An Approach to A Scalable Wide-Area Web Service

An Approach to A Scalable Wide-Area Web Service

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op donderdag 4 december 2003 om 13.00 uur
door

Ihor Theodore KUZ

doctorandus in de informatica
geboren te Toronto, Canada.

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. ir. H.J. Sips
Prof. dr. ir. M.R. van Steen

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. ir. H.J. Sips,	Technische Universiteit Delft, promotor
Prof. dr. ir. M.R. van Steen,	Vrije Universiteit Amsterdam, promotor
Prof. dr. ir. P. van Mieghem,	Technische Universiteit Delft
Prof. dr. A.S. Tanenbaum,	Vrije Universiteit Amsterdam
Prof. dr. P.M.E. de Bra,	Technische Universiteit Eindhoven
Prof. dr. L.O. Hertzberger,	Universiteit van Amsterdam
Dr. A.M. Kermarrec	Microsoft Reasearch Cambridge, Groot-Brittannië

The work described in this dissertation was completed as part of the JERA project.

Typeset by the author with the LaTeX Documentation System.
Author email: ikuz@ikuz.org

Contents

List of Figures

List of Tables

Acknowledgements

The acknowledgements have finally been reached, this means that I am almost done. Personally, I always enjoy reading the acknowledgements in a dissertation. They help to add a human side to the (usually technical) contents. Having said that, I now have to come up with something that I would enjoy reading myself, so here goes.

Doing a Ph.D. is an adventurous journey. There are many obstacles in the way, many ups, many downs, and a great deal of distractions too. Although it leads to an individual accomplishment, there are many people that help out along the way. Whether they help out with the research and writing, or whether they provide moral support, or the much needed distractions, all of them deserve thanks and credit.

The first to get my thanks are both my advisors (promotors) Maarten van Steen and Henk Sips. I have often heard Ph.D. students complaining that their advisors don't give them enough guidance and support. I must be lucky, because I've never had that problem. I have always gotten the best advice and guidance from both Maarten and Henk. Maarten would meet with me weekly and was always involved with everything going on — whether it was research or programming or writing. It was precisely Maarten's close involvement that made Henk's more removed view of the research extremely useful. Whereas many aspects of my work were so familiar to myself (and Maarten) that I never questioned them or felt the need to look at alternatives, Henk always approached the work from a different perspective, seeing problems and possibilities that I had completely missed. Henk also made sure that my descriptions of Globe and GlobeDoc were understandable to more than just a select few members of the Globe group (the readers of this dissertation should thank him for that too).

Next, I would like to thank all the members of my reading commission (Andy Tanenbaum, Piet van Mieghem, Anne-Marie Kermarrec, Bob Hertzberger, and Paul de Bra). I would like to thank you for your time and your comments. It amazes me that such busy people would have time to read this dissertation much less provide such good and insightful comments.

The research described in this dissertation was carried out as part of the JERA project. As such, I would like to thank all the members of the project for making this possible.

Then there is the Globe group at the VU. Globe is a group effort so my thanks go out to all of you. In particular I would like to acknowledge Guillaume Pierre's and Patrick Verkaik's contribution to this dissertation. The experiments described in Chapter 2 were designed and performed together with Guillaume. The description of these experiments

was taken, largely, from a paper written by Guillaume. Most of the object server architecture, as described in Chapter 4, was implemented by Patrick. Some parts (in particular the networking subobject) were designed together with Patrick as well. Patrick was a great asset to the Globe project. Although it was sometimes frustrating to read and modify his code, the code was good, and it worked. I would also like to thank Arno Bakker for imparting to me some of that rare knowledge and wisdom that one gains after personally experiencing (and surviving) the trials and tribulations of the Ph.D. process. Arno also deserves thanks for helping me work out practical details of printing the thesis from overseas.

Being a TU Delft AIO I also spent much of my time at the TU Delft itself. Although I did not collaborate directly with my TU Delft colleagues, they were always open to good discussions (about research and other things). I would like to thank everyone in the PDS group for making my time in Delft a pleasant experience.

Having dealt with the colleagues it is now time to thank friends and family. I don't know if this is normal, but most of my friends (and family) know very little about computers. Because of this it was always a challenge to try to explain what exactly I was doing (I'm pretty sure that most of them still aren't exactly sure what I've spent the past six years doing). Now, in my case, explaining what my research was about was somewhat easier because it had to do with the Web (and everyone's heard of the Web!). As such, I could get away with mentioning the Web, mumbling a bit about performance and replication and they could (realistically) pretend to understand me. Some even went so far as to ask questions about the research and then bearing with me while I went off explaining the intricacies of whatever I was working on at the time. Thanks for humoring me!

Friends also play very important roles as stress relievers. By providing enough distractions to allow me to forget the work and the deadlines you all helped enormously. I appreciate all the climbing, skating, biking, music, partying, swimming, hackying, etc. I wont list names here, you know who you are.

There are a two people who played special roles in my life during my PhD: Margarita and Leonie. I would like to thank both of you for your patience, understanding, motivation, and everything else.

Another important person is Suzan, who has probably had more of an influence than she realizes. She (inadvertently) played an important role in my deciding to become an AIO and pursue my Ph.D. Thank you for that, and thank you for the motivation and friendship since then as well.

After friends comes family. But before I start, I would like to put Ulyana, my sister, in her own category. I couldn't decide whether to put you in the friends or family section, so you get your very own section right here. Although as children we fought a lot, as we've grown older we've become very close. Ulyano, you've always been there for me, even when you lived far away. Thank you very much.

Ok, now the family (not that you are any less important of course). The main role of the family is to keep on asking when I'll be finished already. Well you've all done a great job, and now I can happily say that I'm done! I'd specifically like to thank my Mom, for everything past, present, and no doubt future, my Dad, for always keeping in touch, and

always being close despite the great distances. And finally my thanks to Nick — you have been a great influence in my life, one that I will always appreciate.

This is getting long, I know, but bear with me, we've almost reached the end.

During my PhD I also had the opportunity to join Joost, Frans, Leonard, Lars, and Aiko in SmartHaven, our attempt at .com glory. Although I couldn't put in as much of an effort as you guys did, I want to thank you for letting me join the fun (and understanding how important finishing my dissertation was to me). It was a great ride! On a sadder note, I had never imagined that my Ph.D. would outlive the Smarthaven adventure, but, alas, that's the way it is.

Sydney, September 2003.

Chapter 1

Introduction

The World Wide Web is huge and there are no signs of its growth slowing down in the near future. In terms of the number of available pages, it has been estimated that the Web has grown from thousands of pages in 1993, through 320 million pages in 1997 and 800 million pages in 1998 to more than a billion pages in 2000 [58, 59, 57]. The number of Web servers serving these pages has similarly grown from tens of thousands in 1995, through half a million in 1996 and 8 million in 1999 to over 30 million in 2001 [50, 68]. A similar explosive growth pattern has been seen in the number of users accessing Web pages, with 2001 showing this number to be well over 80 million.

With the size and growth attesting to its popularity, it is safe to say that the Web is the world's first globally used wide-area distributed application. However, when it was first created in the early '90s such massive growth (not only of the Web but of the Internet as a whole) was unexpected. As such, the infrastructural design of the Web and its fundamental protocols was fairly simple.

On an architectural level the Web is based on a **client-server model**. It is organized as a decentralized collection of servers serving pages to browsers and other client applications. A **Web page** is a collection of **Web resources** (files accessible over the Web) and consists of an HTML document and its embedded elements (such as images and media fragments). Related Web pages are often grouped into a Web site. Usually all the Web pages that make up a **Web site** are hosted at the same server, however, this is not a requirement. Each Web resource is identified by a Uniform Resource Locator (URL). A URL identifies the server hosting the resource, the access protocol spoken by the server and the identifier (e.g., path name) of the resource on the server. Given a resource's URL, a client wishing to access that resource must contact the server using the appropriate protocol to request the resource using the given identifier. Upon receiving the request the server fetches the identified resource and returns a copy to the client. The server and client communicate using the HyperText Transfer Protocol (HTTP). HTTP is a simple protocol that allows clients to request resources and servers to return them.

Although the simplicity of the architectural design and the protocols is one of the key factors behind the Web's phenomenal growth, it is precisely this simplicity that caused

problems once the Web's popularity started to rise. As more people started using the Web it became clear that the initial infrastructure did not scale. Servers with popular content became overloaded while networks all over the world became saturated due to Web traffic. Note, however, that although the Web faces scalability problems, the underlying Internet technology and infrastructure has scaled well in the face of its own explosive growth.

Despite all the work done so far, and the fact that both server and network technologies have grown fast enough to keep up with demand, the Web still suffers from severe performance problems. One aspect common to all the research done so far is that it attempts to improve and adapt the *existing* WWW infrastructure. It is my opinion that the current solutions do not go far enough. It is the fundamental client-server nature of the Web infrastructure that prevents it from gracefully scaling to current and future sizes.

The goal of this dissertation is to present an alternative architectural solution to the Web's scalability problems[1]. The central point behind the presented approach is that a truly scalable distributed system cannot be built in an environment (such as the current Web) that forces a 'one-size-fits-all' view of distribution and communication. In order for any distributed system (including the Web) to scale (up or down) it must be flexible with regards to its distribution and communication policies. As will be explained later, the current Web architecture does not fulfill these requirements.

Note that it is not the intention of this work to (attempt to) completely replace the current Web infrastructure. In many cases it works fine as is, so there is no reason to replace it (besides, with such a large installed user base, any proposal to completely replace the current architecture would not stand a chance). Instead, the results of this work can be seamlessly incorporated into the existing Web infrastructure.

1.1 Problems with the Web

There are many problems with the Web. The range of problems affects everyone involved with the Web: users, content creators, content providers and network administrators. These problems can be classified into four categories: content problems, metacontent problems, content-organization problems, and infrastructure problems.

The first category, **content problems**, refers to problems experienced when creating, interpreting and displaying Web content. This includes problems experienced by users when browsers cannot properly display Web pages and problems experienced by content producers when content creation tools and resources are insufficient to achieve desired effects. The causes of these problems vary from incomplete, incompatible or faulty browsers, to the limitations on graphic design imposed by HTML and the attempts by many third parties to overcome these limitations using mutually incompatible solutions.

The second category, **metacontent problems**, refers to problems experienced when searching for and categorizing Web content. Given the large amount of content available on the Web, finding specific content or information is a difficult task. Search engines and Web directories attempt to ease this task by indexing and categorizing the available

[1] These scalability problems, including a precise definition of scalability, will be discussed later on in this chapter.

content. Unfortunately, doing so is not easy. The amount of content available means that manual categorization (such as that done by Yahoo![2]) requires massive resources, and even so only covers a small part of the available content. However, automatic categorization of content suffers from a lack of metadata describing the available content. As such, automatic categorization has to rely on extracting meaning from the content, and techniques for doing so are not yet mature enough to produce good results.

The third category is that of organizational problems. **Organizational problems** are those related to the organization of Web sites and the relationships between the pages that make up a site. These problems most often manifest themselves as missing pages and broken links to pages within a Web site. However, problems caused by a confusing layout of a site also fall into this category. The causes of organizational problems vary from deleted or moved content to complete reorganization of Web sites.

Finally, the fourth category concerns infrastructure problems. **Infrastructure problems** are those problems that are a direct result of the infrastructure (e.g., clients, servers and networks) used. They manifest themselves as performance, fault tolerance, and security problems. Performance problems are those that affect the responsiveness of a Web site, while fault tolerance problems are those that affect the stability and accessibility of a Web site. Security problems relate to the vulnerability of Web sites to unauthorized access and modification.

1.1.1 Performance

In this dissertation we are concerned with the category of infrastructure problems, and in particular the performance aspect of these problems. As mentioned, **performance problems** affect the responsiveness of a Web site. A Web site with good performance is responsive and provides fast downloads of its content. A Web site with bad performance, on the other hand, is slow in responding to user requests and provides slow downloads of its content. In extreme cases low performance can cause a site to become effectively unreachable.

Performance problems are an important category of problems because they affect everyone involved with the Web. Users are affected when the sites they try to reach have slow response times. Content creators and providers are affected when they have to modify the content provided to prevent causing or aggravating decreased performance (for example, a content provider might have to reduce the resolution of a popular film so that downloads of that film do not congest the server's network and thereby affect the whole site's performance). Network and system administrators are affected because they are the ones that have to make sure that performance is adequate.

Performance problems are also important because they affect the heart of the Web. When the available infrastructure cannot keep up with the growing demands, then other problems pale in significance. For example, when a user cannot access a Web site due to poor performance, the problems relating to the organization or layout of the site become irrelevant.

[2] www.yahoo.com

The performance problems experienced on the Web can be further broken down into four subcategories. These are connection problems, latency problems, delivery problems, and rendering problems.

Connection Problems Connection problems are problems related to connecting to a site. These problems manifest themselves as a long wait before a site is reached and content can be transferred. Habib and Abrams [45] have found that connection problems are usually due to delays in DNS queries. As explained later, users may also experience connection problems due to inefficiencies in the HTTP and TCP protocol, as well as network congestion or server overload.

Latency Problems Latency problems are those related to sending a request and receiving (the first part of) a reply. These manifest themselves as a long delay after a site is reached but before content starts being transferred. **Server load** is generally responsible for much of this delay [15, 4]. **Network latency** and **network congestion** also make a significant contribution, especially when the server is not heavily loaded [45]. The effect of server load is usually greater when pages must be generated (e.g., by CGI programs) than when static pages are used.

Delivery Problems Delivery problems are those experienced when transferring the content, but after the connection has been established and the download started. They manifest themselves as abnormally long or unstable downloads. File size and network capacity (available bandwidth) play an important role in causing this delay. Bradford and Crovella found that file transfer delays were heavily influenced by both congestion and packet loss [15]. A higher network load was also found to cause a higher variability in the delay experienced.

Rendering Problems Rendering problems are experienced after the actual content has been received, but before the user can actually view it. This is generally caused by slow and inefficient client-side software or extremely large or complex content data (e.g. HTML pages with complex tables or large images). Allison et al. found that rendering accounted for a large portion of the overall delay experienced by users [4]. Rendering problems will not be further considered in this dissertation and are mentioned only for completeness.

1.1.2 Scalability

The infrastructure developed for the WWW can also be used to create local-area Webs called Intranets. An **Intranet** uses the same technologies as the WWW, but operates on a LAN within a single organization. It usually does not suffer from the same level of performance problems.

In an Intranet, the distance between the client and server is small. In many cases the client and server are even on the same network, in other cases they are a few (generally

two to three) hops away from each other. Either way, the route between the two is short and relatively free of obstacles such as gateways and heavily loaded routers. This means that the network latency between the two is very low. Similarly, the bandwidth of the links between the client and server in an Intranet is high (the bandwidth of local-area networks currently ranges from 10 Mbit/s to 1 Gbit/s). This means, in general, that the connections between client and server will remain uncongested. Also, clients on an Intranet access a limited number of well-known servers, which means that DNS lookups when connecting to a server are unnecessary. When they are necessary, the lookup is quick because it is performed on a local name server. This, combined with the low latency of the network connection, leads to low connection times for the client. Similarly, servers on an Intranet have a limited and stable number of clients. This means that servers often have a predictable load and can be configured to manage that load. Combined with an uncongested network and low latency this leads to low overall latency (i.e., the time to get the first byte of contents). The high bandwidth and low server load also lead to fast content delivery.

In the wide-area case, on the other hand, clients and servers are much further apart. This means that the route between a client and server is longer and more complex, involving many network links connected by routers and gateways. This increases both the latency of the route and the possibility of congestion on the route. Although wide-area network links often have high bandwidths they are generally used by many more concurrent users than are local-area network links. This limits the bandwidth available for any single user and aggravates congestion problems. Clients on the World Wide Web access a wider variety of servers than those on an Intranet. Because they regularly come across unknown servers, expensive DNS queries [28] are usually necessary before a connection to a server can be established. This leads to longer initial connections times. Just as clients have a wider variety of servers that they connect to, so do servers have a wide variety of clients that connect to them. Because of this a server's maximum load is difficult to predict and as such a server must be able to deal with a greater range of loads. WWW servers must also deal with higher frequencies of requests than Intranet servers do. This leads to lower performance of servers, which combined with the higher network latencies leads to a higher overall latency. The lower server performance combined with lower bandwidth also leads to longer delivery times.

The fact that the WWW does suffer from performance problems, while the local-area Intranets do not, indicates a **scalability** problem in the Web infrastructure. According to Neuman [71], the scale of a distributed system has three dimensions: geographical (the distance between the nodes in the system), numerical (the number of users and objects that are part of the system), and administrative (the number of organizations involved in the system). A system's scale has effects on its reliability, performance, and administrative complexity[3]. This dissertation will limit discussion to the effects of the Web's geographical and numerical scale on performance.

As such, there are two scalability problems faced by the Web. The first one is that the infrastructure does not scale well with regards to the geographic distribution of clients. The second is that it does not scale well with regards to the number of client requests

[3] A more formal definition of scalability in distributed systems is provided in [109].

processed. Thus, the infrastructure works well when clients are localized, and when the number of requests is stable and minimized. However, as soon as clients become more widespread, the number of requests increase, or the requests become highly unpredictable, the performance levels drop.

1.2 Requirements for a Solution

Because the Web's performance problems stem from the nonscalability of its infrastructure, a good solution to these problems will provide a scalable infrastructure. The effect of a good solution will be to minimize connection time, latency, and delivery time. Besides simply minimizing these factors, a good solution will make them virtually independent of the geographic distribution of a Web site's clients and of the Web site's access pattern. Being independent of the geographic distribution of clients means that, on average, all clients will experience similar connection time, latency, and delivery time. There will be little distinction between clients closer to or further away from the Web site's server. Similarly, being independent of the access pattern means that despite the server having to handle more requests, all clients experience a similar high level of performance. Furthermore, the traffic and load caused by one site should preferably not affect the performance of another, unrelated site (which currently does happen if the sites are hosted on the same server). However, if the performance is affected (e.g., due to communication over common links), the effects should be minimized.

Some of the performance problems can be solved by applying more hardware, better server software, and more network capacity. For example, the problem of latency can often be solved by deploying a faster server and providing it with a higher capacity link to the Internet. However, this is not a structural solution, and given enough new traffic, the latency problems will return. The application of new hardware or better network connectivity cannot, for example, prevent problems due to sudden, unexpected, increases in the number of client requests received (e.g., flash crowds).

Similarly, some problems may be relieved by improving or fine-tuning the protocols used (HTTP and TCP). For example, a modification to HTTP made in HTTP 1.1 allows persistent connections [36]. This means that clients making multiple requests to the same server can reuse an existing connection. Reusing existing connections improves performance because the costs of creating new connections are avoided. Similarly, TCP has a so-called slow start, where the packet size starts off small and is increased until an optimal transfer rate is reached [100]. This works well for long-lived connections, however, for the short-lived connections[4], which are typical for the Web, this is inefficient and simply increases the latency and delivery time. Eliminating the slow start, therefore, helps to reduce latency and delivery times for HTTP [98][5]. However, because these solutions are

[4] Although the connections used for downloading of large resources (such as software, music, and video) are relatively long-lived, the majority of connections are used to transfer small resources (such as HTML pages, icons, and small images) and are short-lived.

[5] Eliminating slow start provides minor performance benefits for typical HTTP traffic. However, for general Internet traffic TCP slow start plays an important role in avoiding network congestion and improving general performance. Eliminating slow start is, therefore, generally a bad idea.

not structural, increases in the number of requests handled will lead to increased network traffic and server loads causing delays that eventually offset the gains provided by such solutions.

A more effective solution is based on the observation made previously that the Web infrastructure deployed in a local-area scenario (i.e., an Intranet) does not suffer from the same performance problems as when it is deployed in a wide-area scenario (i.e., the WWW). By decreasing the distance between client and server a solution that localizes traffic can help reduce latency and delivery times. Likewise, a solution that reduces the number of requests a single server has to process also reduces connection, latency, and delivery times.

1.2.1 Replication

An effective method for achieving this localization of traffic and reduction of requests is replication. **Replication** involves placing copies of the content (**replicas**) on other servers in the Web. With multiple copies of the content available clients have a choice of servers to send their requests to. Their decision is based on which server will offer the best performance. Assuming that the servers themselves offer similar levels of performance, it is the distance from the client that sets the servers apart. Requests are sent to the server closest to the client. Not only does this spread the total load over multiple servers, relieving the load at any single server, but, because clients send requests to their nearest server, the distance between clients and their servers is shortened allowing for better performance. An added benefit of replication is that it may improve the fault tolerance of a Web site. If a Web site is replicated over multiple servers and one of the servers goes down the Web site may still be reachable at one of the other servers. The number of replicas a Web site needs depends on the distribution of its clients and the number of requests it receives. How, when, and where replicas are created is determined by a **replication policy**.

Unfortunately, replication is not without its drawbacks. One of these drawbacks is that all the replicas must be kept consistent with each other. That is, when any of the replicas is modified they all have to be modified, otherwise they will be out of sync and represent different content. Thus, in order to keep replicas consistent, the content must be copied out to all the replicas whenever it is modified. There are numerous approaches to copying modified data out to replicas. All the approaches can, however, be grouped into one of two categories: those where modified content is pushed to the replicas, and those where replicas pull in modified content when they notice that it has changed. There are also different levels of **consistency** that can be offered. Weak levels of consistency offer little in terms of guarantees about the consistency between the replicas. In this case it may be possible that content received from one replica may differ from content received from another. Stronger levels of consistency make stronger guarantees about the consistency between replicas. The strongest form of consistency guarantees that all replicas will always contain the same content. A **coherence policy** determines what level of consistency is provided and how this level of consistency is achieved. Together, a replication policy and a coherence policy form a **distribution policy**.

Replication has been used as the basis of many proposed solutions to the Web's performance problems. One of the most common approaches is caching. **Caching** involves creating temporary local copies of content. After the first request (which is served from the server and loads the cache) subsequent requests for the content are made to these local copies, if possible. There are numerous mechanisms for keeping the cached content consistent with the main copy. These range from checking with the server every time a request is made, to timeout-based mechanisms where after a specified time limit the content is considered out of date and a new copy is to be fetched from the main server.

Another common approach employing replication is mirroring. **Mirroring** involves hosting replicas of Web sites at other servers. Clients can request content either from the main site or from a mirror site. Most methods for keeping mirrors consistent are manual, that is, the content owner must make sure that updates are also made to the replicas.

A drawback of most proposed replication solutions is that their approach must be applied universally to all Web content replicated using that solution. This means that all Web resources have to be replicated in the same way and with the same coherence policy applied to all of them. However, with the large variety of Web content currently available it is questionable whether such an assumption is valid. For example, strong consistency is very hard to achieve on a wide-area scale and is expensive (in terms of communication between replicas) to implement. For some Web sites (e.g., a site that provides real-time stock quotes) strong consistency is an absolute necessity and the price of extra communication and complex protocols is worth paying. Other sites (such as a conference Web site or a home page) may, however, be satisfied with weaker consistency guarantees and would not be willing to pay the price for stronger consistency. It is not acceptable that sites that do not require strong consistency have to implement it and pay for it. Likewise, it is not acceptable that sites that do require stronger consistency cannot get it.

We claim that in order to be scalable, the Web infrastructure not only needs to allow content to be replicated, but it must be flexible enough to allow the replication and coherence policies used to be determined on a case-by-case basis. An appropriate solution will, therefore, allow content to be replicated, but will not impose any specific distribution policy (that is, a policy with regards to communication, replication, and consistency) on Web sites.

1.2.2 Summary

To summarize the requirements, an appropriate, scalable, solution to the Web's performance problems will have the following effects on access to any Web site:

- Maximize performance by minimizing connection time, latency, and delivery time experienced by clients
- Make the level of performance independent of geographic distribution of the Web site's clients
- Make the level of performance independent of the Web site's access patterns

- Make the level of performance independent of any other Web site's access patterns.

These effects are achieved by:

- Localizing traffic
- Limiting the number of requests handled by (and thus the load on) any single server
- Decoupling a Web site from the server providing its contents.

In order to achieve this, an infrastructure must:

- Allow content to be replicated
- Be flexible and not impose any single distribution policy on all Web sites.

1.3 Example

An example of a situation that is often the cause of extreme performance problems is a flash crowd. A **flash crowd** is a rapid and unexpected surge in the popularity of a Web site. It is caused when a site receives an unusually large number of requests, which causes the Web server's load to become unusually high (a three to four-fold load increase is not uncommon). Despite their severity, flash crowds are relatively short lived; lasting between a couple of hours and a couple of days.

The World Trade Center and Pentagon bombings on September 11, 2001 caused flash crowds at many news Web sites. Shortly after the bombings started (and for the most part of that day), major news sites (such as those of CNN, MSNBC, ABC, etc.) received so much traffic that the loads on their servers became unbearable. CNN[6], for example, had to deal with 9 million page views per hour, as opposed to a normal load of 11 million page views a day. The high loads caused performance to degenerate until most of the sites were rendered unreachable. With the major news sites unavailable, those desiring news about the unfolding events turned to the more numerous, but less prominent, news sites such as Slashdot and Indymedia. Although this helped to spread the load out over more sites, the load on each site was still much higher than normal. For example, Slashdot[7] (a site for mainly technical news, but which also covered the bombings), normally services 18 to 20 requests per second, whereas soon after the bombings its load went up to 40 to 50 requests per second. The problems experienced by most news sites were caused mainly by overloaded Web servers and possibly congested network links at the servers, as opposed to congestion problems on the Internet's backbone networks.

The affected sites reacted in different ways to solve their problems, get back online, and successfully handle the loads. One strategy taken by many pure news sites was to drastically reduce the complexity of their Web sites. CNN, for example, provided a simple, static, HTML page with no tables or frames, no advertising, and no unnecessary graphics; just a list of links to the newest stories. The pages containing the stories themselves were

[6] http://www.cnn.com
[7] http://www.slashdot.org

also plain and simple containing only the text of the story and any relevant pictures. Such stripped down Web sites allowed the servers to run more efficiently and process more requests. Similarly, by removing any unnecessary content, the replies sent were much smaller, meaning that many more replies could be sent before the network links would become saturated. Other sites, such as MSNBC, also added extra Web servers to help handle the loads. Besides actions taken by the news sites themselves, many ad-hoc mirror sites all over the world were also created. These were sites on servers not affiliated with the news sites but which, nevertheless, served copies of all the important stories.

Some news sites, however, provide more than simply the news. A site such as Slashdot also provides discussion forums were people discuss the news stories. For many such sites, including Slashdot, these discussion forums are their main attraction. Because of this, Slashdot could not simply reduce their site to a collection of static pages, updating them only when new stories came in. Instead, its maintainers took an approach that was better suited to their style of content. Besides tuning the software and databases used, they also eliminated many of the seldom used or less relevant functions on their site. Similarly, they switched from serving purely dynamic pages to a mixture of static and dynamic pages. By studying their own usage logs the Slashdot maintainers realized that only a fraction of their users use interactive features that affect their view of the Web site.[8] Based on this observation they decided that some of their servers would be configured to serve only static versions of their pages (that is, pre-generated views of the discussion forums and stories with the interactive features turned off) while the rest of the servers would continue serving dynamic versions. This greatly reduced the load on their database (generating dynamic pages requires database lookups, while serving static pages does not), improving the overall database performance and allowing all of their servers to process requests faster.

These examples highlight the need for a scalable Web infrastructure that can deal with (expected or unexpected) increases in the number of client requests that a Web site receives. They also illustrate that effective solutions can differ per site, and that what is effective for one site may not be so for another. For example, it would not have been appropriate to apply the CNN solution to Slashdot, as it would have rendered one of the most important features of Slashdot useless. Moreover, the examples show that an effective approach may not always remain effective for that same site. A day after the bombings the flash crowds died away and the number of requests processed by the sites had come down to an almost normal level. This allowed most sites to return to their normal modes of operation. It would not have been effective for any of the sites to keep operating with their flash crowd configurations (stripped down pages, decreased dynamic content, etc.) when it was no longer necessary.

[8] The most important of these features are ones that allow users to post stories, change the amount and quality of comments they see in the discussion forums, and filter out particular authors.

1.4 GlobeDoc

The solution proposed in this dissertation is a scalable Web architecture called **GlobeDoc** (Global Object Based Environment for Web Documents). GlobeDoc is an architecture based on **distributed shared objects** (DSOs). DSOs are physically distributed objects, meaning that their state is partitioned across multiple disjoint address spaces (which are usually hosted on separate machines) at the same time. Clients of an object are unaware of such a distribution: they see only the object interfaces available in their own address spaces. **Globe** (Global Object Based Environment) [108] is an existing wide-area distributed system that provides support for DSOs. GlobeDoc is designed as a Globe application and is based on Globe DSOs.

In Globe, each object fully encapsulates its own distribution policy. There is no system-wide policy imposing how an object's state should be distributed and kept consistent. Moreover, clients need not be aware of the details of the distribution policy applied by an object. A Globe object is therefore free to take any possible approach to replication. It can, for example, decide not to create any replicas at all, or it can create replicas and offer a best effort consistency guarantee. An object is even free to dynamically create and destroy replicas as it sees fit. Globe is location transparent, in that clients do not know where a Globe object and its replicas are hosted. A client wishing to use a Globe object will be automatically connected to the replica closest to it. Globe provides both a DSO model and a framework that offers support for using, creating and hosting Globe DSOs.

Conceptually, a **GlobeDoc object** is a distributed shared object that contains a **Web document**, which is a collection of logically related Web pages. Each such page may consist of text, icons, images, sounds, animations, etc., and may also contain applets, scripts and other forms of executable code. Each of these parts is referred to as an **element**. A GlobeDoc object allows clients to add, remove, access and modify its elements. In the GlobeDoc model a Web site consists of a collection of GlobeDoc objects.

The main reasons for basing GlobeDoc on Globe is that the Globe infrastructure encourages replication of content and makes this replication transparent. Similarly, Globe allows each object to have its own associated distribution policy. This makes it possible to determine and implement an optimal policy for each individual Web document.

The fact that GlobeDoc content can be replicated means that a GlobeDoc object's total load will be spread out over its replicas. Likewise, the traffic at each replica will have a more local character (i.e., the distance to clients will be smaller). As stated earlier, this has positive effects on reducing the connection times, latency, and delivery times for Web documents. Similarly, the fact that new replicas can be added and removed means that it is possible for a GlobeDoc object to adapt its replication based on its current situation. For example, if the number of requests dramatically increases a GlobeDoc object may create new replicas to help handle the load. Similarly, a GlobeDoc object may create new replicas placed closer to a large group of clients to reduce the distance between the clients and itself, thus localizing the traffic. This adaptability makes it possible for the performance of a GlobeDoc object to be independent of the geographic distribution of clients and the number of requests they generate.

GlobeDoc objects are also independent of each other. Replicas of different GlobeDoc objects may be colocated on one machine, however this is not necessary. Thus, for example, if a popular GlobeDoc object (replica) causes a high load on a machine, replicas of other GlobeDoc objects being hosted on that machine can be moved to less loaded machines. Because of Globe's location transparency, (potential) clients do not have to be informed of this move.

1.5 Contributions

The main contributions of this dissertation towards solving the Web's scalability problems are as follows.

- It shows that in order to solve many of the Web's performance problems it is necessary to replicate much of the available content. However, besides simply replicating the content using a single policy for every document, optimizing performance requires applying different distribution policies on a per-document basis (Chapter 2).
- It presents GlobeDoc, a Web architecture based on Globe distributed shared objects. GlobeDoc allows distribution policies to be assigned to Web documents on a per-document basis. In order to make it practical to use GlobeDoc, a way to seamlessly incorporate GlobeDoc into the existing Web infrastructure is also presented (Chapters 3 and 4).
- It presents the design of a distributed resource management service. This service is used by GlobeDoc for local administrative purposes as well as to aid in the placing and distribution of replicas (Chapter 5).
- It presents an approach to profiling the performance of components in a (wide-area) distributed system like GlobeDoc (Chapter 6).
- Discussion of the design and implementation of GlobeDoc provides insight into building a scalable wide-area distributed application. Also, because GlobeDoc is based on Globe, this provides insight into building applications using Globe and DSOs in general.

1.6 Outline

The rest of this dissertation is structured as follows. Chapter 2 reviews currently implemented and proposed solutions to the performance problems faced by the World Wide Web and discusses how and why they fail to effectively solve these problems. Chapter 3 presents a system view of the solution proposed in this dissertation. Chapter 4 presents a detailed design of GlobeDoc and Chapter 5 presents a detailed design of the Globe Infrastructure Directory Service, both forming key elements of the proposed solution. Performance experiments and results are presented in Chapter 6, and conclusions are presented in Chapter 7.

Chapter 2

Current Solutions

Many attempts have been made to solve the performance problems of the World Wide Web. Because of the growing importance of the Web the problems have been tackled both in academia and in industry.[1] This has resulted in a wide variety of both experimental and commercial solutions, many of which have been widely deployed throughout the Web.

The available solutions can be divided into three categories. The first category contains solutions that aim to improve performance and accessibility of Web content for clients. Examples of these include browser and proxy caching. The second category contains solutions that are applied at the server-side with the aim of reducing the load on individual servers and improving general server performance. Server caches and server clustering are examples of these solutions. Finally, the third category contains solutions based on replication. They aim to improve both client and server performance by moving content closer to clients, reducing the load on individual servers, and reducing usage of network resources. This category includes technologies such as mirroring and content delivery networks. Each of these categories will be discussed in turn.

2.1 Client-Oriented Solutions

Client-oriented solutions are implemented at (or close to) the clients with the intention of speeding up client access to Web sites and Web resources. **Caching** is the main technique used for client-oriented solutions. Caching involves creating temporary copies of previously accessed Web resources and using these to fulfill subsequent requests for those resources.

By placing copies close to clients, much wide-area network traffic can be avoided, reducing both network-induced access delays and network congestion. By limiting network congestion, requests for noncached Web resources also benefit because they can be

[1] When referring to industry, besides commercial entities I also refer to noncommercial open-source projects. Similarly, when referring to commercial products and solutions I also refer to (possibly noncommercial) open-source products and solutions.

sent faster on noncongested networks. Also, because requests for resources whose home servers are unavailable can often be fulfilled by a cache, caches can help to increase the robustness of the Web.

2.1.1 Cache Architectures

Caches come in several flavors. Browser caches are the simplest type of cache. They are followed in complexity by proxy caches, which introduce the idea of shared caches. Proxy caches can be extended to either hierarchical or distributed caches. Finally hierarchical and distributed caches can be combined to form hybrid caches.

Browser Caches

A **browser cache** is the simplest form of Web cache. As the name suggests it is maintained by a client's Web browser. A caching browser simply stores copies of previously accessed Web resources in its local file system. Subsequent requests for cached resources are fulfilled using the locally stored copies, preventing extra wide-area communication between clients and servers. Figure 2.1 shows a user requesting a Web page through a browser cache. Rather than sending the request over the network to the server, the browser simply returns its cached copy of the requested page. Most major Web browsers support browser caches.

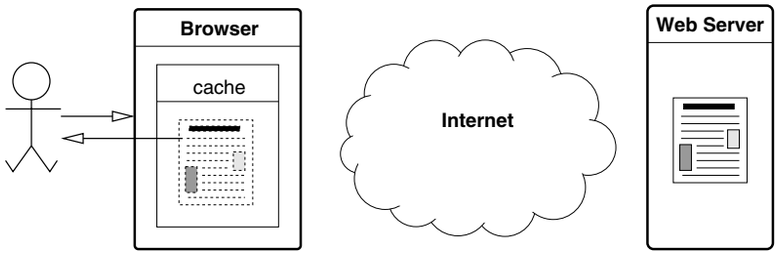


Figure 2.1: Requesting a Web page using a browser cache.

One of the potential drawbacks of browser caches, however, is that the cache contents are not shared between users. This means that if different users in the same organization access the same resources, copies of those resources are stored in all of their browser caches. Not only is this a waste of local disk space, but it can also cause extra wide-area network traffic. When caches are not shared, a user requesting a resource for the first time must always fetch that resource from the home server even though multiple local copies may already be present locally (in other browser caches). By sharing caches, these extra requests can be avoided.

Proxy Caches

Proxy caching [63] allows cached Web resources to be shared by multiple clients and is a solution to the sharing problem. Proxy caches are shared caches stored on and maintained by proxy servers. Figure 2.2 shows a client requesting a Web page through a proxy cache. The proxy server intercepts the client's HTTP request and attempts to fulfill it using resources stored in its local cache. If a request can be fulfilled locally then the cached copy is immediately returned to the client. Otherwise, the proxy server retrieves the resource from its home server and returns that to the client, simultaneously storing a copy in its local cache. Subsequent requests (possibly from other clients) for that resource will receive the cached copy as a reply.

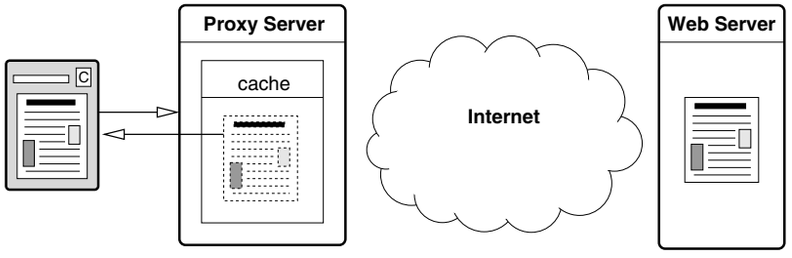


Figure 2.2: Requesting a Web page through a proxy cache.

Proxy caches are generally positioned at the edges of the network, that is, close to the clients that they serve. They are often deployed at an organizational or departmental level, meaning that users within the same organization or department share the same cache. Because proxy caches act as a gateway to the Web, an important issue when designing proxy caches is that the cost of a cache miss should be minimized. In other words, the presence of a proxy cache should not cause a dramatic increase in access latency for cache misses.

Although proxy caches do prevent much wide-area network traffic and do speed up access to many Web resources, they also have some drawbacks. A badly configured, or under-powered proxy server can become a major bottleneck. If, for example, a proxy server receives more requests than it can handle, its performance can degrade rapidly. Because a proxy server intercepts all Web requests, all requests will be delayed due to the server's bad performance. Similarly, the proxy server forms a potential single point of failure. If a proxy goes down, all of its clients may be cut off from the Web.

There are numerous proxy caching solutions in wide use. These include both hardware and software solutions. Many of the proxy servers used today support a wide variety of features besides caching. Examples of extra features include the possibility of forming part of a hierarchical or distributed cache, authentication, firewall services, and content filtering.

Hierarchical Caches

On a cache miss, a proxy cache must directly contact the requested resource's home server. This requires wide-area network communication and is a potential source of performance problems. Hierarchical caches were introduced by the Harvest project [24, 18] in an attempt to avoid this extra communication. In **hierarchical caching**, cache servers are arranged in a tree-like hierarchy at multiple levels of the network.

Figure 2.3 shows a client requesting a Web page through a hierarchical cache. At the lowest level of the hierarchy the servers are similar to the proxy servers described above. They intercept requests (step 1) and attempt to fulfill them using cached resources. However, if a server cannot fulfill a request using cached resources, instead of going directly to the resource's home server it passes the request on to a cache server in a layer above it (step 2). This server attempts to find the requested resource in its local cache, returning it on success or otherwise passing the request on to a server above it. This process continues until either the request is fulfilled by one of the cache servers, or the root of the hierarchy is reached and the request is forwarded to the resource's home server (step 3). When a resource is found (either in a cache or retrieved from its home server) it is passed back down the hierarchy to the client (steps 4, 5, and 6), being stored locally by every cache server it passes through.

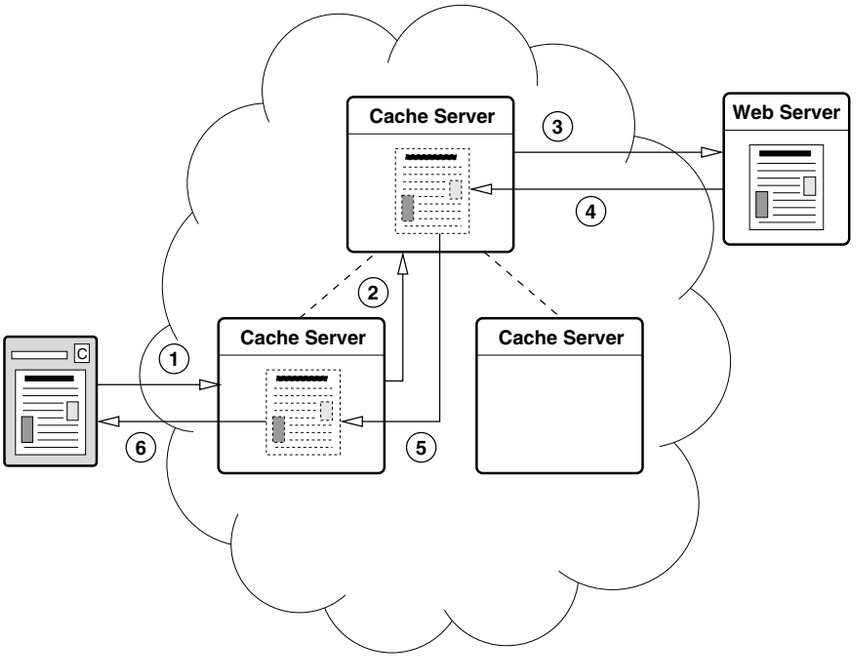


Figure 2.3: Requesting a Web page through a hierarchical cache.

Because popular pages are efficiently disseminated towards the locations that need them, hierarchical caches are potentially more bandwidth efficient than simple proxy caches. They are, however, not without their problems. Due to their multilayered architecture hierarchical caches can introduce significant delays to requests. For example, research has shown that hierarchical caches become inefficient when the number of levels used is greater than four [8]. Also, due to the tree-like nature, high-level servers can become heavily loaded by propagated client requests turning them into bottlenecks. The higher-level nodes are, therefore, often implemented as clusters of servers rather than single servers. There is also an additional cost for storage because keeping copies of Web resources at intermediary levels may introduce a high degree of redundancy. Finally, setting up a cache hierarchy requires placing nodes at key access points in the network. This requires significant coordination among all participating parties.

Despite these problems, hierarchical caching is still widely used. Currently many software caches such as Squid [114] and Netscape Proxy Server [70] support hierarchical caching. Similarly, most hardware caches,[2] such as Cisco Content Engine [25] also offer hierarchical caching facilities.

Distributed Caches

A different extension of the proxy caching idea is to create a single level **distributed cache** [103, 85]. In this architecture all the cache servers are peers, there is no hierarchy. Figure 2.4 shows a client requesting a Web page through a distributed cache. On a cache miss the cache server contacts one of the other cache servers to ask for the resource (step 2). Similar to hierarchical caching, as results are propagated back to the client, intermediate servers store copies in their caches (steps 3 and 4). The servers keep meta-data about the contents of their peers and use this meta-data to decide whom to contact on a cache miss. Distributed caches generally offer an improvement over hierarchical caching because they provide better performance, do not waste as many disk resources, are more robust, and allow for better load balancing.

There are a number of varying approaches to distributed caches, the main difference being the protocols used for routing the requests to peer servers. Inter Cache Protocol (ICP) [114] was designed by the Harvest group for use in hierarchical caching. It allows a cache server to query other servers to determine where a requested resource can be found. Another approach, the Cache Array Routing Protocol (CARP) [105], uses hashing to determine which cache contains a desired resource. CARP hashes URLs to particular servers, thus instead of communicating with other nodes to determine where a resource is located, as in ICP, a server using CARP knows which node to contact based on the hash of the resource's URL. Another approach replaces the upper-level nodes of a hierarchical cache with directory servers [85]. This way, instead of holding copies of resources, upper level nodes simply inform servers where a resource can be found. Cache digests [91] help to reduce inter-cache communication by storing information about peer caches at each server. This way, on a cache miss, a server simply performs a local lookup to find out

[2] A hardware cache is a computer designed to function as a cache and sold with caching software included.

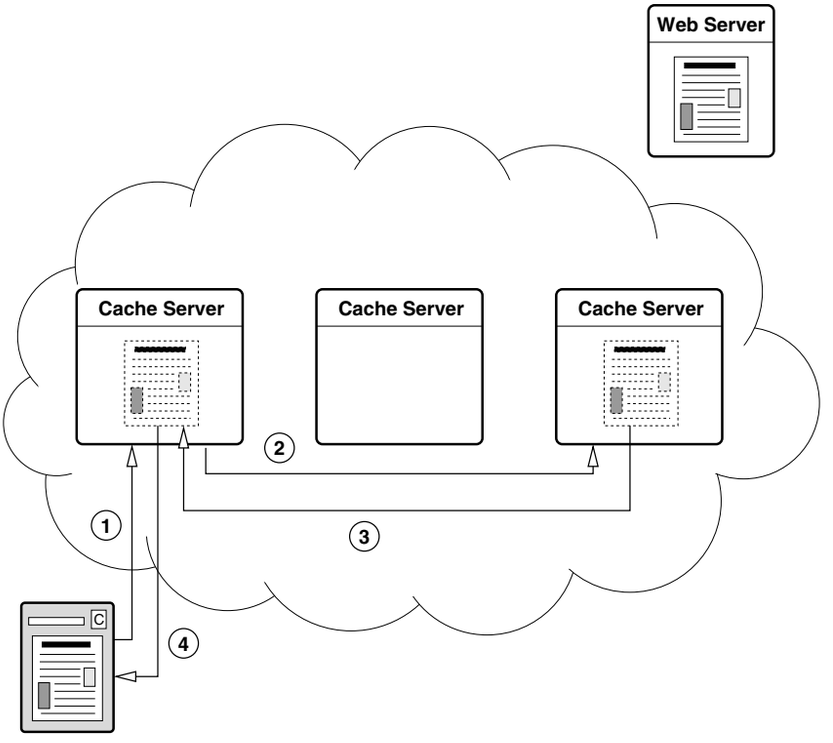


Figure 2.4: Requesting a Web page through a distributed cache.

where the resource is cached. Finally, there is also a central directory approach where information about a group of caches is stored in a central directory [102].

A major drawback of many distributed caching systems is that forwarding requests to peers may actually be slower than going straight to a resource's home server. The performance gained from using distributed caches is highly dependent on the distance between and the performance of other peer cache servers.

Hybrid Caches

The final approach, **hybrid caching**, combines both hierarchical and distributed caching. In this approach caches can forward requests to peers or parents, depending on who offers the best performance. The techniques used are a combination of those from hierarchical and distributed caching. Many hierarchical caches currently act as hybrid caches, for example, in Squid the document is fetched from the parent or neighbor that has the lowest RTT [114]. This allows for improved performance without many of the drawbacks of purely hierarchical or distributed systems.

Peer-to-Peer Caches

A relatively new approach to caching in the Web is based on **peer-to-peer** (P2P) technologies. These are technologies in which end users create distributed services by sharing each others resources, rather than requiring special purpose, or dedicated, servers (such as caching servers or proxies). The Squirrel project [51], for example, applies P2P technology to allow Web browsers to access each others local caches, creating a shared cache network. In Squirrel, browsers use Pastry [93] (a P2P routing protocol) to identify and route to nodes that cache copies of requested resources. The performance of this approach has been found to be comparable to a traditional Web cache solution, without requiring any dedicated caching servers. Note that Squirrel has been designed to work in a small scale environment (such as a LAN). It is unclear whether it would scale to a wide-area environment.

2.1.2 Design Issues

Besides the various cache architectures available, there are a number of architecture-independent issues that affect how caches are designed and deployed.

Cache Coherence

The most important of these issues is **cache coherence**. Caches contain copies of Web resources and in order to be useful the copies they serve should be fresh (i.e., up-to-date). This means that, ideally, when the original Web resource is changed, clients accessing it through a cache should receive the changed, and not the old (or stale), version. The cache coherence problem is similar to the coherence problem in distributed file systems and many of the cache coherency solutions stem from distributed file system research.

Most cache coherency techniques are based on validation checks that verify the freshness of a cached resource. Validation checks are performed using conditional GET messages: HTTP GET requests that have the If-Modified-Since header set. These messages tell the server to return the resource if it has been modified since the given time, and otherwise return a confirmation of the resource's freshness.

Check-Every-Time, the simplest coherency technique, queries the home server every time a cached resource is requested. This guarantees up-to-date data (if the server is reachable), however, it also causes extra network traffic and higher server loads which can lead to significant delays. Client Polling, a less traffic-intensive technique, requires caches to periodically compare a time stamp on the cached copy of a Web resource to the time stamp of the resource at its home server. If it finds that the cached copy is stale then the copy is removed from the cache. Polling does not guarantee that cached copies are always up-to-date. Choosing the right polling interval is, therefore, essential for providing reasonably fresh cached resources.

Callback techniques, or invalidation, require a resource's originating server to keep track of where the resources is cached. When the resource is modified on its home server invalidation messages are sent to each of its cached copies. Although using callbacks ensures that cached copies are kept up-to-date while limiting the bandwidth used, the fact that the originating server has to keep track of and contact all the caches, makes it a nonscalable solution.

Expiration-based coherence techniques rely on assigning an expiration time to Web resources. Requests for resources that have not expired are served from the cache, while requests for expired resources cause the cache to first check with the home server whether the cached copy is still fresh or not before serving the cached copy or retrieving a fresh copy from the home server. The expiration time is usually set as a time-to-live, and is proportional to the amount of time since the resource was last modified. Adaptive techniques modify the time-to-live every time that a cache hit occurs. A popular metric for setting this time-to-live value comes from the Alex file system [23] and is set to one-tenth of the time between when the file was last fetched and the file's creation or modification date (this is generally measured in hours or days).

Check-every-time and callback-based techniques offer the strongest consistency followed closely by adaptive expiration-based techniques. Polling and nonadaptive expiration based techniques offer weaker consistency. Cao and Liu [21] found that supporting strong consistency rather than weak consistency does not necessarily result in increased network bandwidth usage. Other studies, however, show that supporting strong consistency does in fact result in increased network bandwidth use [44]. The difference in results can be attributed to the different workloads and metrics used in each of the studies.

Cache Replacement

Another important issue is **cache replacement**. Caches have practical limits on the amount of space available for storing resources. Often, however, the amount of data that a cache needs to store is larger than the available space, requiring unused cache entries to be replaced with new ones. Cache replacement is concerned with deciding which entries

to replace when new ones must be added. A key requirement for a cache replacement policy is that the cache retains a high hit ratio, that is, that the entries that are least likely to be used are the ones that are replaced.

There are several cache replacement algorithms [115, 2]. They are categorized based on the replacement policy that they implement. The first category includes the traditional replacement policies of Least Recently Used (LRU), Least Frequently Used (LFU), First in First Out (FIFO) and their direct extensions. The second category consists of key-based algorithms, which are based on policies that evict resources using a primary key (using secondary and tertiary keys to break ties). Examples of key-based policies include the SIZE policy, which uses a Web resource's size as the primary key (the largest resource being evicted first) and time since last access as the secondary key. HYPER-G [7] uses frequency of access as a primary key, with time since last access as a secondary and size as a tertiary key. Finally, the third category, cost-based policies, combines a number of different factors in a cost function to determine which resources to evict. Factors used in the cost function include time since last access, entry time of resource in the cache, expiration time, transfer time cost, etc. Many cost functions incorporating these various factors in different ways have been proposed. For example, the Least Normalized Cost Replacement (LNC-R) [96] policy incorporates the access frequency, transfer time and the size, while the Server-assisted policy uses a resource's fetching cost, size, and next request time [29].

The effectiveness of cache replacement is highly dependent on the Web access patterns. For example, the SIZE policy works well in situations where smaller resources are accessed often, however it is not optimal in the opposite situation as the most highly accessed resources are also the once replaced most often. Despite the variety and number of different policies, currently no single policy performs optimally for all access patterns.

Prefetching

An important issue, aimed at improving the hit ratio of caches is prefetching. Research shows that caches reach a maximal hit ratio of only 50% [1]. **Prefetching** is the practice of anticipating future client requests and caching copies of the relevant Web resources before they are actually accessed. There are three types of prefetching: between clients and servers, between clients and proxies, and between proxies and servers. Prefetching algorithms can use locally collected information, such as request patterns at the client or proxy, or information collected at the server, such as a resource's access history, to decide what to prefetch.

Studies [75, 17] show that prefetching directly from the server can be effective and significantly decreases client latency. The prefetching comes at a price, however, and increases both the amount of network traffic and the burstiness of the traffic, which leads to increased general network delays. Prefetching between proxies and servers has also been found to reduce client latency. Pushing content from servers to proxy caches has also been suggested, however, this requires cooperation from the servers. Prefetching between clients and proxies shows promising results in reducing the client latency. In this

scenario the proxies predict which documents the users might be interested in and push these out to the client during periods of idle network activity.

Dynamic Content

A currently unsolved, yet increasingly important issue in caching is how to handle **dynamically generated content**. Active caching [22] attempts to address this problem by introducing applets in the cache that can customize Web documents. Servers provide partially generated documents and applets that can be cached. When these documents are accessed from the cache, the applet locally (on the cache server) generates the missing content and returns the completed document.

Although active caching allows some dynamic content to be cached, it does not solve the problem for all types of dynamic content. Content that is generated based on a database, for example, must still be generated at the server, as the databases are usually centrally located at the servers.

Other issues

An important issue to content providers is accounting for client accesses through caches. For many providers it is important to know how many customers have accessed their pages. For example, if a page has advertisements on it, the content provider needs to know how many clients accessed the page so that the advertiser can be charged appropriately. If a client accesses the page through a cache, the content provider will not see the access, and lose revenue. Many caches, therefore, include some sort of accounting functionality.

2.2 Server-Side Solutions

Server-side solutions are applied at the server-side with the aim of reducing load and improving performance and accessibility of individual Web servers. As opposed to the client-side solutions, which aim to improve access performance of clients to all Web sites, server-side solutions improve access only to the Web sites served by particular servers. There are three main approaches to server-side solutions. The first is upgrading and improving the hardware and software used, the second is server caching, and the third is server clustering.

Upgrading Web server hardware is the most obvious approach to improving Web server performance. Adding more and faster processors, increasing available memory, adding larger capacity disks, etc. can all help to improve the performance of a Web server and prevent server overloading. Like hardware improvements, software changes to the operating system and Web server can also help improve performance. Such software improvements include tuning operating parameters, rewriting and redesigning critical code, optimizing operating system code, and moving core Web server functionality into the kernel.

One problem with simply replacing hardware is that it is expensive and does not take full advantage of previous hardware investments. Moreover it is a short term solution

that does not scale well with regards to growing site popularity (i.e., as the site grows more popular, hardware must constantly be replaced). A long term solution should be incrementally scalable, that is, it should be able to grow as needed.

Another technique that helps ease the load on servers is server caching. **Server caching** involves caching popular resources either in a Web server's main memory or on a separate machine so that they can be retrieved more quickly and without unduly loading the Web server. When implemented outside a Web server, server caches often perform other CPU-bound tasks (such as SSL processing) to further ease the Web server's load. Server caches help to decrease server load, increase the rate of request processing, and decrease response time. Besides the benefits, however, server caching also faces many of the issues and problems involved with caching as described above. It is an appropriate technique for servers where a large portion of their most popular (static) resources receive most of requests. For servers whose content is highly dynamic or whose requests are more evenly distributed they provide less benefit.

2.2.1 Server clustering

Server clustering involves having a pool of servers act as a single Web server when receiving and processing requests. A clustering solution provides incremental scalability, because extra servers can be added to the pool as needed.

Server clusters must be transparent to clients, that is, clients must always see a server cluster as a single server. Clustering is, however, not always transparent to the Web servers themselves. In some clustering solutions the servers must run specialized software to take part in a cluster. The main drawback of this approach is that the servers cannot run commodity Web servers, which may greatly limit the kinds of content that can be served (for example, if the specialized software does not support PHP or JSP, then content providers cannot use these technologies to create their pages). When clustering is transparent to Web servers, regular commodity Web server software can be run on the servers. Although the former approach allows a much greater degree of optimization and therefore performance gains, the latter approach allows a much broader base of Web server technologies to be supported and is more cost effective (because commodity software is usually less expensive than specialized software).

The general architecture of a clustering solution consists of a **dispatcher** and any number of servers in a **server pool**. The dispatcher receives all HTTP requests for the cluster and dispatches the requests to individual servers in the pool. The requests are dispatched based on load balancing and distribution algorithms implemented by the dispatcher. There are three categories of clustering architectures, the major differences being the logical layers in a protocol stack at which requests are intercepted and dispatched to the servers in the pool. This dispatching can be done at the data link layer (OSI layer 2), the network layer (OSI layer 3), and the application layer (OSI layer 7).

Layer 2 clustering

Layer 2 clustering, also called L4/2 clustering because the dispatcher switches on the OSI layer 4 (i.e., TCP) data and forwards packets at layer 2, is one of the more efficient Web server clustering architectures. In this architecture the dispatcher shares the same network layer (layer 3) address with all the servers. In practice this means that the dispatcher and servers all share the same IP address. Figure 2.5 shows a request being dispatched by a layer 2 dispatcher. When a request comes in to the dispatcher, the dispatcher chooses a server for the request (step 2), rewrites the layer 2 information of that packet and forwards the request to the chosen server's layer 2 (MAC) address (step 3). The dispatcher chooses a server based on the incoming packet's TCP data. For example, if the incoming packet is a TCP connection initiation, the dispatcher chooses a server, forwards the packet to that server, and remembers that packets relating to that connection should always be forwarded to that server. When the server receives the request, it processes it and returns a reply (step 4) directly to the client (this is possible because the server has the same IP address as the dispatcher).

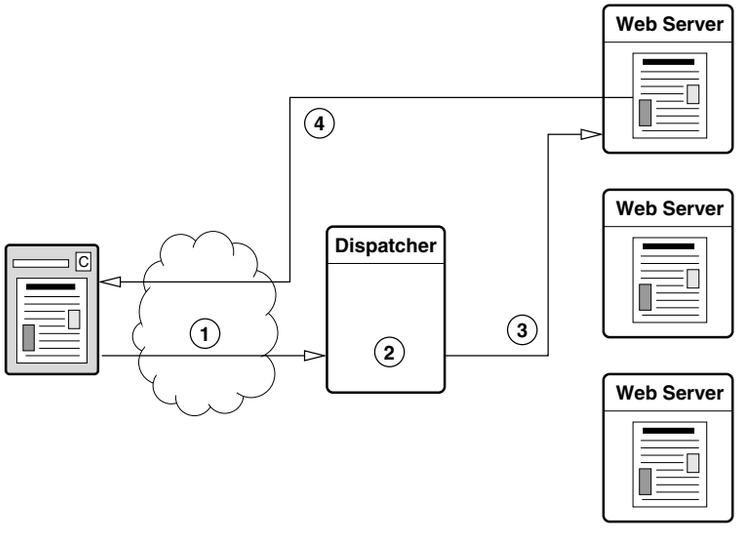


Figure 2.5: Requesting a Web page from a layer 2 server cluster.

Layer 2 clustering is efficient because the dispatcher has very little overhead (for example, because it changes a packet's layer 2 data it does not have to recompute an IP checksum) and does not have to forward or process replies. One restriction on layer 2 clustering is that the dispatcher and servers must be on the same physical network. Examples of systems implementing layer 2 clustering systems include ONE-IP [31] and LSMAC [40].

