-text record which was implemented in 1986[1].

There seem to be no problems inherent in our usage of TXT records as compared to other kinds of DNS records. There also do not seem to be any problems with the semantics of how DNS is meant to work. Therefore, even if there are inefficiencies in BIND[2] which we have not come across or did not deem significant, since there are people actively working on future versions, we believe that such problems will be fixed with time.

Also, we have heard rumors of potential scaling problems with BIND. However, we were not able to find any evidence of any such problems at this scale, and it is not apparent where any such problems will arise in using a database that is, say, ten times larger. Actual trials at larger scales would be required to say anything more about still larger databases.

The following chapter concludes and lays the foundation for future research in this area.

1. This record was used to implement Hesiod at MIT Project Athena.
2. BIND, as most systems, *can* be more efficient. For an example, see Danzig, Obraczka, and Kumar [2].

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

Our implementation of dynamic references based on the Domain Name System seems to work and shows the feasibility of such a system. Unfortunately, some potential problems may not surface until a full-scale implementation is attempted. However, this research indicates that there do not seem to be any fundamental problems with using DNS. We are confident that any potential problems would be implementation issues which can easily be remedied with newer versions of software. Therefore, DNS is a viable, efficient, and feasible system upon which to build dynamic references, and our implementation is an example of how this may be accomplished.

6.2 Future Research

Our research also uncovered areas that could be improved through future work. The most obvious area seems to be in the filling strategy for DNS. Our current method needs to be changed such that it becomes more scalable. One approach is to make the filling incremental. In such a system, authentication, contention, and efficiency issues must be addressed.

Also, the validity of the DNS data will need to be checked or guaranteed in some manner as this system is scaled. Currently, although we do not guarantee a zero percent failure rate, the URN->URL mappings in DNS are never checked for validity. This issue generalizes to the issue of decreasing the probability of failure. Much room remains for future research on this issue.

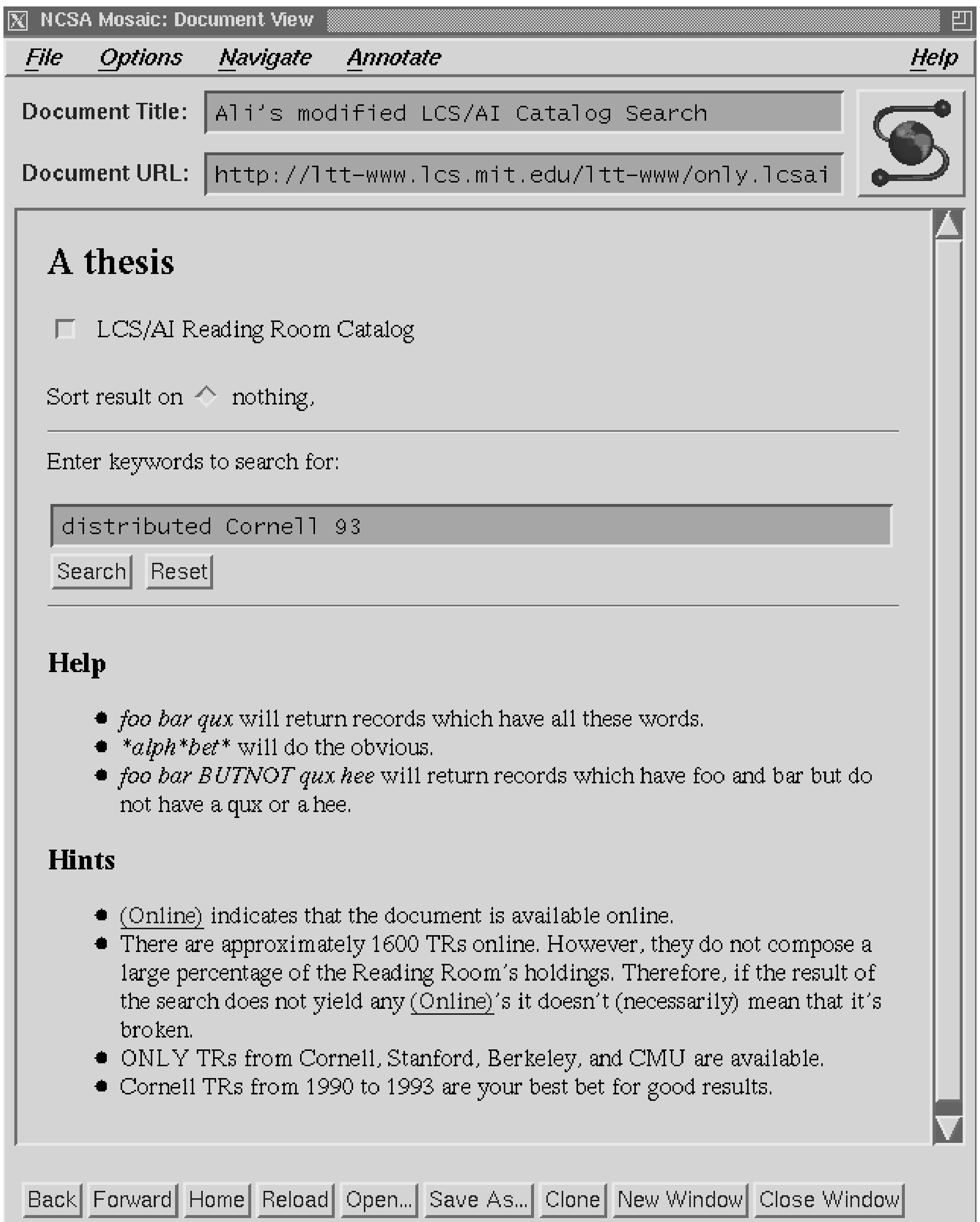
And finally, as our system grows, network latencies will become significant. More intelligent latency-aware search engines need to be created to mask the inevitable latency of such a large distributed system.

