AN OBJECT-BASED SOFTWARE DISTRIBUTION NETWORK

ARNO BAKKER

The cover of this dissertation illustrates the difficulty of content moderation.

VRIJE UNIVERSITEIT

AN OBJECT-BASED SOFTWARE DISTRIBUTION NETWORK

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op woensdag 4 december 2002 om 13.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

ARNO BAKKER

geboren te Zaandam

promotoren:	prof.dr. A.S. Tanenbaum
	prof.dr.ir. M.R. van Steen

Iedereen bedankt!

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

The Internet has radically changed the lives of millions of people. Researchers rarely frequent university libraries anymore, because so many research publications are available “on line.” Grandparents communicate with their grandchildren studying or traveling abroad via electronic mail. Students and companies load their computers with free software distributed via the Internet, developed by programmers from all continents collaborating over that same Internet. More and more people buy books, CDs, concert tickets and clothes via the Net. Some even do their banking and manage their stock portfolio on line.

Many believe this is only the beginning. Over the last few years, large amounts of capital have been invested in Internet companies, trying to invent the next “killer” Internet application. However, building new Internet applications that can provide service to millions of people, distributed all over the world, 24 hours a day is very difficult. The World Wide Web is currently the most frequently used application-development platform, but was never designed to be used on the scale it is today and provides only limited facilities. Add-on technologies such as Java applets, dynamic HTML and plug-ins have failed to address many of the problems of developing large-scale distributed applications. Other application-development platforms, such as object request brokers and DCOM, also have yet to meet these challenges, as they were developed with local-area networks in mind.

The research and development community will have to create better platforms for developers to build Internet applications which are large-scale, secure, reliable and highly available. Designing and building a middleware platform that facilitates the development of new, large-scale Internet applications is the goal of the Globe project. This middleware platform is, like the project, called *Globe*, an abbreviation for **GL**obal **O**bject-**B**ased **E**nvironment.

This dissertation describes the design and implementation of a new Internet application built using the Globe middleware platform. This new application, called the *Globe Distribution Network*, or *GDN* for short, is an application for the efficient, worldwide distribution of freely redistributable software packages. The purpose of this research is to show how a nontrivial distributed application can be built using Globe, and thus evaluate the Globe middleware.

The remainder of this chapter is structured as follows. Sec. 1.1 discusses the problems of developing large-scale distributed applications and the need for a new middleware platform. Sec. 1.2 introduces the Globe Distribution Network and explains why we chose to develop this particular application. Sec. 1.3 presents the contributions of this research and Sec. 1.4 describes the structure of the rest of the dissertation.

1.1. DEVELOPING LARGE-SCALE APPLICATIONS

The scale of a distributed application has three dimensions: numerical, geographical and administrative [Neuman, 1994]. The numerical dimension classifies the application in terms of the number of users and components making up the application. The geographical dimension describes the size of the geographical area over which the users and components are distributed. Finally, the administrative dimension indicates the number of organizations in control of (parts of) the application.

Developing distributed applications that are large in any of the three dimensions in this taxonomy is hard. Dealing with millions of users and components introduces many engineering and management issues. It requires the extensive use of techniques such as caching, replication, and distribution of functionality to reduce and distribute the load over the available infrastructure [Neuman, 1994]. These techniques, in turn, introduce technical and managerial problems of their own, such as maintaining consistency of caches and replicas, and how to keep track of a (replicated) component's current location(s). Large geographical distances introduce unavoidable and significant (by today's performance standards) communication delays, whose impact again have to be minimized by caching, replication, and distribution of functionality. Having to deal with many organizations makes it hard to administer and secure the application, in particular, if these organizations operate in different parts of the world. In addition to the problems introduced by the large scale of the applications, a developer also has to deal with machine and network failures, and heterogeneity in hardware and (system) software.

The key to making large-scale application development easier is therefore to provide the developer with the means for dealing with these complex (nonfunctional) aspects and required techniques in a comprehensive manner. Particularly important for a development platform, in addition to comprehensiveness, is flexibility. To build an application with hundreds of millions of users operating on a worldwide scale, it is necessary that the development platform allows the developer to employ the techniques, protocols and policies that are best suited for the application [Van Steen et al., 1999a]. This implies that the platform should support many different mechanisms and policies and it should also allow new ones to be introduced easily. In short, to accommodate applications of this scale, a platform should allow application-specific optimizations of the middleware itself.