Layer 3 clustering

Layer 3 clustering, also called L4/3 clustering because the dispatcher switches on the OSI layer 4 data and forwards packets at layer 3, was one of the first Web server clustering architectures experimented with. In this architecture the dispatcher and all the servers have unique IP addresses. The dispatcher acts as a gateway for the servers. Figure 2.6 shows a request being dispatched by a layer 3 dispatcher. When a request comes in the dispatcher analyzes it and chooses a server to forward it to (step 2). Once again the decision of where to forward a request is made based on TCP (i.e., layer 4) data. A packet is forwarded by rewriting the layer 3 (IP) address to that of the chosen server and sending it on to that server (step 3). Note that by changing the IP address in the packet, the dispatcher has to recompute all packet checksums. When a server receives a packet, it processes it and returns a reply to the dispatcher (step 4). Upon receiving a reply the dispatcher resets the packet's source address to its own address and sends it on to the client (step 5).

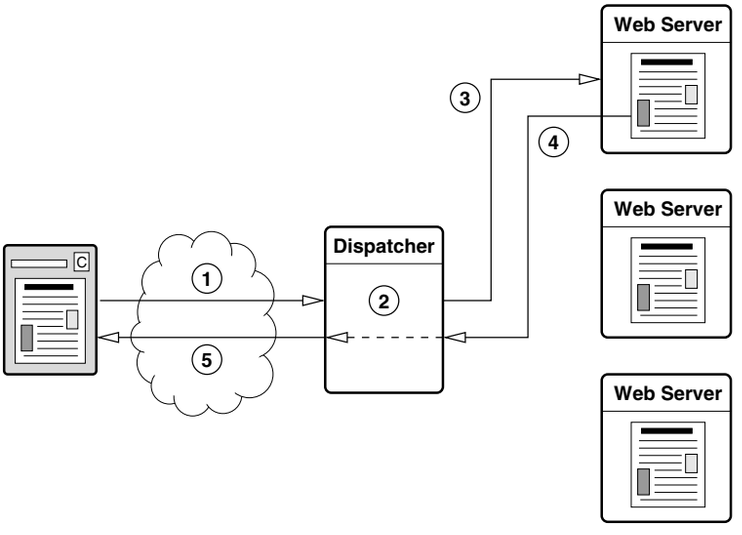


Figure 2.6: Requesting a Web page from a layer 3 server cluster.

Layer 3 clustering is less efficient than layer 2 clustering because the dispatcher not only has to process incoming and outgoing messages, but the modification of packets is more expensive as it involves the recomputing of checksums. Layer 3 clustering is, however, easier to set up on commodity hardware/software. Examples of layer 3 clustering solutions include Magicrouter [6], LocalDirector [26] and LSNAT [41].

Layer 7 clustering

In **layer 7 clustering** (also known as content-based clustering) the dispatcher switches on the application level (i.e., HTTP) data. Unlike the previous two types of clustering

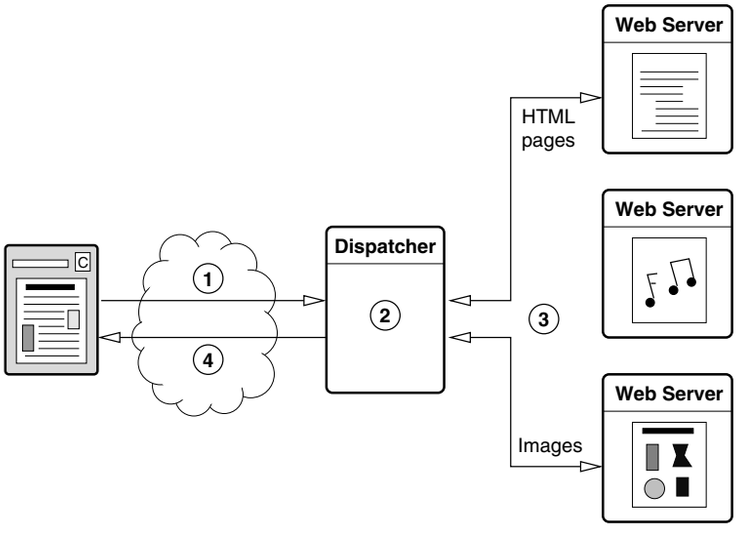


Figure 2.7: Requesting Web resources from a layer 7 clustered server.

where the dispatchers inspect individual network packets, in layer 7 clustering the dispatcher inspects whole HTTP requests. The requests are forwarded to servers depending on the content of the request. Figure 2.7 shows how a layer 7 dispatcher dispatches requests to different servers depending on the resource being requested. Examples of layer 7 clustering solutions include LARD [76] and IBM Web Accelerator [61].

In LARD, each server in the pool is designated to handle a particular class of resources (e.g., HTML pages, images, etc.). As requests come in, the dispatcher classifies the request and forwards it to a server handling that class of resources. Requests are forwarded to servers using TCP handoff, a technique similar to the IP rewriting done in Layer 3 clustering. Unlike layer 3 clustering, however, in order to inspect the HTTP content the dispatcher must first establish a connection with the client, then hand the established connection off to a server. TCP handoff requires modified protocol stacks on the dispatcher and servers.

IBM's Web Accelerator takes a different approach and combines layer 7 switching and server caching with layer 2 packet forwarding. In this approach the dispatcher combines a caching and dispatching functionality. When an HTTP request arrives, the dispatcher tries to fulfill it from its cache. If the requested resource is not in the cache, the request is dispatched to a server using layer 2 clustering techniques.

The main benefit of layer 7 solutions is that it makes it possible to have specialized server nodes. For example, it is possible to assign a very powerful machine to process requests for dynamically generated pages, while a less powerful machine processes simple static requests. A drawback of this type of clustering, however, is that the dispatcher must analyze and process all incoming and outgoing requests. Nevertheless, when combined

with server caches, layer 7 clusters are capable of providing the best performance of all three architectures [97]. In fact, the best layer 7 dispatchers are capable of saturating typical wide-area network links. This development has helped shift the Web performance bottleneck from the ability of Web servers to serve content, to the ability of the networks to deliver content to clients.

Design Issues

Despite the differences among these three categories of clustering solutions there are number of common issues that all clustering architectures must address.

The most important issues are **request distribution** and **load balancing**. When a request arrives at the dispatcher, the dispatcher must decide to which server the request will be forwarded. Requests should be sent out in such a way that all the servers are evenly loaded. The simplest strategies are random distribution and round-robin. Other strategies choose the least loaded server, or the server with the fastest response time. Layer 7 dispatchers often base their decisions on the type of resource requested. With regards to request distribution, **transparency** is also an important issue. As mentioned earlier, server clusters must be transparent, that is, a client must always see a server cluster as a single server.

An important practical issue for clustering systems is **fault tolerance**. Some systems provide a hot spare to take over if anything should happen to the dispatcher. Others allow one of the servers to become a new dispatcher. Similarly, dispatchers must deal with unavailable servers by not forwarding requests to them. However, they must also be able to start forwarding to them when the servers come back up and join the pool.

Dispatchers are the most likely bottlenecks in clusters, thus it is important to make sure that they can handle the load. Unlike the servers in a pool, if a dispatcher becomes overloaded it is not possible to simply add another dispatcher, it must be replaced with a more powerful one. As with server caches, dispatchers may also perform other functions such as being a firewall for the clustered Web servers, performing SSL processing and offering support for other application protocols such as FTP.

2.3 Replication-based Solutions

Between client-oriented and server-side solutions lie replication-based solutions. Replication-based solutions involve creating (semi) permanent copies of a Web site's contents on remote servers and redirecting clients requests to those servers. The goals of replication-based solutions are to improve both client and server performance. Client performance is improved when clients access content from servers closer to them, while servers benefit when a significant part of the load is handled by other machines. Unlike caching, where resource copies are created on behalf of the clients, in replication replicas are created on behalf of the server. Unlike clustering, though, where the servers are

close together (both geographically and network topologically[3]) servers hosting replicas are usually spread widely over the Internet.

2.3.1 Replication Issues

Replication solutions face many of the same issues as caching and clustering solutions. As in caching, one of the most important issues in replication is **coherence**. Replicas of Web sites must be kept consistent with the main site. Unlike caching, however, there is not necessarily only one master copy of every resource. Depending on the policy of the site maintainer(s), changes can possibly be made at multiple replica servers. Furthermore, it is often possible for all the replicas to know about each other allowing coherence policies based on callbacks or invalidations to be used. The coherency problem faced in replication is very similar to that faced in distributed file systems and many solutions can be directly applied to replicated Web sites. One important difference between distributed file system research and Web replication, however, is that the distances (in terms of network topological and geographic distance) on the Web are often much greater, requiring solutions to scale over a large area.

Three other important issues in replication are transparency, routing, and load balancing. **Transparency** and **load balancing** are similar to the equivalent issues in clustering: how to transparently redirect a client to the appropriate replica, and how to make sure that the load is evenly distributed over the available replicas. **Routing** (directing clients to the proper replica) is more difficult in the context of replication than in clustering because of the larger (geographic and network) scale involved. Not only is it necessary to route clients to a replica that contains the content that they want, but it is also important to route clients to replicas that are closest to them, so as to decrease latency and avoid unnecessary bandwidth use. Similarly, many of the low-level techniques applied to make server clustering transparent cannot be applied in wide-area networks. For example, it is not possible to simply rewrite MAC addresses when the servers are not on the same local area network.

As mentioned in the context of caching, dynamic content is playing an ever greater role in the Web. Like caching, replication, of dynamic content poses a great problem. One advantage that replication might have over caching is that replica servers are largely under the content provider's control. This makes it possible for *all* content (including databases) to be replicated so that dynamic content can be created on-the-fly at replicas as well as at the master site. Unfortunately, due to coherence issues, doing so presents problems of its own.

Heterogeneity can also have an important effect on replication. In many situations replicas are hosted on different kinds of servers with different capacities. Taking different characteristics of host platforms into account can further complicate the problem of routing client requests and balancing the load over all replicas. For example, a server with a low bandwidth network connection will be able to handle fewer requests than one with a higher bandwidth connection. This must be taken into account when routing client

[3]Note that host proximity in terms of network topology does not imply geographical proximity. For example, all of a company's offices worldwide may be network topologically but not geographically close.

requests to these servers. Similarly, different operating platforms can affect how dynamically generated data is replicated. For example, if the code used to dynamically generate content relies on libraries available only on Windows platforms, then that code cannot be replicated on Unix based servers.

Finally, as in clustering, **fault tolerance** is an important issue in replication. Despite the fact that replication can improve the fault tolerance of a Web site (i.e., it takes more than one server crash or localized network outage to bring a site down), the replication mechanism must be able to deal with unavailable replica servers. Thus, for example, client requests must not be routed to nonavailable servers.

2.3.2 Mirroring

The simplest form of replication on the Web is mirroring. **Mirroring** involves creating a copy of a site on a remote server and informing clients about the server. There is no formal (nor informal, but widely implemented) method or protocol for mirroring, making it an ad-hoc solution. The simplest and most common approach to mirroring is to create copies of a Web site on multiple servers, include a list of references to these servers at the main site, and ask clients to visit one of the copies instead of the main site. This scheme involves no automated mechanisms to route clients or to keep content consistent. Similarly, this scheme is not at all transparent as it requires effort on a user's part to use a mirror. Figure 2.8 shows a client requesting a Web page form a mirror site. Note that the client directly accesses the mirror server, without having any contact with the home server.

Many attempts have been made to automate mirroring both for the clients and site (or mirror) maintainers. As such, many schemes have been developed for automatically redirecting clients to mirror sites. These range from simple schemes using round-robin DNS scheduling [54] to more complex ones involving analyzing client location (based on domain name or IP address) to help route clients to their closest available replica. In the future, IPv6 anycasting [46, 33] can provide a good way to direct clients to appropriate mirror sites. Similarly, many different schemes for automatically updating content have also been developed. These range from simple schemes that regularly copy the master site to the replicas, to more complex schemes involving network synchronization protocols [104] which allow replicas to remain consistent while limiting bandwidth usage. It is important to note that application of such schemes is ad-hoc and up to the maintainer of a Web site to implement. The extent of solutions implemented is usually dependent on the site's requirements. Thus, for some sites (e.g., those that are not updated often) manual replication and weak consistency are sufficient, while other sites (e.g., large sites that are maintained and updated by a large group of people) require strong consistency and implement automatic replication policies.

2.3.3 Content Distribution Networks

Building and running a mirroring solution as described above generally requires a considerable effort from the content provider. **Content distribution networks** (CDNs) are

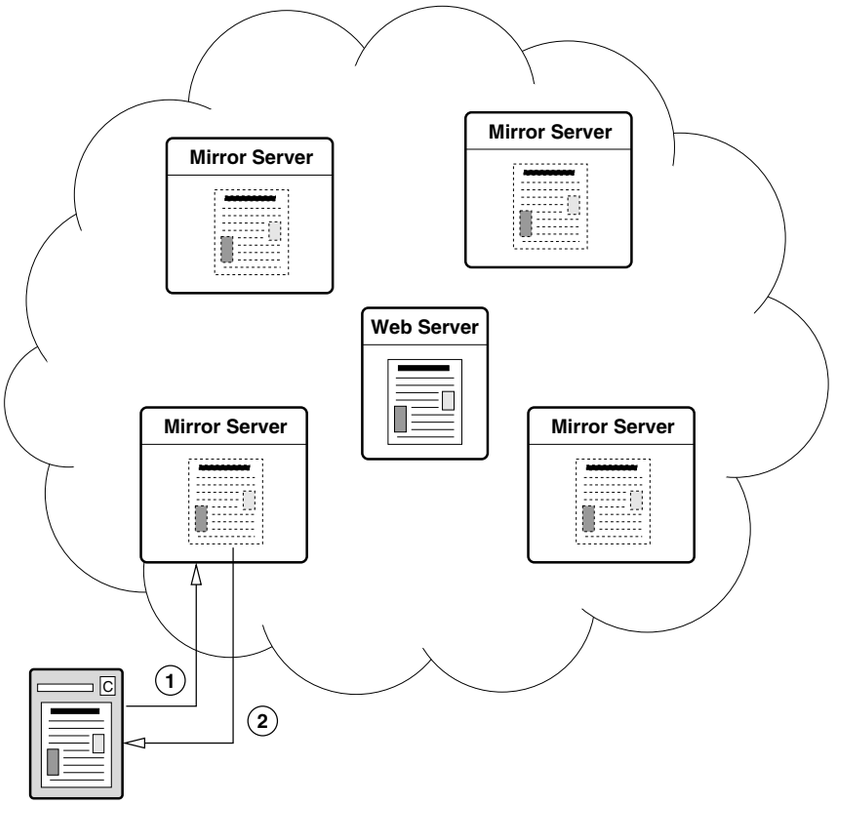


Figure 2.8: Requesting a Web page from a mirror server.

a category of replication solutions that remove this burden from the content provider and place it with a specialized (CDN) service provider. A content distribution network is a widely distributed network of servers that offers replication services to content providers. Content providers who make use of CDN services are assured that their content will be replicated on servers near their clients, thereby providing improved performance to their clients. Likewise, use of a CDN also eases the load on the content provider's own servers and reduces overall congestion on the Internet.

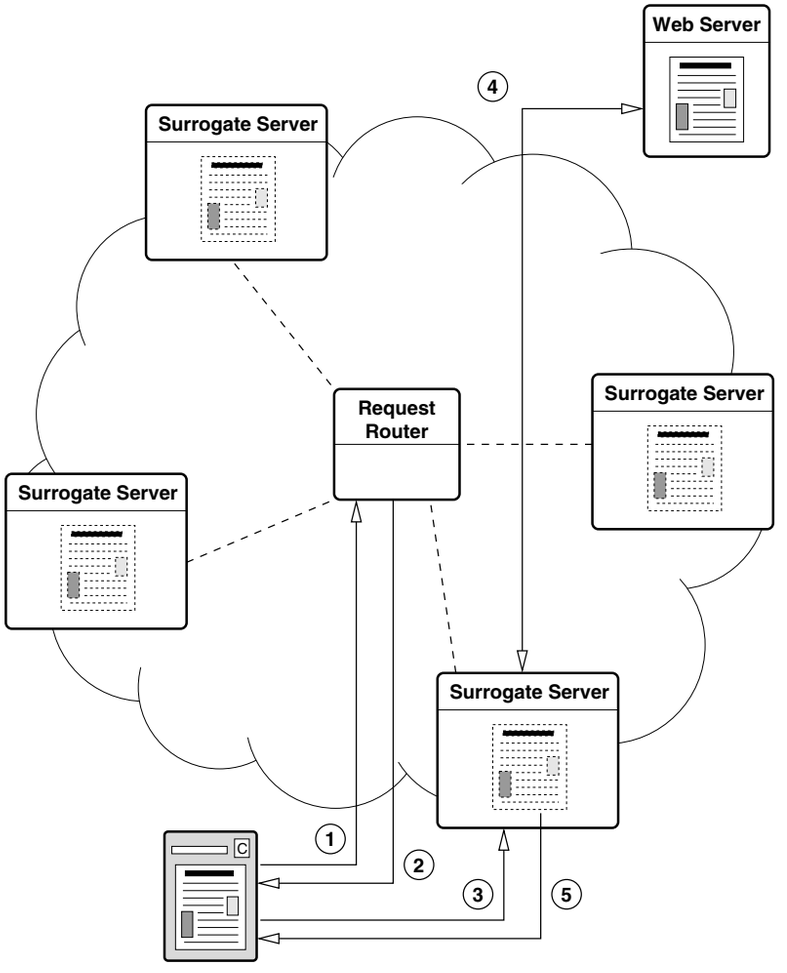


Figure 2.9: Requesting a Web page through a Content Distribution Network.

Figure 2.9 shows a client requesting a Web page through a CDN. When attempting to access a resource hosted on a CDN, a client first sends a request to the resource's home

server. This request is intercepted by the CDN's request router or forwarded by the home server to such a router (step 1).

The **request router** analyzes the request and chooses a surrogate server to forward the request to. A **surrogate server** is a server that stores and serves replicas of hosted Web documents. The surrogate server is chosen such that it fulfills a number of CDN specific criteria such as geographic or network proximity to the client, low load, etc. Once a surrogate is chosen the request router instructs the client (step 2) to redirect its request to that surrogate (step 3). When the request arrives at the surrogate, the requested resource may already be locally available or the surrogate may have to fetch it from the home server (step 4). Once the resource is locally available at the surrogate, the request is fulfilled by returning the local copy of the requested resource directly to the client (step 5). Depending on the design of the CDN, subsequent requests (for the same or different resources) may be handled in the same way or sent directly to the surrogate.

CDN Architecture

At an architectural level a CDN consists of at least two components: the request router and the surrogate servers. In some CDNs the network connection between surrogate servers may be a private network (e.g., a satellite link) in which case it also forms part of the architecture. In other CDNs communication is performed over existing networks such as the Internet. The home server (as shown in Figure 2.9) is generally not part of the CDN architecture.

Request Router A CDN's request router component is responsible for routing client requests to appropriate surrogate servers. There are three types of request-routing mechanism currently used in CDNs: DNS-based routing, transport-layer routing, and application-layer routing.

DNS-based routing involves the use of specialized DNS servers that resolve DNS names to addresses of different surrogate servers based on customized name resolution and routing policies. For example, when a client attempts to resolve the name of a Web server whose content is hosted on a CDN, the DNS server will resolve that name to the address of the client's closest surrogate.

Transport-layer routing involves the inspection of transport layer (e.g., TCP) packet data to make routing decisions. Routing decisions are made based mainly on source and destination IP addresses of packets. This is similar to the approach taken in L4/2 and L4/3 server clustering.

Finally, application-layer routing involves making decisions based on application layer messages. A request router can, for example, determine the type of resource requested based on HTTP headers and route the request to an appropriate server. Other techniques involve using cookies or other site-specific identifiers to determine or remember where to route a client's requests to. A different kind of application-layer technique involves modifying the returned content to control routing of subsequent requests. For example, the URLs embedded in an HTML page may be modified to refer directly to an appropriate surrogate.

Note that these mechanisms are not exclusive, they may be combined to increase request-routing performance. For example, while a first request may be routed using DNS-based routing, the content returned by a surrogate may be modified so that subsequent requests for embedded content will be requested directly from that surrogate.

Besides the mechanism used, another important aspect of request routing is surrogate selection, that is, deciding which surrogate a request should be directed to. While it is important to route requests to good servers, that is, ones that will provide good performance for the client, requests should also be routed in such a way that the overall load on the CDN's surrogates is evenly balanced. The decision of where a request is routed to is generally based on some combination of geographic mapping (finding a surrogate in close geographic proximity to the client), network mapping (finding a surrogate in close network proximity to the client), delay mapping (finding a surrogate with the shortest delay to the client), content type, content size, number of requests and system load. Research indicates that deciding on appropriate proximity metrics is not easy [74, 38]. Furthermore, because the algorithms used for choosing the right surrogate are an important factor of a CDN's performance they are often considered trade secrets.

Surrogate Servers A CDN's surrogate servers host replicas of the content served by the CDN. Important issues with regards to surrogate servers are server placement, replica placement and coherency. Generally, surrogate servers are placed as close to potential clients as possible. For example, in large commercial CDNs, surrogate servers are often hosted by Internet service providers (ISPs) so that the ISP's clients will be in the same autonomous system (AS) as the surrogate. This allows the clients to benefit from better access to content replicated on the surrogate server.

When deciding where to place replicas the goal is to place them such that the access latency perceived by clients and overall network bandwidth consumption when transferring data to clients are minimized. Much research has been done regarding the optimum placement of replicas. Proposed approaches include theoretical models based on graph theory as well as heuristic models that use operational data (such as workload patterns and network topology) to determine where to place the servers [87]. In practice, the theoretical approaches are difficult to apply because they are either too computationally expensive or are too limited (e.g., they consider only proximity and do not consider network characteristics or workload). According to [87] and [53] a heuristic greedy algorithm (which chooses surrogates that best minimize access costs such as latency and bandwidth for all clients) is the best approach for determining where to place replicas.

Besides deciding *where* to place replicas, an important issue is also *when* to create replicas. Replicas can be created on-demand, that is, at the time that a client requests a Web page and that page's content is not available at a surrogate server. Replicas can also be created in advance, so that a surrogate will always contain the right replica when a client is redirected to it. In the first approach, replica placement is dynamic and the choices of where to place replicas can take changing conditions (such as network congestion, Web resource usage, etc.) into account. On the other hand, this on-demand replication means that there is a chance that a client may have to wait until the surrogate actually receives the

contents before being able to access it. This is especially problematic when the content is large (e.g., a video).

Finally, replicas at surrogate servers must be kept consistent with the originals hosted on the home servers. As with cache servers, there are numerous strategies possible, offering a range of consistency levels. Unlike caching, however, in a CDN the location of all replicas is known, making invalidation a viable solution. Likewise, communication between surrogates does not have to comply with the HTTP protocol (or other Web standards), which means that other, more flexible, approaches are also possible. For example, Ninan et al [72] have suggested using leases and cooperation between surrogates to maintain consistency of replicated content.

Examples of CDNs

Akamai With over 13 000 surrogate servers Akamai [3] deploys the worlds largest and best known CDN. Akamai uses DNS-based request routing to route client browsers to surrogate servers. Content providers who distribute their content through Akamai's CDN must adapt their HTML pages. This involves replacing local URL references (i.e., those that refer to the content provider's own server) by URL references that contain an Akamai host name [60]. When a client attempts to resolve an Akamai URL, the Akamai DNS server finds a surrogate close to the client and returns this surrogate's address. In this way, after the initial contact with a Web site's home server, all subsequent resources are retrieved from the nearest surrogate server. Note that, besides network-topological distance, the request routing algorithm also takes load balancing of surrogate servers into account. The details of this algorithm are not publicly available.

With regards to replication, Akamai surrogate servers act as cache servers, that is, content is retrieved by a surrogate server only when it is requested by a client. Consistency of replicated content is maintained using expiration. Once content has been retrieved by a surrogate it is cached for a given time which is determined by a resource's time-to-live value. Akamai also allows cached content to be invalidated by the content provider. When content is invalidated, surrogates must retrieve content from the home server the next time it is requested.

RaDaR Unlike Akamai where content is cached on-demand by surrogate servers, in RaDaR [88] content is replicated to surrogates using a dynamic replication algorithm. This dynamic replication algorithm uses information about server load and client distribution when deciding where to place replicas. Because the replica placement decisions are dynamic they can change in response to changes in access patterns, network characteristics, or the available surrogate servers.

The overall RaDaR architecture is slightly different than the general CDN architecture presented earlier in Figure 2.9. In particular, the functionality of the surrogate server is divided over three different servers: the distributor, the redirector and the replica server, as shown in Figure 2.10.

Like Akamai, RaDaR uses DNS-based request routing. Unlike Akamai, however, RaDaR does not require the rewriting of Web pages. Instead, RaDaR requires the content

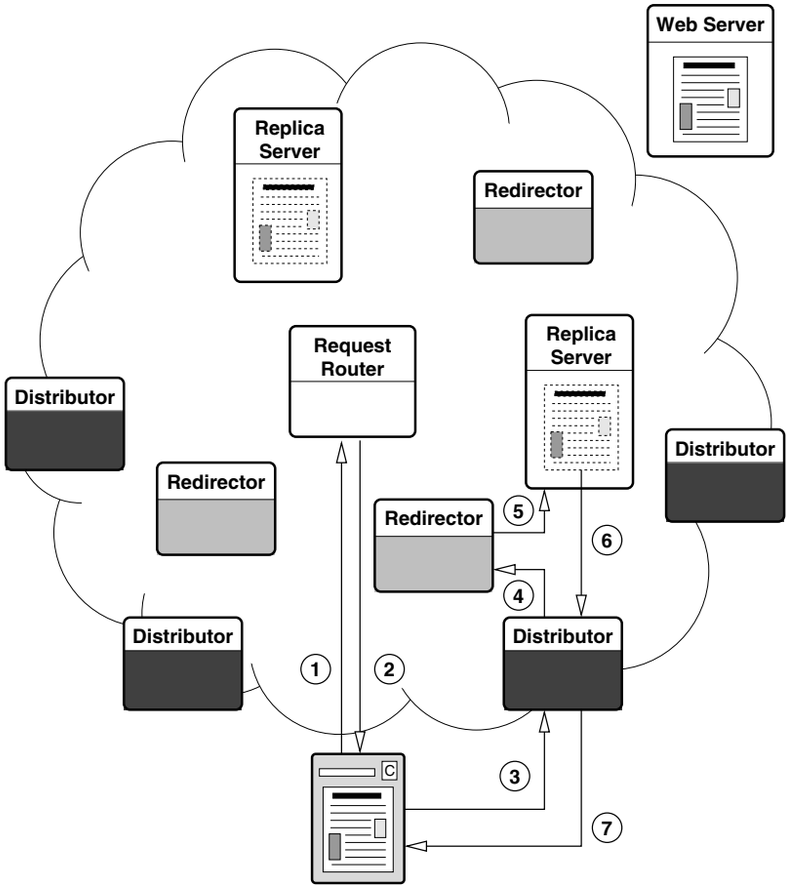


Figure 2.10: The RaDaR architecture.

provider to delegate the task of resolving its Web site address (i.e., resolving the host name used in its URLs) to a RaDaR enabled DNS server. In this way, when a client attempts to resolve the address of a Web site hosted by RaDaR, the RaDaR DNS server determines the closest distributor and returns that distributor's address (steps 1 and 2).

When the distributor receives a request for a Web resource (step 3), it contacts an appropriate redirector (step 4) to determine where a copy of that resource is located. In RaDaR each redirector is responsible for keeping track of the locations of a fixed set of resources. Finding an appropriate redirector for a given resource is done using the hash of the requested resource's name. Given a resource name, the redirector locates the nearest replica server hosting the requested resource and forwards the request to that server (step 5). This replica server returns the resource directly to the distributor (step 6), who in turn returns it to the client (step 7).

Replicas hosted on replica servers are kept consistent with their originals using a primary-backup strategy [5]. In this strategy one of the replicas is designated as the primary replica, while the rest are considered backups. All updates are first processed by the primary and are then propagated to the backups.

2.4 Per-document Solutions

A common characteristic of the solutions surveyed in this chapter is that each proposed solution attempts to impose a **one-size-fits-all policy** on all Web resources (or at least all resources that are accessed by or take part in the solution). However, in the Web, there is a large diversity in Web document characteristics. For example, document sizes, document popularity, the geographical location of clients and the frequency of updates vary greatly from one document to another [83].

As stated in Chapter 1 we claim that no single policy can be good enough in all cases. Thus, instead of attempting to design a single universally optimal policy, several specialized policies should be used simultaneously. Depending on its characteristics, each document should be replicated using the policy best-suited for that particular document.

This section describes a simulation experiment performed to support this claim. In this experiment simulations, based on traces collected from the Vrije Universiteit's Web server, were performed in order to determine whether applying **per-document policies** would result in performance increases when compared to one-size-fits-all policies. The results show a significant performance improvement with respect to end-user delays, wide-area network traffic and document consistency.

This section is organized as follows: Section 2.4.1 describes the experimental setup, Section 2.4.2 describes the configurations used, and Section 2.4.3 discusses the methods designed for associating optimal policies to documents and presents the simulation results.

2.4.1 Experimental Setup

The experiment involved simulating clients performing requests for Web documents (HTML pages) and measuring the performance (i.e., the time it takes to retrieve the doc-

ument) for each of these documents. The documents were distributed over Web servers using various replication configurations. In order to make the simulations as realistic as possible, they were not based on statistical traffic models, but rather on real traces and performance measurements.

The request characteristics (i.e., when and from where Web documents were requested) and the set of Web documents requested were based on traces collected at the Vrije Universiteit's Web server. In the simulation the whole set of Web documents was hosted on a single server while clients (located worldwide) retrieved the documents from the server or from intermediate servers (i.e., caches or replicas). During the experiment we investigated the effect of various replication policies on the quality of service perceived by the users.

A single run of the experiment consisted of simulating clients performing requests for a particular Web document, where the Web document was replicated according to a specific replication configuration. The exact configurations used are described later in Section 2.4.2. One simulation was run per document per configuration. For each run we measured (i) the delay at the clients, (ii) how many clients received stale copies, and (iii) the network bandwidth consumed. We then accumulated each of these values over all runs to determine the performance of any configuration over the entire set of documents.

Document Model

All documents requested were static documents. It was assumed that all documents were updated at the main server. It was also assumed that the server could detect such updates in order to propagate appropriate information to the copies (if the replication policy required it).

Placement of Intermediate Servers

To reliably simulate replication policies, it was necessary to first determine how many document copies were necessary and to decide which client would use which copy. The extent to which this choice reflects reality strongly determines the validity of the final results. Therefore, it was decided to take the actual network topology into account in order to let adjacent clients share copies, minimize bandwidth, and so on.

Clients were grouped based on the autonomous systems hosting them. Autonomous systems (or *ASes*) are used to achieve efficient worldwide routing of IP packets [16]. Each AS is a group of nodes interconnected by network links. Its managers are responsible for routing inside their domain. They export information only about their relations to other ASes, such as which ASes they can receive packets from, and which ASes they can send packets to. Worldwide routing algorithms use this information to determine the optimal route between two arbitrary machines on the Internet.

An interesting feature of ASes is that they generally consist of relatively large groups of hosts that are close to each other with respect to the network topology. As such, it is safe to assume that the network connection performance is much better inside an AS than between two ASes.

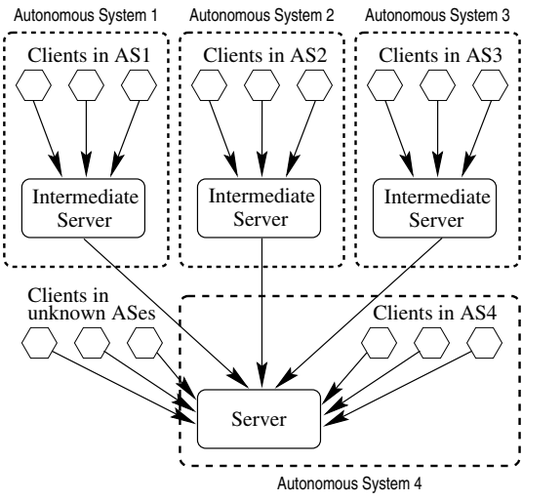


Figure 2.11: Experiment configuration.

This feature of ASes led to the placement of at most one intermediate server (cache or replica) per AS. It was also decided to bind all clients to their AS's intermediate server (see Figure 2.11). This rule has two exceptions. First, it would be pointless to create an intermediate server in the same AS as the master server: clients located in this AS can directly access the master as well. Second, the few clients for which no AS could be determined would also access the master server directly.

Collecting Traces

To simulate the replication of documents, it was necessary to register each event that could happen to a document: creation, update or request. The Web server logs provided the necessary information about the requests for documents: request time and IP address of the clients. The Web server's file system was monitored to detect any creation or update of a file located in the Web server's directories. In this way, it was possible to obtain information about the update times and the new sizes of documents. Document creation was handled as a special case of an update. We also measured the network performance between the master server and each AS in our traces.

Measuring the Network Performance

To measure the network performance from the master server to each AS in the experiment, five hosts were randomly chosen inside each AS. For each of these hosts, a number of *ping* packets of different sizes were sent, and the round-trip time measured. Performing a linear regression analysis on these results provided an approximation of the latency and bandwidth of the network connection to these hosts. The latency corresponds to half

of the round-trip delay for a packet of size 0; the bandwidth corresponds to additional delays due to a packet's size[4]. (see Figure 2.12). Symmetrical network performances are assumed, that is, the performance from the master server to any host is considered equal to the performance from that host to the master server[5].

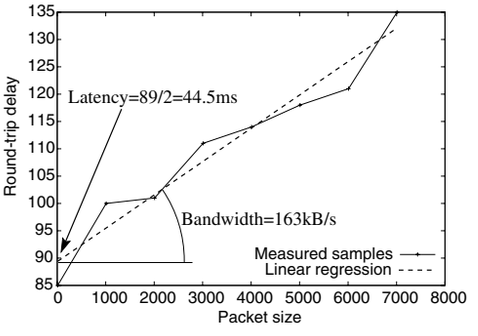


Figure 2.12: Determining the network performance to a host based on ping samples.

The Simulations

The simulations were based on a modified version of Saperlipopette, a discrete event simulator of distributed Web caches [81]. Saperlipopette allows the simulation of any number of caches, each cache being defined by its internal policies (replacement, consistency, cooperation) and its size. Given information about the network performance, Saperlipopette can replay trace files and calculate metrics such as the cache hit rates, document access delays and the consistency of delivered documents. Saperlipopette was extended to implement permanent replicas in addition to caches. Also, more consistency policies, such as invalidation, were added.

Simulating Caching Configurations

A Web cache generally caches Web documents that originate from many different servers. Because the traces used for this experiment reproduce only *part* of the traffic managed by each cache, it was not possible to simulate cache replacement policies; their behavior depends on the entire traffic seen by each cache. Therefore, the caches are simulated without any replacement policy (i.e., caches have an infinite size). To roughly reproduce the behavior of the replacement policies, it was decided that a copy could not stay in a cache for more than seven days, independent of any consistency considerations. This delay is a typical value of any document's time-to live inside a Web cache [73]. When the time-to-live value expires, the corresponding copy is removed from the cache.

[4]Rather than measuring the true bandwidth of the connection, this actually measures the throughput of the connection. This does not, however, affect the validity of the results.

[5]Because paths in the Internet are generally not bi-directional, this leads to rough estimates for the latency measurements, rather than exact values. Once again, this does not affect the validity of the results.

Evaluation Criteria

Choosing a replication policy requires making tradeoffs. Replicating a Web document affects the client access time, the consistency of copies delivered to the clients, the master server load, the overall network traffic, etc. It is generally impossible to optimize all these criteria simultaneously. Therefore, evaluating the quality of service of the system should involve metrics that characterize different aspects of the system's performance. Three metrics, representing the access time, document consistency and global network traffic, were chosen:

Total delay: This is the sum of all delays between the start of a client's request and the completion of the response (in seconds).

Inconsistency: This is the total number of outdated copies delivered to the clients.

Server traffic: This is the total number of bytes exchanged between the master server and the intermediate servers or the clients. This metric measures all the inter-AS traffic, which is taken to be the wide-area traffic; we do not take into account the traffic between the intermediate servers and the clients, as it is considered "local."

An important remark is that all these metrics are additive: it is possible to simulate each document separately and add the resulting values for each document in order to get the quality of service of the complete system. This would not be possible if the metrics were average values, for example.

2.4.2 System Configurations

For each document, a number of setups likely to optimize the access to that document were considered. All configurations are based on the same system model; the only difference between them is the nature of the intermediate servers and the coherence policy they use.

Base Configuration

This configuration acts as a baseline configuration:

NoRepl: This configuration uses no caching or replication whatsoever. All clients contact the server directly, without any intermediate servers.

Caching Configurations

In the caching configurations each intermediate server acts as a proxy cache. These proxy caches may implement the following policies:

Check: When a cache hit occurs, the cache systematically checks the copy's consistency by sending an If-Modified-Since request to the master before answering the client's request.

Alex: When a copy is created, it is given a time-to-live proportional to the time elapsed since its last modification [23]. Before the expiration of the time-to-live, the cache can deliver copies to the clients without any consistency checks. At expiration of the delay, the copy is removed from the cache.

In the simulations, a ratio of 0.2 was used because that is the default in the Squid cache [24]. Thus:

$$\frac{T_{removed} - T_{cached}}{T_{cached} - T_{last\_modification}} = 0.2$$

AlexCheck: This policy is identical to Alex except that, when the time-to-live expires, the copy is kept in the cache with a flag describing it as "possibly stale." Any hit on a possibly stale copy causes the cache to check the copy's consistency by sending an If-Modified-Since request to the master before answering the client's request. This policy is implemented in the Squid cache [24].

CacheInv: When a copy is created, the cache registers it at the server. When the master is updated, the server sends an invalidation to the registered caches to request them to remove their stale copies. This policy is similar to the AFS caching policy [95].

Replica Configurations

In the replica configurations, intermediate servers act as replica servers. Replica servers contain permanent copies of documents. There are a relatively small number of such servers. This allows the application of strong consistency policies, which would be prohibitively expensive in the case of a large number of caches.

The traces collected involve clients from a few thousand different ASes. This led us to consider caching systems with as many caches as ASes. However, it would not be feasible to create that many replication servers. It was decided, therefore, to place replication servers in the autonomous systems where most of the requests came from. The rationale behind this choice being that most requests come from a small number of ASes, as discussed below.

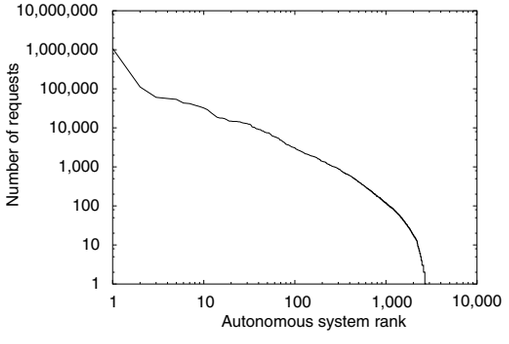


Figure 2.13: Number of requests per autonomous system.

Figure 2.13 shows the number of incoming requests per AS (the ASes are sorted by decreasing number of requests). In our case, the top 10 ASes issued 53% of the requests, the top 25 ASes issued 62% of the requests, and the top 50 ASes issued 71% of the requests.

Replication servers were placed only in the "top ASes." Clients located inside one of these ASes would be bound to their local replica. Other clients would send their requests directly to the server (see Figure 2.14).

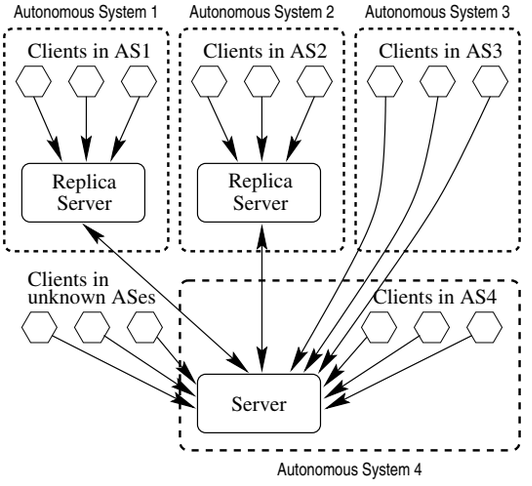


Figure 2.14: Replica configuration.

The three replica configurations were distinguished only by the number of replicas created. These are summarized as follows:

Repl10, Repl25, Repl50: Replicas are created in the top 10, 25, or 50 ASes. Consistency is maintained by pushing updates: when the master is updated, the server sends updated copies to the replica servers.

Hybrid Configurations

In the replica configurations, many clients access the server directly (because they are in lightly populated ASes, for example, clients from autonomous system 3 in Figure 2.14). The AS of such clients generates only a few requests to our server, so it is not worthwhile installing a replica there. It might, however, be worthwhile to install a cache, which is cheaper to maintain than a replica, in these ASes. The hybrid configurations were created to address these situations.

A hybrid configuration is similar to a replica configuration, except that it includes a cache in each autonomous system that does not contain a replica (see Figure 2.15). Two hybrid configurations were defined based on the consistency policy of the caches used.

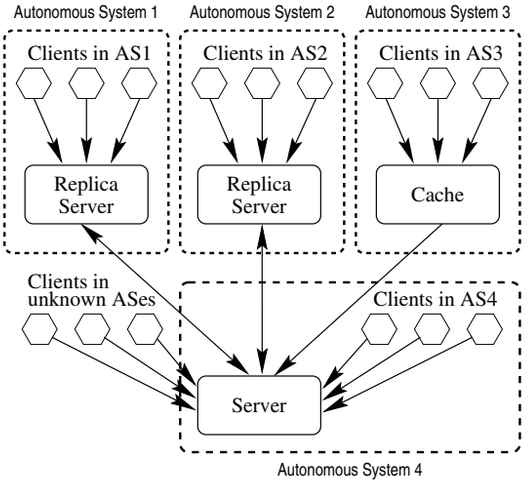


Figure 2.15: Hybrid configuration.

Repl50+Alex: Similar to **Repl50**, but the autonomous systems which do not contain a replica server use an **Alex** cache instead.

Repl50+AlexCheck: Similar to **Repl50**, but the autonomous systems which have no replica server use an **AlexCheck** cache instead.

2.4.3 Results

The results of the experiment are presented as follows. First a brief overview of the traces collected is given. Next the result obtained when the same policy is associated to all documents (one-size-fits-all) is presented. This is followed by a discussion of possible methods for associating each document with its most-suited replication policy. Finally the performance improvements when using per-document policies are presented.

Collected Traces

Number of documents	17,368
Number of requests	2,118,572
Number of updates	9,143
Number of unique clients	107,386
Number of different ASes	2,785

Table 2.1: Characteristics of the collected traces

Traces were collected for 5 weeks, from Sunday August 29, 1999 to Saturday October 3, 1999. Table 2.1 shows some statistics about the resulting traces. The server used handles medium-size traffic and documents are not updated very frequently (the average life-span is 67 days). Large servers, such as electronic-commerce servers, are expected to have more heterogeneous document sets than ours. Therefore, they should benefit even more from the ability to choose the replication policies per document.

One-size-fits-all Policies

Configuration	Delay (hours)	Incons. (no.)	Traffic (GB)
NoRepl	219.0	0	43.91
Check	229.2	0	23.60
Alex	96.4	5211	23.51
AlexCheck	96.6	4821	23.23
CacheInv	93.7	0	23.18
Repl10	177.4	0	43.60
Repl25	145.0	0	48.06
Repl50	121.9	0	55.55
Repl50+Alex	67.5	966	46.93
Repl50+AlexCheck	67.6	941	46.86

Table 2.2: Performance of the one-size-fits-all policies.

Table 2.2 shows the resulting performance when the same policy is applied to all documents (one-size-fits-all policies). As expected, the **NoRepl** policy has bad results compared to the others in terms of delay and traffic. On the other hand, it provides perfect consistency.

Most policies are good with respect to one or two metrics, but none optimize on all three metrics. For example, **Repl50+Alex** and **Repl50+AlexCheck** provide excellent (low) delays. On the other hand, they do not do as well with respect to inconsistency and amount of traffic generated. Other configurations have similar problems.

Assigning Optimal Policies to Documents

The key question is whether it is possible to find a configuration that provides good performance for all documents with respect to all metrics at the same time. In order to answer this question it is necessary to determine, for each document, a custom configuration that optimizes that document's performance. Determining an optimal configuration for a document requires finding an optimal replication policy for that document. The overall performance using these custom configurations is then compared to the overall performance when using the one-size-fits-all configurations.

For a given document, finding the best replication policy consists of deciding which policy provides the best compromise between the different metrics. Policies that are relatively good with respect to all metrics are preferred above those that are very good in one metric and very bad in the others.

Assigning a policy proceeds as follows: based on the simulation results, each policy is given a score. For a given document, the policy with the lowest score is declared "optimal." A policy's score is a weighted sum of the evaluation metrics:

$$score = \frac{delay}{\alpha} + \frac{incons}{\beta} + \frac{traffic}{\gamma}$$

Choosing values for α, β, and γ allows one to control the relative weight of each metric. The larger a weight is, the less the associated metric will influence the final result. Because different metrics are expressed in different units, the factors α, β and γ are expressed such that a score is always dimensionless.

This method is used to assign a policy to each document. Using it with the same parameter vector (α, β, γ) leads to an *arrangement*: a parameter-specific set of *(document, policy)*-pairs. Thus, for a given (α, β, γ), each arrangement has an associated value, which is expressed as a vector $\langle total(metric_1), \ldots, total(metric_n)\rangle$ where $total(metric_k)$ denotes for metric k the value accumulated over all documents in the arrangement. A detailed evaluation of this assignment strategy can be found in [80].

Comparing One-size-fits-all to Custom Policies

Comparing arrangements is somewhat difficult because the values of arrangements actually form a partially ordered set. It is preferable to compare each arrangement with an ideal *target point*. This point corresponds to the best achievable delay (obtained by selecting the policy providing the smallest delay for each document) and the best achievable traffic (obtained by selecting the policy providing the smallest amount of traffic for each document). Of course, for a given document, the best policy in terms of delay and the best policy in terms of traffic are not always the same. The target point is, therefore, generally impossible to reach. Nevertheless, this point acts as an upper bound for performance: it is impossible to obtain a better performance than the target. The target point can also be used to compare the arrangements: the closer it gets to that point, the better an arrangement is.

To simplify matters, β was given a very small value, making the optimization of consistency an important requirement. By subsequently modifying the relative weights for delay and traffic, it was possible to obtain a number of arrangements that implement various delay/traffic trade-offs.

Figure 2.16 shows the performance of arrangements in terms of total delay and server traffic. Each point corresponds to the performance of one particular arrangement. The arrangements where all documents are given the same policy are represented by single points. Custom arrangements provide a set of points, each individual point being obtained with one particular set of weights (α, β, γ), where $\beta \approx 0$.

Among the one-size-fits-all arrangements, some have good performance with respect to delay, but poor performance with respect to traffic; some others behave the other way

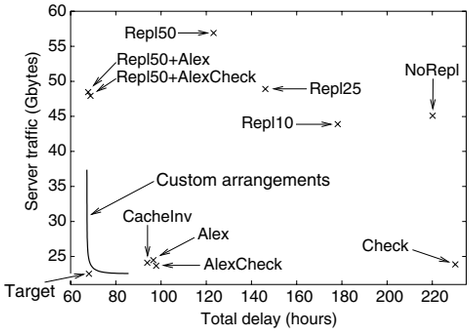


Figure 2.16: Performance of arrangements vs. one-size-fits-all configurations.

round. However, none of them comes very close to the target. On the other hand, custom arrangements do come close to the target. This means that selecting replication policies on a per-document basis provides a significant performance improvement over any one-size-fits-all configuration.

Chapter 3

The GlobeDoc Approach

GlobeDoc is an object-based model and architecture for the Web that provides a solution for the scalability related performance problems discussed in Chapter 1. Like most of the current approaches examined in Chapter 2, GlobeDoc relies on replicating Web resources, thus localizing traffic and reducing individual server loads, to improve performance. Unlike these approaches, however, GlobeDoc does not impose a single global replication policy on all Web resources. As has been shown, the ability to apply resource or document-specific replication policies is necessary to achieve the scalability required to overcome scalability-related performance problems.

This chapter will present the GlobeDoc model and system architecture. Because GlobeDoc is designed as a Globe application, an overview of the Globe distributed system will first be given in Section 3.1. Section 3.2 will present the GlobeDoc object model, Section 3.3 will briefly discuss replication policies, and Section 3.4 will describe the GlobeDoc system architecture.

3.1 Globe Overview

Globe is a middleware infrastructure for building wide-area distributed applications. It provides a distributed shared object model and infrastructural support services for using these objects. Globe distributed shared objects are objects that can be physically distributed over a wide-area network and that completely encapsulate their own distribution policies. This means that a Globe object can determine its own policy for replication, partitioning, and migrating its state, among other things. In Globe, wide-area distributed applications are modeled using application-specific Globe objects.

This section will provide a short introduction to and overview of Globe. More detailed information about the design and implementation of Globe can be found in [108, 47].

3.1.1 Globe Object Model

In the Globe model, processes interact and communicate through **distributed shared objects** (DSOs). A distributed shared object offers (exports) one or more **interfaces**, each consisting of a set of methods. Objects are passive, but multiple processes may simultaneously access the same object. Changes to an object's state made by one process are visible to other processes. A Globe object is physically distributed, meaning that its state may be partitioned and replicated across multiple machines at the same time. However, processes are not aware of this: state and operations on that state are completely encapsulated by the object. All implementation aspects, including communication protocols, replication policies, and distribution and migration of state, are part of the object and are hidden behind its interface.

Distributed Shared Objects

A distributed shared object is built from a collection of **local representatives** (LRs), each of which resides in a single address space and communicates with local representatives in other address spaces (typically on other machines). A local representative is implemented as a **local object** (which, in contrast to a distributed object, always resides within a single address space). Each local representative forms a particular implementation of a distributed object's interface. For example, an LR in one address space might implement an interface by forwarding all method invocations to an LR in another address space, similar to an RPC client stub. Another local representative might, however, implement that same interface as operations on a local copy of the object's state. Together all the local representatives form the representation of one instance of a Globe distributed shared object. The local representatives that form an instance of distributed object are referred to as each other's **LR peers**.

For a process to invoke an object's method, it must first bind to that object by contacting one of the object's **contact points**. A **contact address** describes such a point, specifying a network address and a protocol through which binding can take place. Binding results in a local representative being placed in the client's address space. Figure 3.1 illustrates this model and shows a Globe object distributed across four address spaces (A1 through A4). Note that there is no local representative in address space A5. Consequently, A5 is not part of the object.

Local Object

One of Globe's aims is to let application developers concentrate on designing and implementing their application's functionality in terms of objects. Distribution, being a different concern, should be treated separately. For this reason local objects are constructed in a modular way allowing issues such as replication and communication to be separate from what the object actually does (i.e., its semantics). A local object, therefore, consists of four **subobjects** as shown in Figure 3.2:

The subobjects are:

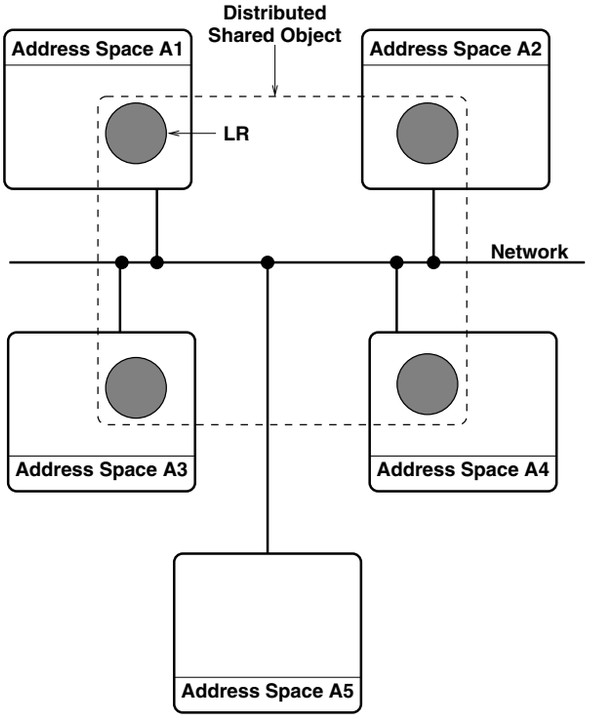


Figure 3.1: Example of a Globe object distributed across four address spaces.

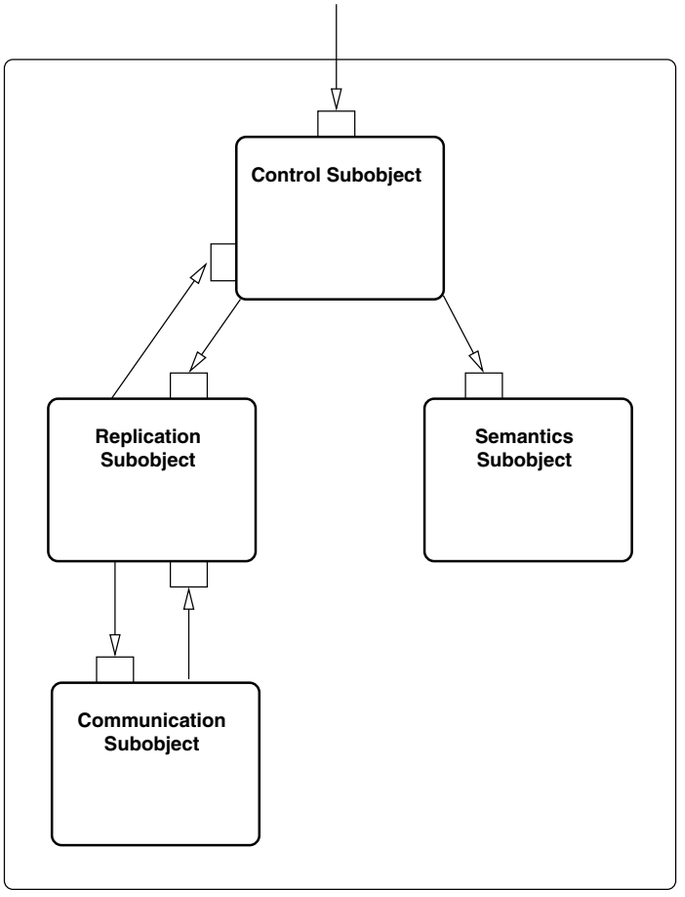


Figure 3.2: General structure of a local object.

- A **semantics subobject** containing the code that implements the distributed shared object's functionality
- A **communication subobject** for sending and receiving messages from other local objects
- A **replication subobject** containing the implementation of a specific replication policy
- A **control subobject** handling the control flow within the local object.

These four subobjects are designed to build scalable distributed shared objects.

Semantics subobject The semantics subobject implements the interfaces that define a particular distributed shared object, thus implementing the semantics of that object. Among a semantic object's primary responsibilities is the storage and manipulation the distributed object's state. A developer writes the semantics subobject in C, C++, Java, or some other language. In contrast to the replication and communication subobjects, the semantics subobject developer is also responsible for defining the interfaces implemented by the subobject. In principle, the semantics subobject is the only subobject a developer needs to construct personally. All other subobjects can either be obtained from libraries, or be generated from interface specifications.

Communication subobject. This is usually a system-provided local object. It is responsible for handling communication between parts of the distributed object that reside in different address spaces (i.e., the LR peers). Depending on what is needed from the other components, a communication subobject may offer primitives for point–to–point communication, multicast facilities, or both. Several communication subobjects can coexist in the same local object. The communication subobject provides a standard interface that allows for a variety of different implementations. For example, it is possible to have implementations for network protocols such as UDP, IP multicast, and TCP, that all provide the same interface.

Replication subobject. The global state of a distributed object is made up of the state of its various semantics subobjects (i.e., the semantics subobjects belonging to the DSO's local representatives). An object's state may be replicated for reasons of fault tolerance or performance. Replicated state is implemented by storing copies of the object state in the semantics objects of the DSO's LRs. For this reason LRs are often referred to as a Globe object's **replicas**. The replication subobject is responsible for keeping these replicas consistent according to some replication policy. The replication subobject operates only on marshaled data. Different distributed objects may contain different replication subobjects, and thus use different replication algorithms. A replication subobject also controls the actions taken by the control subobject.

Control subobject. The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. It is responsible for marshaling parameters and state and passing these to the replication subobject when requested. Incoming invocation requests are also handled by the control subobject.

3.1.2 Globe System Model

Client-to-Object Binding

As mentioned earlier, in the Globe model, processes communicate by sharing a distributed object. To communicate through a distributed object, it is necessary for a process to first **bind** to that object. The result of binding is that a local representative is placed in the address space of the requesting process allowing the process to directly invoke the object 's methods. Binding itself consists of two distinct phases: (i) finding the object, and (ii) installing the interface. This is illustrated in Figure 3.3 . Finding an object is separated into a name-lookup (step 1) and a location-lookup step (step 2); installing the interface requires that a suitable contact address be selected, as well as an implementation for that interface (steps 3 to 5).

Name Lookup

To find an object, a process must resolve the object's name to an object handle. This is done by passing the object name to a **naming service** (step 1 in Figure 3.3). The naming service returns an **object handle**, which is a location-independent and universally unique object identifier, such as a 128-bit number, which is used to locate objects. It can be passed unmodified between processes as an object reference. Whereas an object handle uniquely identifies an object (i.e., an object always has exactly one object handle), an object may have multiple names that resolve to its object handle. An **object name** is simply a human-readable string that represents an object. It is up to the naming service to interpret this name and resolve it to an object handle.

Resolving an object's name leads to an object handle (step 2 in Figure 3.3). An object handle, however, does not identify the location of an object and its replicas. To find the actual object, the object handle must first be passed to a location service, which returns one or several contact addresses (step 3 in Figure 3.3).

Location Lookup

The goal of the **location service** is to find an object's contact points given its object handle. A **contact point** is represented by a **contact address**. The location service is, therefore, responsible for storing contact addresses and resolving object handles to these contact addresses. Besides looking up contact addresses for objects, the location service also allows addition, deletion and modification of contact-address mappings.