Appendix A

Reading Room Catalog Interface

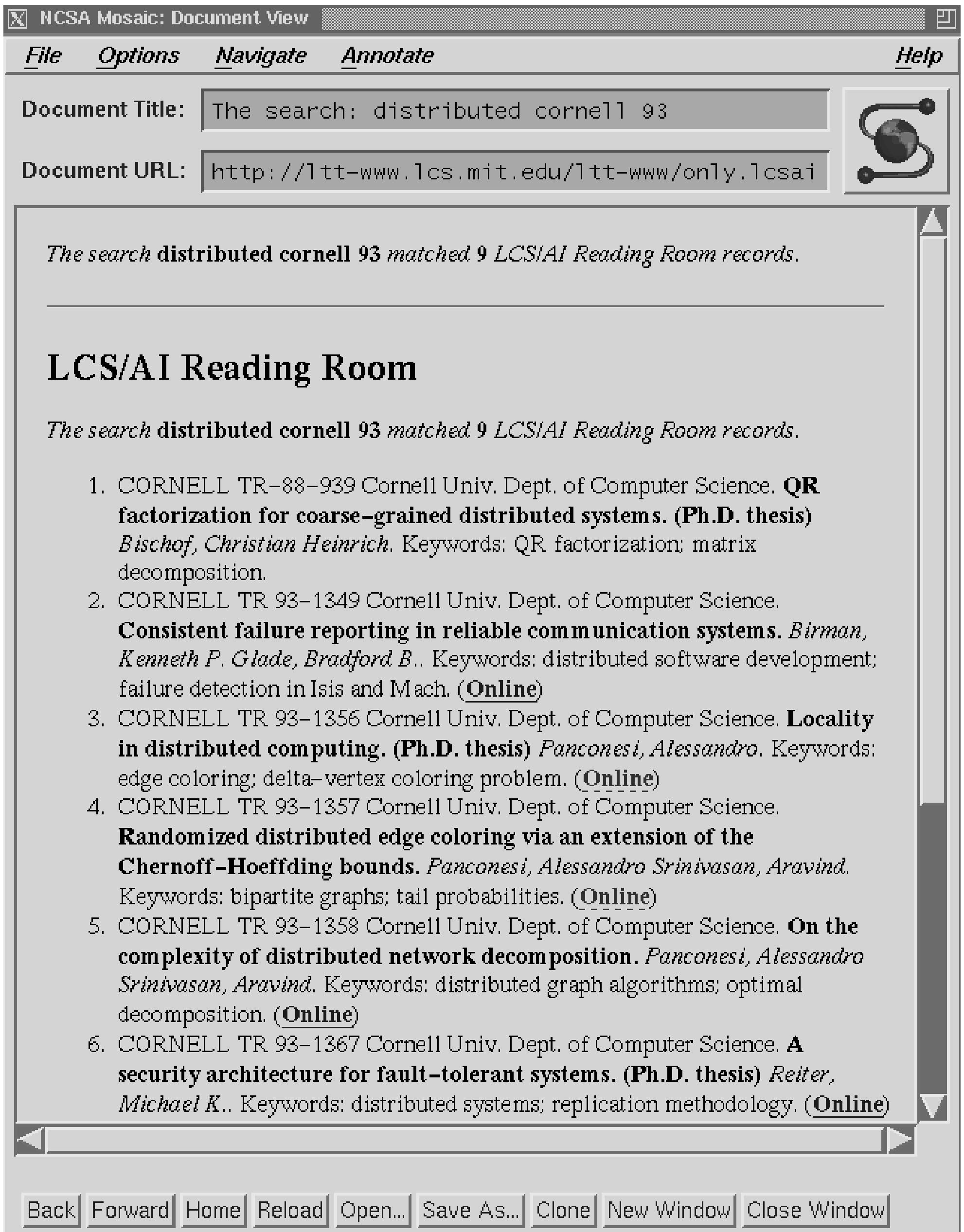
A.1 The Search Interface

This is the interface the user uses to specify a search. In this case, the user was searching for all references containing “distributed,” “Cornell,” and “93.”



A.2 The Search Results

This is part of the result to the previous search. Note that the word "Online" appears after only the references that are available online.


NCSA Mosaic: Document View

File Options Navigate Annotate Help

Document Title: The search: distributed cornell 93

Document URL: http://ltt-www.lcs.mit.edu/ltt-www/only.lcsai

The search **distributed cornell 93** *matched* **9** *LCS/AI Reading Room records.*

LCS/AI Reading Room

The search **distributed cornell 93** *matched* **9** *LCS/AI Reading Room records.*

1. CORNELL TR-88-939 Cornell Univ. Dept. of Computer Science. **QR factorization for coarse-grained distributed systems. (Ph.D. thesis)** *Bischof, Christian Heinrich.* Keywords: QR factorization; matrix decomposition.
2. CORNELL TR 93-1349 Cornell Univ. Dept. of Computer Science. **Consistent failure reporting in reliable communication systems.** *Birman, Kenneth P. Glade, Bradford B..* Keywords: distributed software development; failure detection in Isis and Mach. (**Online**)
3. CORNELL TR 93-1356 Cornell Univ. Dept. of Computer Science. **Locality in distributed computing. (Ph.D. thesis)** *Panconesi, Alessandro.* Keywords: edge coloring; delta-vertex coloring problem. (**Online**)
4. CORNELL TR 93-1357 Cornell Univ. Dept. of Computer Science. **Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds.** *Panconesi, Alessandro Srinivasan, Aravind.* Keywords: bipartite graphs; tail probabilities. (**Online**)
5. CORNELL TR 93-1358 Cornell Univ. Dept. of Computer Science. **On the complexity of distributed network decomposition.** *Panconesi, Alessandro Srinivasan, Aravind.* Keywords: distributed graph algorithms; optimal decomposition. (**Online**)