Because any use of a Globe object requires binding to it (and therefore use of the location service), the location service presents a potential scalability and performance

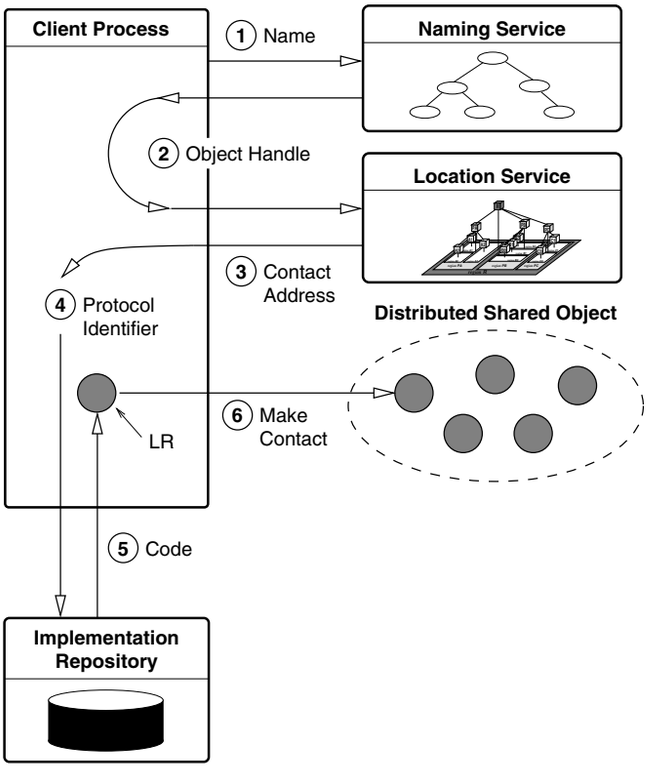


Figure 3.3: Binding a process to a distributed shared object.

bottleneck in the Globe architecture. To overcome this problem, the location service is implemented as a distributed search tree, as shown in Figure 3.4. In this tree, the world is divided into a hierarchical set of domains. At the lowest level there is one domain per site; a collection of sites form a region, etc. An object is recorded at each site where it has a contact address, and recursively in each enclosing region, up to the root of the tree.

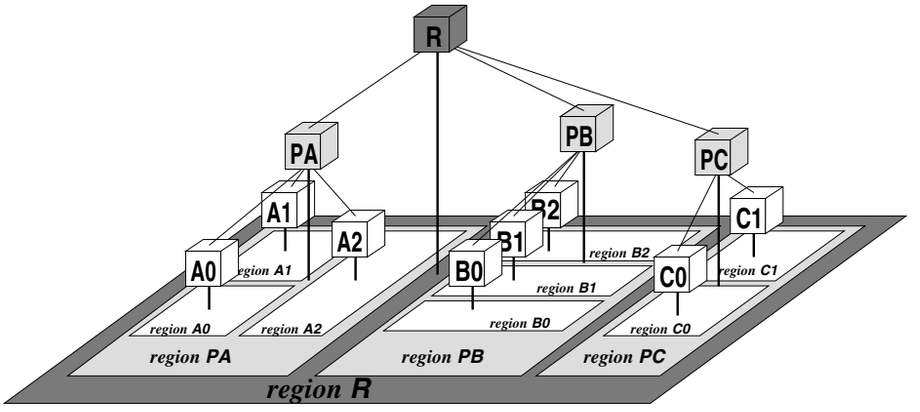


Figure 3.4: Globe's worldwide search tree used for locating objects.

Initially, a record at the site level contains the actual contact addresses and records at higher levels contain pointers to the next lower level. Recording an object at multiple levels allows searches with expanding rings: a search starts at the local site, followed by the local region, then the next higher level region, etc., eventually followed by the root. Searching with expanding rings provides the desired locality (that is, remote or wide-are communication is avoided). If the object has a contact address in the site or local region of the requesting process, then contact addresses will be found using only local communication.

More information about the design, implementation and performance of the location service can be found in [107, 12].

Contact Addresses and Implementation Selection

Once a process knows where it can contact the distributed object, it needs to select a suitable address from the ones returned by the location service. A contact address may be selected for its locality, but there may also be other criteria for preferring one address over another. For example, some addresses may belong to subnets that are difficult to reach, or to which only low-bandwidth connections can be established. Other quality-of-service aspects may need to be considered as well. Note that an address selection service is a *local* service that builds its own administration concerning the quality of contact addresses.

Having selected a contact address a binding process must subsequently find and load a suitable local representative implementation. Besides describing where an object can be reached a contact address also describes how the object can be reached. This is expressed

as a **protocol identifier**. It specifies a complete stack of protocols that must be implemented at the client's side in order to communicate with the distributed object. Such a protocol identifier is mainly used to load code from a (trusted) **implementation repository** and to subsequently instantiate the local representative (steps 4 and 5 in Figure 3.3).

3.1.3 Globe Programming Model

Every Globe distributed shared object exports interfaces that define its functionality. The interfaces exported by a DSO are determined by the interfaces exported by its constituent local representatives. Every local representative exports certain control and management interfaces (used to create, destroy and manage that LR) as well as the interfaces exported by its semantics subobject. Generally all of a DSO's local representatives will export the same interfaces. Note that, while the LRs generally export the same interfaces, the actual implementations of these interfaces may vary between them.

Object interfaces are defined in a platform and programming language independent **Interface Definition Language** (IDL). An LR's structure can be defined using an **Object Definition Language** (ODL).[1] An IDL compiler compiles object interface definitions to a target language creating skeleton implementations of that interface's methods. An ODL description defines the interfaces that an LR will implement and the (communication and replication) subobjects that it will contain. An ODL compiler generates the control subobject, which is responsible for creating and initializing the LR. A programmer is responsible for implementing the methods in the interfaces exported by an LR's semantics object. The combination of the generated and manual code and the communication and replication subobject constitutes all of an LR's code.

An object's code is normally stored separately in one or more **class objects**. A class object is a (local) object that contains the method implementations for objects belonging to the same class. Every local object has an associated class object. By separating code from state it is possible to instantiate several instances of the same class without having to duplicate the code for each instance. A class object also provides a convenient container for packaging and transporting an object's code.

3.2 GlobeDoc Model

In the GlobeDoc model of the Web, a Web site is composed of related **Web documents**, and a Web document itself is a collection of logically related **Web resources**. These Web resources are referred to as a document's **elements** and can be anything that is accessible over the Web (e.g., HTML files, text files, images, audio files, video files, applets, etc.). The relation between the resources contained in a Web document is generally stronger than that between the documents contained in a Web site. For example, an organization's

[1] Currently this is true in theory but not in practice. We have not yet defined an ODL, much less built tools to read an ODL description and generate the appropriate skeleton objects. This means that, currently, control objects are not automatically generated but must be made by hand.

Web site contains a collection of Web documents that are somehow related to that organization, while a Web document representing a news story would contain only the elements directly relating to that story (e.g., the HTML page plus any icons and images relating to the story or the page layout). Note that, in this dissertation, Web documents are considered to be static, that is, none of their elements are generated dynamically upon access.

In GlobeDoc, a Web document is encapsulated in a Globe distributed shared object (called a **GlobeDoc object**) whose state contains a document's elements. A GlobeDoc object also offers methods that allow clients to access and modify this state on a per-element basis. The hyperlinked structure that is normally provided by Web pages is maintained in GlobeDoc. An **internal hyperlink** that is part of some GlobeDoc object refers to an element in that same document. Likewise, an **external hyperlink** refers to an element of another GlobeDoc object.

Every GlobeDoc object assigns one element to be the **root element**. This element provides access to other elements through internal links and is comparable to the index.html file. Because nothing is said about the contents of an element every element has a set of **properties** associated with it. At the least, these properties include a MIME type that describes an element's contents.

By virtue of it being a Globe DSO, a GlobeDoc object can distribute (replicate or partition) its state over multiple physically separated address spaces (or machines). As such, requests for an object's state will be distributed over these various machines, thereby spreading the total load generated by the requests and preventing any single machine hosting the object from becoming overloaded. Similarly, by strategically replicating a GlobeDoc object's state so that it is close to large concentrations of clients, the traffic at each replica will have a local character, increasing responsiveness for clients and decreasing overall network traffic and saturation.

Also, because Globe makes the state distribution transparent to clients and because the distribution policies can be determined independently per Globe object, GlobeDoc makes it possible to apply distribution policies on a per-document basis. In this way GlobeDoc allows replication of Web documents without imposing any single global replication policy on all documents.

3.2.1 GlobeDoc Interfaces

A GlobeDoc object's functionality is defined by the interfaces that it exports. These interfaces define the methods used to access and modify the object's state. There are three main GlobeDoc interfaces. The **document interface** contains methods that act on the document as a whole, allowing elements to be added and removed. The **content interface** contains methods that act on individual elements and allow the contents of elements to be accessed and modified. Finally, the **property interface** allows element properties to be set and retrieved. Tables 3.1, 3.2 and 3.3 show an overview of these interfaces.

An element is added to a GlobeDoc object using the addElement method. This method creates a new element with the given name and type and containing the given raw binary data (passed as the contents parameter). The name given to the element must be used to access, modify or delete that element in the future (Section 3.2.4 discusses the naming of

interface	document	
method	addElement	Creates a new GlobeDoc element
in	name	The element's name
in	elementType	The element's type
in	contents	The element itself
method	startElementAddition	Starts an element creation. A new element is created, which is visible but cannot be downloaded yet. The element is created with the specified size, and with undefined contents.
in	name	The element's name
in	elementType	The element's type
in	dataSize	The size of the element
method	elementAdditionData	Continues an element creation. A block of data is cached or written directly to the element starting at the specified offset.
in	name	The name of the element
in	offset	The offset at which to add the data
in	data	The data to add
method	endElementAddition	Completes an element creation. The element is now available for downloading.
in	name	The name of the element
method	deleteElement	Removes an element
in	name	The name of the element to remove
method	getElementSize	Returns the size of an element
in	name	The name of the element
returns		The size of the element
method	setRoot	Sets the root element
in	name	The name of the element to set as the root element
method	getRoot	Returns the name of the root element
returns		The name of the root element
method	allElements	Returns the list of elements currently in the GlobeDoc object
returns		A list of element names

Table 3.1: The GlobeDoc document interface.

interface	content	
method	getIncarnationID	Returns the current incarnation id of the specified element
in	name	The name of the element
returns		The incarnation id
method	getContent	Performs a complete element download
in	name	The name of the element
returns		The element's contents
method	getContentBlock	Reads a block of data from the element at the specified offset
in	name	The name of the element
in	incarnationID	The element's incarnation id
in	offset	The offset to start reading from
in	maxBlockSize	The maximum amount of data to read
returns		The data
method	putContent	Updates an element in one call
in	name	The name of the element
in	data	The data to replace the element's contents with
method	startPutContent	Starts the update process of an element. Separate storage for the update data is claimed.
in	name	The name of the element to update
in	dataSize	The new size of the element
method	putContentData	Continues the update process of an element. A block of update data is cached or written directly to the element, starting at the specified offset.
in	name	The name of the element
in	offset	The offset
in	data	The data
method	endPutContent	Completes the update process of an element
in	name	The name of the element
method	putAllcontent	Updates all elements in the GlobeDoc object in one call
in	names[]	The names of the elements
in	data[]	The data to update the elements with

Table 3.2: The GlobeDoc content interface.

interface	property	
method	getProperties	Returns all properties of an element
in	name	The name of the element
returns		A list of properties
method	setProperties	Add properties to an element
in	name	The name of the element
in	properties	The properties to add

Table 3.3: The GlobeDoc property interface

elements). Although an element's MIME type is actually a property and could be set using the property interface, it has been chosen to set it in addElement because it is a required property. This way it is guaranteed that an element will always have its type set. The methods in the property interface can, however, be used to manipulate this property once the element has been created.

Because an element can be large (e.g., an audio file, image, or video file) the document interface also includes methods that allow an element's contents to be added in parts. This prevents implementations from having to store the whole element (which may be many times larger than the available memory) in main memory while passing it as a parameter to the addElement method. The startElementAddition method starts the element creation process and causes a new element to be created. The element is created with the specified size and with undefined contents. The elementAdditionData method is used to add contents to the newly created element. It specifies the offset at which to add the given block of data and is called repeatedly until the complete contents have been added. Finally, the endElementAddition method is called to signify that the element is complete. This method can cause local caches to be flushed, modified state to be propagated to replicas, etc. After calling this method an element's contents can be accessed using methods from the content interface. Note that addElement is included as a convenience method and is meant only for adding small elements. Larger elements should be added using the three methods described above.

An element's size (the size of its contents in bytes) can be requested by calling the getElementSize method for a given element. The setRoot method allows an element to be designated as the root element, and getRoot returns the name of the current root element (none if one has not yet been set). A list of all elements contained in a GlobeDoc object is returned by the allElements method. Finally, the deleteElement method is used to remove an element from the GlobeDoc object.

Modifying an element's contents is a three-step process. In the first step, a copy of the element's contents must be extracted with, for example, the getContent method. Next, the element can be modified using an appropriate tool, such as an HTML or image editor. When all modifications have been made, the element is returned to the GlobeDoc object using one of the the putContent methods.

In the content interface, the getContent method retrieves and returns the contents of a named element in one piece. Because an element's contents can be large, the con-

tent interface also contains methods for retrieving an element's contents in parts. The getContentBlock method allows a specified part of an element's contents to be retrieved. In order to retrieve the complete contents, getContentBlock must be called repeatedly until all parts have been retrieved. Because content retrieval using this method spans multiple method calls it is no longer an atomic action. This means that it is conceivable that another client could change the element's contents between successive calls to getContentBlock, resulting in inconsistent data being retrieved. To prevent this problem, an element is given an incarnation id, a unique identifier that is updated every time an element's contents are changed. If an element is modified or deleted, its current incarnation id is invalidated. A client that wants to retrieve an element's contents must obtain that element's latest incarnation id and specify this id with each getContentBlock invocation. If the incarnation id is invalidated during the retrieval (i.e., the element is deleted or its contents changed), the next getContentBlock invocation, and therefore the whole download, will fail. The client should then attempt a new download with a new incarnation id.

The process of replacing an element's contents is similar to that of adding a new element. There is a putContent method that replaces an existing element's contents in one piece. This method is similar to the addElement method in that it is only suited for small elements. There are also three methods that allow content to be added in parts. Invoking the startPutContent method announces the start of the process, which causes the current incarnation id for the element to be invalidated and a new one to be created. It may also cause required resources, such as storage for the new contents, to be claimed. Data is added by repeatedly calling the putContentData method and passing it successive data blocks. When all the data has been added, endPutContent is called completing the process and possibly causing caches to be flushed, data to be replicated, etc. The putAllContent method is a convenience method that invokes putContent for every given element.

Methods for manipulating element properties are defined in the property interface. The getProperties method returns all of a given element's properties, while the setProperties method adds (or replaces) the given properties to the element's set of properties. A property may be removed from an element by replacing its value with a null value.

3.2.2 Transactions and Locking

As previously mentioned, modifying a GlobeDoc element is a three-step process. It involves retrieving the element's contents, modifying those contents, and returning them to the GlobeDoc object. Unfortunately, due to the nature of this process, it is possible for concurrent modifications of an element to interfere with each other. For example, suppose a client A decides to modify element X. Client A would start by retrieving a copy of X and editing it. Now suppose that client B also decides to modify X, it would also retrieve a copy of X and start to edit it. In the meantime, A finishes its modifications and returns the new copy of the element to the GlobeDoc object. However, when B finishes its modifications and returns its new copy of X to the object, all of A's changes are overwritten and therefore lost. The problem is that modification of an element is not an atomic transaction. A fourth GlobeDoc interface, the **lock interface**, attempts to provide modifications with this atomicity property. The interface is presented in Table 3.4.

interface	lock	
method	checkOutElements	Checks out a sequence of elements, for updating purposes
in	names[]	The names of the elements to check out
returns		A list of checkout ids
method	checkInElements	Checks one or more elements associated with the checkout id parameter back in
in	cId	The checkout id
method	getCheckedElements	Returns the checked out elements associated with the checkout id
in	cId	The checkout id
returns		A list of element names

Table 3.4: The GlobeDoc lock interface

The approach is based on checking out elements to lock them for writing. Element modification now becomes a five-step process. In the first step, all elements that are to be modified must be checked out (by invoking the checkOutElements method). Checking out elements locks them for writing and results in a checkout identifier for those elements. The next three steps are the same as described previously (the element contents are retrieved, they are modified, and then returned to the GlobeDoc object), except that all putContent methods now require a valid checkout id for the elements being modified. Finally, once the modifications are complete, the locks on the elements must be released by checking the elements back in (invoking the checkInElements method). An element may only be checked out once at any given time (that is, invoking checkOutElement more than once for the same element without first checking that element back in will result in an error).

This mechanism prevents concurrent modifications of an element from overlapping because it prevents elements from being checked out by more than one client at once. In the example, client B would therefore have to wait until client A was finished with element X before it could successfully check the element out for its own use. This mechanism may, however, fail if the order of operations is modified (for example if a client only checks an element out before returning it to the GlobeDoc object, but not before retrieving its contents).

A serious drawback of this approach is that there is currently no way for a checkout to time out. Thus if a client crashes shortly after checking an element out, but before checking it back in, that element will remain checked out indefinitely, and will become unmodifiable.

Another problem arises when this approach is applied in a distributed environment. Because replication of a GlobeDoc object's synchronization data (e.g., which elements are checked out and their checkout ids) follows the same strategy as replication of its state data, the checkout mechanism may work differently depending on the actual replication and coherence policies used. For example, if state at different replicas is not kept strictly

consistent it becomes possible for an element to be checked out by more than one client at a time.[2]

3.2.3 Persistent and Transient GlobeDoc Objects

A GlobeDoc object's state consists of all its elements, their associated properties and any other relevant data (such as locking data, incarnation ids, etc.). As with all Globe objects, a GlobeDoc object's individual LRs are responsible for storing this state and within an LR it is the semantics subobject that is responsible for storing that state. An object's state can be persistent or transient. An LR that stores persistent state is required to retain its state across (restarts of) the process hosting it. In practice, this means that an LR whose hosting process is brought down and restarted will have the same state before and after the restart. An LR that stores transient state, on the other hand, loses that state when the process hosting it is shut down or ceases to host that LR for some other reason. An LR containing persistent state is called a **persistent LR**, while one containing transient state is called a **transient LR**.

As an example, consider a persistent LR where state is added shortly before the host process is brought down. When the host process is brought back up and the LR recreated, the LR will contain the same state as it did before the process was brought down. Depending on its replication policy, this LR could continue processing requests from clients without having to reinitialize its state. A transient LR, however, would lose all its state when it was recreated, and would first have to transfer a copy of the state from another LR before being able to process client requests.

Persistent LRs generally require the availability of persistent contact addresses to operate. A **persistent contact address** is one that always refers to a particular LR's contact point and is independent of the LR's hosting process. It allows a persistent LR to register itself once with the location service, and not have to worry about re-registering itself again (for example, if it is ever restarted). A **persistent connection** is similar to a persistent contact address in that it is a connection to an LR that remains valid independent of the process hosting the LR. Thus, a client connected to an LR over a persistent connection does not have to create a new connection if that LR's host goes down and it has to be restarted in a new process.

Whether a whole GlobeDoc object is persistent or transient depends on whether its LRs are persistent or transient. A transient GlobeDoc object always has transient LRs. It is, however, possible to have a mixture of transient and persistent LRs within an object. In this case, as long as the object contains at least one persistent LR, the whole object will remain persistent.

Note that a persistent GlobeDoc object is not necessarily a fault-tolerant object. This means that a persistent GlobeDoc object's state is not guaranteed to be maintained if the object goes off-line due to a fault rather than a controlled shutdown. Fault tolerance is a

[2] This may, however, be a problem with the underlying Globe system as it does not make it clear whether an object's underlying distribution policy is allowed to impact the semantics of the object's interfaces. Either way this shows that the distribution issues are not completely hidden from the designer and implementor of Globe semantics subobjects.

property that must be added independently of persistence. It may, for example, be possible to have fault-tolerant transient objects, as long as the fault tolerance guarantees that an LR containing the object state will always be available.

An implementation issue somewhat related to persistence is whether the GlobeDoc semantics subobject should store its state in main memory or secondary storage. The former allows for quicker access to the GlobeDoc object's elements, however it also imposes a limit on an object's size (namely the amount of main memory available). On the other hand, storing the state in secondary storage allows larger objects, which complicates the GlobeDoc interface by making it feasible to store very large elements. The methods for reading and writing state must therefore take into account that the state they are reading or writing may be larger than the available memory, making it necessary to present that state in blocks. This leads to the introduction of the block-wise read and write methods described earlier.

3.2.4 Naming and Binding

To gain access to a GlobeDoc object a client must bind to that object given the object name. Each GlobeDoc object is identified by one or more location-independent human-friendly names. As in Globe, a GlobeDoc object may have multiple names, however, any single name always refers to a single object at any one time. These object names form part of a global GlobeDoc name space.

The organization of the GlobeDoc name space is similar to that used in, for example, UNIX file systems. The name space is organized as a hierarchical rooted tree in which an interior node represents a directory, and a leaf node represents a GlobeDoc object. Every edge is labeled with the (simple) name of the node it points to and an object name is composed of a sequence of the labels representing a path in the name space. As in UNIX, the labels are separated by a slash ("/"). An absolute object name, that is, one that represents a path starting at the root of the name space, always begins with a slash. GlobeDoc object names are always absolute. When used in the Web, GlobeDoc object names follow the Uniform Resource Identifier (URI) syntax and are preceded by a globe: scheme identifier. For example, the GlobeDoc object name /nl/vu/cs/object/foo becomes globe://nl/vu/cs/object/foo in a Web environment. A GlobeDoc object name represented as a URI in this way is a Uniform Resource Name (URN) and is called a **GlobeDoc URN**. Resolving object names is done in the usual (iterative or recursive) way and results in the object handle of the object to which the name refers.

Like GlobeDoc objects, elements within the objects are also named. Each GlobeDoc object contains a separate name space for its elements, which means that element names are valid only in the context of their encompassing GlobeDoc object. Thus, to refer to a GlobeDoc element both the object and element names are required. Unlike the GlobeDoc name space, however, GlobeDoc does not impose any rules or structure on the individual element name spaces. Element names can, for example, reflect the GlobeDoc name space and be arranged in a rooted hierarchy, or they could be hexadecimal strings representing the hashes of the elements themselves. Generally it is up to the clients responsible for adding elements to choose suitable element names. A suggested approach is to mimic the

naming of Web resources in the current Web and name elements as though they were files in a file system (e.g., /main.html, images/image1.gif, etc.)

Although it does not impose a structure on the element name space, a GlobeDoc object does allow one element to be assigned as the root element. Assigning an element as the root element gives it a function similar to the index.html file in many Web servers. It marks that element as an entry point into the GlobeDoc object.

For convenience, it is possible for a GlobeDoc URN to contain both a GlobeDoc object and an element name. In this case the element name is separated from the object name by a colon (":"). The URN globe://nl/vu/cs/object/gdObject:/element.html, for example, refers to an element named /element.html in a GlobeDoc object named /nl/vu/cs/object/gdObject. A GlobeDoc URN with an empty element name implicitly refers to the root element. Note that a client can only ever bind to a particular GlobeDoc object and not to a specific element. To access the element referred to by a GlobeDoc URN a client must first bind to the specified object and then retrieve the element by invoking appropriate methods on the GlobeDoc object.

For integration in the current Web, GlobeDoc URNs can be embedded in HTTP URLs, for example as http://globe.cs.vu.nl/nl/vu/cs/object/gdObject:/element.html. The server name part of the HTTP URL (globe.cs.vu.nl) refers to a GlobeDoc-aware HTTP server that, given the object and element names, can bind to the object and retrieve the named element. The GlobeDoc access point (which is described later) is an example of such a server. These URLs with embedded GlobeDoc object and element information are called **embedded URNs**.

GlobeDoc objects and GlobeDoc elements are often referenced by hyperlinks. As mentioned earlier, there are two types of hyperlinks in GlobeDoc: internal and external. Internal hyperlinks are represented by **relative URNs** (i.e., ones that contain only an element name) and refer to elements inside the same GlobeDoc object. External hyperlinks are represented by **absolute URNs** (i.e., ones that contain both an object name and an element name) and can refer to either internal or external elements.

3.2.5 Alternative Models

Before deciding on the GlobeDoc model described above, other approaches to mapping the Web onto Globe were also considered. The purpose of this section is to present an overview of these other approaches and discuss why they were rejected. The discussion will help shed light on the design decisions that led to the current model.

Fine grained object model

The most straightforward approach to creating an object-based model of the Web is to model every Web resource as a separate object. Such a model introduces separate object classes representing resources such as HTML documents, plain text documents, images, video streams, etc. Each such class implements and exports a resource specific interface. The major benefit of this approach is that each resource can determine its own distribution policy and be replicated (or partitioned) as needed. A secondary benefit is that resources

may offer interfaces and methods particularly suited to their needs. For example, besides simply providing a method for retrieving an HTML object's contents, an HTML object could also provide methods that parsed the HTML and returned a list of links and images contained in the document.

A major drawback of this approach is that it leads to objects that are too fine grained for Globe. Because it requires clients to bind to every Web resource they use, this approach results in a large number of bindings for every Web page accessed (considering that a client has to bind to the HTML object, then to each embedded image and possibly to even more HTML objects embedded in frames). Being a heavyweight operation (requiring contacting a number of external services as well as retrieving and loading code), binding to objects too often would lead to a noticeable performance degradation.

Besides the binding problem, this model also fails to take advantage of the strong relation between HTML pages and their embedded elements. When an HTML page is downloaded there is a strong possibility that related (embedded) images will also be downloaded. By explicitly including this relation in the Web model (as we do in GlobeDoc) this relation information can be taken advantage of to download all related elements in parallel as soon as one of them is requested. Furthermore, including this relation information allows related elements to be replicated together at the same locations, which greatly improves the efficiency and performance of the resulting model.

A final problem with this model is that it is difficult to integrate into the existing Web. In order to take advantage of the possibilities offered by this approach, clients must be aware of and make use of the interfaces offered by all the various classes of resource objects. It is possible to mask this abundance of interfaces using proxies or gateways to translate between HTTP and the various object interfaces, however, this looses much of the flexibility gained by offering each resource its own interface in the first place.

HTTP model

A different approach is to concentrate on Web protocols rather than on Web resources. This approach requires objects to implement interfaces that resemble existing protocols. Thus, for example, an object exporting an HTTP interface allows a client to invoke `GET`, `PUT` and `POST` methods on it. An object exporting such an interface could represent a single Web resource, a Web document or a collection of, possibly unrelated, Web documents.

Unfortunately, simply defining interfaces does not make for a complete model; an underlying object model is still required. The drawbacks of having an object represent a single resource have been discussed above. Having an object represent multiple unrelated Web documents, on the other hand, tends towards one-size fits all distribution policies, and as was shown earlier, this is not beneficial to the goal of improving Web performance. Finally, choosing a model where objects represent Web documents leaves us with a model very similar to the GlobeDoc model, the main difference being the syntax of the methods offered by the interfaces.

The main difference between the GlobeDoc and HTTP interfaces is that GlobeDoc lacks a counterpart to the `POST` operation. In HTTP, the `POST` operation allows a client to send extra data along with a request for a resource. Usually, this data comes as a result

of some user interaction (such as filling in forms) and is used by the receiving end to dynamically generate a reply. The fact that GlobeDoc does not have an equivalent to this operation means that GlobeDoc cannot offer full support for **dynamic contents** (i.e., content that is partly or wholly generated upon being requested).

The decision not to directly support dynamic content in GlobeDoc was deliberate, and not due to an oversight. The following two alternative models represent attempts at including dynamic content in the GlobeDoc model. The reasons for rejecting these models will highlight the reasons for choosing a purely static model for GlobeDoc content. Note that this does not mean that GlobeDoc can not be used for Web sites and documents that require dynamic content. Chapter 7 will discuss a possible GlobeDoc-based approach for handling dynamic content.

Composite Distributed Shared Objects

A first attempt at designing GlobeDoc objects that supported dynamic content is to consider composite DSOs. **Composite DSOs** are DSOs that contain (references to) other DSOs as part of their state. In this model a GlobeDoc object's elements are modeled as separate DSOs, while a GlobeDoc object acts as a container for these DSOs. The goal of the GlobeDoc object is to provide access to its elements' interfaces. These interfaces are regular Globe interfaces and allow a client to invoke methods on an element object.

Because it also models every Web resource as a separate DSO, the composite DSO model resembles the fine-grained object model described above. The difference between the two, however, is that in the composite DSO model, it is the GlobeDoc object and not the client that binds to all the element objects. This avoids the problem, discussed earlier, of a client having to bind to multiple objects to access a single Web page. In this case, the GlobeDoc object is responsible for performing the binding and may bind to all its elements during initialization to avoid the performance degradation associated with binding to the objects on demand. Although the GlobeDoc object can influence the (time and place of) creation and destruction of element replicas, it cannot influence the distribution policies of its individual element objects. This means that the element objects, although they are managed by a GlobeDoc object, remain fully independent Globe objects.

Modeling GlobeDoc using composite DSOs allows for dynamic content, because the individual elements can be either static or dynamic objects. That is, they can either contain static state that is returned by appropriate methods, or they can contain state that is used to generate responses dynamically for every request. For example, Perl CGI scripts could be stored in a Perl CGI object which executes the script every time it is accessed.

The greatest obstacle to this approach, however, is caused by replicated invocations [11]. Replicated invocation occurs when a replicated object invokes the methods of another replicated object. The major problem with replicated invocation is that each replica of the source object may invoke the method on different replicas of the target object. The target must be able to recognize and deal with these replicated invocations (i.e., it cannot treat them as separate and independent method invocations). Replicated invocations can cause a number of problems if not handled properly. First of all, depending on an object's replication policy, the method may end up being executed more often than

intended resulting in potential state corruption (for example, an increment method that is called too often will result in inconsistent state). Secondly, it is possible that different replicas return different responses (for example, due to side effects of the method) whereas in a replicated invocation all invocations should return the same response. Finally, depending on the replication policy, a replicated invocation may cause an explosion in the amount of messages sent between the two objects and an object's LRs. No general wide-area scalable solution to the problem of replicated invocation has yet been found.

Lightweight Distributed Shared Objects

An attempt to overcome the replicated invocation problem of composite DSOs led to the design of lightweight DSOs. A lightweight DSO is similar to a composite DSO, except that the element objects are not actually independent Globe objects. Instead a single semantics object implements all of the required element types (e.g., the most commonly used Web resource types) and exports their interfaces. Clients who wish to access these elements acquire the appropriate interface and invoke methods as though the elements were separate objects.

It is clear that in this approach the problem of replicated invocations is no longer an issue because we no longer deal with separate objects. Similarly, the state of the related objects is always replicated in the same places and in the same ways, thus the relation between GlobeDoc elements is built into the model. However, the major drawback of this approach is that it limits the element types that a GlobeDoc object can contain. This loss of flexibility is unacceptable for GlobeDoc and is the primary reason for rejecting this model.

3.3 Replication Policies

Although applying an appropriate replication policy is an important aspect of creating scalable GlobeDoc objects, the research performed did not focus on designing and implementing the policies themselves. Instead the goal was to design and build an architecture that supported the flexible application of replication policies. The topic of designing and choosing appropriate replication policies will be revisited in Chapter 7.

3.4 GlobeDoc System Architecture

3.4.1 Overview

Implementing Web documents as Globe DSOs requires structural support for the DSOs. This support includes providing address spaces for LRs, providing access to the services used during binding (i.e., naming service, location service, etc.), and providing a means to access objects from clients such as Web browsers. Figure 3.5 shows an infrastructure that provides support for **GlobeDoc-unaware clients** (e.g., standard Web browsers).

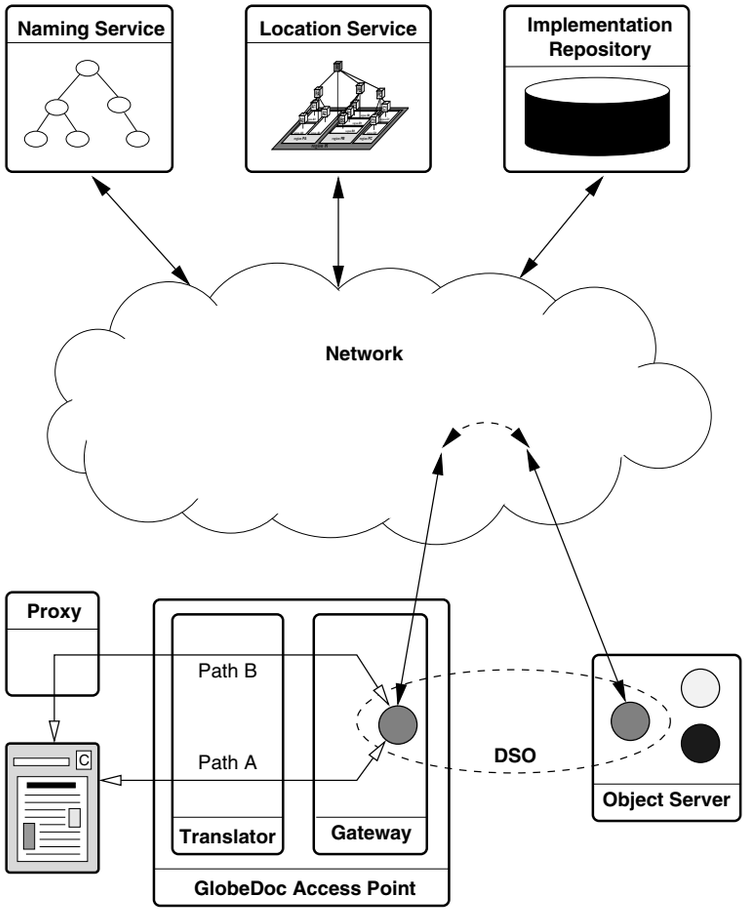


Figure 3.5: The GlobeDoc infrastructure.

In path A of this diagram a client sends an HTTP request for an embedded URN to a **GlobeDoc access point** (GAP). The GAP is responsible for interpreting the request, binding to the appropriate GlobeDoc object and returning the requested element. In path B, the client sends all of its HTTP requests to a GlobeDoc-aware proxy. This proxy filters out GlobeDoc requests (i.e., GlobeDoc URNs or embedded URNs) from other requests, forwarding GlobeDoc requests to a local GAP, and handling other requests in the normal way. A third possibility, shown in Figure 3.6, is that a **GlobeDoc-aware client** binds directly to an object allowing the client to invoke methods directly on that object. The use of GlobeDoc-aware clients will be described later.

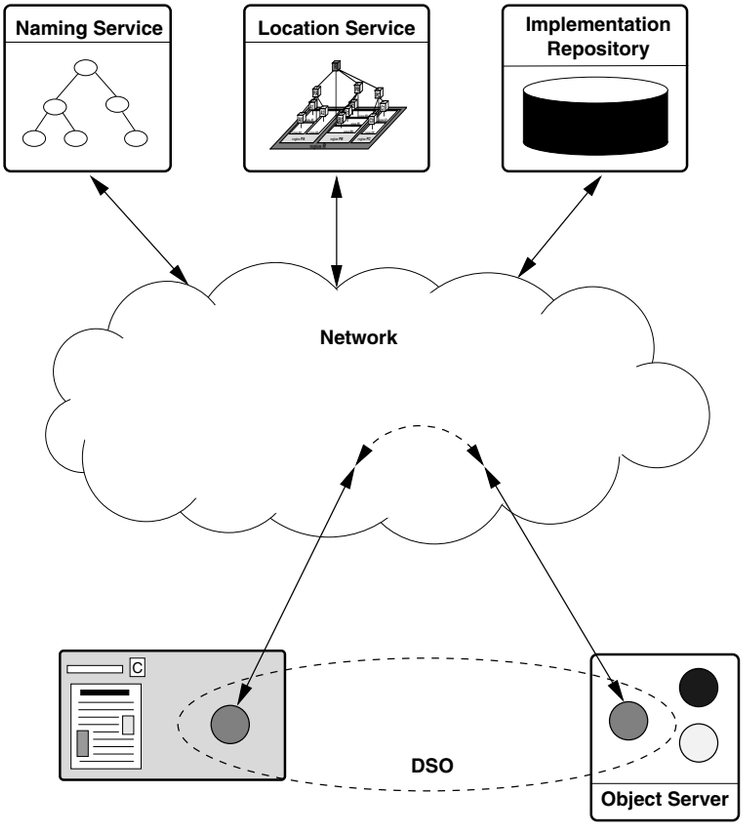


Figure 3.6: GlobeDoc-aware client binding directly to object.

The GlobeDoc access point is a combination of two servers, the **GlobeDoc translator** and the **GlobeDoc gateway**. The translator accepts requests for embedded URNs, translates them to GlobeDoc URNs and forwards the requests to the gateway. The gateway, in turn, accepts the request for a GlobeDoc URN and binds to the object referred to in the URN. Binding to the object causes an LR to be created in the gateway's address space.

Once it is bound to the GlobeDoc object, the gateway invokes methods on the object's LR to retrieve the requested element. In processing the method invocations the LR will generally communicate with one or more of its LR peers. Note that whether this communication actually takes place, and the details of any communication that does take place, is dependent on the replication policy implemented by the LR's replication subobject. When the gateway has retrieved the complete element from the GlobeDoc object, it returns the element to the translator, who passes it on to the client (path A) or the proxy (path B).

3.4.2 The Architecture Elements

Naming and Location

The **naming service** implements a name space for all GlobeDoc distributed objects by mapping **object names** onto **object handles** (which act as unique **object identifiers**). Whereas object handles and the contact addresses that they resolve to are intended for automated processing only, GlobeDoc object names are user-defined and human-readable character strings similar to domain and file names. As in Globe, there is an N-to-1 relationship between these names and object handles, that is, different names can refer to the same object handle, but each name refers to exactly one object handle.

The current name space implementation is largely based on DNS [66] name servers. In this implementation it is assumed that the root as well as (hierarchically) higher-level nodes in the name space correspond to regular DNS domains. In theory, leaf nodes (which represent actual DSOs) as well as lower-level interior nodes also correspond to DNS domains, but these are implemented in a Globe-specific way. Such **Globe domains**, (i.e., Globe-specific as opposed to regular DNS domains) are implemented by **Globe domain servers**.

To adhere to DNS naming syntax, a GlobeDoc object name such as /nl/vu/cs/object/foo is transformed into foo.object.cs.vu.nl. When resolving it, this transformed name is passed to a DNS resolver as though it were a regular domain name. Name resolution proceeds as normal and eventually reaches a Globe name server (e.g., the server for object.cs.vu.nl). The Globe name server recognizes the full GlobeDoc name (foo.object.cs.vu.nl) as an object name and resolves it to the appropriate object handle. More details about the name service and its implementation can be found in [13].

After the name service resolves a GlobeDoc object name to an object handle, the Globe **location service** further resolves the object handle to one or more contact addresses. In GlobeDoc, the protocol identifier part of a contact address contains an **implementation handle**. An implementation handle identifies an **LR implementation** that can be used to contact the object, and data (such as the actual network address of the contact point) used to initialize the LR.

Implementation repository

The **implementation repository** is a service that stores LR implementations and makes them available to binding clients. These implementations are stored and transferred as **class archives**, which are files that contain all the implementation code needed by an LR.

Storing the entire implementation of an LR in a single class archive makes its transportation and management easier compared to having multiple files. In our implementation, a class archive is a Java jar file and contains the Java class files that form an LR implementation.

When an LR implementation is registered at the implementation repository it is assigned an implementation handle. The implementation handle is placed in a contact address and subsequently used by a binding client to retrieve (copies of) the implementation. An implementation handle is an opaque identifier that is generated by the implementation repository.

Currently, only file: URLs are supported as implementation handles, that is, a handle simply contains the path name of a locally available class archive file. Other schemes, such as those based on ftp: or http: URLs, may be preferred for a wide-area system such as the Web. Support for these URLs has not yet been implemented.

Better than URLs, however, are logical names such as URNs, which are globally unique and location transparent. Location transparency has the benefit of allowing one to easily set up a distributed implementation repository without the drawbacks of having to make its distribution visible to the users (as is the case with URLs). For example, it becomes easier to move or replicate class archives without affecting their name as known to users (or stored in contact addresses).

Besides location transparency, URNs also have the benefit of not having to refer to specific class archive files. In other words, a URN can be used as a specification for an implementation *type*. When an implementation handle specifies such a type, the implementation repository is given the freedom to choose an appropriate class archive for the requesting client. A class archive in this sense acts as an instance of the implementation type of the LR. The choice for a specific class archive could, for example, be influenced by the particular platform of a client, or by security requirements. In this way, clients binding to Globe objects can keep control over the code loaded into their address spaces.

Globe Object Server

The Globe object server hosts nonclient LRs, providing them with an address space, contact points, and runtime services. A Globe object server generally runs as an independent process, which means that it is not integrated as part of a client. Clients and other services that offer address spaces and runtime services to GlobeDoc objects will be described later.

A Globe object server can provide a number of different services (such as persistence support, fault tolerance and security facilities, access to local resources, etc.) to the LRs it hosts These services can also be offered at different levels with different guarantees. It is important that LRs are hosted at object servers that provide the services that they require at the levels they require.

Object Server Management Interface Besides simply hosting LRs, the Globe object server also has a remotely accessible interface that allows LRs, other Globe object servers, or administrators to request services from it.

This **object server management interface** (OSMI) is accessed through an RPC-style protocol. It provides functions that allow (object server) clients to request the object server to bind to a specific DSO. Such a bind request causes the Globe object server to bind to the given DSO, resulting in an LR of that DSO being created in the Globe object server's address space. This function is generally used to install object replicas at object servers. An object server can also be requested to unbind from a DSO. In this case all LRs of that DSO which are hosted by the object server are removed from the server's address space.

The OSM interfaces also provide functions that allow objects to be created or destroyed. A creation request is handled similarly to a bind request, except that no name or location lookup is performed, an LR is simply created given an implementation handle and then registered with the location service. The object server does not register a name for the object, it is up to the client to do that. A request to destroy a DSO can be fulfilled only if the object server hosts the only existing LR of that DSO. Destroying a DSO involves destroying all of the DSO's LRs and then unregistering them and the object handle from the location service. Once again, unregistering the object's name (if any) must be done by the client.

Address Space Providing an address space for LRs is straightforward; LRs are passive objects, which means that they do not have an active thread of execution. The Globe object server, therefore, simply needs to provide memory to load the LR code. Memory management is currently handled by the local garbage collector. In addition, the server provides the runtime support needed by LR implementations. For example, a Java virtual machine and accompanying runtime library are needed to support Java implementations of LRs.

Although LRs are not active objects, they do require thread management facilities. For example, a thread is started whenever a message comes in from another LR. The thread facilities are provided by the runtime system.

Contact Points An important property of an LR is that it communicates with other LRs and possibly other services. This means that an object server must provide an LR with access to the network. Besides simply allowing LRs read and write access to the network, an LR must be allowed to create contact points. As mentioned earlier, a contact point is where an LR can be contacted by its LR peers. A contact point is represented by a contact address and registered at the location service. The object server's runtime system provides services for creating, managing and destroying contact points.

Because contact addresses are made publicly available through the location service, it is possible that a contact address survives longer than the object server that provides the associated contact point. Some LRs may require their contact points to be persistent, that is, they may require their contact points to remain valid over multiple restarts of the object server hosting them. It is up to the object server to make sure that the contact addresses for such contact points will always refer to the same (logical) contact point.

Access to external services The naming service, location service and implementation repository are all external services, that is, they are implemented outside of the Globe object server. Because LRs (and other runtime system components) can access only resources in the Globe object server's address space, the runtime system provides local proxies to the external services. These proxies, called **resolvers**, provide local interfaces through which the external services can be used. They can be implemented as simple proxies that forward all requests and replies to and from the actual services, or they can be more complex, storing and manipulating their own local state (e.g., to cache results). The latter are often used to improve system performance. Performance of access to external service is important because it can greatly affect the overall performance of the client-to-object binding process.

Support for binding The Globe object server also provides facilities needed for binding. These are encapsulated in a **binder object**, a local object that is part of the runtime system. As discussed earlier, binding to a GlobeDoc object consists of at least three steps: (i) name resolution, (ii) object handle resolution, and (iii) loading and initialization of an LR. Normally, binding starts at the first step. It is, however, possible to begin binding at any other step, as long as the information needed by that step is present. For example, to start binding at the second step, a client would need to have an object handle to pass to the location service. A Globe object server might store an object handle as previously returned in the first step to avoid a name lookup when it is later requested to bind to that same object again.

When a Globe object server is requested to unbind from a DSO its LR for that DSO has to be disconnected from the rest of the DSO. The process of disconnecting an LR from the rest of a DSO is generally object specific. For example, in some cases it may be necessary to migrate the LR's state to another Globe object server, while in other cases, it may be safe to simply discard the state because the LR is, in fact, a replica. Also, if the Globe object server was offering a contact address for the DSO, the corresponding contact addresses would have to be removed from the location service. Therefore, when unbinding from a DSO, we assume that the DSO implements its own disconnection algorithm. When the LR has been disconnected, the server simply reclaims local resources and removes the LR from its address space.

Local resources A Globe object server must also manage local resources such as secondary storage, memory, network access, etc. The runtime system manages access to many of these low-level resources through standard platform-independent interfaces and abstractions such as communication points (e.g., sockets) and persistent storage (e.g., files on disk). Memory management is done partially through language-specific mechanisms (such as malloc() and free() in C or garbage collection in Java) as well as garbage collection of unused LRs and other Globe-specific constructs. The object server must also manage the number of active (that is loaded in memory) LRs, so that they do not take up all available memory. For example, if the LRs start taking up too much memory, then the object server may decide to remove (unbind) or suspend, the least active ones. Likewise,

communication resources may also be limited (e.g., a limit on the number of open sockets per process) and the object server will have to manage use of these as well.

Clients

GlobeDoc clients are programs that access GlobeDoc objects. They can be used to access (and view) the contents of GlobeDoc objects, or to manage and modify a GlobeDoc object's contents. There are two types of clients: GlobeDoc-aware and GlobeDoc-unaware clients. **GlobeDoc-aware clients** are those that can directly bind to GlobeDoc objects and invoke their methods. These clients are built with Globe runtime support and are capable of loading and hosting GlobeDoc LRs in their address space. These clients also recognize GlobeDoc URNs and can properly interpret them. Generally GlobeDoc-aware clients are custom made, or custom-modified programs.

GlobeDoc-unaware clients, on the other hand, do not know anything about GlobeDoc. They cannot bind to or invoke methods on GlobeDoc objects, nor do they understand nor can they interpret GlobeDoc URNs. Generally GlobeDoc-unaware clients are existing programs such as Web browsers and other Web-based tools. These clients can only access GlobeDoc objects through a GlobeDoc-aware HTTP server, such as provided by a GlobeDoc access point.

GlobeDoc Gateway

A **GlobeDoc gateway** is a gateway between GlobeDoc-unaware clients and GlobeDoc objects. It provides an interface through which clients can bind to a GlobeDoc object and call its methods. This can take the form of a dedicated RPC-style interface, or a server that accepts custom HTTP requests from clients. Upon receiving a request (either via an RPC or HTTP) the gateway binds to the appropriate object, invokes methods on that object, and returns any results to the requesting client.

A gateway is similar to the Globe object server, except that it supports only **client LRs**. Client LRs are similar to regular LRs, except that they cannot be bound to. This means that they do not provide any contact points and are therefore not registered with the location service. Because of this, a gateway does not provide contact points.

Another difference between the gateway and the object server is that the LRs hosted by a gateway are short-lived. Whereas the LRs hosted by an object server are usually created as somewhat permanent replicas, LRs hosted by a gateway usually function as a simple access point to GlobeDoc objects. The gateway generally binds to an object, invokes some of its methods, and then unbinds from the object. Note, however, that it is not always wise to immediately unbind from a DSO once a client request has been satisfied. Consider, for example, a GlobeDoc gateway that has just bound to a DSO in order to retrieve an element for a client. In the same style as HTTP, the gateway could decide to immediately unbind from the DSO as soon as it has received the element from the object. However, if the client immediately request another element from the same object, it would have been much more efficient to stay bound to the DSO, anticipating more requests for that object. In effect, a gateway can decide to **cache bindings** for later

use. In our current implementation, which supports only passive Web documents, the effects of caching bindings turns out to be comparable to that of traditional Web caches.

Translator and Redirector

Ideally, users should be able to use regular Web browsers to access GlobeDoc Web documents. Unfortunately, current browsers are incapable of resolving GlobeDoc URNs as they do not understand globe: URI scheme identifiers. One way around this problem is to use GlobeDoc-aware proxies. These are Web proxy servers that filter out GlobeDoc requests and send them to a (local) GlobeDoc gateway. Any results from the gateway are returned to the user's browser through the proxy. Non-GlobeDoc requests are passed to appropriate servers.

A disadvantage of the proxy approach is that *all* requests from the browser (including nonGlobeDoc requests) may be forwarded through the proxy. As a result, the proxy must be able to handle all the various kinds of schemes supported in URLs, or be able to forward them to a proxy that can. Another problem is that in order to use a proxy a user's browser must be configured to forward requests to the proxy. Users who have not configured their browsers to use an appropriate proxy, will not be able to access GlobeDoc content.

A different approach to integration with the Web makes uses of a **GlobeDoc translator**. This component translates GlobeDoc URNs to embedded URNs and vice-versa. As mentioned earlier, an embedded URN is a regular HTTP URL that contains an object name and a GlobeDoc-aware HTTP server address, such as http://globedoc.cs.vu.nl/nl/vu/cs/foo/object. When an embedded URN link is clicked in the browser, an HTTP request for the embedded object name is sent to the server whose address is contained in the URL. When using the translator, this address will refer to a GlobeDoc translator. This translator converts the embedded URN into a GlobeDoc URN and forwards the request to a local GlobeDoc gateway. The gateway binds to the object and retrieves an element as usual, passing the results back to the translator. At the translator, the element is scanned and each instance of a GlobeDoc URN is rewritten to contain an equivalent embedded URN. The modified element is then passed on to the browser.

A problem with the translator and embedded URN approach is that the translator address is embedded in the resulting URL. Although this is not a problem for users close to the translator, this may be a problem for users further away. Not only does this introduce extra latencies, but it may also cause a problem for translators whose address is included in embedded URNs of popular GlobeDoc objects. The essence of the problem is that embedded URNs are not location independent, they contain the address of a specific translator.

The **GlobeDoc redirector**, an HTTP server that redirects clients to their nearest translator, attempts to solve this problem. When using a GlobeDoc redirector, published embedded URNs contain the redirector's address rather than a translator's address. Requests for such embedded URNs are, therefore, sent straight to the redirector. Upon receiving a request, the redirector finds a translator close to the client, and tells the client to send its request to that translator. The nearest translator is defined as the one that is nearest to the client in terms of geographic distance. To compute this distance, the IP addresses of the

client and known translators are mapped to physical locations (i.e., latitude and longitude coordinates). Note that embedded URNs returned by the translator, do not contain the redirector's address. This allows subsequent requests to be directed straight to the closest translator.

Although the redirector is a centralized service and presents a potential bottleneck, it is only used the first time a client accesses a GlobeDoc object. Afterwards, the translator will ensure that all embedded URNs reference itself, and the redirector will not be used for that GlobeDoc object. The redirector is also a temporary solution, allowing GlobeDoc-unaware clients to access GlobeDoc objects and maintain the locality that is necessary for performance. Ideally clients would be GlobeDoc-aware and bind directly to GlobeDoc objects, avoiding the redirector, gateway and translator, which are all mechanisms intended to make integrating GlobeDoc with the current Web possible.

Globe Infrastructure Directory Service

As we have seen, the use of Globe and GlobeDoc requires a number of external services (naming service, location service, object server, gateway, translator, etc.). Finding these required services may not always be trivial. For example, a translator forwards requests to a gateway and although the gateway may run on the same machine as the translator, this is not required. Ideally when the translator starts, it should be able to dynamically find its nearest gateway. Likewise, all Globe processes make use of the location service. To use the location service they must contact an appropriate location service leaf node. Determining which node that is is not always trivial and should be done dynamically when the process starts. Similarly, when creating an object's replicas, appropriate Globe object servers must be found. What constitutes an appropriate server depends on the LR's resource and location requirements (e.g., what resources the LR will use, and where the LR should be located).

The **Globe infrastructure directory service** (GIDS) is a system for managing the resources and services made available by a world-wide distributed collection of Globe object servers and other Globe and GlobeDoc services. GIDS allows servers to register their resources, services, and other properties while providing clients an efficient search facility for locating the servers and services they need.

In GIDS, a distinction is made between local resource management and global resource management. **Local resource management** deals with managing resources and services within a base region. A **base region** represents a small geographical or network-topological area containing a group of servers (e.g., Globe object servers, a GAP, a location service leaf node, etc.). It is the smallest unit of proximity known to GIDS. In other words, if two processes are in the same base region, GIDS will consider them as being at the same location.

Global resource management deals with grouping the information from base regions into larger units, and making that information available to clients. In particular, global resource management deals with globally tracking resources and services, and providing efficient search facilities for clients, regardless of a client's location.

GIDS is described in detail in Chapter 5.

Chapter 4

GlobeDoc Architecture Details

The previous chapter introduced the GlobeDoc model and the main elements of the associated architecture. This chapter continues where the previous one left off and looks deeper into the design and implementation of those architectural elements. In particular, it includes a discussion of the Globe object server, the implementation repository, GlobeDoc clients, and GlobeDoc cache servers. These are architectural elements and issues that, although generally applicable to other Globe applications, are of particular importance to GlobeDoc.

4.1 Object Server

As mentioned earlier, the **object server** is a process that hosts local representatives of GlobeDoc objects. It allows LRs to be installed on it, provides them with an address space in which to run their code and gives them access to run-time services such as memory management, persistence support, and a local name space. The object server also provides access to local storage and network resources as well as access to external services such as the naming and location services. This section will describe the design and implementation of the Globe object server.

Because it was designed as a general Globe service, the object server supports any kind of Globe object, not just GlobeDoc objects. As such, the text in this section will refer to generic Globe objects rather than specifically to GlobeDoc objects. Where possible particular details regarding the applicability to GlobeDoc and GlobeDoc-specific examples will be provided.

Figure 4.1 shows the object server's internal structure. In this figure the object server's functionality is divided into separate functional components. The **object server management component** provides an entry point through which object-server clients can invoke management operations on the object server. It provides an interface that allows clients to create and destroy LRs as well as access information about LRs hosted on the object server. The **LR management component** is responsible for creating and destroying

LRs as well as keeping track of the LRs that the object server hosts. This component is also responsible for the functionality required to activate and passivate persistent LRs. The **Globe runtime services component** provides all the services required by a standard Globe-aware application. These include providing access to local and remote services such as a binding service, the name service, the location service, and an implementation repository. The **local storage management component** provides access to local storage such as a local file system or a locally-accessed remote file system. This component is also responsible for freeing and cleaning up unused storage space, as well as maintaining persistently allocated storage space. The **network management component** provides access to network resources such as communication and contact points. The following sections will describe each of these components in more detail.

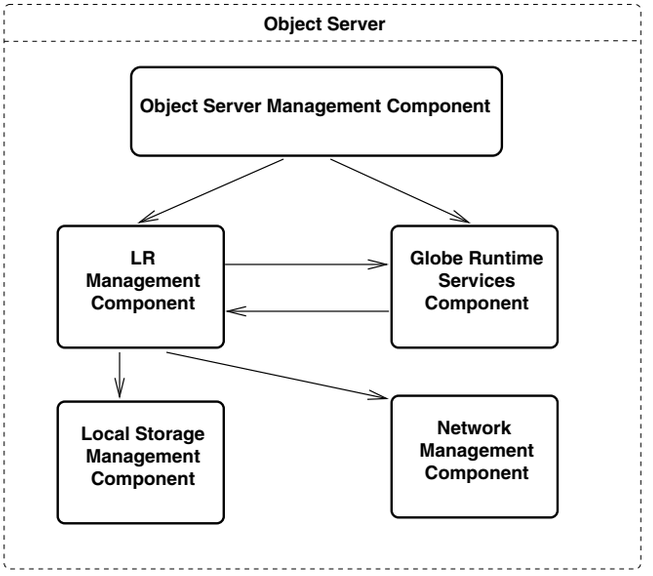


Figure 4.1: Structure of the Globe object server.

4.1.1 Object Server Management Component

The object server management (OSM) component provides clients with access to the management functionality provided by the object server. This functionality is defined in the **object server management interface** as presented in Table 4.1. The OSM component does not actually perform the actions defined in this interface, but simply coordinates their execution and delegates calls to the other components. This management functionality can be accessed in two ways. The first way is locally through a shell interface that the object server presents if it is run in interactive mode. The second way is using an external management client that contacts the object server and communicates with it using

an RPC-style protocol. Such a management client can be a special-purpose object-server management client, but can also include regular clients that need to create or destroy objects, as well as objects themselves if the objects are able to create their own replicas.

The main functionality provided by the OSM interface is creation of LRs. There are two ways that an object server can create LRs. The first is to bind to an already existing Globe DSO and the second is to create an LR given an implementation handle and initialization data.

There are three ways that an object server can bind to an existing Globe DSO. The first way, represented by the bind method, binds to an existing object given its object handle. This causes the object server to perform a location lookup to find an appropriate contact address. Given a contact address the object server extracts the implementation handle, loads the implementation into its address space and initializes it with initialization data also extracted from the contact address. Contact address selection follows a strategy defined in the property selector. Generally such a property selector defines which properties of location service records are to be considered when searching for a contact address. If the object server is already bound to the object identified by the given object handle, then the bind operation is not performed.

The second way of binding to an existing DSO is represented by the bindCAddr method. In this case the object server is given a specific contact address to bind to. The binding process is similar to that followed in the bind method, except that it is not necessary to contact the location service and search for a contact address.

The third way of binding to an existing DSO is represented by the addBinding method. This method is similar to the bind method, except that the operation is performed even if the given object is already bound to. In that case a call to this method results in an extra LR, bound to the same object, being hosted by the object server. This method is generally used when a user requires an LR with properties different from the currently hosted LRs to be added to the object server. For example, this method may be used to cause the object server to host an LR that provides more (or less) security guarantees than already-hosted LRs.

The createLR method is generally used to create the first LR of a DSO. As such it cannot be given an object handle or contact address to bind to because none exist. Instead the method is given an implementation handle and initialization data. The method acquires the implementation code from an implementation repository, loads it into the object server's address space, instantiates the LR and initializes it with the given initialization data.

All four of the above-mentioned methods return an **LR identifier**. This identifier is an object server management specific identifier that can be used to identify a particular LR in subsequent calls on the object-server interface. This includes calls to remove LRs as well as calls to get more information about installed LRs. LR identifiers are also used internally by the object server to keep track of LRs, for example, when keeping track of resources used by an LR.

In all four methods it is possible to specify whether the LR should be persistent or not.

There are two methods for removing LRs. The first, removeLR, causes a given LR to be removed from the object server. The LR identified by the given LR identifier is

interface	objectServerOps	
method	bind	Binds to a DSO given an object handle
in	objectHandle	The object handle to bind to
in	oname	The DSO's name
in	select	Property selector used to select a contact address
in	persistent	Whether the installed LR should be persistent or not
returns		An object-server specific LR identifier
method	bindCAddr	Binds to a DSO given a contact address
in	objectHandle	The DSO's object handle
in	oname	The DSO's name
in	caddr	The contact address to bind to
in	persistent	Whether the installed LR should be persistent or not
returns		An object-server specific LR identifier
method	addBinding	Binds to a DSO (even if already bound to it)
in	ohandle	The object handle to bind to
in	oname	The DSO's name
in	select	Property selector used to select a contact address
in	persistent	Whether the installed LR should be persistent or not
returns		An object-server specific LR identifier
method	createLR	Creates and installs a DSO's first LR
in	ohandle	The object handle to assign to the DSO
in	oname	The DSO's name
in	impl	Implementation handle of the LR to create
in	init	Initialization string for the LR
in	persistent	Whether the installed LR should be persistent or not
returns		An object-server specific LR identifier
method	removeLR	Removes an installed LR
in	lrid	The LR's object-server specific identifier
method	unbindDSO	Removes all of a DSO's installed LRs
in	ohandle	The DSO's object handle
method	listAll	Lists the LRs currently installed in the object server
returns		A list of LR identifiers
method	listDSO	Lists all of a DSO's LRs currently installed in the object server
in	ohandle	The DSO's object handle
returns		A list of LR identifiers
method	statLR	Returns the status of an installed LR
in	lrid	The LR's object-server specific identifier
returns		The status information

Table 4.1: Object Server Management Interface.

destroyed, freeing all resources it had in use. The second method, unbindDSO, removes all of a given DSO's installed LRs. This is an all-or-nothing operation, that is, either all LRs are removed, or none are. Note, however that it is not a fault-tolerant operation, i.e., if the object server were to crash while performing the operation no guarantees are made as to which LRs have been removed and which have not.

Finally, there are three informative methods. The first method, listAll, returns a list of the LR identifiers of all currently installed LRs. The second method, listDSO, returns the identifiers of all LRs of a given DSO that are currently installed in the object server. The third method, statLR returns information about a given LR installed in the object server. The information returned includes the object handle of the DSO that the LR belongs to, information about how the LR was installed (e.g., was this the first LR of the DSO, how a contact address was chosen, etc.) and whether the LR is persistent or not.

4.1.2 LR Management Component

The LR management component is responsible for the creation and destruction of LRs and keeping track of all hosted LRs. It also manages the activation and passivation of persistent LRs as well as the saving and restoring of state that this involves.

In order to host LRs the object server must provide an address space and any system-level support required by the LR implementations. Two important system facilities that must be provided are thread management and garbage collection. Although Globe objects do not have active threads of control, Globe does make heavy use of pop-up threads (for example, in communication subobjects when accepting incoming messages). Because operating systems often limit the number of threads available to a single process, the object server has to make sure that it never uses up all its available threads. The object server manages available threads by providing a pop-up thread pool. Threads are checked out of the pool when needed and returned to the pool when no longer required.

The Globe model supports garbage collection through **reference counting**. In Globe, local objects are accessed through counted references. Whenever a reference to a local object is acquired the count of references increases. Whenever a reference is no longer used it is released and the count decreases. When a local object's reference count drops to zero, the object is no longer referenced and can be removed. In some Globe environments (such as a C implementation) the reference counting and object cleanup is implemented as part of the Globe runtime system and does not rely on system-level support. In other environments (such as a Java implementation) the reference counting and object cleanup relies partially or wholly on system-level support. This reliance on system-level support means that in some cases the exact behavior of certain processes (e.g., destruction of local objects) may differ depending on platform and implementation. For example, in a Java environment, an unreferenced local object will be cleaned up only when the Java garbage collector is run. In a C environment, however, the local object may be cleaned up as soon as its reference count drops to zero.

LR Creation

As mentioned previously there are two main ways that an object server creates LRs. The first is by binding to an existing Globe DSO, corresponding to the OSM interface's bind methods, and the second is by creating a Globe object's initial LR, corresponding to the OSM interface's createLR method.

Invoking the bind method results in the invocation of corresponding methods on the Globe runtime system's **binder object** (see Section 4.1.3). The binder object is responsible for mapping the given object name, object handle or contact address to a corresponding implementation handle and initialization data. The implementation handle and initialization data are then passed on to the LR management component where the actual LR creation takes place. Invoking the createLR method results in the implementation handle and initialization data being passed directly to the LR management component where LR creation takes place. Figure 4.2 shows the steps involved in object creation.

```
in: implementation handle, init data, object handle
out: reference to LR

1. Pass implementation handle to implementation repository.
   Implementation repository returns class loader.  Class
   loader is used to load code and create LRManager class
   object.

2. Create LRManager object.
2.a Invoke create() on LRManage class object to create
    LRManager object.
2.b Register LRManager object in LNS.

3. Initialize LRManager object.
3.a Invoke configurable.configure(init data) on LRManager
    object.
3.b For each subobject implementation handle specified in init
    data:
3.b.i   Get class object from LNS or implementation repository.
3.b.ii  Create the subobject by invoking create() on the class
        object.
3.b.iii Register the subobject in LNS.
3.b.iv  Invoke configurable.configure(subobject init data) for
        the subobject.
3.c Point LRManager's control interfaces to the control subobject.

4. Configure LR.
4.a Invoke distributed.createDistrObject() or
    distributed.bindDistrObject().
4.b Invoke distributed.setObjectHandle(object handle).
4.c Invoke distributed.distributeObject().

5. Register LR (as a reference to the LRManager object) in LR
   table.
```

Figure 4.2: Steps involved in creating an LR.

In the first step the implementation handle is passed to the implementation repository. The implementation repository uses information in the implementation handle to find a suitable implementation. The implementation is returned together with a class loader, which the object server uses to load the implementation into its address space. The object server has two possibilities for loading the code. It can immediately load the code or load it only when it is actually needed. When immediately loading the code, the object server loads all the code required by the LR, including the code for all of the LR's subobjects. When loading it later (called **lazy loading**) the object server only loads the code for the LR's main subobject (called the **LRManager subobject**). The code for other subobjects is loaded only when those subobjects are actually needed. Details of how the class loader works and how code is found and loaded are provided in Section 4.2.

The LRManager is a subobject that is responsible for building and initializing the rest of the LR.[1] Figure 4.3 shows the position of the LRManager subobject in relation to an LR's other subobjects. Because of its administrative role, all interaction with the LR occurs through this subobject. As such, an LR reference is implemented as a reference to the LRManager. This means that any method invocation on the LR is first handled by the LRManager and then passed on to appropriate subobjects. The LRManager, therefore, exports the control subobject interfaces (which are derived from the semantics subobject interfaces). In GlobeDoc this includes at least the document, content, property, and lock interfaces. Besides the semantics subobject interfaces the LRManager also exports the distributed interface (see Table 4.2) which is used for initializing the whole LR. This interface will be described in more detail later.

interface	distributed	
method	createDistrObject	Start of call sequence to create the first LR of a new DSO
method	getContactAddresses	Retrieves the contact addresses of the master replica
returns		The contact addresses
method	bindDistrObject	Start of call sequence to bind a local representative to an existing DSO
method	setObjectHandle	Informs an LR of its object handle
in	ohandle	The object handle
method	distributeObject	Final call after all data has been set
method	prepareImmediateDestruction	Informs this object that it will be destroyed soon, and that destruction should occur quickly

Table 4.2: The distributed interface.

[1] The LRManager subobject was not introduced in Chapter 3 because it serves an administrative role and does not affect the logical model of a Globe LR.

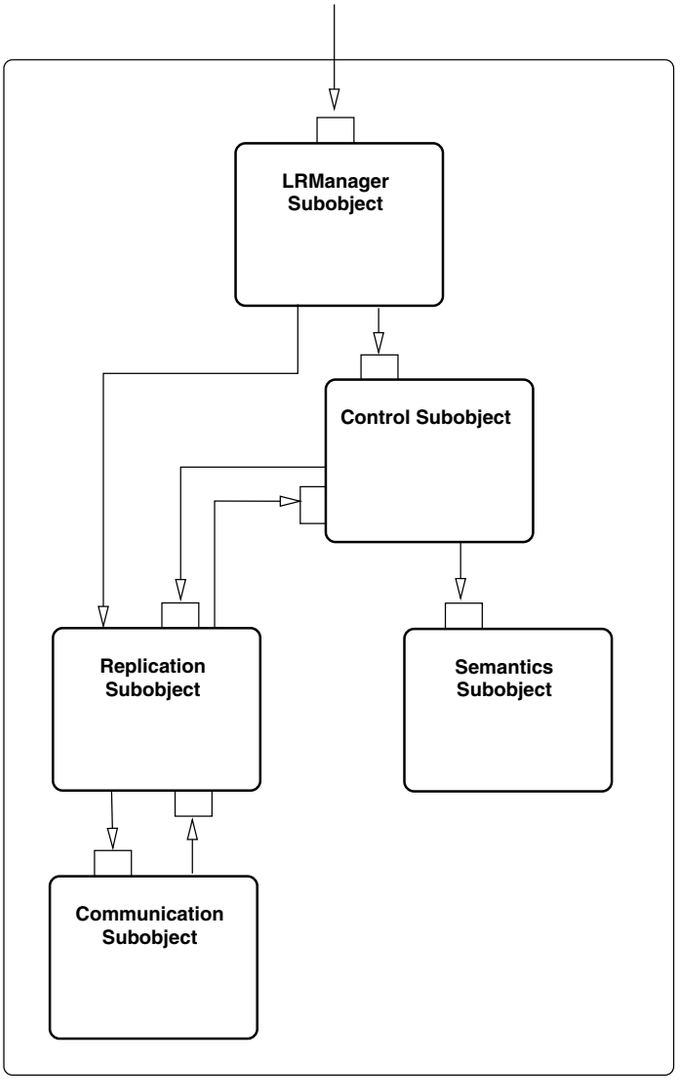


Figure 4.3: Position of the LRManager subobject in relation to an LR's other subobjects.

The second step in creating an LR involves creation of the LRManager subobject and makes use of a Globe class object created in the previous step. Every Globe class object implements a create method that is used to create an object instance of that class. A call to a class object's create method causes a Globe local object to be created and a reference to that object's standard object interface to be returned. The LRManager's class object is used to create an instance of the LRManager. After the LRManager object has been created it is registered with the Local Name Space (LNS - see Section 4.1.3).

After creating and registering an LRManager object, the next step involves initializing this object. Besides the standard object interface, all Globe LR subobjects also export and implement the configurable interface. This interface provides methods that allow a subobject to be configured with subobject-specific initialization data. After being created and registered with the LNS, the LRManager object is configured using this configurable interface. The initialization data passed to the LRManager consists of a list of implementation handles for the rest of the LR's subobjects (semantics, control, replication and communication) and corresponding initialization data for each of the subobjects. The implementation handle and initialization data for the communication subobject are optional. If they are not present then an appropriate system-specific communication subobject, as provided by the Globe runtime system, is used (see Section 4.1.3).[2]

Figure 4.4 shows an example of a GlobeDoc LRManager's initialization data. The data consists of entries in the form of attribute-value pairs. There are two entries for each of an LR's subobjects. The first entry (e.g., sem-imp) specifies the Java class (or classes) that implements the subobject, while the second (e.g., sem-init) specifies initialization data for that subobject. Like the LRManager's initialization data, the initialization data for each of the subobjects also consists of a collection of attribute-value pairs. In this example, the data includes four implementation handles, which specify the Java classes that implement each subobject, as well as initialization information for each of these subobjects. The semantics subobject does not require any initialization data. The communication subobject receives many initialization parameters that deal with setting up and contacting multiplexed contact points. The initialization data for the replication subobject includes the address of an LR peer to contact. Finally, the control subobject, like the semantics subobject, receives no initialization data.

For every subobject specified in the initialization data, the LRManager takes one of two actions depending on whether the implementation repository loaded the LR code lazily or not. In the case of lazy loading the LRManager contacts the implementation repository and asks it to load the given implementation handle and return the associated class object. In the case of nonlazy loading the LRManager retrieves a reference to the appropriate class object from the LNS. Given the class object the LRManager then creates a subobject instance, registers it with the LNS (in a name-space context relative to

[2]Communication subobjects need to access system-specific network resources, which, in many cases, requires system specific knowledge (e.g., which network protocols are supported, how to access the network resources, etc.). Because of this it is usually better for the runtime system to provide the communication subobjects, rather than have the LR provide its own communication subobject. As such, an LR really only needs to specify its own communication object in exceptional cases (e.g., for debugging purposes or when making use of nonstandard network protocols).

```
<sem-impl> =
  <JAVA;vu.globe.globeDoc.globeDocumentImp.globeDocumentImpClassObject>,
<comm-impl> = <JAVA;vu.globe.comm.lwconn.LightConnectorCO>,
<repl-impl> = <JAVA;vu.globe.lr.replication.MasterSlave.msClientCO>,
<ctrl-impl> =
  <JAVA;vu.globe.globeDoc.globeDocumentImp.gdControlClassObject>,
<sem-init> = <>,
<comm-init> =
  < <mux-implementation> = <JAVA;vu.globe.comm.muxconn.MuxConnectorCO>,
    <protocol-stack> = <[ip,tcp,ConnMux0]>,
    <mux-initialisation> =
      < <p2p-initialisation> = <>,
        <p2p-implementation> =
          <JAVA;vu.globe.comm.tcp.TcpConnectorCO> > >,
<repl-init> =
  < <server-address> = <[ip:130.37.192.25,tcp:22999,ConnMux0:-9]>,
    <sec-ownershipcheck> = false>,
<ctrl-init> = <>
```

Figure 4.4: Example initialization data for a GlobeDoc LRManager.

the LRManager's own registration - see Section 4.1.3 for more details about the LNS and name-space contexts), and initializes it with the appropriate initialization data. If no communication object was specified in the LRManager's initialization data, then the LRManager finds a reference to a default communication class object in the LNS and uses this to create a communication subobject. Once all the subobjects have been created the LRManager configures the methods in its own (exported) control interface to call equivalent methods on the control subobject.

In the fourth step, after the LRManager and all its subobjects have been created and initialized, the methods of the LRManager's distributed interface are invoked. These methods complete the initialization of the LR and involve linking all subobjects together, contacting the rest of the DSO, etc. If the LR is being created as a result of calling createLR then the first method called is createDistrObject, which informs the LR that it is being created as a Globe object's primary LR. Otherwise bindDistrObj is called to inform the LR that it will become part of an already existing Globe object. Both methods allow preparatory actions to be taken.

The second method to be called is the setObjectHandle method which tells the LR which Globe object it will be part of. If the LR is being created with createLR then a new object handle for the Globe object is generated by the location service, otherwise the object handle discovered during the binding process is used.

Finally the distributeObject method is invoked. This method causes the LR to make itself available to other LRs and to contact its LR peers. Making itself available to other LRs requires the creation of contact points and the registration of the corresponding contact addresses with the location service. Contact point creation is coordinated by the replication object. As part of the contact point creation process the replication object also determines the communication and replication subobjects required to access them.

After the contact points have been created, the LRManager creates contact addresses for them. As mentioned previously, a contact address contains an implementation handle for an LR and appropriate initialization data. The LRManager includes its own implementation handle as the implementation handle part of the contact address. The initialization data consists of the implementation handles and initialization data for the subobjects. The LRManager determines the implementation handles and initialization data for the semantics and control subobjects, while the replication subobject supplies the information required for the replication and communication subobject parts. Note that the contact point address is included in the replication subobject's initialization data. When a contact address has been created for each of the LR's contact points then these addresses are registered with the location service. Figure 4.5 shows an example contact address. Besides the implementation handle and initialization data, the contact address also contains some location service specific data such as an address identifier (addr_id) and a property mask (props).[3]

```
domain:13,
addr_id:5b69703a3133302e33372e3139322e31362c7463703a32323939392c436f6
        e6e4d7578303a2d315d0300000000000000,
address:
  impl-handle=<JAVA;vu.globe.globeDoc.globeDocumentImp.gdLRImpClassObject>
  init=
    <sem-impl> =
    <JAVA;vu.globe.globeDoc.globeDocumentImp.globeDocumentImpClassObject>,
    <comm-impl> = <JAVA;vu.globe.comm.lwconn.LightConnectorCO>,
    <repl-impl> = <JAVA;vu.globe.lr.replication.MasterSlave.msSlaveCO>,
    <ctrl-impl> =
      <JAVA;vu.globe.globeDoc.globeDocumentImp.gdControlClassObject>,
    <sem-init> = <>,
    <comm-init> =
      < <mux-implementation> =
          <JAVA;vu.globe.comm.muxconn.MuxConnectorCO>,
        <protocol-stack> = <[ip,tcp,ConnMux0]>,
        <mux-initialisation> =
          < <p2p-initialisation> = <>,
            <p2p-implementation> =
              <JAVA;vu.globe.comm.tcp.TcpConnectorCO> > >,
    <repl-init> =
      < <server-address> = <[ip:130.37.192.16,tcp:22999,ConnMux0:-1]>,
        <sec-ownershipcheck> = false>,
    <ctrl-init> = <>,
props:{0, 1}
```

Figure 4.5: Example contact address.

The final step of the LR creation process involves storing a reference to the LRManager in the object server's **LR table**. This is a table that keeps track of an object server's hosted LRs and their properties. The LR table will be described later.

[3]The details of the location service specific data contained in the contact address is beyond the scope of this dissertation. More details about the location service and its implementation can be found in [12].

LR Destruction

As with LR creation there are also multiple ways to destroy an LR. An LR can be destroyed as a result of calling the object server management interface's removeLR or unbindDSO methods. An LR can also be destroyed as a result of the object server shutting down.

Calling the removeLR method results in the destruction of a single, specific, LR. The unbindDSO method, on the other hand, results in the destruction of all LRs hosted by the object server that are part of a given Globe object. This method retrieves a list of LRs belonging to the given Globe object and destroys them one by one (by invoking removeLR on each LR individually). During an object server shutdown, the garbage collection process causes all hosted LRs to be **released**. Note that releasing an LR is not equivalent to permanently destroying it. When released, persistent LRs are given a chance to save their state before the LR is destroyed. For transient LRs release is equivalent to destruction.

Figure 4.6 shows the steps involved in the destruction (or release) process for a transient LR. The differences for a persistent LR will be described later on in this Section.

```
in: LR identifier

1. Get reference to LR given LR identifier.

2. Remove reference from table, also remove all other references
   to LR (including from LNS).

3. LR is garbage collected.
3.a If this is not an object server shutdown
3.a.i All registered contact addresses are unregistered.
3.b Replication subobject is told to clean up.
3.b.i   Possibly update remote states (if not object server
        shutdown).
3.b.ii  Break connections to all LR peers.
3.b.iii Unregister contact points
3.c Semantics subobject is told to clean up.

4. LR object instance is destroyed.
```

Figure 4.6: Steps involved in the LR destruction process.

The process starts by mapping an LR identifier, which is an index into the LR table, to an LR reference (a reference to the LR's LRManager subobject). Next, the LR's LR table entry and all other references to the LR are removed. Once all references to the LR are removed it becomes unreferenced and the LR is garbage collected, initiating the LR cleanup process.

The first step in this process is that all of the LR's registered contact addresses are unregistered from the location service preventing any processes from connecting to or contacting the LR. Note that this is done only in the case of a regular LR removal. In the case of an object-server shutdown the server cannot rely on network communication and therefore cannot reliably contact the location service.

After unregistering the contact addresses, the replication and semantics subobjects are told to clean up their state. The replication subobject starts first. Depending on its implementation and whether this is a server shutdown or not, the replication subobject may attempt to communicate with its LR peers to update the Globe object's state. After completing any communication with other LRs the replication subobject breaks all network connections it has with its LR peers. Once all connections have been broken the communication subobject is told to deallocate all contact points.

After the replication subobject has performed its cleanup the semantics object is allowed to clean up its state. The actions taken by a semantics subobject during cleanup are implementation dependent. The GlobeDoc semantics subobject, for example, removes all state that it has stored on secondary storage before deallocating all memory it has used for storing its state.

After all the subobjects have cleaned up their state, the LRManager along with all the other subobject instances are destroyed and removed from memory.

Note that the above describes the situation where an LR may be destroyed. Depending on a GlobeDoc object's distribution strategy an object may not allow certain LRs to be destroyed, or may require authentication before an LR is destroyed. For example, if a GlobeDoc object's state is partitioned (and not replicated) over its LRs, then destroying an LR would result in the destruction of (part of) the GlobeDoc object's state. Before allowing destruction of such an LR, a copy of that LR would have to be created elsewhere or the state contained in that LR would have to be copied to its LR peers.

LR Administration

The object server keeps track of hosted LRs in an LR table, a table that keeps track of LRs and their properties and maps internal LR identifiers to LR references. The LR table is generally used to get status information about a particular LR, get a list of all installed LRs, distinguish between persistent and transient LRs and map between internal LR identifiers and object handles or names.

The table contains an entry for each LR hosted by the object server. An **LR entry** contains an LR identifier, the object handle and name of the DSO that the LR is part of, how the LR was created (i.e., created directly or through binding), a flag telling whether or not the LR is persistent, the time at which the LR was created, a time stamp specifying when the LR's state was last modified, and a list of all the contact addresses exported by the LR (this may include contact addresses that are not registered with the location service). An entry for a persistent LR contains the above, as well as a field containing the LR's persistence identifier and a reference to an associated persistence manager.

The LR table is a persistent data structure. This means that it is saved to secondary storage when the object server shuts down.

Persistence Management

Persistent and transient LRs were introduced in Chapter 3. Recall that a transient LR is one whose state is lost when the process hosting it ceases to exist (e.g., the object server is

shut down). A persistent LR's state, on the other hand, persists across subsequent restarts of the hosting process.

When shutting down, the object server must make sure that the state of all persistent LRs is safely stored on some sort of persistent storage (e.g., a local disk). The object server must also store a record of all persistent LRs on that same persistent storage. When starting up again, an object server must be able to recreate and reload the state of all persistent LRs that it hosted before shutting down.

A persistent LR can be active or passive. An **active persistent LR** (or simply active LR) has both an active and a passive component. The **active component** is an instance of the LR loaded into the object server's address space. An active component contains the LR's state. Methods invoked on the LR modify that state. The **passive component**, on the other hand, consists of the LR's state as stored on persistent storage. This passive state is usually not directly modified as a result of method invocations on the active component. A **passive persistent LR** (or simply passive LR) has only a passive component. A passive LR can be **activated** to become an active LR, likewise, an active LR can be **passivated** to become a passive LR. A transient LR only ever has an active component, it cannot be passivated. Figure 4.7 shows an example of a transient LR, an active LR and a passive LR. The transient LR only has an active component, the active LR has both an active and a persistent component and the passive LR has only a passive component.

Persistent LRs, their resources, and the process of passivation and activation are managed by the **persistence manager**. The persistence manager has two main responsibilities. The first is the management of persistent LRs, which includes passivating and activating LRs, keeping track of persistent LRs, and cleaning up unused persistent LRs. The second responsibility is the management of persistent resources. The persistence manager makes persistent resources available to persistent LRs and manages the allocation and clean up of persistent resources. Persistent resources, like persistent LRs, are resources that remain allocated and available across restarts of the allocating process. Persistent resources offered by the object server include persistent storage and contact points. Because the administrative information about persistent LRs and resources must survive process restarts, the persistence manager must itself be a persistent object. A third responsibility of the persistence manager is, therefore, the management of its own persistent state.

The persistence manager keeps track of persistent LRs by assigning each one a **persistence identifier** (PID). The PID is used as a reference to a persistent LR. A PID can, however, also be used to refer to a persistent resource. A PID remains valid across process restarts. The persistence manager is responsible for mapping PIDs to their related LRs or resources.

Information about active LRs is stored by the persistence manager in a table of **activation records**. An activation record stores information about a single active LR. It includes: a PID, a reference to the LR's active component, a reference to the LRs passive component (in the form of the PID of its persistent storage resource), and an implementation handle and corresponding initialization data, which can be used to recreate the active component.

When the persistence manager is passivated it stores all the activation records on persistent storage. All the activation record fields, except for the reference to the LR's active

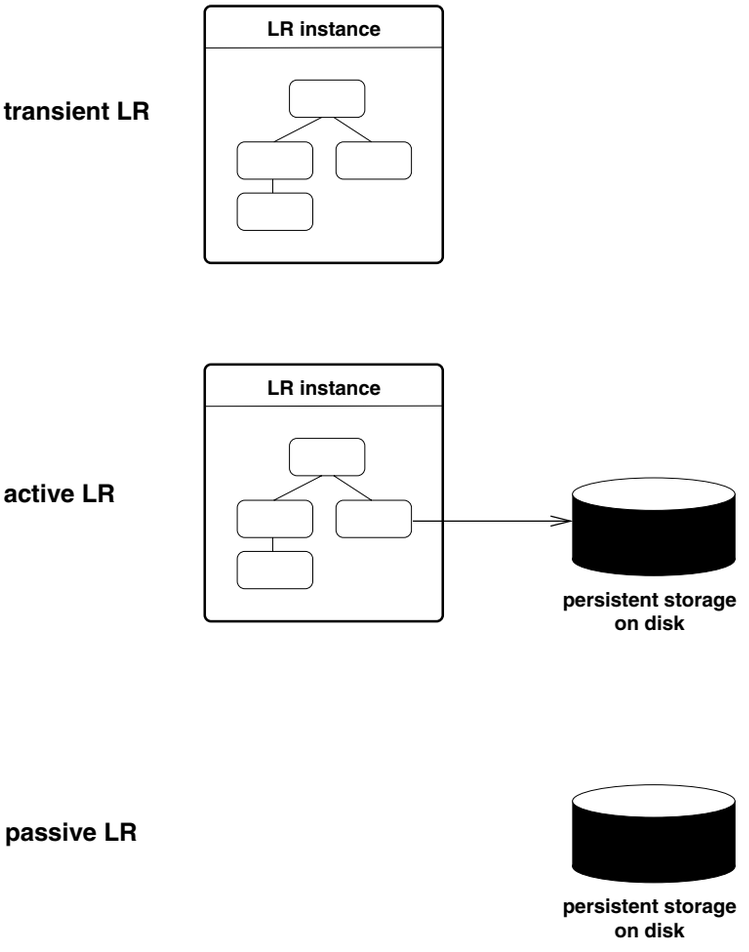


Figure 4.7: A transient LR, an active persistent LR, and a passive persistent LR.

component, are stored. Because an active component is not persistent, the reference to the LR's active component is not persistent and does not need to be stored.

Passivation and object server shutdown. When an object server shuts down, all persistent LRs must be passivated. As mentioned above, this occurs during regular object server shutdown when all installed LRs are released. Whereas releasing a transient LR is equivalent to destroying it, for a persistent LR this results in passivation. Passivation is coordinated by the persistence manager and results in the updating and saving of the passive component to persistent storage and the destruction of the active component.

The steps involved in the passivation process are outlined in Figure 4.8.

```
in: LR reference, LR activation record

1. Store implementation handle and init data in activation record.

2. Save LR state to persistent storage.
2.a Tell replication subobject to save state.
2.b Replication subobject gets state from semantics subobject.
2.c Replication subobject writes state to persistent storage.

3. Remove active component reference from activation record.
```

Figure 4.8: Steps involved in the passivation of an active LR.

In the first step the persistence manager acquires the LR's implementation handle and initialization data from the LR and stores these in the appropriate fields of the LR's activation record. Next, the LR is given the opportunity to save its state to persistent storage. In a GlobeDoc LR the only subobjects that need to save state are the replication and semantics subobjects. The semantics subobject does not have direct access to persistent storage (it does, however, have access to secondary storage through a local storage object as described in Section 4.1.4). As such, its passivation is regulated by the replication object. The replication subobject is told to passivate its state. As part of its passivation process the replication subobject requests a copy of the semantics subobject's state and saves it to persistent storage. The replication subobject then saves its own state (if any) to the same persistent storage and returns control to the LRManager.

After the LR saves all its state to persistent storage, the persistence manager removes the reference to the active component from the LR's activation record. If no other references to that active component exist, then the garbage collector will cause it to be cleaned up and be destroyed. If, however, other references to the active component exist, it might take a while before that component is actually cleaned up and destroyed. Because the LR has already been passivated, the active component's state is no longer valid, and may no longer be accessed. This means that after passivation, an LR may not allow invocations of any methods that access its state.

Activation and object server startup. When the object server is started, one of the first things it does is start up the persistence manager. As part of its startup process, the persistence manager reads in its own passive state and rebuilds its table of activation records. When the table has been rebuilt the persistence manager iterates through all the activation records and activates each of the associated LRs.

Activating an LR involves recreating the active component using the stored implementation handle and initialization data as described earlier. In the last steps, however, instead of initializing the object using the distributed interface the LR is told to restore its state from the given persistent storage. Once the LR has been activated, the persistence manager stores a reference to it in the appropriate activation record.

Persistent storage. Persistent state is stored on persistent storage. Persistent storage can be a local file system, a remote file system, flash ROM, anything that can reasonably be expected to provide the same contents the next time the object server is restarted. Persistent storage is accessed through **storage objects** which are Globe system objects that allow access to storage through methods defined in the storage and persistentObject interfaces. The storage interface defines a file-like interface that allows data to be written to and read from storage. This interface is described later in Section 4.1.4. The getPersistenceID method in the persistentObject interface (shown in Table 4.3) returns the PID associated with a given persistent storage object. Every persistent storage object is associated with one PID.

interface	persistentObject	
method	getPersistenceID	Returns the PID associated with the persistent object that implements this interface
returns		The PID

Table 4.3: The persistentObject interface.

The persistence manager is responsible for creating and managing persistent storage objects. The persistence manager implements the storageManager and perstResourceManager interfaces. The storageManager interface allows storage objects to be allocated and deallocated (the interface is presented in more detail in Section 4.1.4). Resources allocated through this interface must explicitly be flagged as persistent by invoking the makePersistent method in the perstResourceManager interfaces shown in Table 4.4. This method causes a PID to be assigned to an existing storage object.

The persistence manager stores a mapping of storage object PIDs to actual storage resources (such as files) in an internal table. This table is part of the persistence manager's persistent state and is stored on persistent storage when the persistence manager is passivated. When the persistence manager is activated it rebuilds the table and recreates storage objects for the data stored on persistent storage.

interface	perstResourceManager	
method	makePersistent	Assigns a PID to an existing storage object
in	allocator	The PID of the allocating persistent object
in	res	The storage object to flag as persistent

Table 4.4: The perstResourceManager interface.

Persistent Contact Points. Besides persistent storage, persistent LRs also require **persistent contact points**. Like persistent storage, a persistent contact point remains available across restarts of the object server. Besides simply remaining available, a persistent contact point also remains allocated to the same persistent LR across object server restarts. Like regular (transient) contact points persistent contact points are also represented by contact addresses. A persistent contact point's contact address must remain valid (i.e., it must refer to the same contact point) throughout that contact point's lifetime. This means that once such a contact address has been registered with the location service, that registration does not have to be modified until the persistent contact point is explicitly destroyed. Because of this, passivated LRs do not have to unregister their contact address registrations. Those registrations will still be valid when the LR is activated again. Note, however, that the location service does not know whether a contact address refers to a passivated LR or not. It is up to a binding client to determine whether a given contact address is usable or not.

Persistent contact points are implemented using regular transient contact points. When a persistent contact point is created it is assigned a transient contact point (e.g., a TCP socket). The persistent contact point is guaranteed to always have the same transient contact point assigned to it. For example, if a persistent contact point is assigned a transient contact point with TCP/IP address 130.37.16.40:25005 then every time that its hosting process is restarted it is guaranteed to be assigned a transient contact point with the same address. This address will be contained in the persistent contact point's contact address.

If every persistent contact point were assigned a separate transient contact point, then acquiring each of these transient contact points every time the object server started up would be a daunting task. Due to the way that most operating systems assign and allocate network addresses (especially port numbers) this would be next to impossible if other networked processes were started in between successive runs of the object server. The solution to this problem lies in assigning a single multiplexed contact point for all persistent contact points. Multiplexed contact points are explained in Section 4.1.5.

Note that persistent LRs do not have persistent connections to other LRs or clients. When a persistent LR is passivated it breaks all its network connections. Depending on its replication policy, when a persistent LR is activated again it might attempt to reestablish some or all of those connections.

4.1.3 Globe Runtime Services Component

The Globe runtime services component implements the Globe runtime system which provides services needed by any Globe-enabled processes (i.e., any process that can bind to and invoke methods on Globe objects). The Globe runtime system provides the following components.

- The **binder object**, which allows processes to bind to DSOs and implements the binding process as described in Chapter 3.
- The **implementation repository**, which provides access to LR implementations. The implementation repository is responsible for loading implementations into a process's address space. It is a service that can be implemented locally, as a remote service, or some combination of both. The implementation repository will be described in detail in Section 4.2.
- The **local name space** (LNS), which provides a Globe-enabled process with a directory of local objects, services and resources. The LNS is also used by LRs to keep track of and allow communication between their subobjects.
- **Remote service resolvers**, which provide local access to remote services such as the naming and location services.

Binder Object

The binder object is a local object, provided by the Globe runtime system, that coordinates the process of binding to Globe DSOs. The binding procedure, as described earlier, consists of five stages. In the first stage, name resolution, an object name is resolved to an object handle. In the next stage, location lookup (also known as object handle resolution), the object handle is resolved to a set of contact addresses. After this comes the address selection stage, which involves selecting one contact address from the set acquired in the previous step. The fourth stage is instance creation where, given the contact address selected in the previous stage, a local representative of the DSO is created and initialized. Finally, in the last stage, instance initialization, the initialization of the LR created in the previous stage is completed using data, acquired in the previous stages, such as the name and object handle of the Globe object being bound to.

Generally the binding procedure starts with the name resolution stage and proceeds until the instance initialization stage. In some cases, however, it may be possible to start the procedure in a later stage, such as when the requesting process already has an object handle, or has a specific contact address that it must bind to. In these cases the binding procedure would be started from the second and third stages respectively. Likewise, it might be necessary for the binding procedure to stop before reaching the final stage. For example, given an object name a process might require an associated contact address, but not want to actually bind to the associated DSO. In this case binding would start at the first stage and stop after completing the third stage.

interface	binder	
	Methods to get intermediate results.	
method	getObjectName	Retrieves the DSO name
returns		The DSO name
method	getObjectHandle	Retrieves the object handle
returns		The object handle
method	getContactAddressSet	Retrieves the current (nonempty) set of contact addresses
returns		The contact addresses
method	getContactAddress	Retrieves the currently selected contact address
returns		The contact address
method	getInstance	Retrieves the current LR
returns		The LR
	Methods to set and get binding parameters.	
method	setNumOHandleResolutions	Sets the number of object handle resolution retires
in	nres	The number of retries
method	getNumOHandleResolutions	Retrieves the number of object handle resolution retires
returns		The number of retries
method	setNumSelections	Sets number of contact addresses that may be tried by repeated contact address selections
in	nsels	The number of contact addresses tried
method	getNumSelections	Retrieves the number of contact addresses to try
returns		The number of contact addresses to try
method	setLookupProperties	Sets criteria used to look up contact addresses
in	sel	The lookup criteria
method	getLookupProperties	Retrieves contact address lookup criteria
returns		The lookup criteria
method	setPersistence	Sets whether LR should be persistent or not
in	allocator	PID of the allocating persistent object
method	getPersistence	Retrieves the value set by setPersistence
returns		The value set by setPersistence
	Methods to initiate the binding process	
method	bind	Starts binding from the name resolution stage
in	oname	The name of the DSO to bind to
method	bindOHandle	Starts binding from the object handle resolution stage
in	ohandle	The object handle of the DSO to bind to
method	bindCAddr	Starts binding from the contact address resolution stage
in	ohandle	The object handle of the DSO to bind to
in	caddr	The contact address to bind to

Table 4.5: The binder interface.

The binder object implements the binder interface as shown in Table 4.5. This interface provides methods that allow binding to be started and stopped at any stage.

The first set of methods (getObjectName, getObjectHandle, getContactAddressSet, getContactAddress, and getInstance) return the intermediate results from the various binding stages.

The second set of methods sets parameters for the various binding stages. The setNumOHandleResolutions method sets the number of times that the object handle resolution stage may be repeated before failing (which is explained below). The setNumSelections methods sets the number of times that the address selection stage may be repeated before failing. Criteria used by the location service to look up contact addresses is set by the setLookupProperties method. The setPersistence method sets a parameter which determines whether the resulting LR should be created as a persistent or transient object. The corresponding get methods return the value of their respective parameters.

Finally the bind methods initiate the actual binding procedure. The bind method starts the procedure from stage one and is given the name of the Globe object to bind to. Invoking bindOHandle starts binding from the second stage. The method takes the object handle of the Globe object to bind to as a parameter. Invoking bindCAddr starts the procedure from stage three and is given a contact address to bind to. It is also given an object handle to assign to the LR once it has been created and initialized.

Generally the stages are processed in the order given above. Sometimes, however, errors or failures may cause some stages to be retried. The address selection stage is retried after an error in the instance creation or initialization stages. Retrying address selection involves choosing a different contact address and continuing the process using that address. The object handle resolution stage is retried if the address resolution stage runs out of contact addresses to consider without successfully having bound to the Globe object. Retrying this stage involves broadening the scope of the location service lookup to retrieve more contact addresses.

Note that while the binder coordinates the binding process it does not perform all the steps of the process itself. Name resolution, for example, is performed by the name service resolver and object handle resolution is performed by the location service resolver. LR creation is performed by a number of components as described previously.

Local Name Space

The local name space (LNS) [47][106] provides a Globe-enabled process with a directory of local objects (including LRs and their subobjects), services, and resources. It is implemented as a tree of contexts where a context represents a local object in the LNS. A context is associated with exactly one local object. A context can have one or more subcontexts, in which case it is called the parent of those subcontexts.

A context names its subcontexts by defining a mapping of labels to its subcontexts. A label is simply a string that names a context. A context can be referred to using a path name, which is a concatenation of multiple labels separated by slashes (“/”) and represents the path to that context relative to some starting context. A path name may contain special labels such as “.”, “..” and “:” which refer to the current context, the parent context, and

the root context, respectively. A path name, such as :a/b/c, which starts at the root context is called an absolute path name. Other path names, such as b/c, are called relative path names.

Besides referring to a local object, an LNS context can also be empty, a symbolic link, or a delegation. An empty context is one that does not refer to any local object. Generally empty contexts are used for internal nodes in the name space tree. A symbolic link in the LNS is similar to a symbolic link in Unix, it is a context that refers to another context by name. A symbolic link consists of a source context and a target name. The source context is the symbolic link context's name while the target name is a path name that refers to the target context.

A delegation is a context under which an external name space has been placed. An external name space may, for example, be another LNS name space but may also be the name space of a local or remote file system. A delegation is similar to a mount point in Unix file systems. A delegation context implements a delegation interface which provides access to the external name space. The delegation context is responsible for resolving path names that fall within its name space.

A path name can be resolved to acquire a reference to the context that it refers to. Resolution of path names is similar to the resolution of file system path names in, for example, Unix. It involves repeatedly resolving each of a path name's components until the path name's final component has been resolved. A single path name component is resolved by consulting the label mappings of the current context until a table corresponding to the given path name component is found. Resolving a path name component results in a reference to the corresponding context. Resolution of absolute path names always starts at the root context. Resolution of relative path names always starts at some context that is considered to be the current context. In every step of path name resolution the current context is set to the most recently resolved context.

As an example of path name resolution, given the name space shown in Figure 4.9, resolution of the path name :a/b/c would start by consulting the root context's mapping and looking for the label "a." Then, given the context for "a," resolution continues by consulting this new context to find a mapping for the label "b." Finally, given the context for "b," its mappings are consulted to find label "c." The resulting context is the final context and represents the context referred to by the path name :a/b/c.

The Globe runtime system implements the LNS as a name server object. The LNS name server object implements the name server interface, which provides methods that allow the creation of contexts, symbolic links, and delegations. It also allows objects to be inserted into the name space at given contexts and allows pathnames to be resolved. Finally, it allows objects to be unregistered and contexts to be removed.

Access to Remote Services

In order to find and bind to objects Globe relies on two important external services, the naming and location services. Access to these services is provided through the runtime system's remote service resolvers. The resolvers are local objects that act as stubs to the

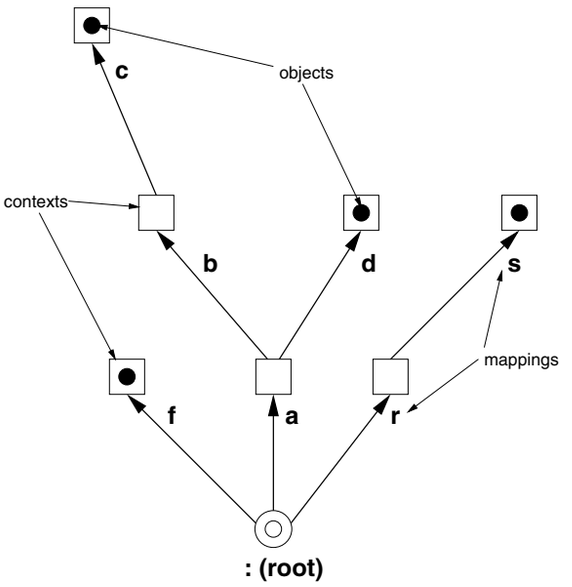


Figure 4.9: Example LNS name space.

remote services, that is, they export and implement interfaces that mirror the functionality of the services they serve.

Another important remote service is the Globe Infrastructure Directory Service (GIDS). GIDS is a service that keeps track of all infrastructure elements (e.g., object servers, GlobeDoc gateways, etc.) for a Globe environment. Like the name and location services GIDS is also accessed through a local resolver object. GIDS and its resolver object are discussed in detail in Chapter 5.

Generally, the Globe runtime system will make a single instance of each resolver available and these will be shared by all entities using them. The resolver objects are made available under well-known names (i.e., they are given previously determined names) in the LNS so that they can be easily found. There are two reasons for providing single instances of resolvers and requiring them to be shared. First, the resolvers contain state that must be shared by all entities using the same runtime system. The location service resolver, for example, provides its clients with identifiers that refer to location service registrations. These identifiers are indexes into a table kept by the resolver that keeps track of all registrations made by it. Because the identifiers can be shared between different local objects it is important that such an ID always refers to the right registration. If different local objects used different location service resolvers, this would not necessarily be the case. The second reason is that using the resolvers requires communication with external services. Creating and initializing a resolver is thus an expensive process. It is more efficient for both the client and the runtime system to create the resolver once and share the resulting object.

4.1.4 Local Storage Management Component

The local storage management component provides access to the locally accessible secondary storage of the machine that the object server runs on. Access to local storage is required by persistent LRs in order to be able to store their state. Access to local storage may also be required by LRs that directly store some or all of their state on secondary storage. Generally, the semantics subobjects are the ones that directly store state on secondary storage. GlobeDoc is an example of a Globe object whose LRs store their state on local storage. As mentioned previously, because a GlobeDoc object may contain many (possibly large) elements, and because an object server may host many such GlobeDoc objects, it quickly becomes infeasible for a GlobeDoc object to store all of its state in main memory. In order to prevent GlobeDoc objects from using up all of an object server's memory resources, the GlobeDoc semantics subobject stores elements on local storage.

An LR can access secondary storage through a local storage object, which is a local object that provides platform-independent access to local storage resources. The storage object is used like a file and provides a file-like interface (see Table 4.6). Note that the storage interface is also implemented by persistent storage objects. The main difference between transient and persistent storage objects is that persistent storage objects also implement the persistentObject interface. Another difference is, of course, that persistent storage objects guarantee that the data they write to storage will be available across restarts of the object server. Data written through regular storage objects is not guaranteed to be available across object server restarts (because, for example, it might have been cached and not flushed out to disk).

Storage objects are allocated using a resource manager object that implements the storageManager interface (see Table 4.7).

Before writing to local storage, storage space must first be allocated by invoking the resource manager's allocateStorage method. This results in storage space being reserved and a **resource identifier** (RID) being allocated for the given space. Before using the allocated space it is necessary to acquire a storage object that can be used to write to the allocated storage space. This is done by invoking the openStorage method. Finally, when the storage space is no longer needed, it can be freed by calling deallocateStorage.

Note that, when storage is allocated for use by persistent LRs, the resource manager must be informed that the storage is persistent. This can be done only if the resource manager also implements the perstResourceManager interface. The result is that the resource manager remembers that that storage is persistent and that it cannot be removed when the object server shuts down. Generally the persistent storage is stored in a different part of the underlying file system than the regular storage (e.g., files containing persistent storage are stored in a different directory than files containing transient object server data). In this way the persistence manager can easily find all of the persistent storage when an object server is restarted.

interface	storage	
method	write	Writes to the storage represented by this object
in	packet	The data to write
method	read	Reads from the storage represented by this object
in	nbytes	The number of bytes to read
returns		The data read
method	getCurrent	Retrieves the current offset of this storage object
returns		The current offset
method	seek	Sets the current offset of this storage object
in	cur	The new offset
method	getLength	Retrieves the size of the storage represented by this storage object
returns		The size of the storage
method	setLength	Sets the size of the storage represented by this storage object
in	len	The size of the storage
method	close	Closes the storage object
method	getResourceID	Retrieves the resource ID of the storage resource represented by this object
returns		The resource ID

Table 4.6: The storage interface.

interface	storageManager	
method	allocateStorage	Allocates storage
returns		The resource ID of the allocated storage
method	deallocateStorage	Releases storage
in	rid	The resource ID of the storage to release
method	openStorage	Creates a storage object through which previously allocated storage can be accessed
in	rid	The resource ID of the storage to access
returns		A storage object through which the storage can be accessed

Table 4.7: The storageManager interface.

4.1.5 Network Management Component

The network management component provides LRs with access to the resources required for communication. An LR accesses available network resources through the standardized interfaces provided by its communication subobject. Because communication subobjects rely on local network resources they are usually supplied by the Globe runtime system. In rare cases, however, communication subobject implementations may be included with an LR implementation. This is usually the case when a custom-made communication subobject is required for debugging or measurement purposes, or when a custom communication protocol is used.

A communication subobject provides contact points and **communication endpoints**. A contact point is a resource through which an LR can be contacted. It is always associated with a network address which is referred to as a **contact point address**.[4] A contact point is always associated with a **native contact point**. A native contact point is a contact point defined by the underlying platform, for example, a TCP/IP listen socket or a UDP/IP socket. The format of a contact point's address is dependent on the protocol of the associated native contact point. An example of a TCP/IP contact point address is 130.37.16.40:25005. A connection-oriented contact point receives connection requests and creates a connection to the sender. The connection is then used as a communication channel. Note that a connection-oriented contact point cannot be used for general communication. Unlike a connection-oriented contact point, however, a connectionless contact point can be used for general communication.

A communication endpoint is a resource through which LRs can communicate with each other. In order for two LRs to communicate, they must both have access to a communication endpoint. In connection-oriented communication a communication endpoint can be used only if it is associated with an existing connection. In connectionless communication a communication endpoint is equivalent to a contact point and is not associated to any sort of connection. Like a contact point, a communication endpoint is always associated with a **native communication endpoint**. A native communication endpoint is a communication endpoint that is defined by the underlying platform, for example, a TCP/IP connection socket or a UDP/IP socket.

There are four types of communication subobjects:

- Connectionless point-to-point communication subobjects provide connectionless communication between two parties.
- Connection-oriented point-to-point communication subobjects provide connection-oriented communication between two parties.
- Connectionless group communication subobjects provide connectionless communication between multiple parties.
- Connection-oriented group communication subobjects provide connection-oriented communication between multiple parties.

[4] Note, that a contact point address is not the same as a contact address.

Group communication has not been implemented yet and will not be described further.

Although it is referred to as a subobject the communication subobject is not a single object but consists of a number of separate Globe local objects. These local objects combine to provide the interfaces and functionality exported by the communication subobject. Looked at this way, it may seem more appropriate to refer to the communication subobject as a subsystem rather than a subobject. However, for consistency with the naming of the other (semantics, control, and replication) subobjects we continue to use the term communication subobject.

The local objects comprising the communication subobject are called **communication objects** or **communication components**.[5] The interfaces exported by a communication subobject depend on the interfaces exported by its constituent communication objects. At the very least, every communication object implements the standard communication interface. This interface provides methods that return information about the particular communication object, such as the protocol stack it implements, and a list of other communication interfaces that it exports.

A connectionless point-to-point communication subobject consists of a **connectionless communication object**. This is an object that is capable of sending and receiving messages without having to establish a connection. It defines a contact point to which messages can be sent by other connectionless communication objects. Besides the standard communication interface, this object also implements the contactExporter and msg interfaces. The contactExporter interface is used to export the contact address of a contact point implemented by the communication object. The msg interface is used to send and receive messages from other connectionless communication objects.

A connection-oriented point-to-point communication subobject consists of connector objects, listener objects and connection objects. A **connector object** is a communication object capable of initiating connections to listener objects. Besides the standard communication interface, the connector object also implements the connector interface. This interface provides methods for sending connection requests to remote listener objects. When a connection request is honored by a remote listener object, the connector object returns a connection object corresponding to the newly created connection.

A **listener object** is a communication object capable of accepting and honoring connection requests from remote connector objects. Besides the standard communication interface, a listener object implements the listener and contactExporter interfaces. This listener interface provides methods for accepting connection requests from remote connector objects. When a connection request is accepted, the listener object returns a connection object corresponding to the newly created connection. The listener represents a connection-oriented contact point.

A **connection object** represents a communication endpoint. It is created by a listener or connector and represents an accepted or initiated connection. Besides the standard communication interface, it implements the connection and msg interfaces. The connection interface provides methods that return information about the connection (e.g., the asso-

[5]Note that for the rest of this section communication subobject and communication object are not interchangeable terms.

ciated remote and local addresses). The connection interface can also be used to close a connection.

In a simple implementation of a communication subobject each exported contact point is implemented as a single native contact point. Similarly each communication endpoint is implemented as a single native communication endpoint. More complex implementations may associate multiple contact points to a single native contact point, or multiplex multiple communication endpoints over a single native communication endpoint.

Multiplexed Contact Points

Multiplexing of contact points involves implementing multiple contact points using a single native contact point. There are two reasons for introducing **multiplexed contact points**. First, it helps to reduce the number of native contact points that a process must allocate. Second, it provides a base for the implementation of persistent contact points. In the first case, the problem is that a process (such as an object server) is limited in the number of native contact points that it can allocate. If such a process hosts many LRs (each of which requires one or more contact points), it may run out of available native contact points. This places a limit on the number of LRs that can be hosted at any one time. By multiplexing contact points a process can reduce the number of native contact points that it allocates and thus increase the number of LRs it can host. In the second case, as described earlier, the object server process allocates one or more native contact points and multiplexes persistent contact points on top of these. The object server is responsible for reallocating these same native contact points whenever it is restarted. By reducing the number of native contact points that must be reallocated, there is less chance that these will be taken by another application.

Both connection-oriented and connectionless contact points can be multiplexed. Figure 4.10 shows the objects involved in multiplexing a connectionless contact point. Recall that in connectionless communication, contact points are equivalent to communication endpoints. At the very bottom of the figure is a native connectionless contact point, in this case a UDP/IP socket. The address of this native contact point is its UDP/IP address, for example: 130.37.16.40:25003. The native contact point is wrapped in a connectionless communication object, which defines a single contact point.

The next object (from the bottom up) is a shared multiplexer object. The task of a multiplexer object is to define logical contact points, which are mapped onto the single contact point below it. Such a logical contact point is called a **multiplex port**. The address of a multiplex port consists of a multiplex port number and the underlying contact point's address. Thus, for example, the address for multiplex port 9000 would be: 130.37.16.40:25003:9000.

Above the multiplexer object are **light-weight communication objects**. These communication objects present a regular connectionless communication object interface to their users. Their task is to hide any multiplexing details from their users. Each light-weight communication object defines a contact point that corresponds to a multiplex port. The address of the light-weight contact point is the same as the address of the multiplex port that it uses, e.g., 130.37.16.40:25003:9000.

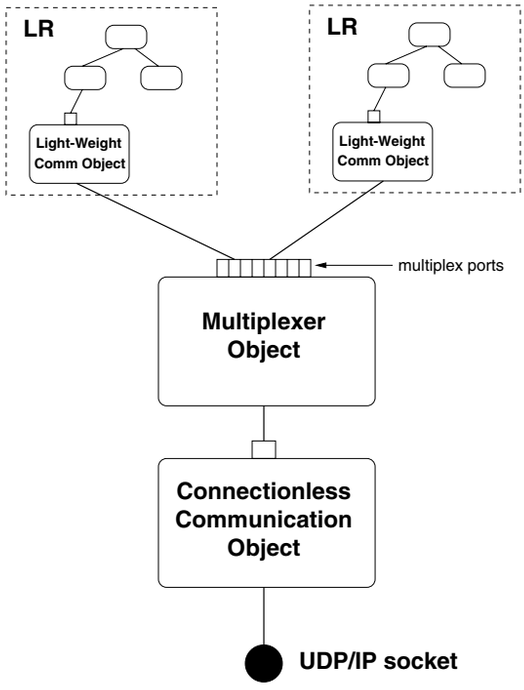


Figure 4.10: Objects involved in multiplexing a connectionless contact point.

Figure 4.11 shows an example of the transmission of a message using a UDP/IP multiplexer. In this example the sender wants to send a message to multiplex port 9000 at address 130.37.16.40:25003. To do this, the sender first invokes the send method (passing the message and destination address) on its light-weight communication object. As a result, the light-weight communication object invokes the multiplexer's send method, passing along the message, destination address, and its own multiplex port number (so the receiver knows who sent the message). Next, the multiplexer creates a new message containing the original message, the receiver's multiplex port, and the senders multiplex port and passes this on to the underlying communication object. This communication object then passes the message on to the native contact point where it is sent as a UDP datagram to the receiver. When the native contact point on the receiver end receives the message, it passes it on to its associated communication object where the message is passed up to the multiplexer. The multiplexer then extracts the receiver's multiplex port from the message and passes the rest of the message on to the light-weight communication object associated with that multiplex port. Finally, the light-weight communication object passes the message on to its user.

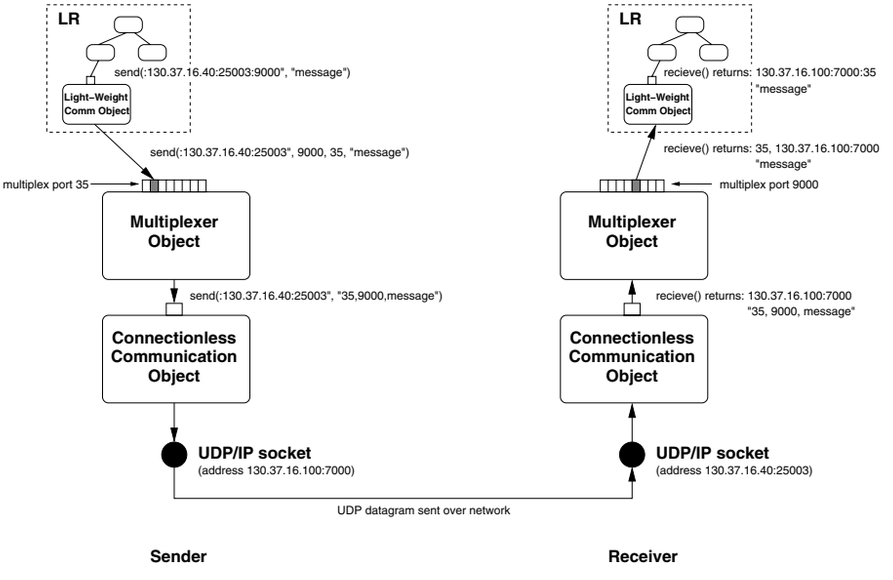


Figure 4.11: Example of transmitting a message using a UDP/IP multiplexer.

Figure 4.12 shows the objects involved in multiplexing a connection-oriented contact point. As in the connectionless case, at the bottom there is a native contact point, in this case a TCP/IP listen socket. The address of this contact point is its TCP/IP address, for example: 130.37.16.40:2500. This native contact point is wrapped in a listener object. Above this object is a **multiplexing listener** object. As in the connectionless case, the multiplexer defines local multiplex ports and maps these onto the listener object (and

native contact point) below it. The address of a multiplexed port consists of the multiplex port number and the underlying native contact point's address. Thus, for example, given a native contact point with address 130.37.16.40:2500, the address for multiplex port 7000 would be: 130.37.16.40:2500:7000

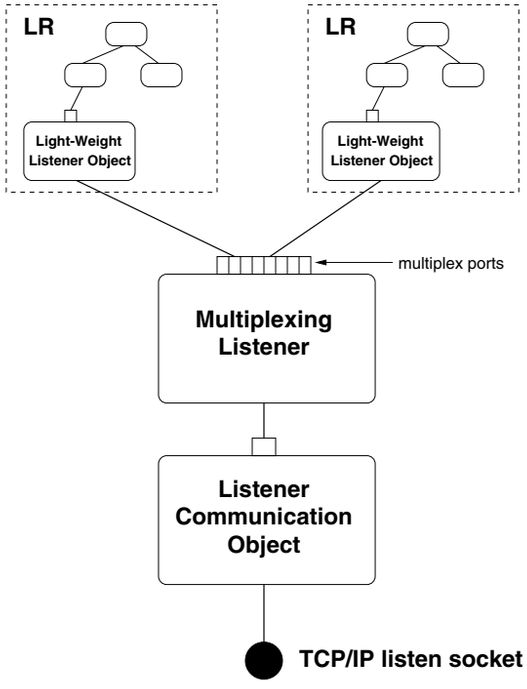


Figure 4.12: Objects involved in multiplexing a connection oriented contact point.

Above the multiplexing listener object are **light-weight listener** objects. A light-weight listener object hides the multiplexing details from its user by presenting a standard listener interface. Each light-weight listener object defines a single **light-weight contact point** that corresponds to a multiplex port managed by the multiplexing listener. The address of a light-weight contact point is the address of the light-weight listener's associated multiplex port.

To connect to a particular light-weight contact point a client must use multiplex-aware connector objects as shown in Figure 4.13. There are three such objects, corresponding to the three kinds of listener object, a **light-weight connector**, a **multiplexing connector**, and a native connector. At the bottom of the figure the native connector is encapsulated in a connector object. In this example the native connector is a TCP/IP socket. In the middle there is the multiplexing connector object that is responsible for initiating connections to multiplexing listeners. At the top the light-weight connector hides multiplexing details from its user.

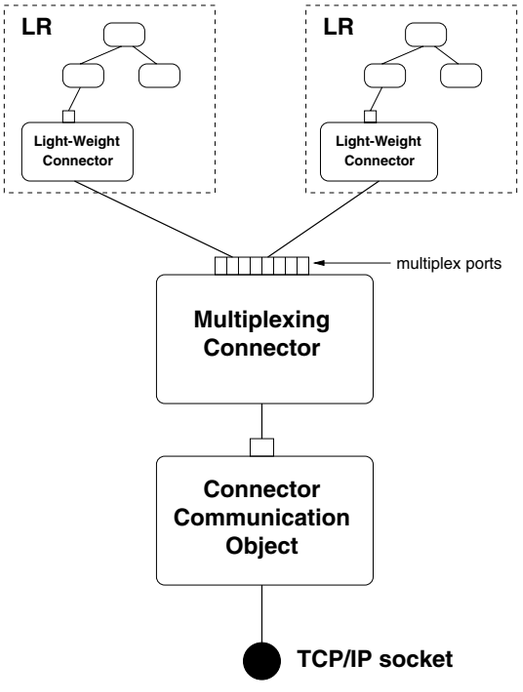


Figure 4.13: Objects involved in multiplexing a connection oriented connector.