6. CORNELL TR 93-1367 Cornell Univ. Dept. of Computer Science. **A security architecture for fault-tolerant systems. (Ph.D. thesis)** *Reiter, Michael K..* Keywords: distributed systems; replication methodology. (**Online**)

Back | Forward | Home | Reload | Open... | Save As... | Clone | New Window | Close Window

References

[1] Berners-Lee, T., "Uniform Resource Locators (URL)," *Internet Engineering Task Force Draft*, October 1993.

[2] Danzig, P. B., Obraczka, K., and Kumar, A., "An Analysis of Wide-Area Name Server Traffic," Proceedings of the ACM SIGCOMM '92 Conference, October, 1992, pages 281-292.

[3] Hardcastle-Kille, S., "X.500 and Domains," *Internet Request For Comment 1279*, University College London, November 1991.

[4] Albitz, P., Liu, C., *DNS and BIND*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[5] Saltzer, J. H., "Technology, Networks, and the Library of the Year 2000," in Future Tendencies in Computer Science, Control, and Applied Mathematics, Lecture Notes in Computer Science 653, edited by A. Bensoussan and J.-P. Verjus, Springer-Verlag, New York, 1992, pp. 51-67. (Proceedings of the International Conference on the Occasion of the 25th Anniversary of Institut National de Recherche en Informatique et Automatique (INRIA,) Paris, France, December 8-11, 1992.)

[6] Weider, C., and Deutsch, P., "Uniform Resource Names," *Internet Engineering Task Force Draft*, October 1993.

[7] Weider, C., and J. Reynolds, "Executive Introduction to Directory Services Using the X.500 Protocol," *FYI 13*, *Internet Request For Comment 1308*, ANS, ISI, March 1992.

[8] Weider, C., Reynolds, J., and S. Heker, "Technical Overview of Directory Services Using the X.500 Protocol," *FYI 14*, *Internet Request For Comment 1309*, ANS, ISI, JvNC, March 1992.

[9] Yeo, A. K., Ananda, A. L., Koh, E. K., "A Taxonomy of issues in Name Systems Design and Implementation," *Operating Systems Review*, July 1993, pages 4-18.