The process of connecting to a lightweight contact point is illustrated by the example in Figure 4.14. In this example, the client LR invokes a connect method on its light-weight connector, passing it a destination address consisting of a TCP/IP address and a multiplex port (step 1). The light-weight connector extracts the multiplex port number and TCP/IP address from the destination address, and passes these as arguments to the multiplexing connector's connect method (step 2). The multiplexing connector calls the underlying connector object's connect method to establish a TCP/IP connection with the destination's native TCP/IP listener (step 3). When the listener receives the connection request it creates a new connection object (and new native communication endpoint) that is associated to a new connection object (created by the connector object) on the sender's side (step 4a). A reference to the new connection object is returned to the multiplexing connector on the sender side and the multiplexing listener on the receiver side (step 4b). The multiplexing connector object (on the sender side) now sends the destination multiplex port number over this connection (step 5a). At the same time it also passes a reference to the newly created connection object to the light-weight communication object that initiated the connect request (step 5b). When the multiplex port number arrives on the receiver's side it is passed up to the multiplexing listener (step 6). The multiplexing listener then contacts the light-weight listener associated with this multiplex port number and passes it a reference to the newly created connection object (step 7). The light-weight communication objects can now use the connection objects to communicate with each other. Note, that in this example the communication endpoints are not multiplexed, only the contact points are multiplexed.

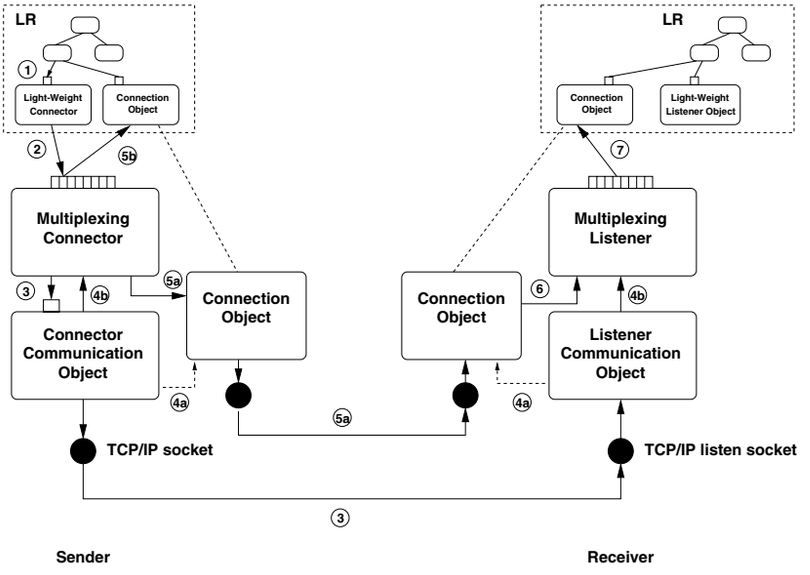


Figure 4.14: Example of connecting to a connection oriented multiplexed contact point.

Multiplexed Communication Endpoints

For the sake of completeness we briefly mention multiplexed communication endpoints. Note however, that neither their design nor their implementation form a part of the GlobeDoc architecture.

As in the case of multiplexed contact points, multiplexing communication endpoints involves creating multiple communication endpoints that make use of a single native communication endpoint. Multiplexing of communication endpoints is useful in object servers that contain many LRs that create connections to peers on a small set of other object servers. In the nonmultiplexed case, this would cause each object server to allocate a large number of native communication points. If the number of hosted LRs is great, then the underlying platform may run out of native communication endpoints. Multiplexing of the communication endpoints greatly reduces the number of native communication endpoints allocated by the object server. Note, that multiplexing of communication endpoints is necessary only for connection-oriented communication. In the case of connectionless communication, multiplexing of communication endpoints is equivalent to multiplexing of contact points.

Communication-Object Manager

The **communication-object manager** is a Globe runtime system object that allows multiplexer objects (i.e., multiplexing listener, multiplexing connector, and connectionless multiplexer object) to be shared by LRs. The communication-object manager is installed under a well-known name in the LNS so that all light-weight communication objects can easily find it. When a light-weight communication object requires access to a multiplexer object it requests a reference to one from the communication-object manager. The communication-object manager keeps track of existing multiplexer objects in an internal table. Each entry in this table represents a multiplexer object and includes information about the native contact point used, the protocol stack implemented and a reference to the actual multiplexer object. If the communication-object manager cannot find an appropriate multiplexer object, such an object must be created by the light-weight communication object requesting it. Once created the new multiplexer object is registered with the communication-object manager. The communication-object manager allows multiplexer objects to be looked up based on their native contact point address, the protocol stack they support, or both.

Persistent Contact Points

Persistent contact points are implemented as multiplexed contact points and are therefore accessed through light-weight communication objects. In order to prevent other (non-persistent) multiplexed contact points from using a persistent contact point's multiplexer object and native contact point, the persistence manager may request the communication-object manager to reserve all addresses that may be used by persistent contact points. Reserving a contact point results in the creation of a native contact point for the reserved

address. It also results in the creation of a multiplexer object associated with that native contact point. A multiplexer object used for persistent contact points may also be requested to reserve multiplex ports corresponding to those used by the persistent contact points.

4.2 Implementation Repository

The **implementation repository** stores and provides access to LR implementations. It is accessible as a local service through the Globe runtime system and provides an interface with methods that map contact addresses and implementation handles to corresponding implementations.

The implementation repository is generally used during the binding process after the binder has selected an appropriate contact address. It is given the selected contact address and maps it to the implementation of an LR capable of connecting to the given contact address. Besides simply finding the appropriate implementation, the implementation repository also takes care of loading the code into an address space. It does this using platform specific **class loaders** — runtime objects capable of loading implementations into an address space.

Although it is always accessed locally, an implementation repository can be a local, remote, or distributed service. An example of a local repository is one where the implementations are all stored on a local file system. A remote repository, on the other hand, might be an FTP archive containing implementations. A distributed repository is one where the implementations can be distributed (and possibly replicated) over many sites.

4.2.1 Class Archive

No matter how the repository is implemented, the code in an implementation repository is always stored in class archives. A **class archive** is a container that stores all the code that makes up the implementation of a single LR. The class archive is also the container in which implementations can be transferred between processes (e.g., when downloaded from a remote repository).

In the Globe runtime system a class archive is represented by a local object, the **class archive object**. This local object provides methods that allow code to be added to, and retrieved from, a class archive. Besides this object representation a class archive may also have an external representation, that is, a representation that is accessible from outside the Globe runtime system. A common external representation of a class archive is a single archive file that contains all the individual platform-specific implementation files. On the Java platform, for example, this could be a Java archive (jar) file containing Java class files. On a Unix platform this may be a library file containing the appropriate (C or C++) object code files. Such class archives can be accessed directly (by accessing the file outside of the Globe environment) or through their associated class archive objects. In this case the class archive object is simply a wrapper that performs operations on the associated archive file. A class archive may, however, be implemented as a class archive

object, with no external representation. In this case, the class archive object would store all the implementation code in main memory and could not be accessed from outside the Globe runtime system.

Besides providing access to implementation code, a class archive object also provides access to a class loader capable of loading the code contained in the class archive. For example, a Java jar format class archive must provide access to a class loader that can open a jar file, extract the appropriate Java class files and load them into the virtual machine. Class loaders and the class loading process will be examined in more detail in Section 4.2.3.

Along with implementation code, a class archive also contains an **implementation catalog**. The catalog contains information needed by a class loader to load and create an LR. It describes the subobjects that make up the LR and where (in the class archive) their implementations can be found. The catalog contains one record per subobject, with each record containing fields that specify the subobject name, the location of the implementation code (e.g., a Java class file name, a C object file name, etc.), the role the subobject plays in the LR (i.e., control, semantics, replication, communication, LRManager, etc.) and subobject-specific initialization parameters. An example of a part of an implementation catalog is shown in Figure 4.15.

```
ObjectName: vu.Globe.GlobeDoc.GlobeDocumentImp.gdControl
ClassObject: vu.Globe.GlobeDoc.GlobeDocumentImp.gdControlClassObject
FileName: /vu/Globe/GlobeDoc/GlobeDocumentImp/gdControlClassObject.class
Position: control

ObjectName: vu.Globe.Runtime.lr.replication.replServer
ClassObject: vu.Globe.Runtime.lr.replication.replServerClassObject
FileName: /vu/Globe/Runtime/lr/replication/replServerClassObject.class
Position: replication
ReplicationPolicy: Client-Server
ReplicationRole: Server
```

Figure 4.15: Fragment of a catalog file.

This fragment shows two records. The first specifies a control subobject and the second a replication subobject. Each record specifies the name of the Java class file containing the subobject's implementation code, as well as the Java object names of the corresponding Globe class object and Globe local object. The second record includes some extra information about the replication subobject such as the replication policy it implements and which role it plays in that policy.

The interfaces implemented by class archive objects are presented in Table 4.8 and Table 4.9.

ClassArchEntry is an interface for accessing the information contained in a class archive's implementation catalog. A classArchEntry describes a single record in the class archive (i.e., a subobject implementation). A record in the implementation catalog is presented as a set of attribute-value pairs, and the value of a particular field can be retrieved

interface	classArchEntry	
method	getValue	Retrieves the value of a particular field
in	fieldName	The field name
returns		The value
method	getName	Retrieves this entry's name
returns		The name
method	getAllFieldNames	Retrieves a list of all fields in this entry
returns		The fields in the entry

Table 4.8: The classArchEntry interface.

interface	classArch	
method	getEntries	Retrieves a list of the entries contained in this archive
returns		A list of classArchEntry objects
method	getMainEntry	Retrieves the archive's main entry
returns		The main entry
method	addEntry	Adds an entry to the archive
in	name	The name of the entry to add
in	ent	The entry data
method	deleteEntry	Removes an entry from the archive
in	name	The name of the entry to delete
method	modifyEntry	Modifies an entry in the archive
in	name	The name of the entry to modify
in	ent	The entry data
method	getClassLoader	Retrieves a reference to a class loader that can load this class archive
returns		A reference to a class loader

Table 4.9: The classArch interface.

by passing the field name to the getValue method. The getName method returns the subobject's name, while the getAllFieldNames method returns a list of all the fields in a record.

A class archive object implements the classArch interface. The getEntries method reads the information in the catalog and returns it as an array of classArchEntrys. An entry representing the whole LR (the LRManager subobject) can be retrieved by invoking the getMainEntry method. The addEntry, deleteEntry, and modifyEntry methods are used to manipulate the contents of a class archive. A reference to a class loader suitable for loading the implementations contained in this class archive is returned by the getClassLoader method.

An important point of discussion relating to class archives is whether all the code needed by an LR (that is the code of all its subobjects) must be contained in a single class archive or whether it may be spread out over multiple archives. The issues involved are similar to those encountered when comparing static and dynamic code libraries on systems such as Unix and Windows.

Monolithic class archives, those that contain all the necessary code in a single class archive, have the advantage that they give the object creator more control over exactly which code will be used in an LR implementation. This means that the object creator can test all components to make sure they work and perform as expected, and can prevent undesirable combinations of subobjects. Similarly, all required code is in a single class archive, so there is no possibility of missing code leading to the loading of a partially complete implementation.

A drawback of monolithic class archives, however, is that the resulting class archives are much bigger than necessary. For example, rather than sharing a single class archive, all Globe objects using a popular implementation of a replication subobject must include the code for that subobject in their own class archives. Because separate archives must be created for every valid combination of subobjects, this not only leads to bloated archives, but also to an increase in the number of available archives.

Modular class archives, those that contain only some of the subobjects needed by an LR, besides decreasing the number and size of class archives, also allow code to be updated dynamically. For example, suppose a bug is found and fixed in the code of a popular replication subobject. When modular class archives are used it is sufficient to update that replication subobject's class archive. All LR implementations that refer to that class archive will automatically load the new code the next time they are loaded. If monolithic class archives are used all class archives that contain the insecure code must be individually updated.

The downside of modular class archives is that combinations of subobjects may be loaded that an object creator had not envisaged, or explicitly wanted to prevent. These may cause problems such as a malfunctioning object or security problems.

Because neither the monolithic or modular approach is ideal in all situations, we allow class archives to be either monolithic or modular. In the case of modular class archives, information about the actual archives needed to load a complete LR is included in an object's contact address.

4.2.2 The Repository

The implementation repository is implemented as a local Globe object that implements the classRepository interface presented in Table 4.10.

interface	classRepository	
method	getArchiveCA	Given a contact address, finds and loads an appropriate implementation
in	ca	The class object
returns		A reference to a Globe class object that can create instances of an appropriate LR
method	getArchiveIH	Given an implementation handle, finds and loads an appropriate implementation
in	ih	The implementation handle
returns		A reference to a Globe class object that can create instances of an appropriate LR
method	getClassObject	Loads an implementation from a previously loaded class archive
in	name	The class name
returns		A reference to a Globe class object corresponding to the given name

Table 4.10: The classRepository interface.

The getArchiveCA method is used during the binding process. After the runtime system selects an appropriate contact address, it invokes this method to load an appropriate local representative for the given Globe object. It takes the Globe object's contact address as a parameter and returns an interface to a Globe class object that can create a local representative for the Globe object. The getArchiveIH method is similar to getArchiveCA, except that it is given an implementation handle directly and does not need to extract one from a contact address. The getClassObject method is used to load an implementation from a previously loaded class archive (this method exists to support lazy loading of code, the details of which will be described later). it is given a class name and returns an interface of the class object corresponding to that name.

Contact Addresses and Implementation Handles

A contact address, as described in the previous chapter, identifies where and how a Globe object can be contacted. It consists of a network address and a protocol identifier. In GlobeDoc this protocol identifier contains an implementation handle and initialization data. The implementation handle is an identifier that specifies a particular LR implementation and the location where a class archive containing a suitable implementation can be found. The initialization data is used to create and initialize instances of the LR once its code has been found and loaded.

The format of an implementation handle is generally dependent on the repository that contains the corresponding class archive. Simple repositories might encode the location of a class archive directly in the implementation handle. More complex implementation repositories might include an object type or class name in the implementation handle, and map this type or name to an appropriate class archive depending on, for example, the platform and the various implementations available for that platform.

Currently, four types of implementation handles have been defined. The first, for locally available class archives, consists of a file: URL that contains the local path name of a class archive. The type of the class archive is determined by the filename's suffix. For example, file:///usr/local/share/globe/repository/MyGlobeDoc.jar specifies a locally available Java jar class archive. The second type of implementation handle, for remotely available class archives, consists of an HTTP or FTP URL that specifies where and on what host the class archive can be found. For example, http://globe.cs.vu.nl/globe/repository/MyGlobeDoc.jar specifies a Java jar class archive available at globe.cs.vu.nl. The third type of implementation handle consists simply of a class or type name, such as class:JAVA:vu.globe.globeDoc.globeDocumentImp.gdLRImp. It is up to the implementation repository to find an appropriate class archive that implements this class. Finally, the fourth type of implementation handle is a composite implementation handle that refers to multiple class archives, all of which are needed to load the code required by a single LR. Figure 4.16 shows an example of such an implementation handle. Note that the implementation handle specifies only class archive locations. It does not specify what types of subobjects are contained in those archives, nor the role that these subobjects play in the final LR implementation. This information is contained in each individual archive's catalog file.

```
http://globe.cs.vu.nl/globe/repository/GlobeDoc.jar;
http://globe.cs.vu.nl/globe/repository/MSRepl.jar;
http://globe.cs.vu.nl/globe/repository/TCPComm.jar
```

Figure 4.16: A composite implementation handle.

The initialization data included in a contact address is used to initialize an instance of an LR. It may include information such as the LR's role in the GlobeDoc object (e.g., client, replica, master, etc.), initialization data for the various subobjects (e.g., buffer sizes, etc.) as well as network addresses of other LRs to contact. There is no general format for this initialization data, it is up to the LR implementation to interpret it.

Retrieving a class archive

Given an implementation handle, the implementation repository must find a suitable class archive and load the code contained in it. Figure 4.17 shows the parts of the repository and the steps involved in retrieving a class archive and loading an object implementation.

Internally, the implementation repository consists of three parts: the **repository table**, the **repository resolvers** and the **class archive pool**. The repository table is where the

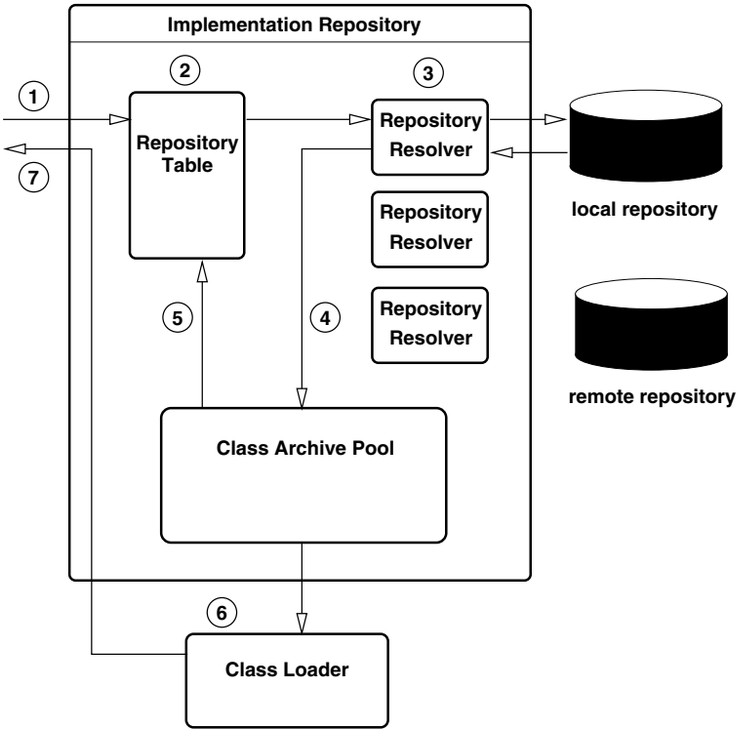


Figure 4.17: Implementation repository and the class loading process.

repository maintains mappings of object names to the class archives that contain their implementations. This table contains an entry per (local) object, with each entry containing an implementation identifier (e.g., in Java, the full Java object name), an implementation file reference (e.g., the name of a file inside the class archive), the implementation handle associated with the class archive, and the location of the class archive (e.g., an interface to the class archive object, or the path of a class archive file's location on the local file system).

Repository resolvers are local interfaces to remote repositories. The remoteRepository interface (see Table 4.11) defines a single method, get, which is used to retrieve a class archive from a remote repository. This method accepts a class archive name, the address of a remote server, and authentication information. A repository resolver fetches a class archive from the given remote repository and returns a reference to a local class archive object. The repository resolver interface can also be used to retrieve class archives from local repositories. In this case the server information is left blank. An implementation repository contains repository resolvers for the various types of implementation handles that it supports. Examples of repository resolvers include resolvers for file: implementation handles, ftp: implementation handles, http: implementation handles, class: implementation handles and composite implementation handles. The implementation repository keeps a table that maps between implementation handle types and the appropriate resolvers. The implementation repository also keeps track of addresses for the remote repositories that it knows about.

interface	remoteRepository	
method	get	Retrieves a class archive from a remote repository
in	name	The class archive name
in	server	The address of a remote server
in	authname	The name to authenticate with
in	authpass	The password corresponding to the name
returns		A reference to a local class archive object

Table 4.11: The remoteRepository interface.

The class archive pool is where class archive objects are stored after they have been loaded through repository resolvers. A class archive object must be kept until the implementation contained in it has been fully loaded (see the section on lazy loading for details). Once an implementation has been fully loaded, the class archive is no longer needed because instances of the associated object can be created using the associated Globe class objects.

Loading an implementation

The class loading process starts with a call to one of the repository's getArchive methods (step 1 in Figure 4.17). The implementation handle (extracted from the contact address parameter if necessary) is used as a key into the repository table to see if a required class

archive is already loaded (step 2). If this is the case, then the process moves on to step 6. Otherwise, if the class archive is not yet loaded, the repository must fetch it using one of the repository resolvers. The repository chooses a resolver and uses it to fetch a class archive (step 3). In the case of a local repository the resolver fetches the archive from the local file system, creates a class archive wrapper object around it and returns a reference to this object. In the case of a remote repository the resolver must first contact the remote repository to fetch the class archive file before wrapping it in a class archive object. Once it has a reference to the class archive object, the repository stores a reference to the object in the class archive pool (step 4), and registers the archive in the repository table (step 5). Next the repository gets a reference to a class loader from the archive and requests the class loader to load the code contained in the archive. The class loader loads the code and creates a Globe class object for the main subobject (step 6). If an appropriate class object already exists, i.e., the code was already loaded, then the class loader simply returns a reference to that class object. Finally, the repository returns a reference to the Globe class object (step 7).

The steps taken while loading code from a class archive using a class loader are shown in more detail in Figure 4.18. In this figure, the repository starts by retrieving information about the class archive's main entry and a reference to a class loader that can load this archive's code. The repository then calls the class loader's load method to load the archive's main entry. If lazy loading is not implemented then the repository proceeds to load the rest of the code contained in the class archive. Otherwise, it simply registers information about the rest of the subobjects in the repository table.

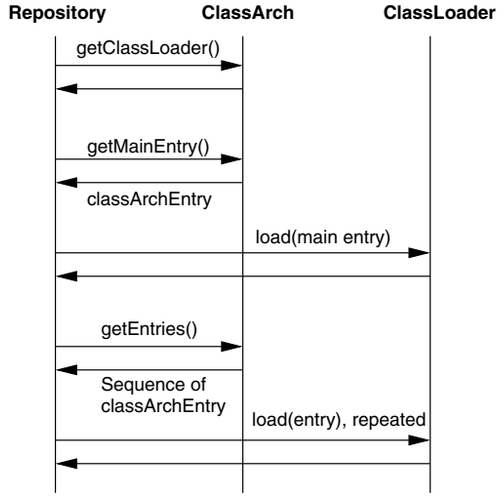


Figure 4.18: Steps involved in loading the code from a class archive.

When the repository's getClassObject method is called, the repository looks in the local name space to see if a Globe class object for the given object name already exists. If such

a class object already exists, then a reference to it is returned. If one does not exist, the repository looks in the repository table to see if an appropriate class archive is known. If this is the case then that class archive's class loader is requested to load the code and create the Globe class object. A reference to the new class object is then returned. If an appropriate class archive cannot be found then an error is returned.

Lazy Loading

For performance reasons an implementation repository might decide not to load all the code in a class archive at once. Sometimes only the required local objects for creating a local representative (usually the LRManager subobject) are loaded right away, while the rest of the subobject implementations are loaded later on when they are actually required.

Some systems allow code to be loaded on demand, that is, the system will automatically load code when it is needed. In Java, for example, if an object P accesses or refers to an unloaded object Q, the code for that object Q will be loaded by the system using the same class loader that loaded object P. This way, once the Globe runtime system loads a subobject P from a class archive, the loading process of other subobjects used by P can be delegated to the Java virtual machine, and the repository does not have to explicitly load those subobjects. The repository must, however, make sure that the implementation code remains available until the object has been completely loaded.

Unfortunately, not all platforms provide dynamic loading mechanisms. On these platforms the repository table plays an important role in implementing lazy loading functionality using the getClassObject method. By keeping track of which class archives contain which subobjects, the repository can find and load the appropriate code whenever getClassObject is called. Whether or not a class archive's code is loaded using lazy loading depends on the implementation of the implementation repository.

4.2.3 Class Loader

The class loader is a Globe runtime object responsible for loading code into the address space and instantiating and initializing a Globe class object from that code. The class loader is class archive and platform specific, that is, every combination of class archive type and platform requires a unique class loader. A class loader does, however, implement a well defined interface. This interface allows new class archive formats and their associated class loaders to be added to a Globe system without requiring modifications to existing code.

Table 4.12 presents the classLoader interface. This interface defines only one method, the load method. It loads the code for a given class from a class archive, creates the Globe class object for that class and returns a reference to that class object. The Globe class object can later be use to create local objects of that class. There is one-to-one correspondence between a class loader and its associated class archive which is why the load method does not require the specification of a class archive from which to load the code.

interface	classLoader	
method	load	Loads the code for a given class from a class archive
in	classname	The name of the class to load
returns		A reference to a class object

Table 4.12: The classLoader interface.

Java Jar File Class Loader

The Java jar file class loader is an example of a platform and class-archive specific class loader. In Java, dynamically loading code is a rather simple process because the Java platform provides built-in support for dynamic loading of code. The class loader's load method reads the jar file and finds the Java class file corresponding to the given class name. It then reads this class file into memory and calls Java's standard java.lang.ClassLoader.defineClass method. This causes the code to be loaded into the virtual machine and a Java class object associated with the given class name to be created from the code.

4.3 Clients

The success of GlobeDoc is largely dependent on the ability of client applications, such as Web browsers, to access GlobeDoc content. Client access to GlobeDoc content should be transparent, that is, the difference between accessing regular Web content and GlobeDoc content should be minimal if not imperceptible. Accessing GlobeDoc content should be as efficient, if not more efficient, than accessing regular Web content. Finally, methods for accessing GlobeDoc content should be nonintrusive (e.g., not involve user intervention or complicated setup procedures) and easy to use so that GlobeDoc content can be accessed by anyone capable of accessing regular Web content.

There are two main categories of GlobeDoc clients to consider. **GlobeDoc-aware clients** are applications that are able to directly access GlobeDoc objects. **GlobeDoc-unaware clients** are applications that have no knowledge of GlobeDoc and cannot directly access GlobeDoc objects.

4.3.1 GlobeDoc-aware Clients

To be GlobeDoc aware, a client application must be able to do two things. It must be able to resolve GlobeDoc URNs and it must be able to bind to a GlobeDoc object and invoke that object's methods. The most interesting kind of GlobeDoc-aware clients are custom made or modified Web browsers. Other specialized GlobeDoc-aware clients (such as tika, a GlobeDoc management tool which was developed as part of the GlobeDoc project) also exist, but these do less to help integrate GlobeDoc with the existing Web.

In a GlobeDoc-aware Web browser the difference between content from GlobeDoc objects and content from the Web should be completely transparent. A GlobeDoc-aware browser should handle GlobeDoc URNs in the same fashion as it handles other Web URIs

(such as HTTP or FTP URLs). There should also be no difference in display of content accessed from a GlobeDoc object or from the Web itself.

Custom Clients

There are a number of possible approaches to making GlobeDoc-aware clients. One of these approaches is to build a client from scratch. The benefits of this approach are that we retain full control over the application features and architecture. This allows GlobeDoc to be supported from the ground up, meaning that GlobeDoc-specific features can be integrated into the architecture of the client, not added as mere afterthoughts. GlobeDoc-related features incorporated in such a client might include: caching of GlobeDoc object bindings for improved performance, GlobeDoc replica hosting services, and management functionality similar to that offered by tika, the standard GlobeDoc management tool.

Unfortunately, building a GlobeDoc-aware browser from scratch also has its drawbacks. The main drawback is that building such a browser takes a lot of work. Not only do GlobeDoc-specific parts and features need to be implemented, but so does the general Web browser functionality (e.g., HTML rendering, HTTP support, JavaScript support, etc.). In general implementing a modern Web browser is a complex task. The Mozilla Web browser, for example, contains over two million lines of code and took a large team years to develop. Compatibility with existing browsers poses another problem. Currently, Web designers already deal with at least three somewhat incompatible browsers. This means that, in order to create pages that work in a majority of browsers, Web designers must test their work in at least three different browsers. Adding yet another browser to this list would make the Web designer's job even more difficult. As such, Web designers generally ignore any new browsers arguing that the new browser's small user base is not worth the effort it takes to assure compatibility. Requiring users to change browsers in order to access GlobeDoc content is very intrusive and will stand in the way of wide-spread adoption of GlobeDoc and GlobeDoc-aware browsers. A further drawback is that, even if the browser was 100% compatible with all existing browsers and Web sites, most users will switch browsers only if the new browser offers enticing features not available in their browser of choice. It is doubtful whether GlobeDoc support is such a feature.

Although it is not practical to build GlobeDoc-aware Web browsers from scratch, it is possible to build smaller, special-purpose GlobeDoc-aware applications and tools. One example of such an application is (the previously mentioned) tika, a GlobeDoc management tool. Tika allows users to create and destroy GlobeDoc objects as well as manage the contents of those objects. Other useful tools include content fetching tools which provide a simple way to access and download a GlobeDoc object's contents. Such a tool could, for example, be used to make GlobeDoc-aware scripts. Likewise, a GlobeDoc client library, which provides functions for binding to and accessing GlobeDoc objects, could provide interested parties an easy way to create specialized GlobeDoc-aware applications.

Applets and Plug-ins

Another approach to creating GlobeDoc-aware browsers is through GlobeDoc applets or browser plug-ins. An **applet** is a small (usually Java) program that can be run within a Web browser. It is downloaded from a Web server and executed by the browser. For security purposes, applets are usually run in a sandbox, which limits their access to most local resources. They do, however, have access to screen space and basic browser functionality.

A GlobeDoc-aware applet contains code (including parts of the Globe runtime system) that lets it bind to GlobeDoc objects and access their contents. Because of its limited access to browser functionality a GlobeDoc applet must also contain code that allows the elements (e.g., HTML pages, images, etc.) to be displayed in the browser.

The applet approach is interesting because it is unintrusive, that is, users do not have to change or patch their browsers, the applet code can be automatically downloaded and executed. The applet should also work in all Java enabled browsers, which includes almost all modern and widely-used browsers. Similarly, a single applet will work on multiple operating systems and hardware platforms as long as an implementation of the Java environment exists for that combination of operating system and hardware platform.

A problem with the applet approach is the fact that applets are generally limited in their use of local resources. In particular, applets may establish communication channels only to the host where the applet was downloaded from. Applets also have limited (if any) access to local resources such as local file systems. These restrictions may be relaxed if the downloaded applet is signed with a trusted certificate and given appropriate rights. Another drawback of the applet approach is that applets generally do not have reliable access to a browser's HTML rendering engine. This means that a GlobeDoc applet must contain its own code for rendering content acquired from a GlobeDoc object. As mentioned earlier, HTML rendering code is generally quite complex and would greatly increase the size of any applets containing it. Likewise, it may introduce incompatibilities with a browser's native HTML rendering engine.

The biggest problem facing the applet is a bootstrap problem. In order to load a document from a GlobeDoc object a user must first download a GlobeDoc applet. The problem is where to find the applet. A possible solution would be to create a regular HTML page that references the GlobeDoc-aware applet and have users load that page first whenever they come across a GlobeDoc URN. This HTML page causes the applet to be loaded after which the user may enter a GlobeDoc URN causing the applet to retrieve and display the corresponding GlobeDoc object's contents. Requiring users to perform an extra action (loading the applet page) to access GlobeDoc URNs is neither transparent nor easy to use and is therefore not a good solution to the problem. Variations of this solution are possible, however, all require some explicit user action which makes them nontransparent and therefore undesirable.

Similar to an applet, a **browser plug-in** is a small program module that a browser runs to display a particular type of content. Generally, a plug-in is associated with a specific type of content (i.e., a MIME type). When content of that type is downloaded by the browser, the browser finds an appropriate plug-in and passes control to the plug-in code. The plug-in is responsible for reading and displaying the content. For example, to

display a QuickTime movie a browser must first load a QuickTime movie plug-in. The plug-in is responsible for reading the QuickTime movie file and playing the resulting movie in the browser window. Whereas applets are usually provided by content providers themselves, a plug-in is generally provided by a single software developer while content providers provide content that can be displayed using that particular plug-in. The QuickTime plug-in, for example, is created by Apple and must be downloaded from Apple's Web site. Because plug-ins are provided from central, generally trusted, sources, they are usually allowed access to more local resources including networking facilities and local file systems.

A GlobeDoc plug-in is a plug-in that is capable of binding to and requesting elements from a GlobeDoc object. Like an applet, a GlobeDoc plug-in is also responsible for displaying the GlobeDoc content. In order for the GlobeDoc plug-in approach to work, a browser must recognize GlobeDoc URNs and pass these on to the GlobeDoc plug-in. Most browsers support recognition of content type based on file extensions or reported MIME types. Generally, however, browsers do not support recognition of URI types based on their scheme identifier. This is a large obstacle to implementing and deploying GlobeDoc plug-ins. The plug-in approach also shares other problems with the applet approach, namely that it may be limited in its access to browser resources such as the HTML rendering engine.

Because most plug-ins are implemented in C or C++ another drawback of the plug-in approach is that a separate plug-in must be implemented for each combination of supported platform and browser (e.g., separate plug-ins for Netscape on Linux, Netscape on Windows, Internet Explorer on Windows, etc.).

Extending Existing Clients

A third approach to creating a GlobeDoc-aware browser is to extend an existing, widely-used, browser. There are two ways of doing this. The first involves directly modifying source code to add GlobeDoc support into the browser. The second involves creating dynamically loadable modules, which modify a browser's behavior without having to directly modify its code.

Directly modifying a browser's code can be done only with browsers whose source code is available for modification, or in cooperation with the browser's vendor. In the absence of cooperation with a browser's vendor, this approach currently excludes two popular browsers, Internet Explorer and Opera, leaving Mozilla (and Mozilla-based browsers such as Netscape, Galeon, etc.) and Konqueror as the candidate browsers. Although it limits the choice of browsers, being able to modify browser code directly offers a lot of flexibility. By doing so, any extra code or functionality required for GlobeDoc support can simply be made part of the browser. Likewise, if necessary, the browser architecture may be altered (to a small degree) to allow better integration with GlobeDoc. Unlike creating a browser from scratch, however, it is not necessary to redesign or implement nonGlobeDoc related functionality such as the HTML rendering engine.

Browser modifications can be distributed as separate patches or merged into the official browser distributions. Due to the complexity and space requirements of compiling

a browser, users generally rely on binary distributions of browsers and rarely build them from source. Because of this, distributing source code patches is not a feasible way to widely distribute a GlobeDoc-aware browser. Merging GlobeDoc specific changes into an official browser distribution requires convincing maintainers of the code that the changes are indeed beneficial and justify the extra maintenance work that their inclusion entails. Until GlobeDoc becomes widely used, this may not be easy to do. A third option is to create and maintain a separate GlobeDoc-aware distribution (or fork) of the browser software. Besides the fact that this may not be possible due to license constraints, it also requires much maintenance and administrative work and is not a recommended solution.

Besides distributing the changes, another problem with directly modifying existing code is that users must upgrade their browsers to take advantage of GlobeDoc functionality. Users who do not upgrade will still be unable to access GlobeDoc content. A final problem is that, to achieve wide coverage, multiple browsers need to be modified. This requires a large programming and maintenance effort.

A different approach to modifying existing Web browsers involves creating **protocol handler plug-ins** in the form of dynamically-loadable code modules. A dynamically-loadable code module allows functionality to be added to a browser without requiring the browser's source or binary code to be modified. Such a module is dynamically loaded by the browser either at startup or later when the module's functionality is first needed. In this way, dynamically-loadable modules are similar to the content-specific browser plug-ins described above, except that they are more general in nature (i.e., they are used for more than just displaying content). A protocol handler plug-in is a module that handles a particular type of URI (as identified by the URI's scheme identifier). For example, a mailto: handler would be responsible for handling all mailto: URIs. Part of this handler's functionality may be to start up a mail client, or to automatically send email to the address specified in the URI.

A GlobeDoc protocol handler is responsible for handling GlobeDoc URNs. It is the GlobeDoc protocol handler's responsibility to split the URN into separate object and element names, bind to the object, retrieve the specified element's contents and return this to the browser. The browser is then responsible for displaying that element. The use of a GlobeDoc protocol handler plug-in in a particular browser depends on the support offered by that browser for dynamically loading modules. Although not all browsers support this, a number of popular and widely used ones (Mozilla, Netscape, Internet Explorer, and Konqueror) do.

The benefit of protocol handler plug-ins is that they make it much easier (for the developer and end user) to extend a browser's functionality. The developer is presented with a standard (but browser-specific) API to work with and does not have to be familiar with the inner workings of the browser. Because none of the browser's code is actually modified, the developer is also freed from dealing with browser related maintenance issues. The end user simply downloads and installs the plug-in to create a fully GlobeDoc-aware browser. Once the plug-in is installed the user is offered full transparency. GlobeDoc content can be accessed and used in the same ways as regular Web content.

One of the drawbacks of the protocol handler plug-in approach is that every browser has its own, incompatible, plug-in format. This means that separate plug-ins must be

made for all browsers. Likewise, the APIs offered to plug-in writers are C or C++ APIs which means that the plug-ins will be compiled to platform-specific binaries. As a result separate plug-ins must be created for every combination of supported platform and browser. Another drawback of the protocol handler plug-in approach is that users who have not downloaded the plug-in will not be able to resolve GlobeDoc URNs. The users must somehow be made aware that the plug-in can be downloaded before being able to resolve GlobeDoc URNs.

Mozilla protocol handler plug-in support. Mozilla (and many Mozilla based browsers such as Netscape Navigator) offers two possibilities for adding protocol handlers. The first involves associating a protocol handler plug-in with a given URI type. The protocol handler is loaded and given control whenever a URI of that type is encountered. The second possibility involves associating an application or server process with a given URI type. The associated application or server process is responsible for handling URIs of that type.

The first approach requires creating an nsIProtocolHandler class and associated nsIChannel class. The nsIProtocolHandler provides information about the protocol that it is responsible for, as well as providing access to an appropriate nsChannel object when needed. The nsChannel object provides a connection to the resource identified by a particular URI. For a GlobeDoc protocol handler, the GlobeDoc specific nsIProtocolHandler registers itself with Mozilla's networking service and informs it that it is responsible for handling GlobeDoc URNs. When a globe: scheme identifier is encountered, the GlobeDoc specific nsProtocolHandler is called and asked to provide an nsChannel to the URN. The GlobeDoc protocol handler creates an nsChannel object and returns it to the browser. The browser then uses this nsChannel to access the resource associated with it. The nsChannel object is responsible for binding to and accessing the specified GlobeDoc object. More information about creating protocol handle plug-ins for Mozilla can be found in [94].

The second approach requires a third-party extension to Mozilla called Protozilla. Protozilla is a Mozilla specific dynamically loadable module that allows URI scheme identifiers to be associated with external programs or processes. By associating a URI scheme identifier to an external program, whenever a URI with that scheme is encountered, the associated program is started and the URI is passed to it as a command-line parameter. This program is responsible for accessing the resource specified by the URI and returning its contents (preceded by a MIME header), to Protozilla where it is passed on to the browser. Note that the actual contents (e.g., an HTML page, an text file, an image file, etc.) and the format they are returned in are determined by the protocol handler. When a URI scheme identifier is associated with an existing process, then whenever a URI with that scheme identifier is encountered Protozilla opens a bi-directional communication channel to the associated process and sends it the URI. The process is responsible for accessing the resource specified by the URI and returning it to Protozilla.

Creating a GlobeDoc-aware browser using Protozilla involves associating the globe: scheme identifier with a GlobeDoc-aware content-fetching tool or a GlobeDoc proxy server. A GlobeDoc-aware content fetching tool is a program that takes a GlobeDoc URN as an argument, binds to the specified object, retrieves the specified element, and prints

the contents of that element on standard output. A GlobeDoc proxy server is a process that binds to and retrieves content from GlobeDoc objects on behalf of other GlobeDoc-unaware processes. A drawback of the content-fetching tool is that it is restarted for every GlobeDoc URN encountered. This means that it cannot store session information, and must therefore rebind to an object every time that object is requested. Because binding to an object is a time-consuming process this may cause performance problems. A GlobeDoc proxy server, on the other hand, can cache GlobeDoc object bindings, thus avoiding expensive rebinds every time an object is accessed. Another benefit of the proxy approach is that it is not bound to a single instance of a browser. It is, therefore, possible that a GlobeDoc proxy server can act as a local cache to users on the same machine or local network.

A particularly useful aspect of the Protozilla approach is that it does not impose implementation language requirements on the GlobeDoc specific parts. Thus, for example, it becomes possible to integrate a Java-based GlobeDoc environment into the C++ based Mozilla browser without worrying about language incompatibilities. More information about Protozilla can be found in [86].

Konqueror protocol handler plug-in support. Konqueror is a Web browser built for the KDE desktop environment. Konqueror and most other KDE applications make use of the KDE I/O (KIO) library to access external resources such as files and Web pages. The KIO library uses separate modules called kioslaves to access different kinds of resources. When opening a resource identified by a URI a KDE application passes that URI to the KIO library. Internally, the KIO library keeps a list of known kioslaves and the scheme identifiers that they are responsible for. When it receives a URI the KIO library extracts the URI's scheme identifier and finds an appropriate kioslave to handle that protocol. It then starts the kioslave (as a separate process), passes it the URI and returns a reference to the kioslave to the application. The application accesses the associated resource by communicating with the kioslave.

Making GlobeDoc-aware KDE applications involves creating and registering a GlobeDoc-aware kioslave. This kioslave will be responsible for handling all globe: URNs. When given such a URN the kioslave must extract the object and element names, bind to the object and retrieve the specified element. Note that, rather than directly binding to and accessing an object, a kioslave may also contact an existing GlobeDoc proxy as described above. The benefits of such an approach are that GlobeDoc object bindings may be cached, and bound objects may be shared between users. Note, also, that kioslaves offer both read and write operations. It should, therefore, also be possible to use a kioslave to update a GlobeDoc object's elements as well as read them. More information about kioslaves can be found in [79].

Internet Explorer protocol handler plug-in support. Like Mozilla, Internet Explorer also offers two possibilities for adding protocol handlers. The first involves associating a URI scheme identifier with an external application. All attempts to open a URI with that scheme identifier cause the specified application to be launched. The second possibility is

to use the Asynchronous Pluggable Protocols API to associate a scheme identifier with a protocol handler capable of accessing URIs of that type.

Associating a URI scheme identifier to an external application involves adding an appropriate entry to the Windows registry in HKEY_CLASSES_ROOT. This entry specifies the scheme identifier as the key, and the application and application parameters as the values. For GlobeDoc the key is globe: and the values specify an application that can handle requests for GlobeDoc URNs.

Mapping a URI scheme identifier to a protocol handler involves creating a protocol handler using the Asynchronous Pluggable Protocols API. Such a protocol handler is a COM object that handles any requests involving URIs containing the scheme for which it registered. To register a protocol handler an entry must be added to the Windows registry in HKEY_CLASSES_ROOT\PROTOCOLS\Handler\⟨protocol_scheme⟩ (where ⟨protocol_scheme⟩ is replaced with the actual scheme identifier, for example, globe:). The entry specifies the scheme identifier as the key and an identifier of the protocol handler object class as the value.

Both of these approaches provide GlobeDoc support to any Windows applications that use URL monikers to access URIs. It is likely, therefore, that adding a GlobeDoc protocol handler in this way will also make applications other than Internet Explorer GlobeDoc-aware. More information about the Asynchronous Pluggable Protocols API and extending Internet Explorer can be found in [64].

4.3.2 GlobeDoc-unaware Clients

A GlobeDoc-unaware client is any client application that cannot resolve GlobeDoc URNs and cannot directly bind to or access elements of a GlobeDoc object. Examples of GlobeDoc-unaware clients include unmodified Web browsers, unmodified file managers, search engine spiders, etc. In order to reach the goal of seamlessly incorporating GlobeDoc into the existing Web infrastructure it is necessary to allow existing GlobeDoc-unaware applications to access GlobeDoc contents as transparently as possible.

GlobeDoc Gateway

The most common way to allow GlobeDoc-unaware clients to access and use GlobeDoc objects is through the **GlobeDoc gateway**. The GlobeDoc gateway is a GlobeDoc specific HTTP server. It accepts HTTP requests for GlobeDoc URNs, binds to the appropriate object, retrieves the specified element and returns that element's contents as an HTTP response.

Figure 4.19 shows an example of the GlobeDoc gateway in use. In step 1, the client sends an HTTP request for a GlobeDoc URN to the gateway. There the object and element names are extracted (step 2) and the gateway binds to the object (step 3). In step 4 it retrieves the specified element's contents and in step 5 returns these to the client as an HTTP response.

Besides functionality for retrieving a GlobeDoc object's elements, the GlobeDoc gateway also contains functionality to browse through the GlobeDoc name space. For ex-

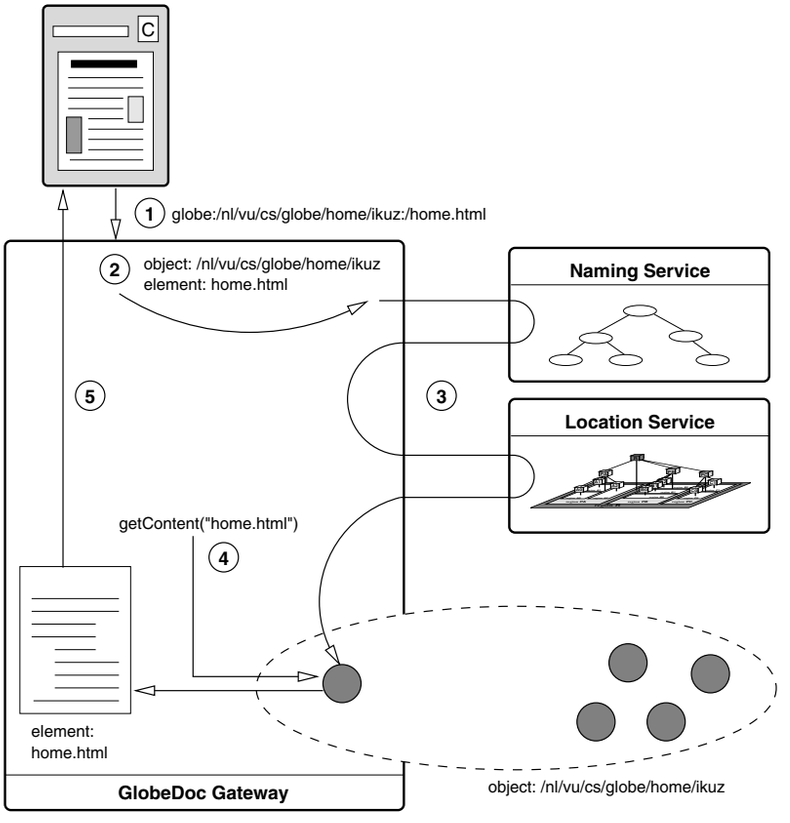


Figure 4.19: Example of a GlobeDoc gateway in use.

ample, when it receives a request containing an incomplete object name such as globe://nl/vu/cs/globe/, the gateway contacts the naming service to find out what objects that part of the name space contains. The gateway then creates an HTML page containing links to the root elements of all these objects and returns this page to the client. The links to the objects are returned as GlobeDoc URNs.

For performance reasons the GlobeDoc gateway maintains a cache of object bindings. This helps reduce the number of name-to-object binding operations for frequently accessed GlobeDoc objects. In the GlobeDoc gateway, a cache entry expires (and the corresponding GlobeDoc object is unbound from) if the corresponding object is not referenced within a given interval. More advanced caching and cache replacement strategies that can be applied in the gateway will be discussed later in Section 4.3.5.

GlobeDoc Translator

A problem with the GlobeDoc gateway is that in order for it to be useful, a client application must recognize that all requests for GlobeDoc URNs are to be sent to the gateway. This is only possible if the client recognizes these URNs. Unfortunately, most GlobeDoc-unaware clients do not know of, and are therefore unable to recognize, GlobeDoc URNs.

The **GlobeDoc translator** is meant as a solution to this problem. The GlobeDoc translator is an HTTP server that accepts requests for embedded GlobeDoc URNs. Recall that an embedded GlobeDoc URN is is an HTTP URL that contains the address of a GlobeDoc-aware HTTP server in its address part and a GlobeDoc object and element name in its path part. For example: http://globe.cs.vu.nl/nl/vu/cs/globe/ikuz/home:/projects.html refers to a GlobeDoc-aware HTTP server at globe.cs.vu.nl, an object named /nl/vu/cs/globe/ikuz/home and an element named /projects.html. When it receives a request, the translator translates the embedded URN into a regular GlobeDoc URN and forwards the request to an associated GlobeDoc gateway. When the gateway returns the requested element contents, the translator transforms the results by replacing any GlobeDoc URNs with embedded GlobeDoc URNs. The address part of these replacement URLs contains the translator's own address.

Besides acting as an HTTP server, the translator can also act as an HTTP proxy server. In this case, the translator accepts all HTTP URLs and filters out any that it recognizes as embedded GlobeDoc URNs. Embedded GlobeDoc URNs are processed by the translator as described above, while all other URLs are forwarded to their respective servers. All results, whether they come from the GlobeDoc gateway or regular HTTP servers, are filtered for GlobeDoc URNs which are replaced by equivalent embedded GlobeDoc URNs before being returned to the client.

The combination of a GlobeDoc translator and a GlobeDoc gateway is called a **GlobeDoc access point** (GAP). Generally the gateway and translator of a GAP run on the same machines and have the same IP addresses. They do, however, run on different TCP ports. A GAP's address is the same as its associated translator's address.

GlobeDoc Redirector

Use of a GAP allows GlobeDoc objects to be accessed from any HTTP enabled application. There are, however, problems with referring directly to a particular GAP in embedded GlobeDoc URNs. First, because a GAP's address is included in the URL, there is no longer any location transparency, nor is there any locality. A client resolving such an embedded GlobeDoc URN will send its request to the GAP specified in the URL causing the GAP's gateway to bind to the associated object. If the client is far away (either geographically or network topologically) from that GAP then the locality provided by the location service will be lost. Similarly, if an embedded GlobeDoc URN becomes widely spread so that many clients access the associated GlobeDoc object using that URL (e.g., the embedded GlobeDoc URN is posted in a story on Slashdot), the translator and gateway at the referenced GAP may become overloaded. Worse yet, the GlobeDoc object replica used by the GAP's gateway may become overloaded, while other replicas remain largely unused. Another problem is that the GAP referenced in an embedded GlobeDoc URN may form a single point of failure. If the GAP is down, or has moved to a new address, the embedded GlobeDoc URN no longer useful as a reference to the associated GlobeDoc element.

The **GlobeDoc redirector** solves these problems. It is an HTTP-server that redirects clients to their nearest GAP (where the nearest GAP is defined as the GAP that is nearest to a client in terms of geographical distance). To make use of the redirector, embedded GlobeDoc URNs contain the redirector's address rather than a specific GAP address. When a client resolves an embedded GlobeDoc URN, a request is sent to the redirector. Upon receiving a request, the redirector first converts the client's IP address to a latitude and longitude (e.g., by using the NetGeo [69] service). Next, the redirector calculates the distance between the client and each known GAP and chooses the GAP closest to the client. The redirector then sends a redirect message to the client, causing it to contact the chosen GAP and request the GlobeDoc element there.

The redirector supports two methods for redirecting a client. The first method is to return an HTTP redirect message to the client specifying an embedded GlobeDoc URN containing the address of the selected GAP. The second method involves returning an HTML page that immediately reloads an embedded GlobeDoc URN containing the address of the selected GAP.

In addition to redirecting the client, the redirector also stores two HTTP cookies on the client side. The cookies are used to improve the redirector's performance by storing previously calculated locations and GAP choices. The first cookie is called the GAP cookie and contains the address of the client's nearest GAP. The second cookie is called the location cookie and contains the geographical coordinates associated with the client's IP address. When the redirector receives a request from a client that has these cookies set, it will use the information from the cookies to redirect the client, rather than recalculating everything.

Figure 4.20 shows an example of a GlobeDoc object's element being requested through the GlobeDoc redirector. In the first step the client sends a request to the redirector for the embedded GlobeDoc URN: http://enter.globeworld.org/nl/vu/cs/globe/

ikuz/home:/projects.html (note that enter.globeworld.org is the redirector's address). The redirector converts the client's IP address into a latitude and longitude and finds the nearest GAP (step 2). Next, in step 3, the redirector returns a redirect page to the client, passing it an embedded GlobeDoc URN containing the chosen GAP's address. Finally, in the last step, the client follows the redirect and sends a request to the address contained in the new embedded URN.

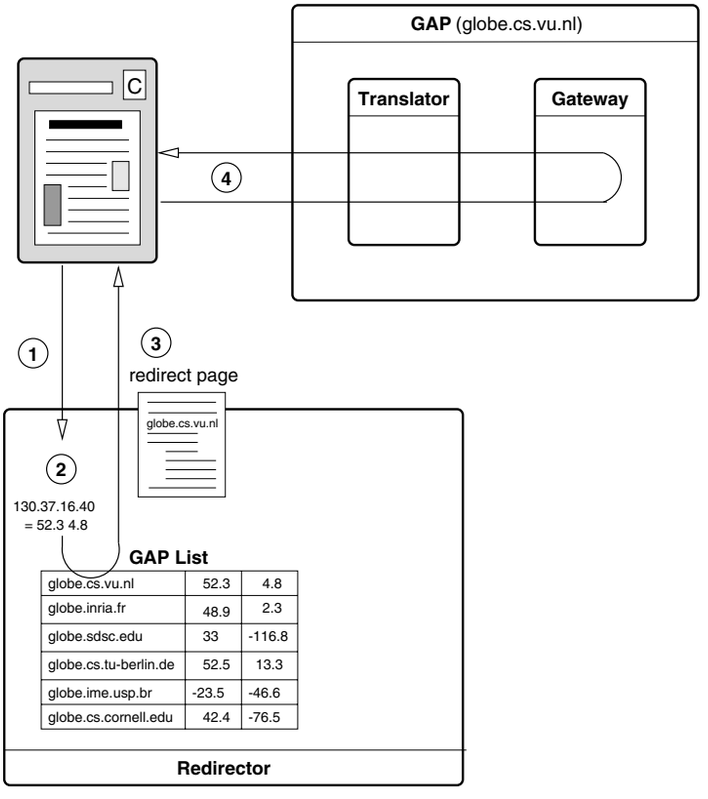


Figure 4.20: Example of the GlobeDoc redirector in use.

4.3.3 GlobeDoc-aware Clients versus GlobeDoc-unaware Clients

While the GlobeDoc-aware approach allows true seamless access to GlobeDoc objects, currently, the GlobeDoc-unaware approach is preferred as it allows unmodified clients to access GlobeDoc content. Allowing unmodified clients to access GlobeDoc objects will lower the threshold for use and deployment of GlobeDoc and thereby increase the chances of its wider adoption.

The drawback of the GlobeDoc-unaware approach is, however, that many more services (e.g., gateways, translators, and the redirector) must be deployed and managed. Clients that do not have GlobeDoc access points in their vicinity are also unable to fully benefit from replication policies implemented by the GlobeDoc objects that they access. Note, however, that deploying the services required for use by GlobeDoc-unaware clients does not prevent future GlobeDoc-aware clients from being implemented and deployed.

4.3.4 Partially GlobeDoc-aware Clients

Besides fully GlobeDoc-aware clients and completely GlobeDoc-unaware clients, it is also possible to have partially GlobeDoc-aware clients. Partially GlobeDoc-aware clients are applications that recognize GlobeDoc URNs, but cannot actually bind to or interact with a GlobeDoc object. An example of such a client is a Web browser that recognizes GlobeDoc URNs but forwards all requests for these to a (preconfigured) GlobeDoc gateway. Although the browser does recognize GlobeDoc URNs, it is not responsible for actually binding to the GlobeDoc object, the gateway takes care of this.

Another example of a partially GlobeDoc-aware client is one that rewrites GlobeDoc URNs into embedded GlobeDoc URNs and uses these to access the desired GlobeDoc content. Protozilla, the Mozilla plug-in mentioned earlier, makes it possible to turn Mozilla into a such partially GlobeDoc-aware client. Protozilla offers a function that allows URIs to be rewritten based on their scheme identifier and some user-defined rules. For example, a possible rule for GlobeDoc URNs is to replace globe: with http://enter.globeworld.org. This transforms all GlobeDoc URNs into embedded GlobeDoc URNs that refer to a GAP (or redirector) at enter.globeworld.org.

4.3.5 GlobeDoc Object Caching in Clients

All GlobeDoc-aware applications (including GlobeDoc-aware clients and the GlobeDoc gateway) must bind to a GlobeDoc object in order to retrieve GlobeDoc content. Because binding is a relatively expensive procedure, applications generally cache bindings to GlobeDoc objects. By caching object bindings, applications that frequently access the same objects can avoid much of the performance degradation (due to the number of lookups involved, the retrieving and loading of LR implementations, the fetching of state from remote LRs, etc.) associated with (re)binding.

Applications retain bindings to GlobeDoc objects by storing object LRs in an **object reference cache**. Due to Globe's reference counting mechanism, as long as a counted reference to an LR exists, that LR cannot be removed by the Globe runtime system. As long as a GlobeDoc object's LR is available in an application's address space, that application is considered to be bound to the object and can access the object without having to rebind to it.

Ideally, an application would be able to cache (and possibly prefetch) references to all the GlobeDoc objects that it ever uses. Unfortunately, keeping a GlobeDoc object's LR in cache costs resources, such as memory, storage space, and network bandwidth, all of which are usually available in a limited supply. This means that it is rarely possible for

an application to cache the LRs of all the GlobeDoc objects that are ever accesses. To allocate the resources required by new GlobeDoc LRs a client may have to free up cache space and other resources by unbinding from one or more previously cached GlobeDoc objects.

Applications must also be able to place limits on the amount of resources that cached LRs can consume. For example, an application should be able to limit the amount of memory and disk space used by cached LRs. Similarly, an application might choose to limit the amount of network traffic a cached LR may generate (for example, by only caching objects that rarely send and receive messages). These kinds of limits can be imposed on the whole LR cache or on individual LRs. Limits can be used to determine which LRs to cache as well as to determine if the cache is full and whether some LRs must be evicted to make room for new LRs.

When caching object bindings, besides taking the application's requirements into account, it is also necessary to take the GlobeDoc object's requirements into account. For example, depending on its distribution policy, an object might place a limit on how long a replica may be cached. Likewise an object's owner must be able to destroy all its replicas, including those stored in clients as cached object bindings.

Web Cache Replacement Policies

The problems involved in caching GlobeDoc object bindings are similar to those involved with caching Web resources in the World Wide Web. The problem of Web cache replacement, and in particular cache replacement in Web clients and Web proxies, has been widely studied and is well represented in the literature [20, 48, 90, 2]. Web cache replacement has already been introduced in Chapter 2. This section will look in more detail at particular strategies that can also be applied to the caching of GlobeDoc object bindings.

Studies of cache replacement policies are generally concerned with the size and replacement cost of the cached resources. In these studies the size of a resource refers to how much cache space the resource takes up while the replacement cost refers to a value representing the difficulty of bringing the resource (back) into the cache. The higher the replacement cost, the more difficult it is to bring the resource into the cache. In the case of Web caches, the replacement cost can be affected by numerous parameters such as the latency of the connection to the resource's originating host, the network bandwidth available, the resource size, etc. Traditional nonWeb cache replacement policies (e.g., those used for virtual memory) deal with data that has a fixed size (e.g., memory pages) and fixed replacement cost (e.g., accessing blocks on a local disk). When looking at Web caches, on the other hand, the data (Web resources) does not have a fixed size, nor does it have a fixed replacement cost. As such, it has been found that traditional cache replacement policies are not well suited as Web cache replacement policies.

The job of a cache replacement policy is to determine which cached resources to remove from the cache when a new resource must be inserted into a full cache. In general, a cache replacement policy always attempts to minimize or maximize various cost metrics. These cost metrics include the cache's miss rate, its hit rate, the average (client) access latency, total access cost, etc.

It is important to note that the implementation of a cache replacement policy must be reasonably efficient, otherwise much of the benefit of caching (quick access to Web resources) is lost. It has been shown that determining the optimal replacement policy in the off-line situation (i.e., when all requests are known before hand) is an NP complete problem [48]. This means that there is no efficient optimal solution to the problem, in fact, the performance of the optimal off-line algorithm degrades exponentially as the number of requests grows. It is clear that the problem of determining the optimal replacement in the on-line situation (i.e., when all requests are not known before hand) is harder than that of the off-line situation. As such, it follows that the performance of any optimal on-line algorithm will be similarly limited. To find efficient on-line cache replacement algorithms it is therefore necessary to look at heuristic algorithms that approximate an optimal on-line cache replacement policy.

Both Cao [20] and Hosseini [48] have proposed heuristic algorithms that take locality, size, and replacement cost factors into account. Both algorithms are based on the concept of specific cost, which is defined as the ratio of replacement cost over size. Specific cost reflects the fact that a good replacement algorithm must take both a resource's replacement cost and size into account.

Hosseini, besides looking at the specific cost of a cached resource, also takes the resource's activeness into account. A resource's activeness reflects the resource's degree of activity. This may be related, for example, to the resource's age, recency, or access frequency. Note that a resource's activeness is not a fixed value but changes over time. Hosseini combines a resource's specific cost and its activeness in the Double Sort algorithm [48]. The general effect of Double Sort is that the most active resources remain in cache and, of the less active resources, the ones with the lowest specific cost are removed from the cache.

Double Sort works in three phases. In the first phase the list of cached resources is sorted according to their activeness at that time. In the second phase, a new list consisting of a portion of the least active resources from the first list is created and sorted by specific cost. Finally, in the third phase, resources, from the second list with the lowest specific cost are chosen for eviction. Different variations of the Double Sort algorithm are possible. For example, varying the model used to calculate replacement cost influences a resource's specific cost and therefore influences a resource's chances of being evicted. Similarly, varying the factor that influences a resource's activeness (e.g., whether activeness measures age or frequency of access) will also change a resource's position in the activeness list and thus its chances of being evicted.

With the Greedy Dual-Size algorithm [20] Cao proposes an algorithm that takes a resource's specific cost and its frequency of access into account. In this algorithm each cached resource is associated with a value, H, which is originally equal to the resource's specific cost. On a cache miss resources with the lowest H values (H_{min}) are chosen for eviction. After resources have been evicted, the H values of the remaining resources are decreased by H_{min}. On a cache hit, the requested resource's H value is reset to its specific cost. As in the case of Double Sort, the effect of the Greedy Dual-Size algorithm can be altered by using different cost models to determine a resource's specific cost.

GlobeDoc Cache Replacement Policies

There are a number of similarities between the caching of GlobeDoc object bindings and the caching of Web resources. In both cases the cached resources have a nonuniform size and nonuniform replacement costs. Similarly in both cases policies and algorithms may optimize for resource or object hit rate, byte hit rate, access latency and network traffic. Likewise in both cases the access patterns (e.g., patterns of activeness, popularity of particular resources or objects, source (location) of requests, etc.) are very similar.

Despite the similarities, there are some major differences between the caching of Web resources and GlobeDoc object bindings. Unlike a Web resource's size, a GlobeDoc object LR's size changes over time. This is because an LR, unlike a traditional Web resource, is not read-only. This means that the LR's state can be updated while the LR is cached, but not actively used. Note that, because a cached LR's size can change when it is updated, cache entries may need to be removed on an update as well as on a cache miss. Unlike a Web resource, a GlobeDoc LR can also cause extra incoming traffic, even when it is not being actively used (e.g., an actively replicated LR may receive and process update requests from its peers). Likewise, a GlobeDoc LR can generate extra outgoing network traffic (e.g., an LR might regularly poll for updates or forward updates to other peers it knows about). Unlike Web resources which are merely data, a cached GlobeDoc LR can use up CPU cycles while processing incoming messages. Another big difference is that while in Web caching it is up to the (proxy) server or client to determine whether a resource is stored on disk or in memory, in GlobeDoc this is up to the GlobeDoc object itself. Also, in GlobeDoc, whole documents (GlobeDoc objects) are cached, whereas in the Web resources are cached individually. Finally, while a Web proxy (or client) generally determines and implements a global replication and coherence policy for its cached resources, in GlobeDoc this is something that is determined individually by each GlobeDoc object.

Also important to note is that GlobeDoc objects have a nontrivial binding cost associated to them. This is a uniform cost based on the time it takes to contact the naming and location services and retrieve, load and instantiate an LR implementation. This does not include the time it takes for the LR to contact any of its peers or to transfer state to or from those peers. The binding cost can also be expressed in terms of the network traffic generated during binding.

Given these differences, it is clear that when developing cache replacement policies for GlobeDoc object bindings we must look at more than just hit rates, miss rates, latency and network traffic. It is also important to consider the extra network traffic that a cached GlobeDoc LR can generate, the extra CPU cycles that an LR may use up and the main memory and secondary storage that the LR will require.

Note, also, that there is a large degree of interaction between the replication policies, access patterns, and update patterns. This means that subtly different combinations can often cause very different results. For example, using active replication in a GlobeDoc object that receives few updates will result in an LR whose size is relatively stable. Likewise, such an LR will use up little processing power and cause little network traffic (either incoming or outgoing). On the other hand, active replication in a GlobeDoc object that is

updated regularly will result in an LR whose size changes regularly. Such an LR will also use up more processing power and will receive a regular stream of incoming messages, increasing the network traffic to its host.

To apply the Web cache replacement algorithms described above (Greedy Dual and Double Sort) to GlobeDoc object caching it is necessary to determine a cost function that can take the new GlobeDoc specific metrics, much of which are affected by an object's replication policy, into account. According to the design of Globe, however, an object's replication policy is meant to be transparent, which means that GlobeDoc does not provide any mechanisms to find out about an object's policy. Fortunately, this may not be as bad as it seems. Given that a GlobeDoc object is free to implement any replication policy it desires, having the cache replacement strategy depend on knowledge of a limited set of replication policies will limit how LRs replicated according to a new policy are dealt with. Worse, such a strategy may interfere with the proper functioning of the cache by making incorrect assumptions about the LR. Ideally the cache replacement strategy should work independently of the object's replication policy (in the same way that the semantics subobject works independently of the replication policy).

There are two approaches to solving this problem. The first involves adding a cache management interface to the control object. This interface provides methods that return information about an LR's size, network cost, main memory use, secondary storage use, processing cost, latency cost, etc. It would be up to the LR (a collaboration between the replication, communication, semantics, and control subobjects) to implement this interface by recording, calculating, and providing the required information. The information returned by these methods could be used to calculate values for the LR's specific cost and activeness. Although the current suggestion is specific to the problem of LR caching, the interface can be generalized into a statistics interface, which would provide a wide variety of information about the LR and the GlobeDoc object it is part of. Such an interface could be used by other interested parties to monitor and manage LRs.

The second approach involves devising a cache replacement algorithm that is independent of the effects of an object's replication strategy. Such an algorithm would most likely resemble one of the Web cache replacement algorithms described above, with the replacement cost function modified to take into account the fixed costs associated with binding to an object. Although this approach ignores many of the GlobeDoc specific (and in particular replication-strategy specific) issues presented earlier, it is possible that such an algorithm may nevertheless provide effective results. As Cao has shown, many algorithms that depend on unstable data (such as latency data) provide worse results than those that rely solely on stable data. Experiments must be performed to verify if this is the case for GlobeDoc and to determine which of the two solutions is most feasible.

4.4 Shared Local Replicas

Often, when clients from the same location access the same Web document, it is preferable to create a local shared copy of that document than to have every client keep its own copy. This sharing of a local replica is the idea behind proxy caches in the Web In GlobeDoc

the effect of a proxy cache can be achieved through the sharing of LRs. A **shared local replica** (SLR) is a replica (i.e., a regular LR) of a GlobeDoc object that is created to provide a locally accessible LR for clients to bind to. Unlike a Web proxy cache, the use of SLRs must remain transparent to the client. That is, a client need not explicitly specify that it is going to use a shared replica. Moreover, a client will generally not know whether it is using a shared replica or not.

This transparency is achieved through the use of a **bind-through service** (BTS). The essence of the BTS is that it performs the first phase of binding (mapping an object's name to an object handle, and mapping an object handle to a set of contact addresses) on behalf of the client, while possibly contacting an existing LR or creating a new LR as a side-effect. From the outside, it seems like the client simply passes the BTS an object name or object handle and the BTS returns a contact address. On the inside, however, the BTS determines whether an SLR is available to be used, or if one should be created. As a consequence, whether the returned contact address is for an existing SLR, a newly created SLR, or a remote replica, is transparent to the client.

When binding using a BTS, a client transparently forwards its bind request to the BTS (step 1 in Figure 4.21). Note that the request may include extra information required to choose an appropriate contact address (e.g., required consistency guarantees). The BTS first checks whether a local object server is already bound to the object. If so, the BTS returns a contact address for the LR at that object server (step 5) and the client continues with the binding process.

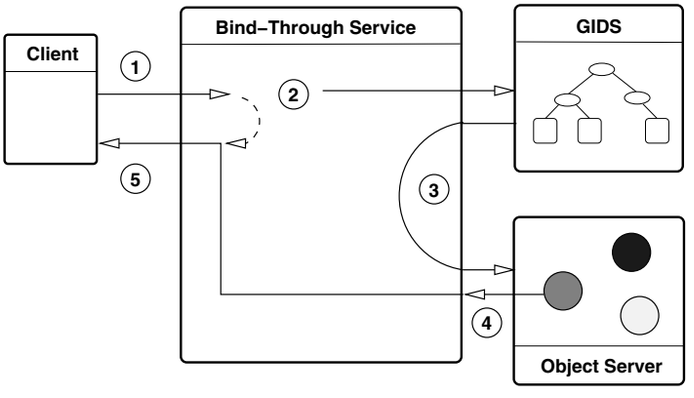


Figure 4.21: Creating a shared local replica.

If no appropriate, locally bound LR is found, however, the BTS attempts to create a new SLR. It first tries to find an appropriate local object server through GIDS (step 2). If a server is found, a replica is created on that server (step 3). This will cause the server to load an LR in its address space and connect to the rest of the object. After completing the binding, the object server returns a contact address for the new LR (step 4). This contact address is then used by the client to complete the binding process (step 5).

If no appropriate local object server is found, the BTS simply forwards the bind request to the Globe location service. The contact address it receives is returned to the client, allowing the client to proceed with binding as usual. After receiving one or more contact addresses from the BTS, the client proceeds as normal with the next phase of the binding process.

Chapter 5

Globe Infrastructure Directory Service

As we have seen in the previous chapters, the GlobeDoc architecture relies on many services. Examples include LR hosting services, the location service, the naming service, etc. These services are generally implemented by one or more servers, either as a single independent server or as a distributed service where multiple servers work together to provide the service. LR hosting, for example, is generally done by a single object server. The location service, on the other hand, is implemented by many cooperating widely distributed location service nodes.

For clients it is usually irrelevant whether a service is implemented by a single server or by multiple servers. A client accesses a service through a **service access point**, that is, a server (usually a local server) that provides access to the service. Depending on the service and its implementation, service access point functionality may be implemented by multiple servers or by a single, dedicated, access point server. In the location service, for example, all leaf node servers provide access point functionality. Clients connect to their closest leaf node when they need to access the location service.

Services generally provide access to **resources**. These resources may, for example, be primary or secondary storage, a high-bandwidth network connection, large amounts of processing power, etc. Services may also perform tasks on behalf of clients. The naming service, for example, resolves object names to object handles and the implementation repository resolves implementation handles to implementation archives.

Services can be identified by a list of **properties** that describe the resources they provide and tasks that they perform. Besides practical information such as the addresses of a service's access points and access restrictions, the properties also include information such as the resources made available by the service and limitations placed on the availability of those resources. The list may also include other information about services offered such as performance guarantees and service costs.

In GlobeDoc, (potential) clients[1] often require access to this kind of service information. For example, when creating a GlobeDoc object, a user must know the address of a suitable object server where the object's initial LR can be hosted. Likewise, when creating a new replica of a GlobeDoc object, a user must also be able to find the address of a suitable object server to host the replica LR. Often this requires not only finding an object server that offers support for the right combination of platform and OS, but also one that is close to where client requests (for the object) originate. Likewise, in order to perform proper maintenance, GlobeDoc site maintainers (i.e., those responsible for maintaining a local group of object servers, GlobeDoc access points (GAPs), location service nodes, and naming service nodes) also require information about the servers running in their domain. Besides simply accessing this information they may also need to modify it if the status of servers or services changes.

In GlobeDoc we call the process of maintaining and accessing such service information **resource management**. The **Globe infrastructure directory service** (GIDS) is a service that allows resource management. It is a service that keeps track of all available GlobeDoc services and their properties. Clients can access this service and request information about any given service, or they can discover services given a set of search criteria. Services also access GIDS to register themselves and to keep data about themselves up-to-date.

This chapter describes the design and implementation of GIDS. The next section presents the GIDS architecture model and is followed by a section containing a practical overview of an implementation of GIDS using DNS and LDAP. The chapter closes with a review of related technologies and an evaluation of the GIDS system (in comparison to these other technologies).

5.1 GIDS Architecture

GIDS makes a distinction between local resource management and global resource management. **Local resource management** deals with managing resources and services within a base region. A **base region** represents a small geographical or network-topological area containing a group of services (object servers, location service nodes, GAPs, etc.). It is the smallest unit of proximity known to GIDS. In other words, if two processes are in the same base region, GIDS will consider them as being at the same location.

Global resource management, on the other hand, deals with grouping the information from base regions into larger units (called **regions** and making that information available to clients. In particular, global resource management deals with globally tracking resources and services, and providing efficient search facilities for clients, regardless of a client's location.

[1]Note that in this context a client is not a process requesting a GlobeDoc object's elements but a user or process that is in charge of managing (the replicas of) a GlobeDoc object

5.1.1 Local resource management

Base Regions and the Region Service Directory

Every base region has a **region service directory** (RSD). An RSD is the central access point for a base region, storing, managing, and making available data about all the services in that region. A service that wishes to make its information available through an RSD must register itself with that RSD by providing details of its resources, services, and additional properties. A service may register at an RSD only if it was previously authorized by the base region's administrator. Clients that make use of GIDS are also assumed to belong to a base region. Unlike services, however, clients do not register themselves at an RSD.

In the RSD, services are identified by a set of **property attributes** that describe their resources and services. General properties include information such as the service name, service type, address, communication protocol, the platform it runs on, etc. Services also have service-specific properties. For example, an object server has properties that describe whether it offers persistence, how much disk space it makes available to each object, how much memory is available, the available local network bandwidth, authentication and authorization information, etc.

The RSD stores a single record per service. Each such record contains all of a service's properties and a unique **service identifier**. This service identifier is set when a record is created and can be used to retrieve the contents of that particular record. The RSD is also searchable. By specifying a list of attribute-value pairs as search criteria a client can search for and retrieve all **service records** that match that criteria.

Table 5.1 lists the operations that an RSD provides for accessing and modifying service information.

The add_service, delete_service, and modify operations allow management of an RSD and the service information that it stores. The search operation initiates a search in the RSD for services with the given properties. In contrast, the remote_search operation is used to search for servers *outside* the RSD (i.e., in other RSDs). Both return a set of service entries that match the criteria. We return to these operations later.

An RSD makes its operation available to clients through an RSD-specific network protocol. This means that an RSD can be accessed both from within and from outside its base region, that is, it can be queried by clients from its own base region, as well as by clients from other base regions.

Naming and locating base regions

Base regions are grouped into nonoverlapping regions, which, in turn, are grouped into larger (nonoverlapping) regions. This organization leads to a **region hierarchy**, similar to the organization of domains in DNS. An important difference with DNS domains, however, is that, in GIDS, regions *always* represent a notion of proximity. For example, the base regions within a city can be grouped into a separate region representing that city, which, in turn, can be part of a region representing a province, which itself may be grouped into a country region, etc.

local operations		
method	add_service	Adds a service registration entry
in	serviceID	Identifier of the service to be added
in	properties	Properties of the service
method	delete_service	Removes a service registration entry
in	serviceID	Identifier of the service to be removed
method	modify	Modifies a service registration entry
in	serviceID	Identifier of the service to be modified
in	properties	New properties of the service
method	search	Finds a service with the given properties
in	properties	Properties to search for
returns		A list of identifiers for services matching the given properties
remote operation		
method	remote_search	Finds a service with the given properties
in	properties	Properties to search for
in	location	An extra location property
returns		A list of identifiers for services matching the given properties

Table 5.1: RSD operations

In this sense, each (base) region has an associated **location**, which is represented by a name that reflects the hierarchical organization of regions. For example, if the campus of the Vrije Universiteit forms a base region, a possible name for this region could be vucampus.amsterdam.nl.eu, with each component name representing a region.

In GIDS, a base region is the smallest grain of location representation, meaning that every service inside a base region has the same location as the base region itself. A full location name always refers to a single base region. A partial name (which is always a suffix of a full name), on the other hand, can refer only to a group of (base) regions, and never to a base region.

It is not necessary that the region hierarchy forms a balanced tree. As such, some branches in the hierarchy may be deep and broad, while others may be shallow and narrow. For example, in a region hierarchy that reflects geographic location, branches representing areas with a high density of services will tend to be deep and broad, while those representing areas with few services will be shallow and narrow.

Figure 5.1 shows an example of a region hierarchy and some associated full and partial names. Note that the partial name sara.amsterdam.nl.eu refers to an organization called SARA[2] that has been registered as a region, but does not (yet) have any base regions. In this case the set of base regions belonging to sara is empty. This feature (that a branch of the region hierarchy does not always end in a base region) is necessary to support

[2]SARA is a computing and networking service provider for the universities in Amsterdam.

the extensibility and flexibility of the hierarchy. It allows the hierarchy to be built up dynamically by adding regions and base regions as they are needed. For example, the sara.amsterdam.nl.eu region may acquire base regions (or even regular regions) in the future, in which case its part of the hierarchy may be extended.

Note, also, that in this example the naming scheme changes from a proximity based scheme to a logical scheme. For example, the names dunet1, vucs, and bionet all refer to logical networks as opposed to physical locations. This combining of naming schemes will be explained later in Section 5.2.2.

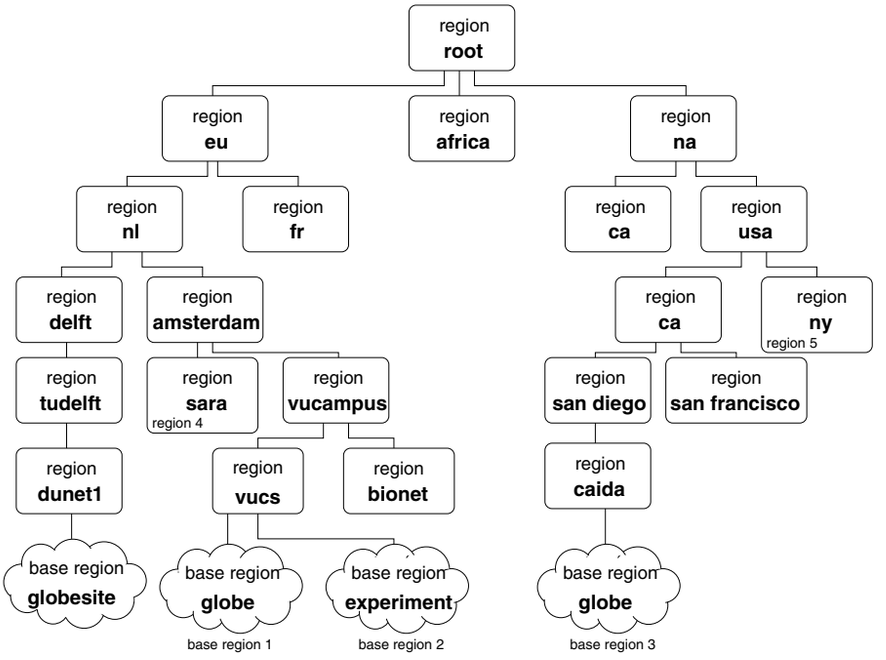


Figure 5.1: Base regions and a simple hierarchy.

Local management in a base region

Like GlobeDoc services, every client also belongs to a base region and has an associated location. Unlike services, however, clients do not register with an RSD; they simply contact their local RSD to determine their location. Besides GlobeDoc service details, an RSD also contains a wide variety of other information about its base region. For example,

an RSD can also provide the address of a local or nearby DNS server, or otherwise that of a proxy to such a server. Similarly, information on Web (proxy) servers, local file servers, and so on, can be readily provided by an RSD. In this sense, an RSD is similar to domain controllers as used in the Windows 2000 Active Directory [62]. There are, however, important differences with Active Directory that are presented later.

Security in a base region

There are a number of tasks (such as adding, updating and deleting service records), which can be performed with respect to an RSD, that must be protected against abuse. The RSD's security model deals with authenticating clients and confirming that they are authorized to add, remove, or modify service records. Authentication can be performed automatically by the RSD. However, granting authorization is arranged out-of-band (e.g., as part of a service contract). Note that the authorization needed for modifying an RSD is separate from the authorization that a client might need to actually use a service in the base region (which is determined by the service itself). There are also, currently, no mechanisms for verifying the validity of the data returned by an RSD.

5.1.2 Global resource management

While the previous section described management of servers and resources on a local scale (i.e., within a single base region), this section deals with the management of services and resources on a global level involving multiple regions.

Searching for servers

One of the main functions of GIDS is that it allows users to search for services with given properties. In doing so, it distinguishes between two kinds of searches: local searches and remote searches. A **local search**, initiated at a specific base region, is limited to the services in that base region, while a **remote search** allows a GIDS client to search for services in base regions other than the one where the search was initiated.

As described above, a local search is invoked through an RSD's search operation. A remote search is invoked through a similar remote_search operation. During a local search, the RSD searches through its internal database of service records trying to find records that match the given search criteria. The client may limit the number of results it wants returned, ranging from one, to all the possible matches.

Remote searching is somewhat more complicated, and proceeds in two steps. The first step involves resolving the given location name into a set of RSD addresses that represent all the base regions referred to by that name. The second step involves performing a local search on each of the RSDs found above for services that match the given search criteria. A remote search returns a set of service records for services within the specified location that fulfill all the specific client requirements.

Because locations are represented as names, the first step, location-to-name resolution, is done using a name service. This name service resolves a location name to the set

of base regions (actually, the addresses of the RSDs belonging to those base regions) that fall under that location. As an example, Figure 5.1 shows a number of base regions and a partial location hierarchy. Resolving the location name globe.vucs.vucampus.amsterdam.nl.eu in this hierarchy would result in the RSD of base region globe being returned. Resolving the name nl.eu, on the other hand, would result in the RSDs of base regions globe, experiment and globesite being returned because these three base regions lie in region nl. Finally, resolving the location sanfrancisco.ca.usa.na would result in an empty set being returned because there are no base regions in region sanfrancisco.

Such a name service can be built by associating a **region name server** (RNS) with every region. An RNS stores the following information about its associated region. For each subregion, it maintains a mapping of that subregion's location name to the address of the server representing that subregion. Such a server is an RNS if the subregion is not a base region, and otherwise an RSD. To resolve a location name, name resolution starts at the root RNS, and moves down the hierarchy following the components of the location name, completely analogous to DNS name resolution. A partial name is always resolved to an RNS, while a full name is resolved to an RSD.

When name resolution ends in an RNS, the naming subtree rooted at that RNS is traversed by resolving each pathname to a leaf node. This traversal results in a set of RSDs, which is subsequently returned to the client that initiated the original search operation. When name resolution ends in an RSD, that RSD is returned. Attempting to resolve a nonexisting name (i.e., one that refers to a nonexisting region) returns an error.

Once the location name is resolved to a set of RSDs, these RSDs are searched for appropriate services using their local search operations. Clients may limit the number of results they wish to receive from the remote search. In this case, the search stops when that number of services is found, otherwise it continues until all RSDs have been searched. The RSDs are searched in a random order to prevent any one RSD's servers from being chosen more often than others. This introduces simple load balancing and prevents servers from being favored or overloaded because their RSD is always found first during the location hierarchy traversal.

Scalability

Although having a root RNS through which all requests are made may pose a problem to scalability, well known techniques (from DNS), such as caching of mappings at every RNS, can be applied to prevent every request from going through the root. Similarly, as in DNS, the root RNS may be replicated over multiple physical servers to prevent overloading of any single server.

Global region management

The region hierarchy is dynamic, meaning that there is no predefined structure determining the number and locations of all (base) regions. Rather, the hierarchy is built up as regions and base regions are added and removed. This addition and removal of regions is controlled by a **global management policy**. The management policy determines which

clients are authorized to add or remove regions, and where and when such operations can take place. For example, it determines whether it is possible to add regions at any RNS or only at the leaves of the hierarchy, whether it is possible to remove or add whole subtrees at once, etc. In GIDS, the management policies can be implemented either internally (e.g., by the RNSs themselves) or externally (e.g., as an external management service). They can also be automated (e.g., a region can be added by performing an operation on an existing RNS) or manual (e.g., an administrator must edit a configuration file and restart the parent RNS for a region to be added). As explained later, the current implementation of GIDS adopts the policy implemented in DNS.

Region Hierarchies

Although GIDS provides a design for the region hierarchy, it does not impose any particular hierarchy. The only requirements placed by GIDS on a region hierarchy is that it reflects some notion of **proximity**. This may, for example, be geographic proximity, where regions represent geographic locations. Proximity in such a hierarchy would represent proximity in the real world. Another possibility is for the hierarchy to represent network proximity. Thus, regions that are close together in the hierarchy will also be close together in terms of network connectivity (e.g., number of network hops). Yet another region hierarchy might be based on Internet routing data, with the higher regions representing Internet backbone connections and autonomous systems, and the lower regions representing individual networks and subnetworks.

It is not clear that any single hierarchy is better than another. For example, although a hierarchy based on geographic data provides a good measure of geographic proximity, this does not always translate well to network proximity. It is possible that two points, although geographically close, may be on separate networks and therefore far removed in terms of network topology. On the other hand, it is currently difficult to build a hierarchy representing network proximity. Unlike the world's geographic topology, the Internet topology is not yet well mapped and is constantly changing.

Currently GIDS supports a region hierarchy that represents a combination of geographic and network proximity. This hierarchy is described in more detail in Section 5.2.2.

Multiple Region Hierarchies

Because no single hierarchy is ideal for all situations, GIDS has been designed so that it can support multiple region hierarchies. In this case a base region (and services registered with it) can have multiple locations - one location for each hierarchy. Figure 5.2 shows an example of some base regions that are accessible through three different hierarchies. One of the hierarchies is based on geographic proximity, one is based on Internet routing information and one is based on DNS domain names. In this example base region 1 has the following location names: globeexp.WI.bl1081.VU.amsterdam.nl.eu, globeexp.vucs.VU-NET.AS1103 and exp.globe.cs.vu.nl. Because base regions can have multiple location names they are considered independently of the (proximity based) region hierarchy that they are part of.

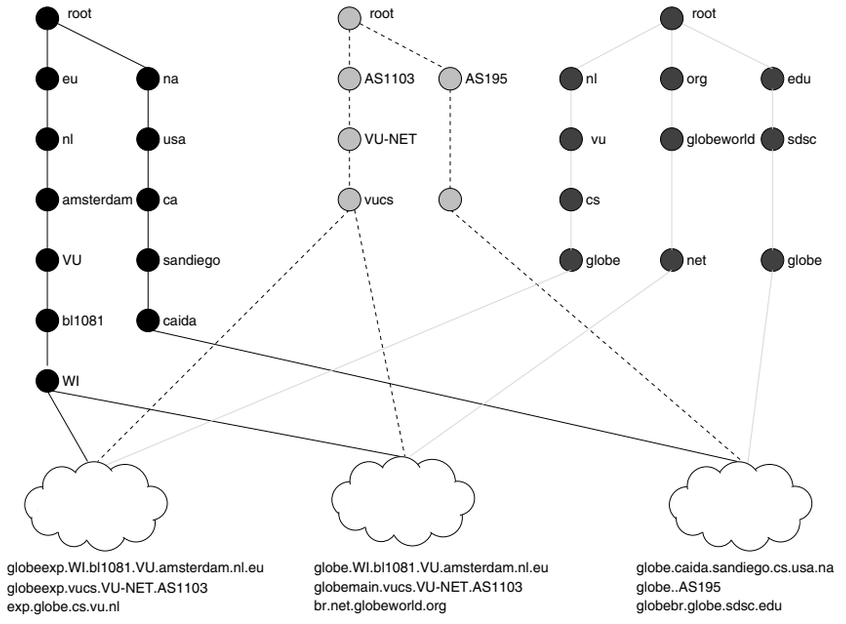


Figure 5.2: GIDS with multiple region hierarchies.

An important requirement for supporting multiple hierarchies is that the different hierarchies share the same base region granularity. That is, in each hierarchy the established base regions must be the smallest grain of location. This means that a base region in one hierarchy may not be a (inner) region in another hierarchy. Another requirement is that all RSDs must have access to at least one RNS for each of the hierarchies that they are part of. Note that not all base regions need to be part of every hierarchy.

5.2 Implementation

This section describes an implementation of GIDS based on widely available LDAP [112] and DNS [66] technology.

5.2.1 LDAP

LDAP (the Lightweight Directory Access Protocol) is a lightweight protocol for accessing X.500 directory services. The protocol is based on the IP protocol stack and as a result has become popular on the Internet. Despite being a protocol for accessing X.500 directory services, LDAP's popularity has led to implementations of stand-alone (that is, independent of X.500) LDAP directory servers.

The RSD, as described in the previous section, is implemented using a stand-alone LDAP server provided by the openldap project.[3] We use LDAP because it is a standardized directory protocol that provides all the functionality required for the RSD. By basing the RSD on a standard protocol, existing LDAP clients can be used to interface with our RSDs. Similarly, many libraries for building LDAP applications exist which greatly facilitates the building of custom GIDS clients.

LDAP defines both a network protocol for accessing information in a directory and a data model, which defines the form and character of that information. The data model is based on *entries* which contain typed *attributes* and their associated values. An LDAP **schema** defines the classes of available entries and the attributes contained in them.

The data model is object based, meaning that entries in the directory are instances of classes defined in the schema. Because of the object-based nature, entries are often referred to as **directory objects**. Following the object-oriented model, class definitions are hierarchical, a class may have a parent class and attributes are inherited from parent classes. Besides identifying an object class's attribute names and types, a schema also specifies whether attributes are mandatory or optional in an object class. Directory objects must always contain values for attributes that are declared mandatory in their class definitions.

As part of the implementation we have created a schema for GIDS. This schema contains class definitions for the various services (e.g., object server, location service, naming service, etc.) that an RSD keeps track of.

While LDAP directories are usually structured as trees (which means that schemas define both container objects and leaf objects, where container objects are used to create a tree structure and leaf objects represent actual resources, services and other entities) GIDS does not follow this model. In GIDS, the schema defines a flat object model, which means that all objects stored in GIDS are leaf objects. Each object represents a service, a resource or a configuration record. Note, however, that the object class *definitions* are hierarchical. The schema makes use of super and subclasses in defining object classes.

Logically, the GIDS schema can be divided into three parts. The first part contains general GIDS attribute type definitions. This includes, for example, definitions of attribute types for an IP address, a host name, a port number, etc. The second part of the schema contains definitions of attributes and object classes used for service registration, for example, object server registrations, gateway registrations, etc. The third part of the schema contains definitions of attributes and object classes for storing administrative and configuration information. This includes information such as a base region's location, the address of a DNS server that can be used to resolve location names, the port number that an object server should listen on, etc.

Figure 5.3 shows a fragment of the schema that defines attributes and object classes used to store service registrations. The schema includes attributetype declarations and objectclass declarations. Each declaration contains an **object identifier** (OID), a name, which acts as an alias to the OID, and a brief description. The OID is a unique identifier which is composed of an organizational part, a GIDS specific part, and a definition specific

[3] see http://www.openldap.org

part. The organizational part is a globally unique OID prefix assigned to an organization by an authorized registry such as IANA[4] or ANSI[5]. The Globe project has been assigned 1.3.6.1.4.1.10020 as its organizational OID prefix. The GIDS part is used to identify the OIDs used in the GIDS schemas. GIDS attribute type definitions have 2.1.2 as their GIDS part, while GIDS object class definitions have 2.2.2 as their GIDS part. Attribute type definitions are therefore assigned OIDs starting with 1.3.6.1.4.1.10020.2.1.2 while object class definitions are assigned OIDs starting with 1.3.6.1.4.1.10020.2.2.2.

```
1  ####################  General GIDS AttributeTypes ####################
2
3  attributetype ( 1.3.6.1.4.1.10020.2.1.2.17 NAME 'admin' SUP mail
4          DESC 'The email address of an administrator.' )
5
6  attributetype ( 1.3.6.1.4.1.10020.2.1.2.3 NAME 'host'
7          DESC 'A DNS hostname or IP address'
8          EQUALITY caseIgnoreMatch
9          SUBSTR caseIgnoreSubstringsMatch
10          SYNTAX 1.3.6.1.4.1.1466.115.121.1.44 ) # printable string
11
12
13  ################### Base server registration entry ###################
14
15  objectclass ( 2.3.6.1.4.1.10020.2.2.2.1 NAME 'server'
    SUP top STRUCTURAL
16          DESC 'Base class for server registration entries.'
17          MUST ( tag )
18          MAY ( admin $ description $ host $ port ) )
19
20  ################ Globe object server registration entry ##############
21
22  objectclass ( 1.3.6.1.4.1.10020.2.2.2.3 NAME 'GOS'
    SUP server STRUCTURAL
23          DESC 'GOS registration entry'
24          MUST ( host $ port )
25          MAY (objectBandwidth $ objectDiskspace $ objectMemory $
26                objectCPU $ totalBandwidth $ totalDiskspace $
27                totalMemory $ totalCPU $ load $ persistence $ os $
28                persistentContactPoints $faultTolerance $ runtime ) )
```

Figure 5.3: Fragment of the schema that defines service registration attributes and object classes.

The schema fragment shows the definitions of some general attribute types (that is attributes shared by multiple services), the top level server object class definition, and the object server (GOS) object class definition.

All service registration classes descend from the server object class. The server object class definition on line 15 declares the basic attributes that every service registration may

[4]Internet Assigned Numbers Authority, http://www.iana.org/

[5]American National Standards Institute, http://www.ansi.org/

contain. It specifies which attributes must be defined in a service registration and which attributes are optional. From the definition we see that a service entry must contain a tag attribute and that it may also contain admin, description, host, and port attributes. Note that these attribute types are defined elsewhere in the schema and their definitions are not shown here.

The attribute type definitions on lines 3 and 6 define the administrator email (admin) type and the host address (host) attribute type respectively. The admin attribute type definition, besides specifying the OID, name, and description, also specifies that it is a subclass of the mail attribute type (SUP mail). This means that it inherits all the characteristics of the previously defined mail attribute type. Unlike the admin definition, the host definition does not descend from a superclass. Thus, besides OID, name, and description fields, it also contains information about the characteristics of this type of attribute. The definition states that values of this type must conform to a specific syntax (the SYNTAX field), and may be compared in specific ways (the EQUALITY and SUBSTR fields). Note that the SYNTAX field contains an OID value that refers to a syntax OID as defined in [111].

The GOS object class definition on line 22 defines an object server registration entry. Like the other definitions, it also contains OID, name, and description fields. It also specifies that this definition is a subclass of the server object class. Furthermore it specifies the compulsory and optional attributes for registrations of this type. Thus, starting on line 25 we see that an object server may, for example, register information about its bandwidth as well as the amounts of secondary storage and main memory available to hosted LRs. On line 24 we also see that it is compulsory for an object server to include its address attributes (host and port) when registering itself.

As mentioned earlier, besides holding server registration information, GIDS may also be used to provide configuration information to services in particular base regions. Figure 5.4 shows a fragment of the GIDS schema where the service configuration object classes are defined. The schema defines two basic types of configuration classes: a base region configuration class and a service configuration class. Both of these are subclasses of the config object class defined on line 1. The base region configuration class, defined on line 7, specifies a base region's general characteristics. This includes information such as the base region's location (latitude and longitude) as well as the address of a DNS server which can provide access to the rest of the GIDS region hierarchy (dns). Service specific configuration information is contained is separately defined service configuration object classes, which are subclasses of the serverConfig object class defined on line 16. An example of a service configuration class is the GOSConfig object class defined on line 24. Like other object class definitions this definition specifies an OID, a name, a description, and both compulsory and optional attributes.

The GIDS schema is extensible. This means that new attributes can be added to entry definitions without invalidating data stored in the RSD based on older schemas. This allows new service classes and new service attributes to be defined when new services or service properties are introduced into the GlobeDoc architecture.

Note that in an LDAP directory an object is uniquely identified by a **distinguished name** (DN). A DN is usually constructed by combining the relative distinguished names (RDNs) of the object and all of its container objects. An RDN is a name that identifies an

```
1   objectclass ( 1.3.6.1.4.1.10020.2.2.2.8 NAME 'config'
    SUP top STRUCTURAL
2           DESC 'Base class for general and server config entries.'
3           MAY ( cn $ admin $ description ) )
4
5   ################### Base region configuration entry ##################
6
7   objectclass ( 1.3.6.1.4.1.10020.2.2.2.9 NAME 'baseRegionConfig'
8           SUP config STRUCTURAL
9           DESC 'Config parameters that apply to a whole base region'
10          MUST ( latitude $ longitude $ leafNodeID $ treeHost $
11                 treePort $ globeca )
12          MAY ( gnsRoot $ dns $ clientca ) )
13
14  ###################### Server configuration entries ###################
15
16  objectclass ( 2.3.6.1.4.1.10020.2.2.2.10 NAME 'serverConfig'
17          SUP config STRUCTURAL
18          DESC 'Base class for server configuration entries.'
19          MUST ( tag $ enabled )
20          MAY ( host $ port $ preferredIP $ serverOptions ) )
21
22  ################ Globe object server configuration entry #############
23
24  objectclass ( 2.3.6.1.4.1.10020.2.2.2.14 NAME 'GOSConfig'
25          SUP serverConfig STRUCTURAL
26          DESC 'GOS configuration entry.'
27          MAY ( lsPort $ securityFile $ lsLookupTimeout $
28                lsUpdateTimeout $ checkpointInterval )
29          MUST ( cpPort $ tlsCpPort $ tlsClientAuthCpPort ))
```

Figure 5.4: Fragment of a schema that defines service configuration attributes and object classes.

object but does not place it in the directory tree hierarchy. It is constructed out of one or more of an object's attributes. An RDN does not have to be unique, however, a DN must always be unique in a directory. Because GIDS does not model its directory as a tree of containers and leaves, a GIDS object's DN is equivalent to its RDN. In GIDS, an object's DN is constructed out of the objectClass attribute, a site-specific tag attribute and the RDN of a ficticious GIDS LDAP container object. For example, an object server might have the following DN: objectClass=GOS_server_top, tag=my_site, dc=gids. Note that the tag attribute is used to distinguish multiple instances of a service running in the same base region. Note also, that, due to the architecture of GIDS, it is not necessary for the DNs of GIDS records in different base regions to be unique. A DN only needs to be unique within a single base region.

The local RSD operations, as presented in Section 5.1, map directly onto equivalent LDAP operations. Thus, add_server maps onto the LDAP Add operation and delete_server maps onto LDAP's Delete operation. Similarly, modify maps onto Modify and search maps onto Search. Unfortunately, however, there is no direct mapping for the RSD's remote operation remote_search. To solve this problem we overload the LDAP Search operation and map both the local search and remote remote_search operations onto it. This means that the RSD interface, as implemented, does not strictly follow the syntax of the interface as presented in Section 5.1. We felt that this is acceptable given the benefits that using the LDAP protocol brings.

The LDAP Search operation uses a **search filter**, which defines the conditions that must be fulfilled for the search to match a given entry. The filter specifies attributes, attribute values, and match criteria for the search. Composite filters can be created by combining other filters using *and*, *or*, and *not* operations. LDAP also defines a string format for a human-readable filter representation [49]. This representation uses a prefix notation for the three logic operations (*and*, *or*, and *not*), and an infix notation for the attribute/value matching criteria. Figure 5.5 shows an example string representation of a filter where the persistence attribute's value must be true and the objectMemory attribute's value must be greater or equal to 32.

```
(&(persistence=true)(objectMemory>=32))
```

Figure 5.5: Example of an LDAP search filter

In order to distinguish between a local and remote search operation we require that the filter for the remote operation always contains a *location* attribute. The value to match for this attribute corresponds to the location parameter from the remote_search operation as specified in Section 5.1. Figure 5.6 shows an example of the search filter for a local and a remote search. The only difference between the two is the presence of a location attribute in the remote search filter.

Because both the local and remote search operations are represented as regular LDAP Search operations, an RSD must distinguish between the two before actually processing search requests. This means that, upon receiving a search request, the RSD must first scan the search filter to see if a location attribute is present. If it is, then the RSD must perform

```
local:
     (&(persistence=true)(objectMemory>=32))
remote:
     (&(persistence=true)(objectMemory>=32)(location=amsterdam.nl.eu))
```

Figure 5.6: Examples of a local and remote LDAP search filter

a remote search, otherwise it can continue with a local search. Because this scanning functionality is not a normal part of LDAP it is necessary to extend the functionality of the LDAP server used. In order to extend the LDAP server functionality without directly modifying the LDAP server code, the GIDS RSD has been implemented as two separate units as shown in Figure 5.7. The LDAP back end is an unmodified LDAP server such as the openldap server. It is responsible for storing all of the RSD's directory data. The **preprocessor frontend**, on the other hand, is a custom GIDS server that is responsible for scanning LDAP requests and filtering out and performing remote searches.

The preprocessor implements the LDAP interface and protocol and receives all of an RSD's LDAP requests. It scans all incoming messages and separates search requests from other requests. All nonsearch requests are forwarded on to the LDAP back end where they are processed. Results of these requests are piped through the preprocessor back to the RSD client. For search request messages, the search filter is extracted and examined to see if it contains a location attribute. If not, then it is assumed to be a local search and is passed on to the LDAP backend like all other requests. If, however, the filter does contain a location attribute, then it is treated as a remote search request.

Remote search requests are handled directly by the preprocessor. The preprocessor starts by resolving the location search attribute to a list of remote RSDs. It then sends each of the RSDs in this list an LDAP search query. The search query contains a search filter that is the same as the original filter except without the location search attribute. Results from these remote searches are grouped together and returned as a single LDAP reply to the RSD's client.

5.2.2 DNS

The implementation of the region hierarchy is based on DNS. In this implementation RNSs are implemented as standard DNS name servers, and are arranged in a hierarchy similar to that used in DNS. While domain names in DNS are similar to region names in GIDS, it is important to note that DNS domains do not always represent a notion of proximity. This means that it is not sufficient to simply map the region hierarchy onto the existing DNS domain hierarchy. For example, addresses in the .com domain represent administrative domains, rather than domains reflecting geographic or network topological proximity.

Currently, GIDS supports a region hierarchy that is completely separate from the DNS domain name hierarchy and represents a combination of geographic and network proximity. At the top, the hierarchy represents geographic proximity. The root node

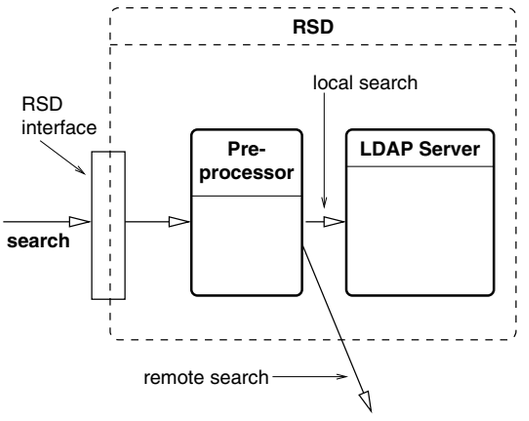


Figure 5.7: RSD with pre-processor and LDAP server.

represents the world, with the next level representing the continents, followed by countries and then (possibly) provinces or states and cities. At the city level the hierarchy changes into an organisational and network topological hierarchy. Figure 5.1 shows a part of this hierarchy. In this figure base region 1 is represented by the location name globe.vucs.vucampus.amsterdam.nl.eu. The topmost three elements represent geographic location (Amsterdam in the Netherlands in Europe), while the next two elements represent network topological location (the Vrije Universiteit campus network and the Computer Science department's LAN). Finally the last element represents the Globe group's base region. Note that the root region is not explicitly contained in the name. When explicitly referred to, this region is called . or root.

This hierarchy is independent of the existing DNS domain name hierarchy. Although this means that GIDS cannot reuse any of the existing DNS infrastructure, keeping the two name spaces separate does have advantages. First, when setting up the GIDS region hierarchy, it is not necessary to modify any existing DNS domain name servers. This is important from a practical point-of-view because many name server administrators are busy enough administering the DNS domain name space that administering a second name space will not be a high priority for them. Second, the current DNS domain name space does not represent proximity in the nongeographical top-level domains (e.g., .com, .org, etc.). Likewise some top-level country domains have been appropriated for commercial purposes thereby losing their geographic significance (e.g., .to, .tv, etc.). This makes it difficult to integrate the two name spaces. A third advantage is that separating the two unrelated name spaces prevents name clashes. Finally, while GlobeDoc and GIDS are not in widespread use, the Globe group can keep the management of the GIDS region hierarchy in its own hands. This facilitates expansion of the current hierarchy and allows experimentation with alternative hierarchies.

A major disadvantage of separating the GIDS and DNS domain name spaces, however, is that in order to resolve a GIDS region name, it is necessary to have access to

a GIDS-specific DNS resolver. Until GIDS becomes widely used, finding such a resolver will not be trivial (e.g., an address will have to be retrieved from a central site). To overcome this problem the GIDS hierarchy can be integrated into the DNS domain name space (without merging the two name spaces). This is done by rooting the GIDS region hierarchy at an existing DNS domain name, for example, gids.globeworld.org. GIDS region names can then be converted into DNS domain names by appending this domain name. For example, globe.vucs.vucampus.amsterdam.nl.eu would become globe.vucs.vucampus.amsterdam.nl.eu.gids.globeworld.org. This new name can be resolved by any DNS domain name resolver.

The actual GIDS DNS-based region hierarchy is implemented as follows. The DNS servers at the leaves of the hierarchy contain mappings from base region names to corresponding RSD addresses. These mappings are stored in DNS SRV records. An SRV record [42] is a DNS **service record** and is used to specify the location of named network services. This enhances DNS by identifying the specific services provided at a particular domain name, allowing different services to share a logical domain name but be hosted on different physical servers. A particular service is identified by prepending a service and protocol name to a domain name. Thus, an RSD server at the location globe.vucs.vucampus.amsterdam.nl.eu would be identified as gids.tcp.globe.vucs.vucampus.amsterdam.nl.eu, where gids identifies the GIDS service and tcp the TCP/IP protocol. Storing RSD mappings in SRV records allows GIDS to be implemented alongside other DNS based services without interfering with each other. A GIDS SRV record contains the hostname (or IP address) of an RSD server and the port that it listens to.

When an RSD performs a remote search, it acts as a DNS resolver and looks up the given location name. It first tries to look up the location name as though the name represented a base region, by prepending gids.tcp to it. If this works, the RSD receives the remote RSD's address and can contact it directly to perform a local search on it. If it fails, however, the RSD performs a zone transfer on the domain server represented by the location name thus acquiring a list of domain names managed by that name server. The RSD then repeats the above process on each these domain names until it finds all the underlying base regions and their corresponding RSD addresses.

5.2.3 Security

LDAP provides mechanisms that secure access to the directory information stored in LDAP directory services. Although there has not been any standardization on an actual access control model for LDAP (i.e., different implementations implement different models), LDAP does provide standardized authentication mechanisms. Authentication mechanisms supported by LDAP include anonymous (i.e., nonauthenticated) access, simple authentication, simple authentication via SSL/TLS, and SASL-based authentication. An LDAP server providing anonymous access allows everyone to access the LDAP directory. Simple authentication allows clients to authenticate themselves to an LDAP server by sending a cleartext password. This form of authentication is not very secure because the password is sent unencrypted over a network connection. Simple authentication via

SSL/TLS combines the simple password approach with an encrypted connection. Finally SASL (Simple Authentication and Security Level) based authentication [67] provides an extensible security model that allows an authentication method and optional encryption to be specified for protocol interaction.

Currently, RSDs provide anonymous read access and simple authentication for write access. This allows anyone to search through an RSD's directory, but only those clients that know an RSD's password to add or modify entries. Generally all processes that need to register themselves in their base region will be configured with their RSD's password.

The region hierarchy, being based on DNS, inherits DNS' security model. This implies that only administrators of GIDS DNS region servers can add, remove or modify regions and their (SRV) entries. Thus, for example, in order to add a base region to GIDS it is necessary to contact the administrator of the parent region's DNS server and request the addition of an SRV record for the new base region. There are currently no automated tools for inserting and modifying region and base region entries.

5.3 Related work

Different approaches have been proposed and implemented to keep track of distributed resources. One of the simplest approaches is manually distributing files containing resource information to interested parties. As resource information changes, new copies of the files must be redistributed to those parties. This approach works well for systems were few copies must be made and resource information rarely changes. However, when the number of interested parties increases and becomes more dispersed, or the resource information becomes more dynamic, such a manual approach becomes infeasible. Systems such as NIS [99] attempt to overcome this problem by automatically distributing these files, however, their scalability is generally limited to local networks. DNS provides a highly scalable service for mapping domain names to Internet addresses, however, it is not designed to contain general directory information nor to provide the flexibility required for a general resource directory (i.e. it does not support attribute-based naming [19]).

X.500 [89] is a standard that defines a wide-area, extensible directory service. Although X.500 provides a suitable directory service model, it is complex and requires the ISO protocol stack. LDAP (the Lightweight Directory Access Protocol) is a lightweight version of the X.500 directory service. It does not require the ISO protocol stack, but defines a protocol based on the IP protocol suite. Data encoding is also simplified and a standard API for directory access is defined. Because it can be used over IP, LDAP is becoming the directory service of choice for the World Wide Web. LDAP provides the capability to create distributed hierarchies of the directory data, however, this capability does not scale to the extent needed for GIDS. The reason for this is that LDAP places a directory server at every node in a hierarchy, which makes it a more heavyweight solution than our approach of using DNS for finding location-bound directory servers. GIDS does not benefit from the extra directory servers in the hierarchy because it does not store any information about the hierarchy. As such, the extra servers provide unnecessary overhead.

Because of its nature as a directory service and its implementation using LDAP, GIDS bears a strong resemblance to both Microsoft's Active Directory [62] and the Globus project's Metacomputing Directory Service (MDS) [37]. It is important to stress that although all three (GIDS, Active Directory, MDS) are based on the same technologies, and therefore have many resemblances, each is tailored to best suit its own particular environment, and none of them renders the others obsolete.

Active Directory, in addition to being based on LDAP, also makes use of DNS to organize its directory servers into larger groups or domains. The main difference between GIDS and Active Directory is between the role of domains in Active Directory and base regions in GIDS. In Active Directory organizations define hierarchical domains where each node in the hierarchy is represented by an LDAP directory server. Given an operation on a domain, DNS is used to find the address of the directory server on which the operation is to be performed. Whereas in Active Directory directory servers can represent internal nodes of the hierarchy, in GIDS the directory servers are limited to the leaves of the hierarchy. The reason for this difference is that, being primarily a system for storing administrative information, Active Directory stores information about the domains and subdomains themselves, as well as information about the servers in those domains. In GIDS, however, we are interested only in storing information about actual servers and services and these are present only in the leaves. Thus, in Active Directory the domain hierarchy is part of the directory structure, while in GIDS the location hierarchy is simply an efficient mechanism for finding directory servers. Active Directory also provides a domain-wide index called the Global Catalog, which provides a quick way to find information in a domain. This index is a centralized summary of the information found in the various directory servers in a domain. Such a centralized index would not be feasible in GIDS because GIDS's location hierarchy is meant to cover a much larger (both logical and physical) area and is therefore meant to contain a greater number of directory servers.

Unlike Active Directory, which is primarily an administrative system, Globus MDS is similar to GIDS in that both are resource directories; they provide information about available resources in distributed systems. MDS is a resource directory service for metacomputing environments. As such, one of its properties is that the data maintained by it are highly dynamic, which leads to the requirement that data are made available in a timely fashion. Similarly, the data returned by the system may come from multiple sources and might even be generated on the fly (as opposed to simply being stored in a database on the LDAP server). Because GIDS is used in an environment where the server data are less dynamic and timeliness is a less important issue, we do not share these requirements with MDS. Similarly, the data stored in MDS are of a much finer grain than that needed for GIDS. MDS also relies on the distribution capabilities of the LDAP protocol to create distributed hierarchies of the resource data. As mentioned above, we feel that the distribution provided by LDAP does not scale to the extent needed by GIDS.

Another system that is similar to GIDS is Jini [113], an environment that allows Java programs to provide and use remote services. The Jini infrastructure provides mechanisms for devices, services, and users to join a network, detach from a network, and find each other in that network without the need for any manual administration. A main component of this infrastructure is the lookup service. This is a service that, like the RSD in

GIDS, stores and publishes information about the services available on a network. Unlike the RSD, the lookup server in Jini is not based on LDAP, but on associative memory technology similar to JavaSpaces [39]. Although an associative memory approach is better suited to distribution (for example, through the use of hashing [92]), Jini is targeted for smaller-scale networks (at the workgroup level) and as such, wide-area scalability is not a goal in the Jini architecture. Jini, being an extension of the Java architecture, is also heavily dependent on Java and Java Remote Method Invocation (RMI).

5.4 Evaluation

GIDS is meant to be used by clients looking to find servers that can host their GlobeDoc object replicas. Because these clients generally know where to place their replicas, location plays an important role in the search process. As such, GIDS is designed as a location-aware resource management system. In contrast, most (general-purpose) resource management systems, such as MDS, are location unaware. That is, given characteristics of a service, they attempt to find any matching services, regardless of their location. In fact, such systems are often specifically used to find the locations of desired services. GIDS is not a general-purpose resource management system.

In practice this means that GIDS works well when searching for services whose (approximate) location is known. It does not, however, work well when performing global searches for services whose location is not known. Such searches require a search of the whole region (DNS) hierarchy, which is a highly inefficient operation. Note that attribute-based searches in large areas are inherently expensive.

GIDS is designed as a system for managing both global and local resources. Using GIDS for local resource management is equivalent to using it for global resource management, except that the location involved is implicit.

In the GlobeDoc environment, GIDS is also used by services as a means for finding configuration information. This makes the job of managing a local GlobeDoc environment easier as there are considerably less configuration files to be maintained. However, using GIDS in this way, we found that the limitation of storing data only in base regions prevented us from easily sharing information between base regions. For example, currently, when object servers in multiple base regions share an external service, copies of that service's address details must be stored separately in the RSDs of each of these base regions. When this information changes, modifications must be made to each RSD individually. Ideally this information would be stored at a higher level in the region hierarchy, where it could be shared by multiple base regions. Storing information in the hierarchy in this way resembles the approach taken in directory services (such as Active Directory) used for administration.

One thing that GIDS lacks is an RSD discovery protocol. Such a protocol would allow any GIDS-enabled client to find its nearest RSD and in doing so automatically determine its base region. RSD discovery would allow local services to startup, find a local RSD, and fully configure themselves without any interference from an administrator. Various local-area discovery protocols [78] [43] exist that can readily be used for this purpose.

Chapter 6

Performance Evaluation

Previous chapters have described and motivated the design and implementation of GlobeDoc. In this chapter the performance of the GlobeDoc architecture is evaluated. Recall from Chapter 1 that the performance of Web servers is generally affected by two types of scalability problems: problems due to the geographic distribution of clients and problems due to the number of client requests received. It was argued that an effective solution to both of these problems is to localize traffic by replicating Web documents. In Chapter 2 it was shown, using simulation based experiments, that one-size-fits-all approaches to replication cannot provide optimal (or sometimes even good) solutions in all scenarios. As such, the base of our claim that GlobeDoc provides a scalable architecture for the World Wide Web lies in its support for per-document replication policies.

These experiments have been repeated using the current implementation of the GlobeDoc infrastructure instead of simulations, and have reconfirmed the results presented in Chapter 2. Due to the similarity (both in setup and in results) of the repeated experiments and the previous simulation based experiments, they will not be further discussed in this dissertation. Instead this chapter will focus on profiling experiments performed to evaluate the performance of GlobeDoc's architectural components.

Because the overall performance of any architecture relies on the performance of its individual components, this chapter evaluates the performance of the GlobeDoc architecture's individual components. This is done by **profiling** the GlobeDoc architecture to determine the amount of work done by each component while processing requests. The results of this profiling experiment allow us to determine the contribution made by each component to the delay experienced by a client when requesting a Web document.

Note that this chapter does not compare the GlobeDoc architecture to existing HTTP servers, existing caching solutions, or existing content delivery networks. GlobeDoc code has not yet been optimized for performance, whereas software used in practice is generally highly optimized. As such a comparison between these systems would not provide a balanced evaluation.

Readers nevertheless interested in a direct comparison of Globe based applications and existing technologies, are referred to the experiments described by Bakker in [10]. In

Chapter 7 of this dissertation, the author compares the performance of the Globe Distribution Network (GDN), a Globe based network for the distribution of freely redistributable software, to that of the Apache HTTP server. In these experiments the throughput of GDN objects serving software packages is compared to that of Apache serving similar data. Because GlobeDoc uses the same code base as GDN, and because GlobeDoc and GDN objects offer similar functionality, the results presented also reflect on GlobeDoc's performance. Note, however, that GlobeDoc, unlike GDN, makes use of extra services such as the redirector, translator, and gateway. These services may introduce extra delay into the system, meaning that the results of a comparison between GDN and Apache cannot be directly transferred to GlobeDoc but can only serve as a guideline.

The simulation experiments presented earlier do, however, show that the GlobeDoc approach can provide a fundamental advantage over solutions that apply a one-size-fits-all approach to replication. Also, by providing a global view of GlobeDoc's performance, the profiling experiments allow existing and potential bottlenecks to be located and as such provide clues about where to start optimizing.

The rest of this chapter presents the experiments performed to evaluate the performance of the GlobeDoc architecture and is organized as follows. Section 6.1 introduces the profiling experiments used to examine the performance of individual GlobeDoc components. The experiment setup is described in Section 6.2. Section 6.3 explains how measurements were made and Section 6.4 presents the results of the experiments. Finally, Section 6.5 draws conclusions about the performance of the GlobeDoc architecture based on the experiment results.

6.1 Profiling

Because the GlobeDoc architecture consists of many separate components (e.g., the redirector, the translator, the gateway, etc.) there are many opportunities for performance problems to occur. Furthermore, some of the components are more sensitive to these problems than others. A component that is heavily used during the processing of a request will be more sensitive than one that is hardly used. The redirector, for example, is used only the first time that a Web document is accessed. Its performance will be less crucial to overall performance than the performance of components like the translator or gateway, which are used each time a Web document is accessed.

The primary goal of the profiling experiments described in this chapter will be to determine which components are most heavily used and therefore most crucial to the performance of the GlobeDoc architecture. The experiment results will be summarized in a number of functions, which can be used to predict the work done by any single component based on the properties of requests it must process. Analysis of the experiment results will provide an overview of how request properties (such as the size of the requested data or the number of elements requested) affect the amount of work done by each component. Finally, the experiments will also help to determine each component's contribution to the total time required to fulfill a client request.

The profiling experiments are performed using a GlobeDoc implementation based on the designs as presented in the previous chapters. The experiments consist of a single client performing requests for Web documents whose elements are stored in one or more GlobeDoc objects.

In these experiments we make a clear distinction between a **Web document request** and an **element request**. An element request is a request for a single GlobeDoc element as represented by an embedded GlobeDoc URN. An example of such a request is expressed by the following URL:

http://enter.globeworld.org/object1:/element1.html

A Web document request is a request for all the elements that make up a Web document. A Web document request is represented by a list of embedded GlobeDoc URNs, such as:

http://enter.globeworld.org/object1:/element1.html
http://enter.globeworld.org/object1:/element2.gif
http://enter.globeworld.org/object1:/element3.gif

Unless stated otherwise, in the rest of this chapter *request* will refer to a complete Web document request.

Note that in these experiments a Web document can span multiple GlobeDoc objects, that is, the elements making up a Web document can be partitioned over multiple GlobeDoc objects. In this case the list of URLs that represent the Web document will contain URLs that reference different objects. For example, the following Web document request contains requests for elements from GlobeDoc objects object1 and object2:

http://enter.globeworld.org/object1:/element1.html
http://enter.globeworld.org/object2:/element1.gif
http://enter.globeworld.org/object2:/element2.gif

Web documents have three identifying properties. The first, n, is the number of elements that a Web document contains. The second property, s, is the total size of the Web document, i.e., the combined size of all the document's elements. The third property, g, is the Web documents granularity, i.e., the number of GlobeDoc objects needed to contain all of the Web document's elements.

When requesting a Web document a client sequentially requests all of that document's elements. There is no delay between subsequent element requests, so as soon as a reply for the previous request is received the next request is sent out. A run of the experiment consists of repeated requests for a single Web document. The client may insert delays between these Web document requests. However, if such a delay is inserted the same delay must be inserted between all Web document requests.

Unlike the models offered by other Web load generators, which attempt to simulate realistic Web loads (e.g., SURGE [14]), our request model does not intend to be realistic in the same sense. A property of realistic request streams is that there is much variance between subsequent requests. Thus, for example, the delay between requests may vary, the size of the elements requested may vary, etc. In our case, a steady stream of identical

requests allows the system's reaction to particular types of requests (i.e., requests for particular types of Web documents) to be analyzed. A variable request stream does not make it possible to analyze the effects of various request parameters on system performance in this way.

For each Web document request made we measure the total **round trip time** (RTT) for that request and the amount of **work done** by each component to fulfill that request. RTT is defined as the time between the client's sending of the first element request and receiving the last reply for a single Web document. Work done to fulfill a request is defined as the amount of CPU time used by a component to process that request. Both RTT and work done are measured in seconds.

Figure 6.1 shows an overview of the experiment setup and the steps taken when requesting a Web document. As mentioned above, a Web document request is executed as a sequence of separate element requests. A Web document request is initiated by sending a request for the first element to the redirector (step 1). Upon receiving this request the redirector finds a GAP near to the client and returns a redirect message instructing the client to send all future requests to that GAP. In response to this redirect message, the client sends an HTTP request for the element to the given GAP's translator (step 2) where the request is immediately forwarded to the associated gateway (step 3). When it receives this request, the gateway binds to the GlobeDoc object referenced in the request. The binding process causes a local representative (LR) to be loaded into and created in the gateway's address space. After binding, the gateway invokes methods on the new LR to retrieve the requested element. These method invocations cause the LR to communicate with a replica hosted on a nearby Globe object server (step 4a), resulting in the transfer of the element contents from that replica to the LR. Depending on the replication policy implemented by the GlobeDoc object, the replica may contact other replicas before sending the element contents to the LR (step 4b). Once the gateway has received the complete element, the element is returned to the translator (step 5) where it is optionally processed before being returned to the client as an HTTP reply (step 6).

Note that the client contacts the redirector only the first time that it requests a Web document's element. After this first request, the client remembers its nearest GAP and does not consult the redirector for subsequent element requests. Every time a new Web document request is started, however, the client forgets its nearest GAP and always sends the first element request to the redirector.

Similarly, although the gateway can cache many object bindings, in this experiment it caches object bindings only within a single Web document request. This means that after the first request for a GlobeDoc object's element, subsequent requests for that object's elements do not cause the gateway to re-bind to the object. All cached object bindings are flushed after a Web document request is completed.

6.2 Experiment Setup

Figure 6.2 expands on Figure 6.1 and shows all 8 components involved in this experiment: the client, the redirector, the translator, the gateway, the naming service, the location ser-

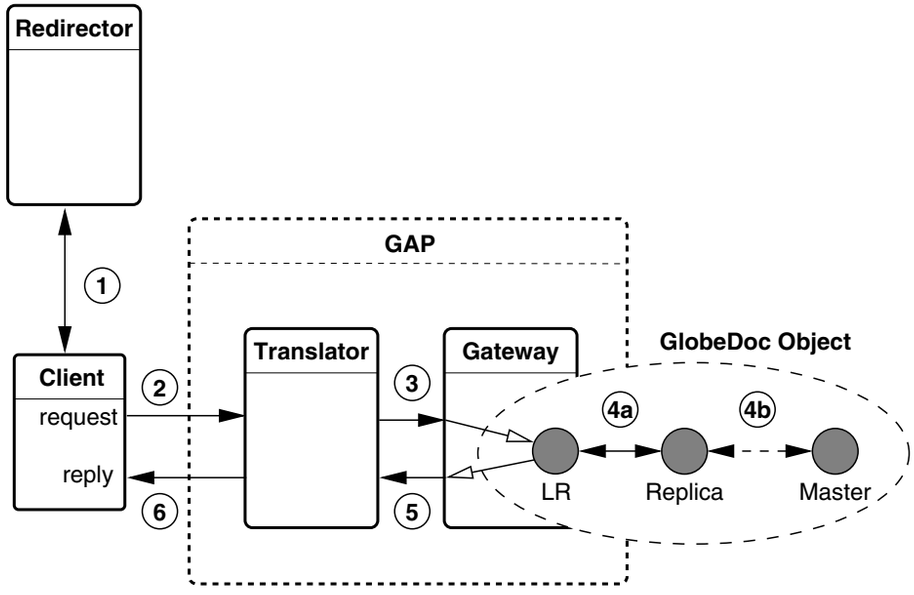


Figure 6.1: Overview of request processing.

vice, the replica object server, and the master object server. Note that, although the location service is treated as a single component, it is actually implemented as four separate components: the replica leaf node (used by the gateway and the replica object server), the master leaf node (used by the master object server), the root node, and the location service tree node (see [12] and [110] for more details about the implementation of the location service and the roles of these components).

The experiments were run on the DAS-2 [32] distributed supercomputer. This is a cluster computer consisting of a large number of PC nodes (of which 11 were used for this experiment). Each DAS-2 node is a dual 1-GHz Pentium III with 1 gigabyte of memory. The nodes are connected together by 100Mbit/sec Fast Ethernet as well as a Myrinet-2 network. However, only the Ethernet was used for this experiment. The nodes run RedHat Linux version 7.2 with kernel version 2.4.19-pre10. The version of the Globe software used is 1.0.6. To prevent GlobeDoc components from interfering with each other and to simplify the gathering of the performance data, each component runs on a separate node.

When a run of the experiment is started each component is assigned to a separate DAS-2 node. Next, each component is individually started on its respective node. The client component waits until all the other components are running before starting to send requests. Once the client has finished sending its requests and the last request has been processed, all running components are terminated. After all the components have ter-

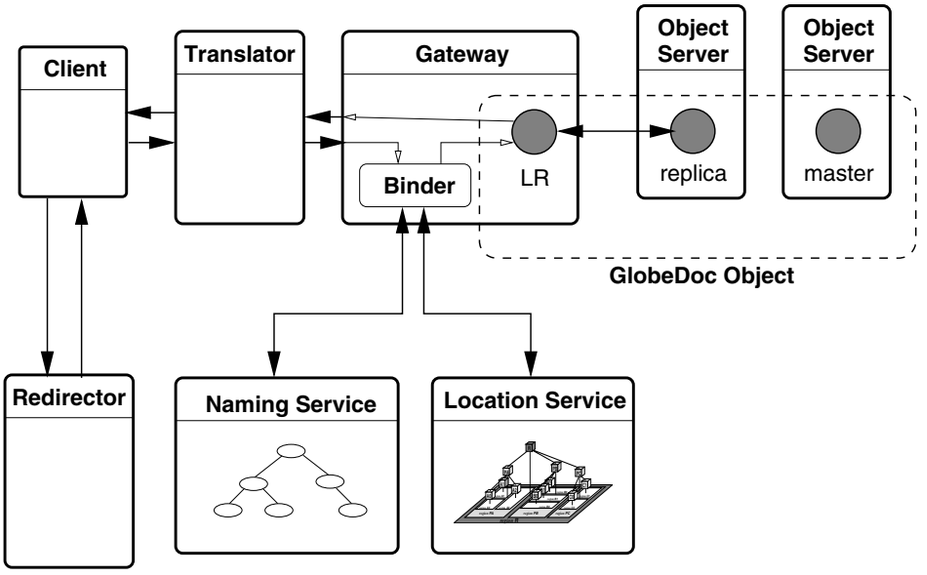


Figure 6.2: Setup of performance analysis experiment.

minated a preliminary analysis of their activity logs is made and stored on a central file system.

When running this experiment it is assumed that each GlobeDoc object is optimally replicated. This means that the replica to which the gateway connects is the closest replica available. It is also assumed that this replica is close enough to the gateway and that these two components can make use of the same location service leaf node. Furthermore, this replica will already contain the most up-to-date copy of the object state. Thus, when an LR makes a request for an element, that element's contents can be served directly by the replica without first contacting a master or other replica. A GlobeDoc object's elements do not change during the experiment.

Another assumption made is that the location service leaf node used by the gateway already contains the contact address for the GlobeDoc object replica at the replica object server. This means that the root and master location server nodes are used only to set up the experiment, and are not used during the experiment to locate a GlobeDoc object's replica. Likewise, because the location service tree server is used only when a location service node starts up, the tree server is used only to set up of the experiment and is not used during the experiment itself.

A run of the experiment is identified by five parameters as shown in Table 6.1. In this table, n, s, and g identify the Web document being requested, i specifies the length of the delay between Web document requests, and r specifies the number of complete Web document requests performed.

Parameter	Description
n	Number of elements in the requested Web document.
s	Total size of the requested Web document (in kilobytes).
g	Granularity of a the requested Web document.
i	Delay between Web document requests (in seconds).
r	Number of Web document requests to perform.

Table 6.1: Experiment run parameters.

6.3 Measurements

Two kinds of data are logged during a run of the experiment: the round trip time (RTT) for each Web document request and the amount of CPU time used by each component to process the requests. Logging the RTT is reasonably straightforward and is done by the client. The client records the time when a Web document request is started, it records the time when the request is completed, and then it calculates the elapsed time. Determining the CPU time used by each component is more complicated and requires the **monitoring** of a component's activity during request handling.

There are three possible approaches to monitoring a component's activity. The first approach is to strategically place timers in a monitored component's code. With this approach, whenever a monitored part of the component's code is executed, the timers keep track of the time spent in that part of the code. By placing the timers in parts of the code that are responsible for processing requests, it is possible to determine exactly how much time each component spends processing requests. By differentiating between timers placed in different parts of the code (e.g., parts responsible for network communication, parts responsible for file system access, etc.), it is possible to get a detailed breakdown of the work done while processing requests. Such a detailed breakdown is often used to spot bottlenecks and find code that needs to be optimized. There are two drawbacks to this approach. First, it is very time consuming (and error-prone) to insert timers into existing code. Second, the timers are not always accurate and are susceptible to the effects of other system activities. For example, if a process is stopped by the scheduler while in a timed part of the code, the timers will register more work being done than was actually the case.

Execution profiling is another approach to monitoring a component's activity. In execution profiling, a program is run in an environment that periodically interrupts the running program to gather details of the program's current state. These details are logged and analyzed to determine which function was being executed at the time, what data structures were being used, how much memory was allocated, etc. When the program is finished executing, the logged data are analyzed to provide an overview of the program's execution. Execution profiling is often used to provide a detailed analysis of the functions called during the execution of the program as well as the frequency and duration of those calls. Given such a detailed analysis and access to the profiled program's source code it is possible to determine the amount of time spent in the parts of the code responsible for specific

behavior (such as processing requests). A major benefit of this approach compared to the timer approach is that it is not necessary to modify the program in order to profile it.

There are, however, a number of drawbacks associated with the use of an execution profiler. First, the data collected is generally quite fine grained. This means that analyzing the output can be painstaking and usually requires intimate knowledge of the code being profiled. Second, because it periodically interrupts the running program to collect data, the profiler degrades the performance of the program being profiled. Third, because the profiler collects data at specified intervals, time spent executing small blocks of code that are short enough to fall between two consecutive profiling intervals may be missed. Missing such small blocks of code may skew the results produced by the profiler.

A third approach to component activity monitoring is to monitor system behavior using data made available by the operating system. In this approach a component's activity is monitored by periodically logging operating statistics (such as, running time, CPU time used, memory used, etc.) for its processes.[1] Because the data is gathered at the process level (e.g., CPU time used per process, as opposed to CPU time used per function), the data provided is of a coarser grain than that provided by the previous two approaches. On the other hand, when analysis of fine-grained data is not required, it is much easier to get an overall view of a component's activity using this method. Another benefit to this approach is that access to, or modification of, a monitored program's source code is not required. This means that the effort required to monitor a component using this method is much lower than in the previous approaches.

Because we are looking for an overall view of how components process requests, and because modifying or analyzing all of GlobeDoc's source code would be a unwieldy undertaking, we decided to use this third approach to monitor a component's activity. During a run of the experiment component activity data is collected by a monitoring program that regularly copies every running process's stat file from the /proc file system.[2] The stat file provides a snapshot of a process's status at the time that the file is read. Note that because each component runs on a separate computer, a separate monitoring program must be run for each component.

When an experiment run is finished and the components stopped, the data collected by the monitor programs is filtered and analyzed. Filtering the data for a single component involves removing data not relevant to the component (i.e., data from processes not related to the component) and isolating the parts of the data that are of interest in this experiment (e.g., CPU time used). Analyzing the data involves analyzing process hierarchies and calculating total CPU time used by the component's main process and all of its children at every measurement point.

This initial analysis of the monitor data provides separate activity logs containing the CPU usage data for each component. Each line in a log file represents a single measure-

[1] A single component may be implemented by more than one process. On Linux, for example, a component usually consists of many active processes due to the implementation of (Java) threads as separate processes.

[2] On Linux the /proc file system is a virtual file system that provides general and per process operating information in the form of files. This allows data to be gathered from the operating system using regular file based operations rather than obscure, kernel specific system calls. For more details about the /proc file system see [65].

ment point and specifies the absolute time at which the measurement was taken, the time since the program started, the time since measurement started, the total CPU time used so far, and the percentage of available CPU time used since the last measurement. All time measurements are recorded in (fractions of) seconds.

Besides an activity log file the client component also creates a separate log file where it stores information about requests performed and the RTT of those requests. This file contains a record for each Web document request as well as a record for each element request. Each record specifies the time that the request was sent, the time that a reply was received, and the RTT of the request.

After the monitor data has been filtered and analyzed, the next step is to calculate the work done by each component due to request processing. The easiest approach would be to match request information from the client log file to the data from the component activity logs. Thus, given the start and end times of a request the data corresponding to that time period would be extracted from each component's activity log and used to calculate the work done processing that request. Doing this for every request performed and taking the average of the results would provide per component values for the average work done to process a request. Unfortunately, in a distributed environment such as the DAS-2 where the clocks of the separate computers are not synchronized, this approach is not feasible. Instead, the approach taken is to determine the work done by a component over a given period of time and compare that to the number of requests performed in a similar period of time. This approach provides the same results (average work done to process a request) but does not rely on synchronized clocks.

Concretely, for each component, the CPU time used is calculated for consecutive blocks of 60 seconds.[3] Next, the average over these blocks is taken providing, for each component, a value for the average CPU time used per 60 seconds. Following this the average number of requests made in 60 seconds is calculated using data from the client logs. Finally, for each component, the CPU time used per 60 seconds is divided by the number of requests performed in 60 seconds, giving the average CPU time used (or work done) per request.

Note, that by working with fixed-size time blocks the exact starting and stopping times of the client and components do not have to be the same. In fact, in this approach, it is best to cut off the head and tail of the logs as the head may include initialization activity unrelated to request processing, while the tail may include (idle) activity after all requests have been processed.

A problem encountered with this approach to component activity monitoring is that it measures both the activity due to request processing as well as activity that is independent of request processing. This is illustrated in Figure 6.3, which shows a plot of the gateway component's activity over time. The Y axis of this graph represents the percent of available CPU time used and the X axis represents the time at which measurements were made (relative to the start of the experiment). In this graph we see that the component generally performs a steady amount of work, interspersed with a few (somewhat regularly

[3] Depending on the experiment parameters, a client can perform between 20 and 400 requests in a 60 second block. This provides a large enough sample size of measurement points to confidently calculate the average CPU usage per request.

distributed) peaks of greater CPU utilization. The frequency of these peaks seems to be unrelated to the frequency of the requests processed (the requests were generated approximately every 0.2 seconds, whereas the peaks occur approximately every 50 seconds).

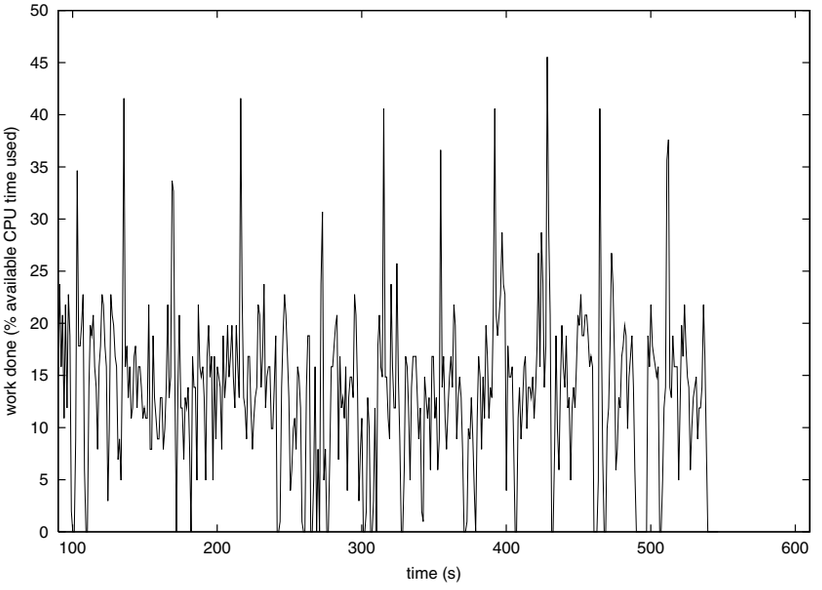


Figure 6.3: Work performed by a component when processing requests.

We presume that these peaks are caused by background work (such as running the Java garbage collector for the components written in Java) done by the component that is not related to request processing. To verify the hypothesis that these peaks represent background work, a similar experiment was run with no client requests generated (and thus no request processing done by the components). The results (once again for the gateway) from this run of the experiment are shown in Figure 6.4. This graph shows similar peaks to those seen in Figure 6.3. Note, however, that other than these peaks, the component performs no other work.

The work done by a component when not processing requests is referred to as that component's **background activity**. In order to determine how much work a component does due to request processing alone this background activity must be filtered out from the total request processing activity. To do this it is first necessary to calculate the amount of work done due to background activity.

To calculate the background activity the experiment is run without performing any client requests and with component activity monitored as described above. After the experiment run is completed the monitor data is analyzed providing, for each component, a value for the average amount of work done (per 60 seconds) due to background activity. To determine the amount of work a component does due to request processing, we subtract the average work (per 60 seconds) due to background activity from the average

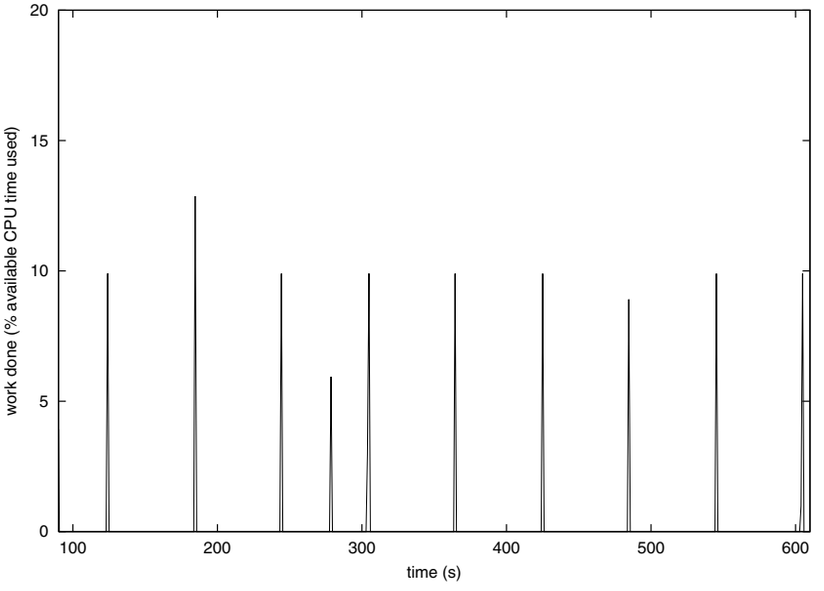


Figure 6.4: Work performed by a component when not processing requests.

work (per 60 seconds) measured while processing requests. Dividing this value by the number of requests performed in 60 seconds provides a value for the average amount of work done per request solely due to request processing.

6.4 Experiments and Results

This section presents an overview of all the profiling experiments run and gives a detailed description of the experiment parameters and results. Two groups of experiments were performed. The first has been briefly described above and provides the data used to filter out noise due to background activity. The second provides the main profiling data for the experiment. It consists of many runs of the experiment each using different values for the parameters.

6.4.1 Experiment 1: Background Activity

In this group of experiments a single run of the experiment with no client requests was performed. As explained above, this experiment provides values that are used (in subsequent experiments) to filter out noise due to background activity. Because no Web requests are performed, this experiment takes only one parameter: the duration of the experiment. The duration parameter specifies how long the components will run before being terminated.

Results

The experiment was run with a duration parameter of 16 minutes. Note that although only a single run needs to be performed to measure the background activity, multiple runs were actually performed and the average background activity calculated. This allows the stability of the measured values to be verified. Each component's average over the multiple runs is presented in Table 6.2. We use the calculated mean values as the base background activity in the following experiments. Note that the standard deviations in this table are all low (compared to the means), which shows that the values between runs did not vary significantly. It is also interesting to point out that the naming service, the only nonJava process, does not have background activity.

	Redirector	Translator	Gateway	GOS	NS	LS
mean	0.02413	0.02167	0.12867	0.13313	0	0.11280
std. dev.	0.00712	0.00781	0.00735	0.00625	0	0.00724

Table 6.2: Results from experiment 1. Note that the GOS column refers to the replica object server and the LS column refers to the replica location service leaf node.

6.4.2 Experiment 2: System Performance

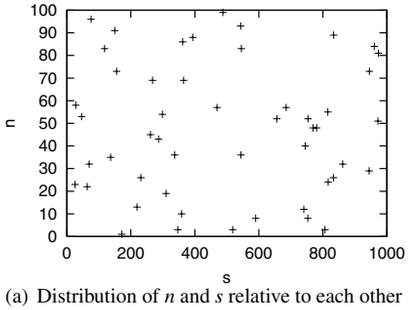
In this group of experiments, a number of runs were performed, each with different experiment parameters. The values of parameters used were chosen randomly per run such that all types of combinations of n, s, and g are accounted for. For example, during runs of the experiment large documents consisting of few elements and large documents consisting of many elements should be encountered. Likewise small Web documents consisting of few elements and small documents consisting of many elements should also be encountered.

The plots in Figure 6.5 show the distribution of the experiment parameters relative to each other. In this experiment the n parameter was distributed between 1 and 100, the s parameter between 1 and 1000 and g between 1 and 100. Note, however, that g has an upper bound, namely the total number of elements in a Web document (a Web document cannot span more GlobeDoc objects than it has elements). The plots show that the parameters are all well distributed over the available parameter space.

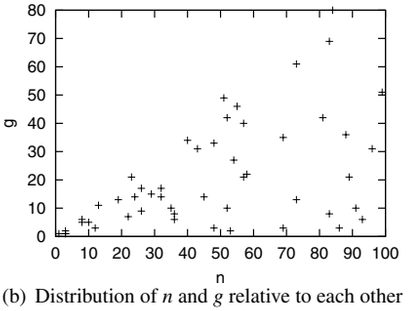
In all the runs the delay between Web document requests is kept constant at 0.1 seconds. Also, the number of requests actually performed varies between 2400 and 240 depending on the parameters. This is done to keep the duration of an experiment run reasonable (under one hour - including post-processing and analysis) and is based on previously observed effects of the experiment parameters on the request RTT.

Results

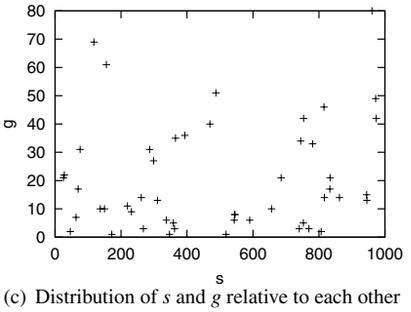
The raw results of the experiment are presented in the form of scatter plots of the work done by a component against the experiment parameters. For example, Figure 6.6 shows



(a) Distribution of *n* and *s* relative to each other



(b) Distribution of *n* and *g* relative to each other



(c) Distribution of *s* and *g* relative to each other

Figure 6.5: Distribution of experiment parameter values in their parameter spaces.

plots of the work done by the translator component against the experiment parameters *n*, *s*, and *g*. These plots suggest a strong relationship between the work done and *n* and a possible relationship between the work done and *g*. It is not clear from the plots, however, whether there is a relationship between the the work done by the translator and the *s* parameter. Result plots for other components will be presented later when analyzing the results for each specific component.

The relationship between the experiment parameters and the work done by a component can be modeled by the following equation.

$$work = C_n * n + C_s * s + C_g * g + K$$

This equation represents a **cost function** where K is a constant and C_n, C_s, and C_g are coefficients that specify the relative importance of each parameter in determining the amount of work a specific component does. For some components one or more of the coefficients may have a value of zero. This means that the corresponding parameters do not affect the amount of work done by that component.

The values of the coefficients in these functions (i.e., the statistical model for the component) are calculated using multiple linear regression analysis [35] [52]. A regression analysis is performed for each component involved in request processing. In each analysis, the work done per request is taken as the response variable, while the relevant param-

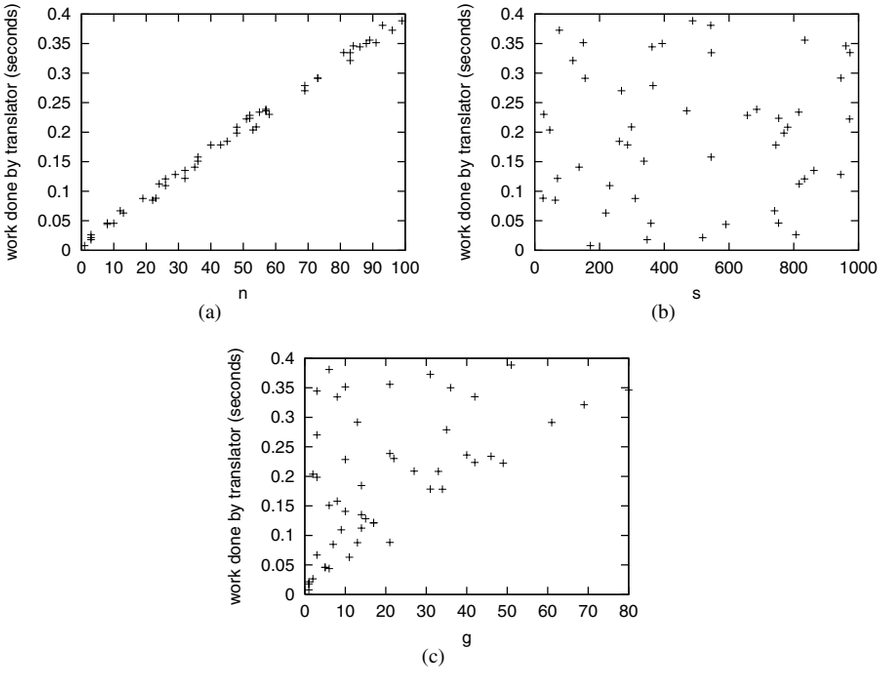


Figure 6.6: Work done by the translator component.

eters n, s, and g are taken as the predictor variables (or factors). The regression analysis is performed hierarchically, starting with the n parameter, then adding the s parameter and finally the g parameter. The regression analysis results in a model that provides values for K and each of the three coefficients in the cost function.

Details of the analysis results are presented separately for each component. These details include two tables, one presenting the coefficients of the calculated model and one containing a summary of the model's validity. The first table presents the coefficient for each predictor variable, as well as the results of the null-hypothesis test and the significance of this test. The null-hypothesis test is a t-test that tests how much the calculated coefficient differs from zero. The result of the t-test, the t value, represents the difference between the calculated coefficient and zero; the higher the t value, the greater the difference between the coefficient and zero. The value in the significance column is based on the t-test result and tells us the probability that the given difference (i.e., the t value) could have occured by chance.[4] Coefficients with significance values less than 0.05 are considered to be significantly different from zero, those with a significance greater than 0.05 are not significantly different from zero and can be ignored.

[4]The higher the t value, the lower the probability of it occuring by chance.

The second table shows the R, and R squared values for the analysis. These values indicate how much the model contributes to any observed differences between cases. A high R value means that most observed differences can be accounted for by the model. A low R, on the other hand, means that a significant part of the differences cannot be accounted for by the model. The R squared value represents the amount of variance in the model relative to the amount of variation in the measurements. Multiplying R squared by 100 gives us the percentage of variation in the measurements that can be explained by the model. An R squared value of 0.95, for example, means that 95% of the variation between cases can be explained by the model.

Analysis 1: Redirector

By design the redirector is used only once per Web request. This means that the work it does per request must be independent of the *n*, *s*, and *g* parameters. As such, a regression analysis of the redirector is not necessary, instead it is sufficient to present the average work done per request. This (along with variation and standard deviation) is shown in Table 6.3. Because it is in not dependent on any of the experiment parameters, the equation for work done by the redirector is a constant, namely, the mean from Table 6.3. Thus:

$$work\_red = 2.456 * 10^{-3}$$

Mean	Variance	Standard deviation
$2.456 * 10^{-3}$	$5.424 * 10^{-7}$	$7.365 * 10^{-4}$

Table 6.3: Statistical summary of the redirector.

Analysis 2: Translator

Presented earlier, Figure 6.6 shows plots of the work done by the translator compared to all of the experiment parameters. These plots suggest that there is some sort of relationship between the translator and the experimental parameters. As such it is it is sensible to perform a regression analysis on the translator's activity data. The regression analysis is performed with the translator's average work done per second as the response variable and the *n*, *s*, and *g* experiment parameters as the predictor variables (or factors).

The results of the regression analysis are shown in Table 6.4 and Table 6.5. Table 6.4 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that both the *n* and the *s* coefficients are significantly different from zero, while the *g* coefficient and the constant are not. This means that only the *n* and *s* parameters play a significant role in affecting the work done by the translator. The equation for work done by the translator therefore becomes:

$$work\_trans = 3.843 * 10^{-3} * n + 2.014 * 10^{-5} * s$$

Table 6.5 shows the R, and R squared values for the model. These values are high which, in this case, means that 99.8% of the variation in the measurements can be explained by the model (and therefore by the effect of the *n* and *s* factors).

Factor	Coefficient	t-test	Significance
Constant (K)	$2.356 * 10^{-3}$	1.348	.184
C_n	$3.843 * 10^{-3}$	135.522	.000
C_s	$2.014 * 10^{-5}$	8.574	.000
C_g	$4.405 * 10^{-5}$	1.006	.320

Table 6.4: Coefficients for regression analysis of the translator.

R	R squared
.999	.998

Table 6.5: Model summary for regression analysis of the translator.

Analysis 3: Gateway

Figure 6.7 shows plots of the work done by the gateway against the experiment parameters. Like the translator, these scatter plots show a possible relationship between the work done and the experiment parameters. It is sensible, therefore, to perform a regression analysis on the gateway's activity data. The regression analysis is performed with the gateway's average work done per second as the response variable and the *n*, *s*, and *g* experiment parameters as the predictor variables (factors).

The results of the regression analysis are shown in Table 6.6 and Table 6.7. Table 6.6 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that all the coefficients are significantly different from zero. This means that all the parameters, *n*, *s*, and *g*, and the constant play a significant role in affecting the work done by the gateway. The equation for work done by the gateway becomes:

$$work\_gatew = 3.314 * 10^{-3} * n + 6.442 * 10^{-5} * s + 8.984 * 10^{-3} * g + 2.891 * 10^{-2}$$

Table 6.7 shows the R, and R squared values for the model. These values are high which, in this case, means that 99.4% of the variation in the measurements can be explained by the model (and therefore by the effect of the *n*, *s*, and *g* factors).

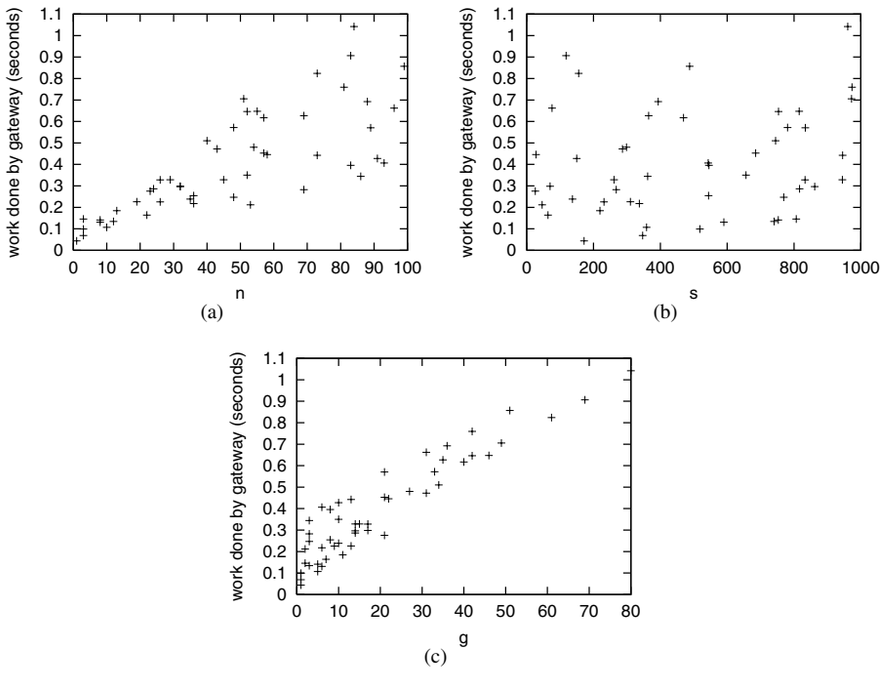


Figure 6.7: Work done by the gateway component.

Factor	Coefficient	t-test	Significance
Constant (K)	$2.891 * 10^{-2}$	4.407	.000
C_n	$3.314 * 10^{-3}$	31.123	.000
C_s	$6.442 * 10^{-5}$	7.306	.000
C_g	$8.984 * 10^{-3}$	54.643	.000

Table 6.6: Coefficients for regression analysis of the gateway.

R	R squared
.997	.994

Table 6.7: Model summary for regression analysis of the gateway.

Analysis 4: Naming Service

Plots of the work done by the naming service compared to the experiment parameters are shown in Figure 6.8. These plots show a clear relationship between the work done and the g parameter. As with the previous two components it is sensible to perform a regression

analysis on the naming service's activity data. The regression analysis is performed with the naming service's average work done per second as the response variable and the *g* experiment parameter as the predictor variable (or factor). Note that the *n* and *s* parameters are not included in the analysis. This is because the naming service does not deal with the actual requests and therefore the number of requests and size of the elements requested cannot affect the amount of work that it does. Although in Figure 6.8(a) there seems to be a relationship between the work done by the naming service and *n*, this only reflects the fact that the values of *g* are bounded by *n*.

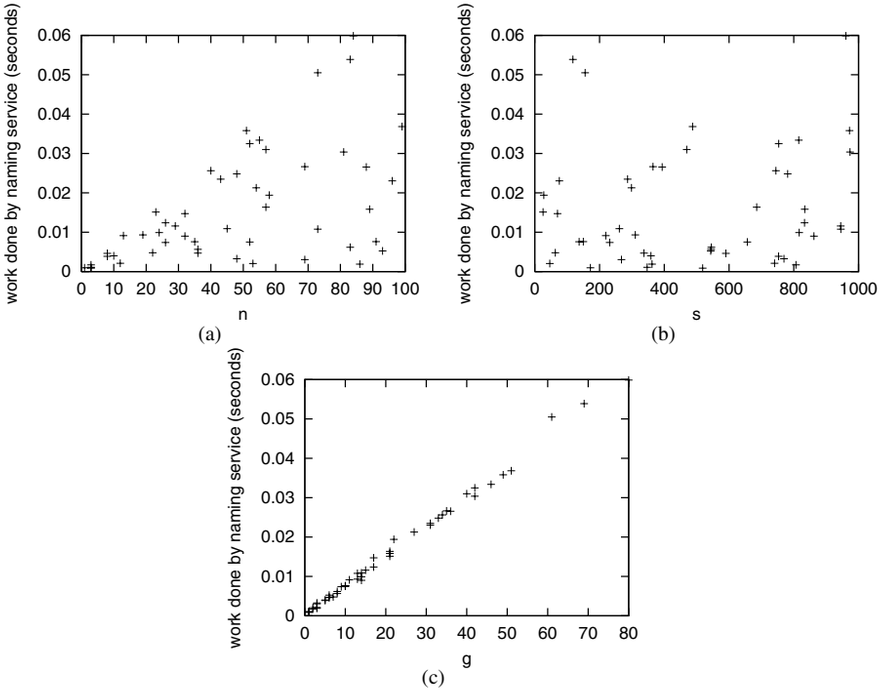


Figure 6.8: Work done by the naming service component.

The results of the regression analysis are shown in Table 6.8 and Table 6.9. Table 6.8 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that only the *g* coefficient is significantly different from zero. The equation for work done by the naming service becomes:

$$work\_ns = 7.577 * 10^{-4} * g$$

Table 6.9 shows the R, and R squared values for the analysis. These values are high which, in this case, means that 99.5% of the variation in the measurements can be explained by the model (and therefore by the effect of the *g* factor).

Factor	Coefficient	t-test	Significance
Constant (K)	$1.320 * 10^{-4}$	.602	.550
C_g	$7.577 * 10^{-4}$	96.437	.000

Table 6.8: Coefficients for regression analysis of the naming service.

R	R squared
.997	0.995

Table 6.9: Model summary for regression analysis of the name server.

Analysis 5: Location Service

Figure 6.9 shows plots of the work done by the location service components over time (the plots include background activity). From these graphs it is clear that only the replica leaf node performs any work during a Web request. It is therefore only necessary to perform an analysis on the replica leaf node data. Figure 6.10 shows plots of the work done by the replica leaf node and the experiment parameters. As with the naming service, there is a clear relationship between the work done by this leaf node and *g*. The regression analysis is performed with the location service's average work done per second as the response variable and the and *g* experiment parameter as the predictor variable (or factor). Note that the *n* and *s* parameters are not included in the analysis. This is because the location service, like the naming service, does not deal with the actual requests and therefore the number of request and size of the elements requested cannot affect the amount of work that it does.

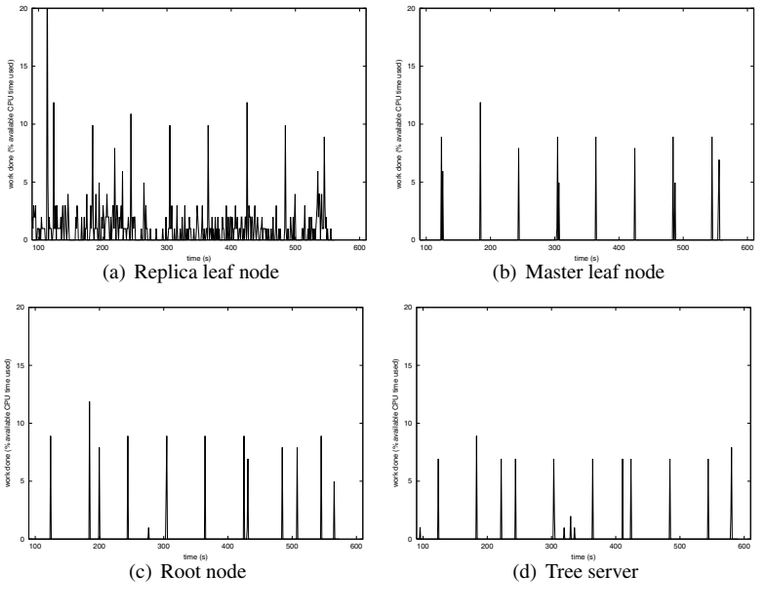
The results of the regression analysis are shown in Table 6.10 and Table 6.11. Table 6.10 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that both the *g* coefficient and the constant are significantly different from zero. The equation for work done by the location service becomes:

$$work\_ls = 9.925 * 10^{-4} * g + 6.958 * 10^{-3}$$

Table 6.11 shows the R, and R squared values for the analysis. These values are high which, in this case, means that 96.2% of the variation in the measurements can be explained by the model (and therefore by the effect of the *g* factor).

Factor	Coefficient	t-test	Significance
Constant (K)	$6.958 * 10^{-3}$	8.780	.000
C_g	$9.925 * 10^{-4}$	34.957	.000

Table 6.10: Coefficients for regression analysis of the location service replica leaf node.



(a) Replica leaf node (b) Master leaf node

(c) Root node (d) Tree server

Figure 6.9: Work done by the location service components.

R	R squared
.981	.962

Table 6.11: Model summary for regression analysis of the location service leaf node.

Analysis 6: Replica Object Server

Scatter plots of the experiment results for the replica object server are shown in Figure 6.11. These plots suggest a relationship between the work done by this component and the *n* and *g* parameters. It is unclear whether there is a relationship with the *s* parameter. We perform perform a regression analysis on the replica object server's activity data to shed light on the possible relationships. The regression analysis is performed with the object server's average work done per second as the response variable and the *n*, *s*, and *g* experiment parameters as the predictor variables (factors).

The results of the regression analysis are shown in Table 6.12 and Table 6.13. Table 6.12 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that all three parameters and the constant are significantly different from zero. This means that *n*, *s* and *g* play a significant role in affecting the work done by this object server. The equation for work done by the replica object server becomes:

$$work\_repl = 6.287 * 10^{-4} * n + 6.327 * 10^{-5} * s + 2.445 * 10^{-3} * g + 2.499 * 10^{-2}$$

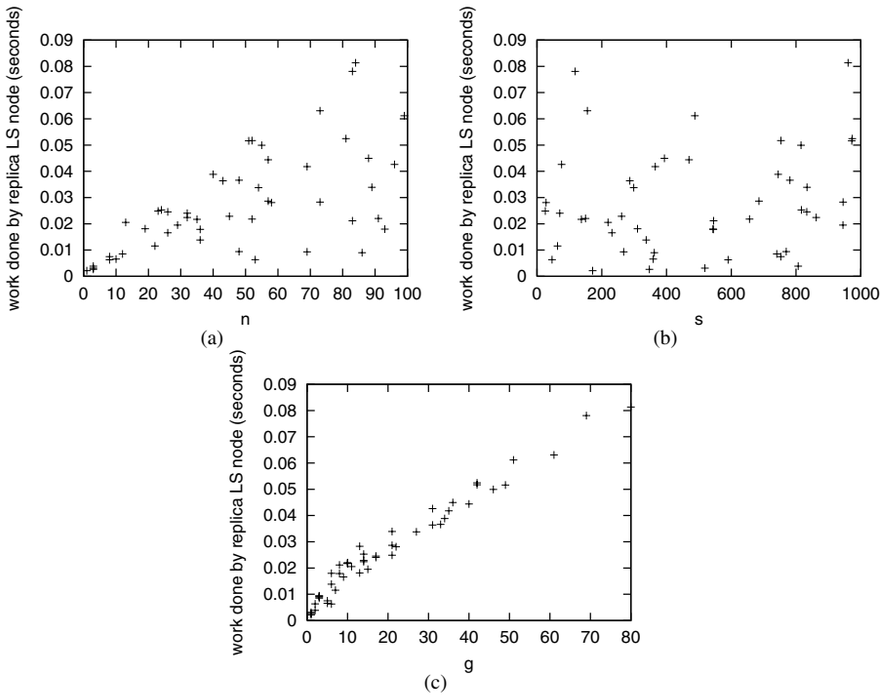


Figure 6.10: Work done by the (replica) location service leaf node component.

Table 6.13 shows the R, and R squared values for the analysis. These values are high which, in this case, means that 87.6% of the variation in the measurements can be explained by the model (and therefore by the effect of the *n* and *s* factors).

Factor	Coefficient	t-test	Significance
Constant (K)	$2.499 * 10^{-2}$	2.852	.006
C_n	$6.287 * 10^{-4}$	4.419	.000
C_s	$6.327 * 10^{-5}$	5.370	.000
C_g	$2.445 * 10^{-3}$	11.129	.000

Table 6.12: Coefficients for regression analysis of the replica object server.

Analysis 7: Master Object Server

Figure 6.12 shows a plot of the work done (including background activity) by the master object server over time. From this graph it is clear that this object server does not perform

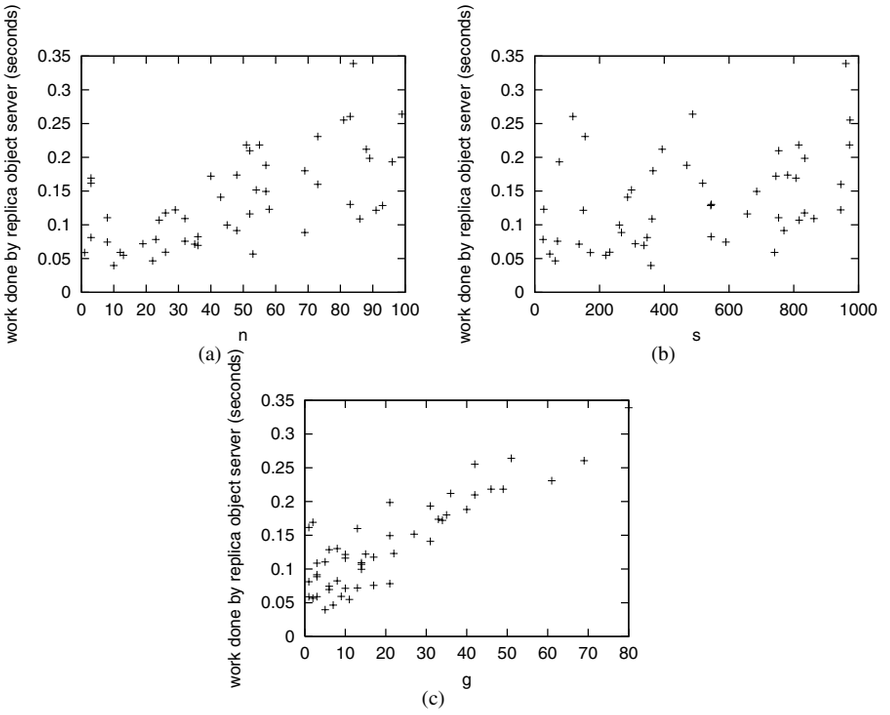


Figure 6.11: Work done by the replica object server component.

R	R squared
.936	.876

Table 6.13: Model summary for regression analysis of the replica object server.

any work related to Web document requests. As such it is not necessary to perform a regression analysis on the master object server data. The equation for work done by the master object server becomes:

$$work\_master = 0$$

reflecting the fact that the master object server is not involved in processing Web document requests.

Analysis 8: Request Round Trip Time

As with work done by the individual components it is also possible to make a cost function that relates RTT to the experiment parameters. Determining the coefficients for this

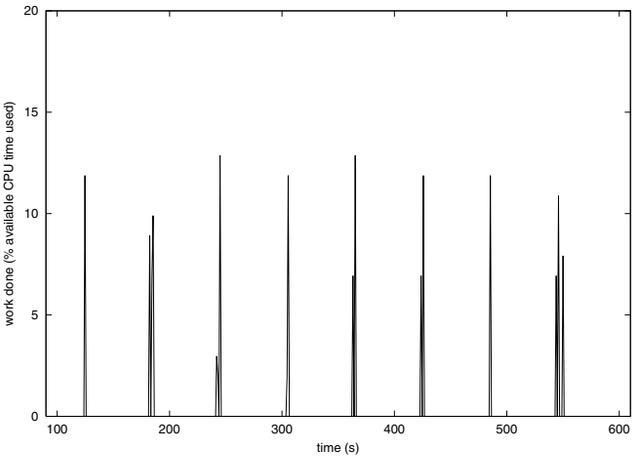


Figure 6.12: Work done by master object server while processing requests.

cost function is also done using multiple linear regression. The regression analysis is performed with the RTT per request as the response variable and the *n*, *s*, and *g* experiment parameters as the predictor variables (or factors).

The results of the regression analysis are shown in Table 6.14 and Table 6.15. Table 6.14 shows the coefficient for each factor, as well as the results of the null-hypothesis test and the significance of this test. Note that all the coefficients (except the constant) are significantly different from zero. This means that all of the *n*, *s*, and *g* parameters play a significant role in affecting the RTT. The equation for RTT becomes:

$$RTT = 1.820 * 10^{-2} * n + 6.174 * 10^{-4} * s + 1.467 * 10^{-2} * g$$

Table 6.15 shows the R, and R squared values for the analysis. These values are high which, in this case, means that 86.9% of the variation in the measurements can be explained by the model (and therefore by the effect of the *n*, *s*, and *g* factors).

Factor	Coefficient	t-test	Significance
Constant (K)	$-6.451 * 10^{-2}$	$-.603$	.550
C_n	$1.820 * 10^{-2}$	10.476	.000
C_s	$6.174 * 10^{-4}$	4.290	.000
C_g	$1.467 * 10^{-2}$	5.467	.000

Table 6.14: Coefficients for regression analysis of the RTT.

Note that the coefficient for *s* implies a transfer rate of 1.5Mbit/sec. This is very low compared to the 100Mbit/sec capacity of the network used. Recent work on the Globe

R	R squared
.932	.869

Table 6.15: Model summary for regression analysis of the RTT.

implementation has found and removed a significant bottleneck with regards to transfer of data that has significantly improved the transfer rate.

6.5 Conclusion

This chapter examined the performance of GlobeDoc by profiling the system components to determine the amount of work each component does while handling a single Web document request. Based on the measurements made, cost functions relating Web document properties to the amount of work required to process a request for that Web document have been derived.

Based on these cost functions, it is possible to determine the relative amount of work done by each component. This requires calculating the amount of work done by each component as a percentage of total work done by all the components. Based on the cost functions it is also possible to examine the effects that the various Web document request parameters have on the relative amount of work done by the components. Together, this provides us with a good overview of which components do the most work when processing requests and, therefore, a starting point for code optimization. Figures 6.13, 6.14, and 6.15 show how the percentage of total work done changes as a function of the individual experiment parameters.

Figure 6.13 shows that as the number of elements in a Web document increases that most of the work load increase is experienced by the translator and gateway components. Figure 6.14 shows that as the total size of a Web document increases, the object server and the gateway components end up performing a majority of the request processing work. Likewise Figure 6.15 shows that when a Web document spans many GlobeDoc objects most of the work is done by the gateway component. In all three of these graphs we see that the gateway is one of the most active components.

Reducing the amount of work done by the gateway should, therefore, be the first priority when optimizing the GlobeDoc architecture. Furthermore we see that keeping the granularity of the Web documents low (that is storing most if not all of a Web documents elements in a single GlobeDoc object) reduces the number of binds needed and hence the work done by the gateway.

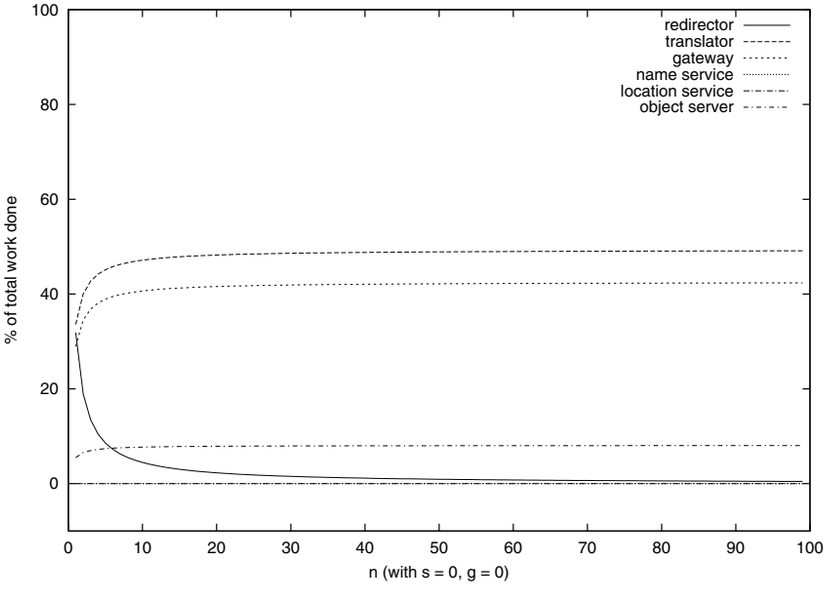


Figure 6.13: Overview of the relative work done by components as the *n* parameter increases.

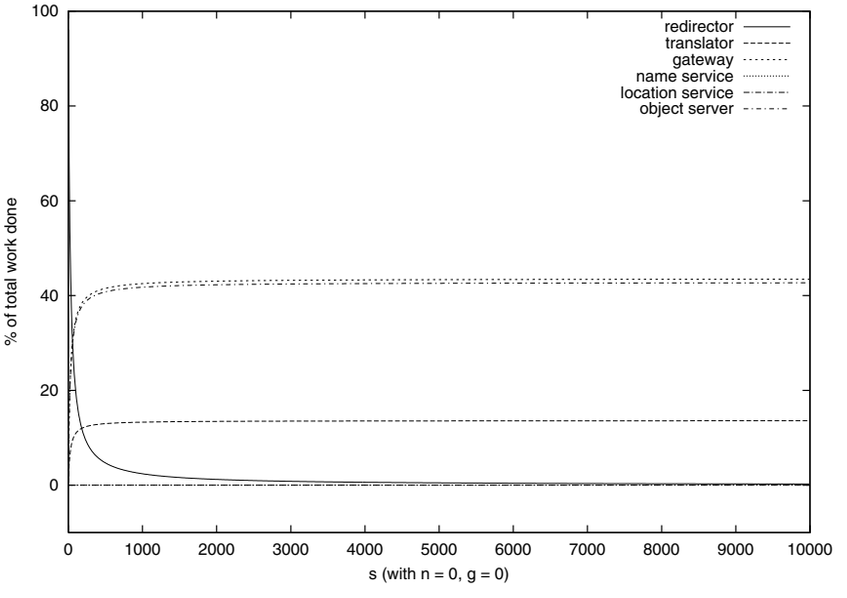


Figure 6.14: Overview of the relative work done by components as the *s* parameter increases

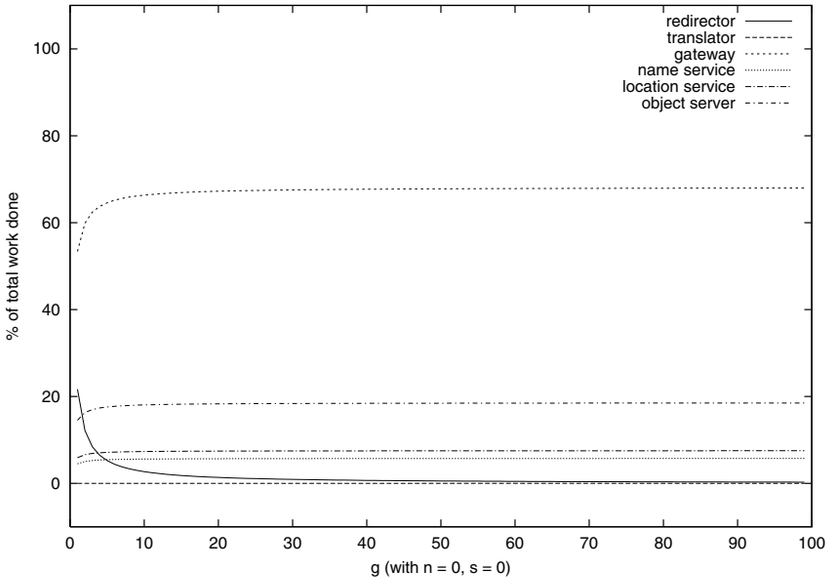


Figure 6.15: Overview of the relative work done by components as the g parameter increases

Chapter 7

Summary and Conclusions

7.1 Summary

The performance problems plaguing the World Wide Web were introduced in Chapter 1. Among these problems are those caused by poor scalability of its architecture. In particular there are the problems of scaling with regards to the geographic distribution of clients as well as scaling with regards to the number of client requests a single server must process. It was argued that an effective solution to these scalability problems must localize traffic, that is, it should reduce the distances between clients and the content they wish to access, and reduce the number of requests a single server must process.

An effective technique for achieving localization is replication. Although replication has been used as the basis of many proposed solutions to the Web's performance problems, it is usually provided it as a one-size-fits-all solution, that is, the replication must be applied universally to all Web documents. The central claim of this dissertation is that, in order to be scalable, it is not enough for content to be replicated. The underlying architecture must be flexible enough to allow replication policies (and associated coherence policies) to be applied on a case-by-case basis. The first chapter closed by introducing GlobeDoc as such an architecture.

Chapter 2 discussed current approaches to solving the Web's scalability related performance problems. These solutions are divided into three categories: client-oriented solutions such as browser and proxy caching, server-oriented solutions such as server caching and clustering, and replication-based solutions such as mirroring and content delivery networks. A common characteristic shared by all these solutions is that they take a one-size-fits-all approach to applying relevant policies. In order to illustrate the inferiority of one-size-fits-all solutions, this chapter presented a simulation-based experiment that compared the performance of per-document replication policies to one-size-fits-all policies. The results of this experiment show that per-document policies provide better (i.e., closer to optimal) performance gains than one-size-fits-all policies.

GlobeDoc, the approach for a scalable Web infrastructure covered in this dissertation, was presented in Chapter 3. The GlobeDoc architecture is based on Globe distributed

shared objects. Globe's key characteristic is that every Globe object is responsible for determining and implementing its own replication and coherence policy. This property of Globe allows GlobeDoc to provide support for per-document replication policies.

The GlobeDoc model groups logically related Web resources into Web documents. Each Web document is represented by, and contained in, a GlobeDoc object (which is implemented as a Globe distributed shared object). Web resources are stored in GlobeDoc objects as elements. Every GlobeDoc object implements methods that allow clients to access and modify the object's elements. However, in order to invoke methods on a GlobeDoc object, a client must first bind to the object. Binding causes a local representative of the GlobeDoc object to be placed in the client's address space. The client invokes methods on this local representative and the GlobeDoc object's replication and coherence policies determine how that invocation is processed. A method invocation may, for example, cause a message to be sent to all of a GlobeDoc object's replicas, or it may simply cause local state to be updated.

The GlobeDoc architecture provides the runtime support and external services required to access and use GlobeDoc objects. The architecture also provides all the services necessary to access GlobeDoc objects from traditional (i.e., GlobeDoc-unaware) Web clients. The GlobeDoc architecture was presented in detail in Chapter 4. This chapter described the design and implementation of key components and services of the GlobeDoc architecture. These include the object server, the implementation repository, clients, and support for shared local replicas. Core services, such as the naming service and the location service, which were not designed as part of the GlobeDoc project are described elsewhere [47] [12].

The Globe infrastructure directory service (GIDS) is a meta service that keeps track of all other essential GlobeDoc services, it stores meta-data that describes other servers and the services that they provide. This meta-data can, for example, be used to find servers to host GlobeDoc objects and their replicas. GIDS was discussed in detail in Chapter 5. Besides a description of the design of GIDS, this chapter also describes an implementation of GIDS based on existing LDAP and DNS technologies.

Chapter 6 provided a performance evaluation of the architecture and its components. In particular this chapter evaluated the relative performance of each component in the architecture by profiling the component's CPU usage while processing client requests. This performance evaluation provided insight into the relative impact of each component on the total delay a client experiences when requesting a Web document using GlobeDoc. In particular it showed that the gateway component is crucial with regards to total performance as it is responsible for most of the work done. The evaluation also showed that grouping related Web resources into a single GlobeDoc object reduces the load on key components and leads to improved performance.

7.2 Conclusions

7.2.1 Evaluation

Chapter 1 outlined the requirements that an infrastructure wishing to provide an effective solution to the Web's performance problems must fulfill. Such an infrastructure must:

- Localize traffic
- Limit the load on any single server
- Decouple Web documents from the servers providing their contents.

Chapter 1 also specified the effects that an effective solution must have on performance. Performance, as perceived by a Web document's clients, must be:

- Maximized (i.e., connection time, latency, and delivery time must be minimized)
- Independent of the geographic distribution of clients
- Independent of a Web document's access patterns
- Independent of any other Web document's access patterns.

The following sections evaluate GlobeDoc based on these requirements.

Localize traffic

When traffic is localized, client requests and associated replies travel only a short distance over the network. GlobeDoc achieves localization of traffic in two ways. The first is by widely distributing Globe access points (GAPs) and allowing each client to communicate with its nearest GAP. Ensuring that GAPs are widely distributed increases the chances of a GAP being close to any particular client, thus reducing the average distance between client and GAP. By running a GAP locally (i.e., on a user's workstation or on a machine on the local network), a user can effectively reduce the distance to the nearest GAP to zero. The second way that GlobeDoc achieves localization is by allowing content to be replicated on widely distributed object servers. Because a GAP will always connect to the nearest replica of a GlobeDoc object, replicating GlobeDoc objects on object servers close to expected clients considerably reduces the distance between replicas and GAPs. Depending on the replication policy implemented by the object, the choice of where to place a replica can be made manually by the GlobeDoc object owner, or automatically by the object itself. In order to effectively localize traffic, it is also important to have widely distributed object servers where replicas can be placed. GIDS is used to find object servers at appropriate locations.

Limit server load

In GlobeDoc, creating widely distributed replicas of GlobeDoc objects helps to limit the load on individual object servers. With widely distributed replicas, the number of requests processed by any single replica is kept low, thus reducing the load on the object server hosting the replica.

Ideally, in order to reduce a particular object server's load, it should also be possible to move a replica from a heavily loaded object server to a lightly loaded one. Although technically possible, this is a complicated procedure in the current implementation of GlobeDoc. Moving a replica requires that all clients connected to the replica first be unbound from it and then be rebound to the replica at its new location.

Decouple servers and Web documents

A client normally refers to a GlobeDoc object by its name (usually embedded in a URN). Because of this, and due to Globe's separation of naming, identity, and location, GlobeDoc objects are not bound to any particular server. As such, creating a new replica or destroying an existing replica affects only a small portion of clients (e.g., those connected to the destroyed replica). Clients connected to other replicas are not affected. Likewise, new clients can still bind to the object without having to use a new name. This is important because it makes it possible to add, remove, and move replicas during the lifetime of a GlobeDoc object without invalidating existing URNs for that object.

Maximized performance

Maximizing performance requires minimizing connection time, latency, and delivery time. For regular (GlobeDoc-unaware) Web clients, connection time refers to the time it takes to resolve a URL and contact the corresponding server. When accessing a GlobeDoc using an embedded URN this time is divided into three parts: the time it takes to contact the redirector, the time it takes to receive a reply from the redirector, and the time it takes to contact the GAP. The redirector, being a centralized component, is a potential bottleneck and can greatly increase a client's connection time (either because it is far away from the client, or because a heavy load delays request processing). The GlobeDoc architecture minimizes the impact of the redirector in two ways. By replacing the redirector's address with its own address in all embedded URNs that pass through it, the translator reduces the number of times a client must contact the redirector. In this way, a client contacts the redirector only once for every GlobeDoc object it accesses rather than for every element it accesses. In order to minimize the time it takes for the redirector to return a reply, the redirector stores information about the client's location and its nearest GAP in a cookie on the client's machine. Every time a client contacts the redirector the information in the cookie is used to return a quick reply. This way the redirector avoids having to look up the client's location and having to find its nearest GAP. Finally, the time it takes a client to connect to a GAP is minimized by having clients always connect to their nearest GAP and by placing GAPs as close to clients as possible.

For GlobeDoc-aware clients, connection time refers to the time it takes to resolve a GlobeDoc URN and bind to the corresponding GlobeDoc object. Caching object bindings, as described in Chapter 4, minimizes the effect of binding on connection time.

Latency is affected in two ways. It is affected by the latency of underlying networks, and it is affected by the time it takes a server to process a request and start returning results. In the first case latency is minimized by reducing the amount of traffic sent over higher latency networks (such as wide-area network links). This is a direct result of localizing network traffic.

In the second case the latency is affected by the time it takes for the gateway to bind to the appropriate GlobeDoc object, the time it takes for the gateway to request and receive the requested element. The time it takes for the gateway to bind to the requested object can be minimized by caching object bindings in the gateway. The time it takes the gateway to request an element from the replica and receive a reply is largely dependent both on the replication policy implemented by the object (e.g., whether the replica has to contact a master to verify the freshness of its state, whether the replica has to retrieve the state form another replica first, etc.) and the network connection between the gateway and the replica. By strategically placing replicas close to GAPs the effect of the network connection between the gateway and replica is minimized. Likewise, by choosing an appropriate replication policy, the amount of work done by the replica before an element can be returned can also be minimized (e.g., by transferring state to the replica as soon as a replica is created so that when a request comes in the state is already locally available). This will generally require some sort of tradeoff, for example, between speed and consistency guarantees.

Delivery time for Web clients is dependent on the time it takes to return an element from the GAP to the client. This is generally affected by the network link between the client and GAP. By localizing the network traffic, this time is kept to a minimum.

Independence of performance from geographic distribution of clients

Keeping performance independent from geographic distribution allows all clients (no matter where they are located) to experience similar levels of performance when accessing a Web document. This means that there are no clients closer to a Web document who experience better performance, or clients further away who experience worse performance. GlobeDoc provides this independence in two ways. By widely distributing GAPs a large number of potential clients can be brought closer to the access points. This helps to localize traffic, which helps to improve performance. Also, by placing replicas in areas where client requests are expected to come from, clients will generally find themselves close to a replica of the GlobeDoc object they are accessing. This also helps to localize traffic and thus helps to improve performance. Both help to maximize the number of clients that are close to a Web document and minimize the number of clients that are far away from a Web document.

Independence of performance from access patterns

When performance is independent from access patterns, a Web document's clients experience consistent levels of performance despite changing access patterns. Thus, for example, clients accessing different Web documents do not experience different levels of performance because one document is more popular than another. GlobeDoc allows access to Web documents to be independent from the document's access patterns through its flexible per-document approach to replication. By tailoring the replication policy used, as well as the placement of replicas, to the expected popularity of a document, a consistent level of performance can be provided (and maintained).

This works well for Web documents with stable access patterns, however, many Web documents have access patterns that change over time. It is important that the performance experienced by clients remains stable even as access patterns change. For example, clients accessing a Web document during a flash crowd should not experience a degraded level of performance compared to those accessing that document under normal circumstances. GlobeDoc's support for flexible application of replication policies makes this possible. Slowly changing access patterns can be handled by manually adding or removing replicas, while rapid and unexpected changes in access patterns must be handled by dynamic and adaptive replication policies. These are parameterized replication policies whose behavior can be adapted as access patterns change. Note that dynamic and adaptive replication policies have not been examined in this dissertation. As such, no claim can (yet) be made for the independence of GlobeDoc from access patterns. Studying the application of dynamic and adaptive replication patterns in GlobeDoc will be the subject of future work, and will be described briefly in Section 7.3.

Independence from other Web documents

When a Web document becomes popular, the number of request for its elements and therefore the load on the servers hosting the contents increases. Such an increase in popularity should not affect other, unrelated, Web documents. Thus, the increased load caused by a popular document should not affect access to other Web documents which may be hosted on the same server. A fundamental property of the GlobeDoc architecture is that GlobeDoc objects are not dependent on any single server. Because of this, replicas of popular Web documents can be placed on dedicated servers, so that the increased load does not affect the performance for other documents. These replicas can be added and removed as needed, depending on a document's popularity. Likewise, by widely replicating popular Web documents, the load on, and the traffic to and from, any single server can be kept at a reasonable level.

7.2.2 Observations

Designing and implementing GlobeDoc as a Globe-based Web architecture has provided some insights into building a scalable wide-area distributed application, as well as practical insights into building an application using Globe.

Separation of Concerns

An important software engineering principle to follow when building a scalable wide-area distributed system is separation of concerns [77]. This involves splitting a problem into a number of concerns and addressing each concern individually. It allows a developer to identify, encapsulate, and develop those parts of a system that are relevant to a particular concern. In Globe, for example, we see that the concerns of object naming, object identification and object location have been separated. This has led to separate services for naming and location, as well as separate means of identifying objects. Likewise by decomposing the local representative into a semantics, control, replication and communication subobject, the issues of replication and communication can be addressed separately from an object's semantics.

By separating concerns in this way, it is possible to create general solutions for issues that apply to all distributed systems. Designing GlobeDoc, for example, did not require the design of a new naming and location scheme. Furthermore implementations of replication and communication policies can also be shared between different object types. For example, both GlobeDoc and GDN [10] use the same implementations of replication and communication policies.

Separation of concerns is evident in the design of the GAP and GIDS. In the GAP the translator and gateway are designed as separate components because they have clearly independent tasks. Likewise, in GIDS, locating resource information and storing this information are separate concerns. This is reflected in the GIDS design where the location aspect is separated from the resource management aspect.

Hierarchies

Although a hierarchical structure is generally a good approach to providing scalability, it is not without problems. For one, hierarchies, as in the case of hierarchical caches, may cause more harm than good if they become too deep (i.e., when it takes longer to traverse the hierarchy than to simply connect to a central or remote server). Hierarchies also form a problem from an administrative point of view. A hierarchy (or at least the upper levels of the hierarchy) must generally be administered by a trusted organization or person. This means that to set up an effective worldwide hierarchy (as is necessary for the location service and for GIDS) it is necessary to find enough trustworthy nodes to form a stable base for the hierarchy. Also, in the case of the location service, a central node must administer the tree structure and always be available to new nodes joining the network. It would be interesting to look for alternative (nonhierarchical) models that could be used in the location service and GIDS. Such nonhierarchical models could, for example, make use of peer-to-peer networks and related distributed hash table technology [101, 93].

Integration of Different Architectures

Providing seamless integration between GlobeDoc and the World Wide Web has shown that it is difficult to integrate architectures that are based on fundamentally different models. The Web is based on a client-server model: resources are located on single servers and

clients request resources from these servers. In the Web, resources are identified by URLs, which identify a resource location. As such, the Web does not separate the concepts of naming and location. GlobeDoc, on the other hand, does separate the notions of name and location. A GlobeDoc object's name does not specify the location of that object, nor is an object bound to any single location. Seamlessly integrating these approaches to naming and location resulted in the introduction of the redirector, which introduced a centralized component (and potential bottleneck) to an otherwise decentralized architecture.

Besides different naming and location models, GlobeDoc and the Web also differ in their basic connection models (and protocols). This difference led to the introduction of the GAP. As mentioned earlier, in order for the GAP approach to be scalable, many GAPs have to be deployed, widely spread over the network. This network of GAPs is in fact a second network, parallel to the network of object servers required to host GlobeDoc objects. It also means that clients must find a nearest GAP, which requires the deployment of a second location service, besides the Globe location service. This second location service is implemented in the redirector and is a centralized service which maps IP addresses to locations (geographic coordinates) and then finds the closest GAPs.

Globe

Using Globe to build a distributed application has many benefits, the greatest being that Globe provides much of the architectural support and services needed to build such an application. In theory building a Globe application does not require a programmer to understand or worry about the distributed nature of the application. It should suffice to define an interface and implement a semantics object. In practice, however, the separation between an application's interface, its semantics, and its distributed nature is not quite as clean as desired (this has also been pointed out in [10]). When designing the interface for a Globe object, one must keep in mind that the object state may be remote and possibly replicated. This is reflected in the limited type support of the Globe IDL (e.g., no support for passing pointers or references). GlobeDoc's lock interface is an example of an interface where the semantics may fail depending on the distribution (and in particular the coherence policy) used. Likewise, when handling large data one must take into account the possible cost of such operations (e.g., that the operation might block for a long time if a large amount of data is sent over the network) and the various ways that such operations may fail. When implementing the semantics object there are also restrictions that remind the developer that the application may be distributed. For example, it is not possible to bind to other Globe objects from a within semantics object.

Because Globe requires many services to be run and because each component must connect to other remote components, it is not trivial to install, configure, and run a Globe site. In fact, despite clear documentation, setting up a Globe site is still a complex task. Setting up a Globe site from behind a firewall or NAT gateway is even more difficult. Work is being done to make this easier and good defaults as well as integration with GIDS has already made Globe setup easier. It is important that, in the future, globe and GlobeDoc clients can be installed and run without requiring the user to do any configuration.

7.3 Future Work

7.3.1 Replication Policies

During the development of GlobeDoc we looked at the subject of choosing appropriate replication policies. Initially an attempt was made to assign replication policies (or at least determine replication requirements) based on Web document characteristics (such as size, type of content, number of elements, etc.) [55, 56]. Unfortunately, it was found that these kinds of characteristics are not good predictors of a Web document's access patterns. In the end, replication policies were assigned based on common sense prediction of a Web document's popularity. Assigning policies in this way is, of course, not very accurate. A related project was set up to look at the possibility of predicting future access patterns based on analysis of past access patterns [82]. The results of this research showed that such prediction was possible and has led to the development of an **adaptive replication policy** (i.e., a policy whose behavior is modified by feedback about its past performance). Because an adaptive policy requires data about past access patterns to decide on an appropriate policy, there is a significant chance that the policy will perform badly when a Web document is first deployed. Given this start-up problem it would be interesting to re-examine the effect that a Web document's semantic characteristics have on its (initial) access patterns and replication requirements. If a relationship (even a weak relationship may be useful) between the two is found, this can be used to initialize the starting state of an adaptive policy. This would provide a starting scenario which was better than the common sense approach currently used, and would allow quicker convergence to an optimal situation.

7.3.2 GlobeDoc-aware Clients

Different approaches to developing GlobeDoc-aware clients were discussed in Chapter 4. Building and deploying one or more of the described clients would provide extra insight into the ease or difficulty of building GlobeDoc-aware applications. Also, because a GlobeDoc-aware client binds directly to an object, the redirector and GAP would not be required to access GlobeDoc content. Comparing the current GlobeDoc-unaware approach to a GlobeDoc-aware approach would, therefore, provide more insight into the actual cost of the GAP and redirector components. Furthermore, by bringing the LR into the client, GlobeDoc-aware browsers would also allow different replication policies to be explored. For example, policies that cache content locally in the client could replace the one-size-fits-all browser caching policies used in clients today.

7.3.3 Experiments

Chapter 6 presented experiments that examined the performance of individual GlobeDoc architectural components. The work done by these components is not the only factor that may affect the performance of the GlobeDoc system as a whole. Properties of the network connections between components will likely affect overall performance as well. It would

be interesting to perform experiments similar to those described in Chapter 6 but where characteristics (e.g., bandwidth, latency, etc.) of the network connections between the components are modified. Also, given the results from Chapter 6 it would be interesting to examine the performance of components such as the gateway in more detail (e.g., perform more detailed instruction level profiling) The object server also plays an important role in the architecture, and it would be interesting to determine how its performance is affected by the number of object's hosted. Similarly, it would be interesting to examine how the load on one object affects the performance of other (unrelated) objects.

7.3.4 Security

Security of the GlobeDoc architecture has not been discussed in this dissertation. Nevertheless security is an important aspect in any architecture. As such, security of GlobeDoc and the Globe architecture are currently the subject of ongoing research. In particular, we are developing an extension to GlobeDoc that allows clients to verify that elements received from a GlobeDoc object have not been tampered with (e.g., by a malicious object server). This extension allows the contents of a GlobeDoc object to be signed by the creator, and allows the signature to be retrieved along with the contents whereupon the contents can be checked for validity. More on this approach can be found in [84].

7.3.5 Support for Dynamic Content

As mentioned in Chapter 3 the GlobeDoc model does not support dynamic content generation. Nevertheless dynamic content is gaining importance and makes up a large part of current Web content. For the future success of GlobeDoc it will be necessary to incorporate dynamic contents into the model. Chapter 3 presented two approaches to integrating static and dynamic contents in GlobeDoc. A sketch of a third approach, called the **Globe Web component** (GWC) approach, is presented here.

Generally, dynamic content generation is used to create Web pages that are a mix of predefined user interface elements and dynamic data elements. While processing a request the data elements are either generated or retrieved from a database, merged with predefined interface elements, and returned as a complete Web page. This is achieved by executing a server-side program that is responsible for generating or accessing the data and building the page. For example, in an online photo album, the data, photos and photo meta-data, such as captions, are stored in a database. When an index page is requested, the server executes code that queries the database for a list of photos and then adds HTML code representing a list of the photos with links to their thumbnails to an index page HTML template. The completed in template is then returned to the client as a regular HTML page.

Depending on the server used, the program responsible for generating the content can be stored as an executable on the server, or it can be embedded in an HTML page. When the server is asked to retrieve a page corresponding to an executable, that executable is run and the output returned to the client. Examples of this approach include CGI binaries [27] and Java Servlets [30]. When the server is asked to retrieve an HTML page with embedded

code it processes the page, executes any code it comes across, and replaces the code with the execution results. Examples of this approach include PHP [9], and JSP [34].

Generally, no matter which approach is taken, the code for generating data and the interface elements are highly entwined. In the photo album example, the code must perform database queries, generate HTML code, and insert HTML snippets into an HTML template. It is this entwined nature that makes it difficult to successfully replicate dynamically generated Web pages. If such a page is replicated, then the database content must also be replicated, otherwise the code cannot execute properly. Replicating a database is difficult. Not only is it difficult to ensure that the data remains consistent, but it also requires all potential replica sites to run the same database software. Besides the fact that replicating the data is difficult, it may also be unnecessary to replicate the data at the same sites as the HTML and executable code.

The GWC approach to dynamic contents proposes to separate the interface part of a dynamic Web document from the data part. All the static elements that make up the interface, including HTML templates and the code that generates and manipulates HTML, are stored in a GlobeDoc object, while the code that generates or manages data is encapsulated in an application-specific Globe object. In the photo album example, the HTML template and the code that transforms a list of photo names to HTML is stored in a GlobeDoc object. The photos themselves are stored in a photo album Globe object, a Globe object that implements a special photo album interface (with methods to add photos, retrieve photos, generate thumbnails, list photos, etc.).

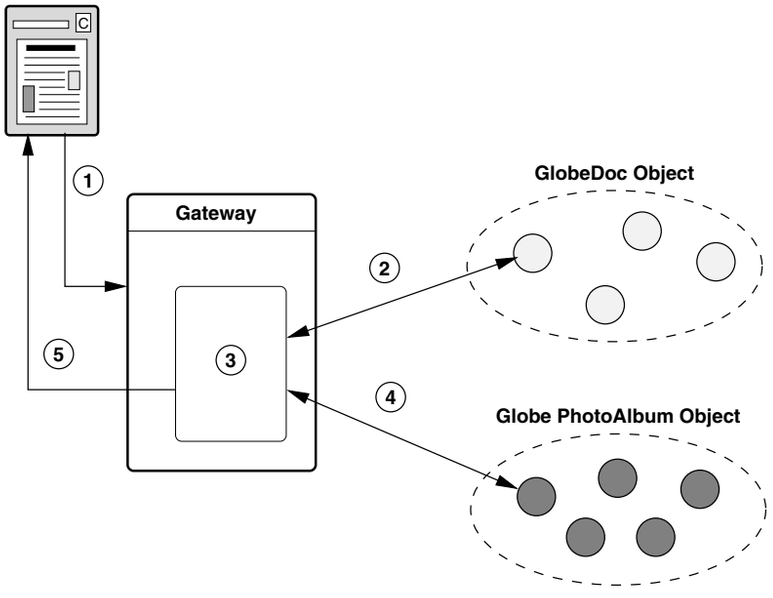


Figure 7.1: GWC approach to dynamic content.

Figure 7.1 shows a client accessing dynamic content through a GlobeDoc object and a GWC enabled GlobeDoc gateway. In this figure the client contacts the gateway with a request for a particular GlobeDoc URN (step 1). The requested elements are retrieved from the GlobeDoc object (step 2), and are processed by the gateway. Any code embedded in the retrieved elements is executed as described above (step 3). The code causes the gateway to bind to a photo-album Globe object and invoke methods on that object. The invoked methods set or retrieve photos and photo meta-data (step 4). The data retrieved from the photo album object is converted to HTML and merged into the HTML template. Finally, the completed HTML document is returned by the gateway to the client (step 5).

The GWC approach has a number of advantages over the traditional approach to dynamic content. First the interface logic is clearly separated from the application logic. Thus, for example, the creator of the photo album HTML interface does not have to know the best way to store, manipulate and replicate photos. Likewise, the creator of the photo album object need not be concerned about issues related to user interface design. A second benefit is that the application object can be used with many different interfaces. For example, the same photo album object could be accessed by a user with a specialized photo album GUI, a user using a regular HTML interface and a user using a simplified mobile device HTML interface. Likewise the interfaces can be used with different implementations of an application object, as long as the object's interfaces remain the same. Thus, even if the photo album object is updated to contain a radically different storage mechanism, none of the existing interfaces need to be modified.

This was a sketch of a possible approach to providing dynamic content using GlobeDoc and Globe. The GWC approach needs to be worked out further and implemented as a prototype to identify potential problems and find solutions to these problems.

Bibliography

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, MA, USA, Dec. 1995.

[2] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):95–107, 1999.

[3] Akamai Technologies Inc. http://www.akamai.com/.

[4] C. Allison, M. Bramley, and J. Serrano. The World Wide Wait: Where does the time go. In *Proceedings of the Euromicro 98 Conference*, Vasteras, Sweden, Aug. 1998.

[5] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, San Francisco, CA, USA, Oct. 1976.

[6] E. Anderson, D. Patterson, and E. Brewer. The MagicRouter, an application of fast packet interposing. Technical report, University of California, Berkeley, CA, USA, May 1996.

[7] K. Andrews, F. Kappe, H. Maurer, and K. Schmaranz. On second generation hypermedia systems. *Journal of Universal Computer Science (Pilot Issue)*, 0(0):127–135, 1994.

[8] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World-Wide Web Caching – The Application level view of the Internet. *IEEE Communications Magazine*, 35(6):170–178, 1997.

[9] S. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lerdorf, A. Zmievski, and J. Ahto. *PHP Manual*. The PHP Documentation Group, Apr. 2003.

[10] A. Bakker. *An Object-Based Software Distribution Network*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2002.

[11] A. Bakker, M. van Steen, and A. Tanenbaum. Replicated invocation in wide-area systems. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[12] G. Ballintijn. *Locating Objects in a Wide-area System*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2003.

[13] G. Ballintijn, P. Verkaik, E. Amade, M. van Steen, and A. S. Tanenbaum. A scalable implementation for human-friendly URIs. Technical Report IR-466, Vrije Universiteit, Amsterdam, the Netherlands, Nov. 1999.

[14] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems - Performance Evaluation Review (SIGMETRICS '98/ PERFORMANCE '98)*, Madison, WI, USA, June 1998.

[15] P. Barford and M. Crovella. Measuring Web performance in the wide area. *Performance Evaluation Review*, 27(2):37–48, Aug. 1999.

[16] T. Bates, E. Gerich, L. Joncheray, J.-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP routing policies in a routing registry. RFC 1786, Mar. 1995.

[17] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3):3–11, Sept. 1996.

[18] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, Dec. 1995.

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

[20] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the First USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, Dec. 1997.

[21] P. Cao and C. Liu. Maintaining strong cache consistency in the World Wide Web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[22] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents (objects) on the Web. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, UK, Sept. 1998.

[23] V. Cate. Alex – a global file system. In *Proceedings of the USENIX File System Workshop*, Ann Arbor, MI, USA, May 1992.

[24] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX Technical Conference*, San Diego, CA, USA, Jan. 1996.

[25] Cisco Systems Inc. Network caching white paper. Published on-line (http://www.cisco.com/warp/public/cc/pd/cxsr/500/tech/cds_wp.pdf), 2000.

[26] Cisco Systems Inc. Data sheet: Cisco Local Director 417 and 417G. Published on-line (http://www.cisco.com/warp/public/cc/pd/cxsr/400/prodlit/dsld4_ds.htm), 2002.

[27] K. A. L. Coar and D. Robinson. The WWW common gateway interface version 1.1. Internet Draft <draft-coar-cgi-v11-03>, June 1999.

[28] E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *Proceedings of the First International Symposium on Applications and the Internet (SAINT)*. IEEE, Jan. 2001.

[29] E. Cohen, B. Krishnamurthy, and J. Rexford. Evaluating server-assisted cache replacement in the Web. In *Proceedings of the European Symposium on Algorithms*, Venice, Italy, Aug. 1998.

[30] D. Coward. Java servlet specification, version 2.3. Technical report, Sun Microsystems, Inc, Palo Alto, CA, USA, Sept. 2001.

[31] O. P. Damani, P. E. Chung, Y. Huang, C. M. R. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, USA, Apr. 1997.

[32] The Distributed ASCI Supercomputer 2. http://www.cs.vu.nl/das2/.

[33] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, Dec. 1998.

[34] E. P. L. (editor). Java server pages specification, version 1.2. Technical report, Sun Microsystems, Inc., Palo Alto, CA, USA, Aug. 2001.

[35] A. Field. *Discovering Statistics using SPSS for Windows*, chapter Regression. SAGE Publications, London, UK, 2000.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Jan. 1997.

[37] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Sixth IEEE International Symposium on High Performance Distributed Computing*, Portland, OR, USA, Aug. 1997.

[38] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An architecture for a global Internet host distance estimation service. In *Proceedings of IEEE INFOCOM*, New York, NY, USA, Mar. 1999.

[39] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, USA, 1999.

[40] X. Gan and B. Ramamurthy. LSMAC: An improved load sharing network service dispatcher. *World Wide Web*, 3(1):53–59, 2000.

[41] X. Gan, T. Schroeder, S. Goddard, and B. Ramamurthy. LSMAC vs. LSNAT: Scalable cluster-based Web servers. *Cluster Computing: the Journal of Networks, Software Tools and Applications*, 3(3):175–185, 2000.

[42] A. Gulbrandsen and P. Vixie. A DNS RR for specifying the location of services (DNS SR V). RFC 2728, Feb. 2000.

[43] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2165, June 1999.

[44] J. Gwertzman and M. I. Seltzer. World Wide Web cache consistency. In *Proceedings of the USENIX Technical Conference*, San Diego, CA, USA, Jan. 1996.

[45] A. Habib and M. Abrams. Analysis of sources of latency in downloading Web pages. In *Proceedings of WebNet*, San Antonio, TX, USA, Nov. 2000.

[46] R. Hinden and S. Deering. IP version 6 addressing architecture. RFC 2373, July 1998.

[47] P. Homburg. *The Architecture of a Worldwide Distributed System*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2001.

[48] S. Hosseini-Khayat. *Investigation of generalized caching*. PhD thesis, Washington University, St. Louis, MO, USA, 1997.

[49] T. Howes. The string representation of LDAP search filters. RFC 2254, Dec. 1997.

[50] Inktomi Corporation. Inktomi webmap. Published on-line (http://www.inktomi.com/webmap/), Jan. 2000.

[51] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer Web cache. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, Monterey, CA, USA, July 2002.

[52] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, NY, USA, 1991.

[53] J. Kangasharju, J. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. In *Proceedings of the Sixth International Web Caching and Content Distribution Workshop*, Boston, MA, USA, June 2001.

[54] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. In *Proceedings of the First International World Wide Web Conference*, Geneva, Switzerland, May 1994.

[55] A.-M. Kermarrec, I. Kuz, M. van Steen, and A. S. Tanenbaum. A framework for consistent, replicated Web objects. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, the Netherlands, May 1998.

[56] A.-M. Kermarrec, I. Kuz, M. van Steen, and A. S. Tanenbaum. Towards scalable Web documents. Technical Report IR-452, Vrije Universiteit, Amsterdam, the Netherlands, Oct. 1998.

[57] D. Lake. The Web: Growing by 2 million pages a day. *The Industry Standard*, Feb. 2000. Published on-line (http://www.thestandard.com/article/0,1902,12329,00.html).

[58] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.

[59] S. Lawrence and C. L. Giles. Accessibility of information on the Web. *Nature*, 400(6740):107–109, 1999.

[60] F. Leighton and D. Lewin. Global hosting system. United States Patent 6,108,703, Aug. 2000.

[61] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. M. Dias. Design and performance of a Web server accelerator. In *Proceedings of IEEE INFOCOM*, New York, NY, USA, Mar. 1999.

[62] A. Lowe-Norris. *Windows 2000 Active Directory*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[63] A. Luotonen and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, 1994.

[64] About asynchronous pluggable protocols. Published on-line (http://msdn.microsoft.com/workshop/networking/pluggable/pluggable.asp).

[65] M. Mitchell, J. Oldham, and A. Samuel. *Advanced Linux Programming*, chapter The /proc File System. New Riders Publishing, Indianapolis, IN, USA, June 2001.

[66] P. Mockapetris. Domain names – concepts and facilities. RFC 1034, Nov. 1987.

[67] J. Myers. Simple authentication and security layer (SASL). RFC 2222, Oct. 1997.

[68] Netcraft. Netcraft Web server survey. Published on-line (http://www.netcraft.com/survey/Reports/index.html).

[69] NetGeo - the Internet geographic database. Published on-line (http://www.caida.org/tools/utilities/netgeo/).

[70] Netscape Communications. Netscape Proxy Server 3.5 data sheet. Published on-line (http://wp.netscape.com/proxy/v3.5/datasheet/), 1999.

[71] B. C. Neuman. Scale in distributed systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[72] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative leases: scalable consistency maintenance in content distribution networks. In *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[73] NLANR caches "vital statistics", 1999. http://www.ircache.net/Cache/Statistics/Vitals/.

[74] K. Obraczka and F. Silva. Network latency metrics for server proximity. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, San Francisco, CA, USA, Dec. 2000.

[75] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of ACM SIGCOMM*, Stanford University, CA, USA, Aug. 1996.

[76] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, Oct. 1998.

[77] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[78] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Nov. 1993.

[79] C. Pfeiffer and S. Kulow. The omnivore: Kde's flexible i/o architecture. Published on-line (http://www.heise.de/ct/english/01/05/242/).

[80] G. Pierre, I. Kuz, M. van Steen, and A. S. Tanenbaum. Differentiated strategies for replicating Web documents. Technical Report IR-467, Vrije Universiteit, Amsterdam, the Netherlands, Nov. 1999.

[81] G. Pierre and M. Makpangou. Saperlipopette!: a distributed Web caching systems evaluation tool. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, UK, Sept. 1998.

[82] G. Pierre, M. van Steen, and A. S. Tanenbaum. Dynamically selecting optimal distribution strategies for Web documents. *IEEE Transactions on Computers*, 51(6):637–651, June 2002.

[83] J. E. Pitkow. Summary of WWW characterizations. *Computer Networks and ISDN Systems*, 30(1–7):551–558, Apr. 1998.

[84] B. C. Popescu, I. Kuz, M. van Steen, and A. S. Tanenbaum. Security for replicated Web documents. Technical Report IR-498, Vrije Universiteit, Amsterdam, the Netherlands, June 2002.

[85] D. Povey and J. Harrison. A distributed Internet cache. In *Proceedings of the 20th Australasian Computer Science Conference*, Sydney, NSW, Australia, Feb. 1997.

[86] Protozilla: pipes, protocols, and p2p in mozilla. Published on-line (http://protozilla.mozdev.org/white-paper.html).

[87] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of Web server replicas. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, USA, Apr. 2001.

[88] M. Rabinovich and A. Aggarwal. RaDaR: a scalable architecture for a global Web hosting service. *Computer Networks*, 31(11–16):1545–1561, 1999.

[89] S. Radicati. *X.500 Directory Services: Technology and Deployment*. International Thomson Computer Press, London, UK, 1994.

[90] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[91] A. Rousskov and D. Wessels. Cache digests. In *Proceedings of the Third International Web Caching Workshop*, Manchester, UK, June 1998.

[92] A. Rowstron. Run-time systems for coordination. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies and Applications*, pages 78–96. Springer-Verlag, Berlin, Germany, Aug. 2001.

[93] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[94] B. Ryner. Writing a mozilla protocol handler. Published on-line (http://www.mozilla.org/projects/netlib/new-handler.html).

[95] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–18, May 1990.

[96] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of Web documents. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, USA, Apr. 1997.

[97] T. Schroeder, S. Goddard, and B. Ramamurthy. Scalable Web server clustering technologies. *IEEE Network*, (3):38–45, May 2000.

[98] S. Spero. Analysis of HTTP performance problems. Published on-line (http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html), July 1994.

[99] H. Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1991.

[100] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, Jan. 1997.

[101] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, San Diego, CA, USA, Aug. 2001.

[102] M. R. Syam Gadde and J. Chase. Reduce, reuse, recycle: An approach to building large Internet caches. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997.

[103] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the Internet. Technical Report TR98-04, Department of Computer Science, University of Texas, Austin, TX, USA, Feb. 1998.

[104] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Canberra, ACT, Australia, 1999.

[105] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. Internet Draft <draft-vinod-carp-v1-03.txt>, 1998.

[106] L. van Doorn. *The Design and Application of an Extensible Operating System*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2001.

[107] M. van Steen, F. Hauck, G. Ballintijn, and A. Tanenbaum. Algorithmic design of the Globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[108] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January–March 1999.

[109] M. van Steen, S. van der Zijden, and H. J. Sips. Software engineering for scalable distributed applications. In *Proceedings of the 22nd International Computer Software and Applications Conference*, 1998.

[110] P. Verkaik, E. Amade, A. Bakker, and I. Kuz. *Globe Operations Guide*. Vrije Universiteit, Amsterdam, the Netherlands, Aug. 2002.

[111] M. Wahl, A. Coulbeck, T. Howes, and S. Kille. Lightweight directory access protocol (v3): Attribute syntax definitions. RFC 2252, Dec. 1997.

[112] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, Dec. 1997.

[113] J. Waldo. *The Jini Specifications*. Prentice Hall, Englewood Cliffs, NJ, USA, 2nd edition, 2000.

[114] D. Wessels and K. Claffy. ICP and the Squid Web cache. *IEEE Journal on Selected Areas in Communications*, 16(2):345–357, 1998.

[115] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of ACM SIGCOMM*, Stanford University, CA, USA, Aug. 1996.

Summary

An Approach to A Scalable Wide-Area Web Service

Scalability and the World Wide Web

Many problems plague the World Wide Web, one of the greatest being poor performance which affects the responsiveness of Web sites. A Web site with good performance is responsive and provides fast downloads of its content. A Web site with poor performance, on the other hand, is slow in responding to user requests and provides slow downloads of its content. In extreme cases poor performance can cause a site to become effectively unreachable.

An important cause of many of the Web's performance problems is poor scalability of the underlying architecture. Scalability refers to how well the architecture can handle increased growth. On the architectural level the Web is based on a client-server model; it is organized as a decentralized collection of servers serving pages to widely distributed clients. Although the simplicity of the architectural design is one of the key factors behind the Web's phenomenal growth, it is precisely this simplicity that is at the root of the Web's scalability problems. This client-server nature leads to two types of scalability problems.

The first scalability problem is that the Web architecture does not scale well with regards to the (geographic or network topological) distribution of clients. This means that clients further away from a Web site's server generally experience worse performance than those close to the server. As such, a Web site with widely distributed clients often provides decreased performance compared to a site with mostly local clients.

The second problem is that the Web architecture does not scale well with regards to the number of client requests processed. Because a Web site is generally served by a single server, the more clients that request its contents, the higher the load on that server becomes. Likewise, the more requests a server receives, the more saturated its network becomes. Both result in decreased performance.

Requirements for A Scalable Web Architecture

In order to solve the Web's scalability problems it is necessary to provide a scalable architecture. Such an architecture will address the two scalability problems by making the performance of a Web site virtually independent of the (geographic or network topological) distribution of that site's clients and of their access patterns. This means that there

will be no distinction between clients closer to or further away from the Web site's server. Furthermore, clients will experience the same (high) level of performance, independent of the amount of requests the Web site's server must process.

An effective method for achieving this is replication. Replication involves placing copies (i.e., replicas) of a Web site's content on other servers in the network. With multiple copies of the content available clients have a choice of servers to send their requests to. Their decision is based on which server will offer the best performance. Assuming that the servers themselves offer similar levels of performance, it is the distance from the client that sets the servers apart. Requests are, therefore, sent to the server closest to the client. Not only does this spread the total load over multiple servers, relieving the load at any single server, but, because clients send requests to their nearest server, the distance between clients and their servers is shortened thus allowing for better performance.

A central claim of this dissertation is that, in order to be scalable, a Web architecture not only needs to allow content to be replicated, but it must be flexible enough to allow the replication policies used to be determined on a case by case basis. We distinguish between one-size-fits-all solutions, which impose a single policy on all Web sites (or, more precisely, Web documents), and per-document solutions, which allow policies to be chosen on a case-by-case basis.

In order to illustrate the differences between the effects of one-size-fits-all and per-document solutions on performance a simulation-based experiment was performed. This experiment compared the performance of per-document replication policies to one-size-fits-all policies. The results show that per-document policies provide better (i.e., closer to optimal) performance gains than one-size-fits-all policies.

GlobeDoc: A Scalable Web Architecture

Replication has been used as the basis of many proposed solutions to the Web's performance problems. These solutions include caching, mirroring, server clustering, and content delivery networks. A drawback of these proposed solutions is that their approach must be applied universally to all Web content (i.e., they provide one-size-fits-all solutions). This means that all Web resources have to be replicated in the same way and with the same coherence policy applied to all of them.

In order to address the sclability problems described above we have designed and implemented a scalable Web architecture called GlobeDoc (Global Object Based Environment for Web Documents).[1] GlobeDoc is an architecture based on distributed shared objects (DSOs). DSOs are physically distributed objects, meaning that their state is partitioned across multiple disjoint address spaces (which are usually hosted on separate machines) at the same time. Clients of an object are unaware of such a distribution: they see only the object interfaces available in their own address spaces. Globe (Global Object Based Environment) is an existing wide-area distributed system that provides support for DSOs. GlobeDoc is designed as a Globe application and is based on Globe DSOs.

[1] It is not the intention of this work to (attempt to) completely replace the current Web infrastructure. In many cases it works fine as is, so there is no reason to replace it. Instead, the results of this work can be seamlessly incorporated into the existing Web infrastructure.

Conceptually, a GlobeDoc object is a distributed shared object that contains a Web document, which is a collection of logically related Web pages. Each such page may consist of text, icons, images, sounds, animations, etc., and may also contain applets, scripts and other forms of executable code. Each of these parts is referred to as an element. A GlobeDoc object allows clients to add, remove, access and modify its elements. In the GlobeDoc model a Web site consists of a collection of GlobeDoc objects.

In GlobeDoc, each object fully encapsulates its own distribution policy. There is no system-wide policy imposing how an object's state should be replicated and kept consistent. Moreover, clients need not be aware of the details of the distribution policy applied by an object. A GlobeDoc object is therefore free to take any possible approach to replication. It can, for example, decide not to create any replicas at all, or it can create replicas and offer a best effort consistency guarantee. An object is even free to dynamically create and destroy replicas as it sees fit. GlobeDoc is location transparent, meaning that, from a client's point of view, location is not relevant. As such, clients do not know (nor do they need to know) where a GlobeDoc object and its replicas are hosted. A client wishing to use a GlobeDoc object is automatically connected to the replica closest to it. GlobeDoc provides both a DSO model and a framework that offers support for using, creating and hosting GlobeDoc DSOs.

In order to access a GlobeDoc object's contents a client must bind[2] to the object and invoke appropriate methods on it. Naturally, this can only be done by GlobeDoc-aware clients. Unfortunately, there is no currently existing Web browser that is GlobeDoc-aware. The GlobeDoc architecture, therefore, also includes mechanisms and services that allow GlobeDoc-unaware clients to access the contents of GlobeDoc objects, while preserving the scalability benefits provided by GlobeDoc objects and the GlobeDoc environment.

This dissertation provides a detailed description of the GlobeDoc model, the GlobeDoc architecture, and its associated services. In particular it describes the design and implementation of the Globe object server, the implementation repository, GlobeDoc clients (and the services providing access to GlobeDoc objects by GlobeDoc-unaware clients), and the Globe infrastructure directory service. Furthermore we devised and performed profiling experiments to measure the performance of each service used in the GlobeDoc architecture. These experiments gave an indication of the effect of different request characteristics on the work done by each of the services. They also provided information about which GlobeDoc services form potential bottlenecks and are prime candidates for optimization.

Conclusions

The main reasons for basing GlobeDoc on Globe is that the Globe infrastructure encourages replication of content and makes this replication transparent. Similarly, Globe allows each object to have its own associated distribution policy. This makes it possible to determine and implement an optimal policy for each individual Web document.

[2]Binding is a Globe specific process and involves contacting Globe specific naming and location services, which are responsible for finding a client's nearest replica.

The fact that GlobeDoc content can be replicated means that a GlobeDoc object's total load will be spread out over its replicas. Likewise, the traffic at each replica will have a more local character (i.e., the distance to clients will be smaller). As mentioned earlier this has a positive effect on performance. Similarly, the fact that new replicas can be added and removed means that it is possible for a GlobeDoc object to adapt its replication based on its current situation. For example, if the number of requests dramatically increases a GlobeDoc object may create new replicas to help handle the load. Similarly, a GlobeDoc object may create new replicas placed closer to a large group of clients to reduce the distance between the clients and itself, thus localizing the traffic. This adaptability makes it possible for the performance of a GlobeDoc object to be independent of the geographic (or network topographic) distribution of clients and the number of requests they generate.

GlobeDoc, therefore, provides a scalable architecture for the Web and when used with appropriate replication policies can help to improve the performance of Web sites. The GlobeDoc architecture can be seemlessly integrated into the current Web architecture, allowing GlobeDoc-based Web sites to be accessed by regular GlobeDoc-unaware browsers. To show this we have implemented GlobeDoc and used this implementation to host a number of existing Web sites (in particular the Globe project's Web site). Future work for GlobeDoc includes studying, designing, and applying new replication policies, and in particular looking at adaptive replication policies.

Ihor Kuz

Samenvatting

Een Aanpak voor een Schaalbare Wereldwijd Web Dienst

Schaalbaarheid en het World Wide Web

Er zijn veel problemen die het Web teisteren. Een van de grootste problemen wordt gevormd door slechte prestaties. Prestatieproblemen beïnvloeden de reactiviteit van een Web site. Een Web site met goede prestaties is reactief en biedt snelle downloads van zijn inhoud. Een Web site met slechte prestaties daarentegen, reageert langzaam op verzoeken van gebruikers en biedt langzame downloads van zijn inhoud. In extreme gevallen kunnen slechte prestaties er zelfs voor zorgen dat een Web site geheel onbereikbaar wordt.

Een belangrijke oorzaak van veel van de prestatieproblemen van het Web is de slechte schaalbaarheid van de onderliggende architectuur. Met schaalbaarhaeid wordt bedoeld hoe goed een architectuur bestand is tegen toenemende groei. De Web architectuur is gebaseerd op een client-server model. Dit houdt in dat het Web georganiseerd is als een gedecentraliseerde collectie servers die Web paginas leveren aan een groep wijdverspreide gebruikerapplicaties (zoals Web browsers). Hoewel de eenvoud van deze architectuur een van de belangrijkste redenen voor de snelle groei van het Web is, is het meteen ook de oorzak van de slechte schaalbaarheid van het Web. De client-server aard van het Web leidt tot twee soorten schaalbaarheidsproblemen.

Het eerste probleem is dat de Web architectuur niet goed schaalt met betrekking tot de (geografische of netwerk-topologische) verspreiding van gebruikers. Gebruikers die verder verwijderd zijn van een server ervaren meestal slechtere prestaties dan gebruikers die dichter bij een server zijn. Dit betekent dat een Web site met wijdverspreide gwbruikers over het algemeen slechtere prestaties levert dan een Web site met minder wijdverspreide gebruikers.

Het tweede probleem is dat de Web architectuur niet goed schaalt met betrekking tot het aantal verzoeken dat afgehandeld moet worden. Omdat de inhoud van een Web site meestal door een enkele Web server wordt bediend zal de server meer werk moeten verrichten wanneer er meer verzoeken voor de inhoud binnen komen. Het is ook zo dat hoe meer verzoeken worden afgehandeld hoe meer verzadigd het het netwerk wordt. Beide zorgen voor verlaagde prestaties.

Vereisten voor een Schaalbare Web Architectuur

Een oplossing voor de genoemde schaalbaarheidsproblemen moet aangeboden worden in de vorm van een schaalbare architectuur. Zo'n architectuur pakt de twee schaalbaarheidsproblemen aan door ervoor te zorgen dat een Web site onafhankelijk wordt van de (geografishce of netwerk topologische) verspreiding van zijn gebruikers en het aantal verzoeken dat afgehandeld moet worden. Dit betekent dat er geen onderscheid wordt gemaakt tussen gebruikers die zich dichter bij of verder weg van een bepaalde server bevinden. Verder zullen alle gebruikers dezelfde (hoge) prestaties ervaren, onafhankelijk van hoeveel verzoeken een Web site moet afhandelen.

Een effectieve methode om dit te bereiken is replicatie. Replicatie houdt in dat kopieën (replicas) van de inhoud van een Web site op andere servers in een netwerk geplaatst worden. Door meerdere beschikbare kopieën van de inhoud hebben gebruikers een keuze uit servers waar ze hun verzoeken naar toe kunnen sturen. De beslissing van waar een verzoek naar toe gestuurd wordt wordt gebaseerd op de prestaties die geleverd kunnen worden. Wanneer er vanuit gegaan wordt dat de servers zelf gelijke prestaties kunnen leveren, is het de afstand tussen een gebruiker en een server die (nu) bepalend wordt. Verzoeken worden naar de dichtsbijzijnde server gestuurd. Dit heeft als effect dat de belasting veroorzaakt door alle verzoeken verspreid wordt over meerdere servers. Daarnaast zal de afstand tussen gebruikers en servers verkort worden. Waardoor de prestaties beter worden.

Een belangrijke bewering die gemaakt wordt in dit proefschrift is dat wil een Web architectuur schaalbaar zijn, dan moet zo'n architectuur het mogelijk maken om inhoud de repliceren, maar moet het flexibel genoeg zijn om replicatie gedifferentieerd toe te passen. Wij maken een onderscheid tussen een "confectie-maat" aanpak en een "per-document" aanpak. Een "confectie-maat" aanpak legt één specifiek replicatiebeleid op alle Web sites (of beter gezegd, individuele Web documenten), terwijl een "per-document" aanpak het mogelijk maakt om elk document apart volgens een optimaal beleid te repliceren.

Om te laten zien dat er een belangrijk verschil is tussen de "confectie-maat" en "per-document" aanpakken hebben we simulatie experimenten uitgevoerd. De resultaten lieten zien dat een "per-document" aanpak significant betere (dwz dichter bij de optimale) prestaties biedt.

GlobeDoc: Een Schaalbare Web Architectuur

Replicatie wordt vaak gebruikt als de basis van veel voorgestelde oplossingen voor de prestatie problemen van het Web. Onder deze oplossingen bevinden zich onder andere caching, mirroring, server clustering en content delivery netwerken. Een nadeel van al deze voorgestelde oplossingen is dat hun aanpak universeel toegepast moet worden op alle Web inhoud (dat wil zeggen, het zijn "confectie-maat" oplossingen). Dit betekent dat alle Web resources(?) op dezelfde manier moeten worden gerepliceerd en op dezelfde manier consistent moeten worden gehouden.

Als oplossing voor de eerder genoemde schaalbaarheidsproblemen hebben we een schaalbare Web architectuur genaamd GlobeDoc (Global Object Based Environment for

Web Documents) gemaakt.[3] De GlobeDoc architectuur is gebaseerd op *gespreide gedeelde objecten* (oftewel distributed shared objects — DSOs). DSOs zijn objecten die fysiek gespreid zijn, wat betekent dat hun data ten alle tijden verspreid kan zijn over verschillende processen (op verschillende machines). Gebruikers van een object zijn zich niet bewust van het feit dat de objecten zo verspreid zijn: zij zien alleen een lokale programmeerinterface in hun eigen proces. Globe (Global Object Based Environment) is een bestaand (hoewel nog in ontwikkeling zijnd) gespreid systeem dat ondersteuning biedt voor gespreide gedeelde objecten. GlobeDoc is ontworpen als een Globe applicatie.

Een GlobeDoc object is een gespreid gedeeld object dat de inhoud van een Web document bevat. Een Web document is een verzameling van logisch gerelateerde Web paginas. Zo'n pagina kan bestaan uit verschillende onderdelen zaols tekst, plaatjes, geluiden, films, enzovoort. Deze onderdelen van een pagina heten elementen. In het GlobeDoc model bestaat een Web site uit een verzameling van GlobeDoc objecten.

In GlobeDoc bepaalt ieder object zijn eigen replicatiebeleid. Er is geen globaal beleid dat bepaalt hoe elk object zijn data (dwz alle elementen) moet repliceren of consistent moet houden. Het is verder zo dat gebruikers van objecten niets hoeven te weten van het replicatiebeleid dat door een object wordt uitgevoerd. Een GlobeDoc object is dus vrij in zijn keuze van replicatiebeleid. Een object kan bijvoorbeeld beslissen om helemaal geen replicas te creëren, of hij kan beslissen replicas te creëren en zijn best doen om deze replicas consistent te houden. Een object kan zelfs dynamisch nieuwe replicas creëren en bestaande replicas verwijderen afhankelijk van hoe veel verzoeken verwerkt moeten worden. In GlobeDoc is locatie transparant. Dit betekent dat vanuit het oogpunt van een gebruiker de locatie niet relevant is. Gebruikers weten dus niet waar een GlobeDoc object en zijn replica's zich bevinden. Een gebruiker die een GlobeDoc object wil gebruiken wordt automatisch verbonden met de dichtstbijzijnde replica. GlobeDoc biedt zowel een DSO model als een omgeving voor het creëren en gebruiken van GlobeDoc objecten aan.

Om de inhoud van een GlobeDoc object op te vragen moet een gebruiker eerst aan een GlobeDoc object binden.[4] Tijdens het binden wordt er een lokale kopie van het object (vergelijkbaar met een stub) in de address space van de gebruiker gemaakt. De gebruiker kan methoden op het lokale object aanroepen om de benodigde GlobeDoc inhoud op te vragen. Binden aan een object en het aanroepen van methodes kan alleen worden uitgevoerd door GlobeDoc-afhankelijke applicaties. Helaas zijn er weinig GlobeDoc-afhankelijke applicaties, en bestaan er op dit moment geen GlobeDoc-afhankelijke Web browsers. De GlobeDoc architectuur biedt daarom ook mechanismen en diensten aan die het mogelijk maken dat GlobeDoc-onafhankelijke applicaties op een schaalbaare manier toegang kunnen krijgen tot de inhoud van GlobeDoc objecten.

Dit proefschrift bevat een gedetailleerde beschrijving van het GlobeDoc model, de GlobeDoc architectuur en de daarmee verbonden diensten. De nadruk wordt in het bijzonder gelegd op de beschrijving van het ontwerp en op de implementatie van fundamentele

[3] Let op dat het niet de bedoeling is om het huidige Web infrastructuur te vervangen. In veel gevallen werkt het Web infrastructuur goed. Er is dus geen reden om de infrastructuur dan te vervangen. Het is de bedoeling dat de resultaten van dit onderzoek naadloos geïntegreerd kunnen worden in de bestaande Web architectuur.

[4] Binden, oftewel *binding*, is een proces gedefinieerd door Globe en omvat het gebruiken van Globe specifieke naam- en locatiediensten om een gebruiker de dichtstbijzijnde replica te kunnen vinden.

aspecten van de architectuur zoals de Globe object server, op de opslag van implementaties, op GlobeDoc gebruikers applicaties, en op GIDS (een gids van alle GlobeDoc diensten en servers). Verder worden er experimenten beschreven die zijn uitgevoerd om te bepalen hoeveel werk iedere component van de architectuur verricht tijdens het verwerken van een verzoek. De resultaten van deze experimenten geven aan wat de effecten van verschillende karakteristieken van de verzoeken zijn. Er is ook informatie vergaard over de componenten die potentiële prestatie problemen kunnen veroorzaken en dus het best kunnen worden geoptimaliseerd.

Conclusies

De belangrijkste reden voor het gebruiken van Globe als basis voor GlobeDoc is dat Globe het gebruik van replicatie mogelijk maakt en dat hij de replicatie transparant maakt voor gebruikers. Evenzo maakt Globe het mogelijk dat iedere GlobeDoc een eigen replicatie beleid kan uitvoeren. Hierdoor kan per Web document een optimaal beleid worden bepaald en geïmplementeerd.

Het feit dat de inhoud van een GlobeDoc object kan worden gerepliceerd zorgt ervoor dat het werk dat een object moet doen verspreid kan worden over meerdere servers. Tevens zal het netwerkverkeer bij ieder van deze servers een lokaal karakter hebben (d.w.z. dat verzoeken zullen van dichtbij gelegen gebruikers afkomstig zijn). Zoals eerder gezegd heeft dit een gunstig effect op de prestaties. Het feit dat nieuwe replicas dynamisch gemaakt kunnen worden en dat bestaande replicas ook verwijderd kunnen worden betekent dat het mogelijk is voor een GlobeDoc object om zijn replicatie beleid aan te passen aan zijn huidige situatie. Als het aantal verzoeken dat een GlobeDoc object moet verwerken drastisch toeneemt, kan een GlobeDoc object bijvoorbeeld nieuwe replicas creëren om zodoende het werk over meerdere servers te verspreiden. Een GlobeDoc object kan ook nieuwe replicas creëren in de buurt van een hoge concentratie gebruikers. Hierdoor kan het de afstand tussen gebruikers en het object (in de vorm van replicas) kleiner maken waardoor het verkeer een lokaler karakter wordt gegeven. Deze aanpasbaarheid maakt het mogelijk om de prestaties van een GlobeDoc object onafhankelijk te maken van de (geografische of netwerk topologische) distributie van gebruikers en van het aantal verzoeken dat zij genereren.

GlobeDoc, samen met geschikte replicatie beleid, biedt een schaalbare Web architectuur aan, en kan de prestaties van Web sites helpen te verbeteren. De GlobeDoc architectuur kan naadloos geïntegreerd worden in de huidige Web architectuur, waardoor het mogelijk is GlobeDoc objecten vanuit bestaande Web browsers te gebruiken. Om dit hard te maken hebben wij GlobeDoc geïmplementeerd en gebruikt voor een aantal Web sites (waaronder de Web site van het Globe project). Toekomstig onderzoek zal zich richten op het replicatiebeleid en met name op de dynamische en adaptieve replicatie.

Ihor Kuz

Curriculum Vitae

Ihor Kuz was born in Toronto, Canada on September 15, 1972. He lived in Canada until he was 15 years old, when he moved, with his family, to the Netherlands. He completed his secondary education in the Netherlands receiving a European Baccalaureate in 1990 at the European School in Bergen. After taking a year off to do some travelling in North America, Ihor returned to the Netherlands in 1991 where he studied Computer Science at the Vrije Universiteit, Amsterdam. After graduating in 1996 (receiving his M.Sc. degree cum laude) he worked for a while as a programmer before deciding, in 1997, to enroll with the Parallel and Distributed Systems group at the TU Delft as a Ph.D. student. The topic of his Ph.D. research was "A scalable Web Architecture" and resulted in the work described in this dissertation. In 2001 and 2002, while writing his dissertation, Ihor also worked as a scientific programmer at the Vrije Universiteit, Amsterdam.

Ihor is currently employed as a researcher at National ICT Australia in Sydney, Australia.

Publications

- *The Globe Infrastructure Directory Service* by I. Kuz, M. van Steen and H.J. Sips in Computer Communications vol. 25(9), June 2002, pp. 835-845.
- *Differentiated Strategies for Replicating Web Documents* by G. Pierre, I. Kuz, M. van Steen and A.S. Tanenbaum in Computer Communications vol. 24(2), February 2001, pp. 232-240.
- *A Distributed-Object Infrastructure for Corporate Websites* by I. Kuz, P. Verkaik, M. van Steen and H.J. Sips in Proc. IEEE Distributed Objects and Applications (DOA'00), Antwerp, Belgium, September 2000.
- *The Globe Distribution Netowrk* by A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen and A.S. Tanenbaum in Proc. 2000 USENIX Annual Conf. (FREENIX Track), San Diego, CA, USA, June 2000.
- *A Scalable Middleware Architecture for Advanced Wide-Area Web Services* by M. van Steen, A.S. Tanenbaum, I. Kuz, and H.J. Sips in Distributed Systems Engineering, vol. 6(1), March 1999, pp.34-42.

- Replicated Web Objects: Design and Implementation by I. Kuz, A.M. Kermarrec, M. van Steen, and H.J. Sips in Proc. Fourth Annual ASCI Conf., Lommel, Belgium, June 1998.
- *A Framework for Consistent, Replicated Web Objects* by A.M. Kermarrec, I. Kuz, M. van Steen, and A.S. Tanenbaum in Proc. 18th Int'l. Conf. on Distributed Computings Systems, Amsterdam, May 1998.
- *Towards a Taxonomy of Distributed-Object Models* by A. Bakker, I. Kuz and M. van Steen in Proc. Third Annual ASCI Conf., Heijen, The Netherlands, June 1997.

Index