Related to flexibility is *scalability*. Homburg [2001] defines scalability of a design as the "ability to support smaller and larger implementations of that design with adequate performance." Scalability can therefore be considered an additional requirement for an application-development platform: it should allow applications to evolve from small-scale (either numerically, geographically or administratively) to large-scale, while preserving the design and retaining required performance levels. Supporting scalability enhances flexibility for the application developer because his[1]application is then able to accommodate future changes in scale which could not have been predicted in advance. It should be clear that supporting scalability further complicates the design of the development platform. More on developing scalable distributed applications can be found in [Van Steen et al., 1998b].

For successful application development more is needed, however, than flexible and comprehensive support for scaling nonfunctional complexity. During application development, a developer should be able to separate the functionality of the application from its nonfunctional aspects, such as performance and fault tolerance. A middleware platform for large-scale distributed applications should therefore provide not only the means for dealing with the nonfunctional complexity of such applications, but should also make these aspects transparent, such that a developer is able to focus on just functionality in the relevant stages of development. In other words, a middleware platform should provide transparency for distribution, migration, replication and failure [ISO, 1995].

Many researchers have recognized the usefulness of the object-oriented programming (OOP) paradigm in providing these transparencies [Jul et al., 1988; Dasgupta et al., 1991; Birrell et al., 1993; Mitchell et al., 1994; Makpangou et al., 1994]. The OOP paradigm is based on the concept of an object combining data and the operations on that data into a single package. The fact that the object's data can be accessed only via the object's operations provides a natural mechanism for shielding the application programmer using the object from the details of how and where the operations on the data are actually performed. Using this mechanism we can also hide distribution, migration, replication of the object and (in many cases) mask failures from the client. Generally, the operations on the ob-

[1] Please read "his" as "his or her" throughout this dissertation.

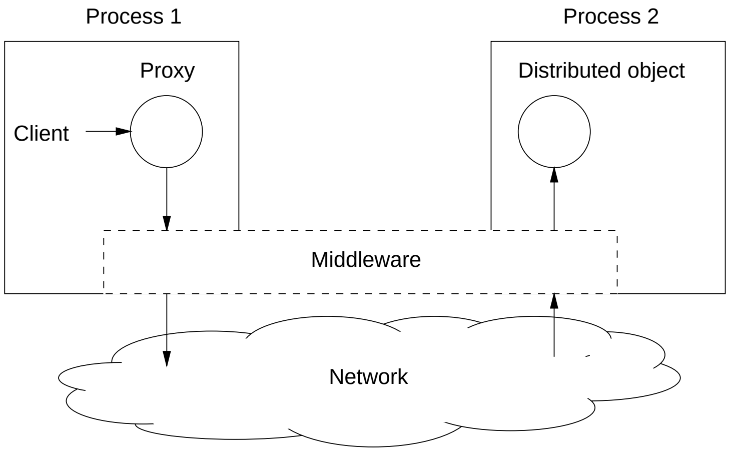


Figure 1.1: The remote-object model.

ject's data can be implemented without considering the distribution of the object and the application developer can design and implement his application as a collection of interacting objects as in the nondistributed case. Of course, he still faces the problems of software engineering in-the-large if there is much functionality to be provided [Tichy, 1992].

Although many of today's middleware platforms are object-based, they currently lack the extensive and flexible support we argue are necessary for developing large-scale distributed applications. An important reason for this lack of support is that these platforms were developed with local-area networks in mind, which place less demanding requirements on the middleware due to their smaller scale. Extending these platforms to make them suitable for increased scale may be possible, but we believe that this will result in platforms with are hard to use, and instead suggest a different approach, which entails adopting a different model of what a distributed object is [Bakker et al., 1999].

The current object-based middleware platforms, such as CORBA-based object request brokers [Object Management Group, 2001], Microsoft's DCOM [Eddon and Eddon, 1998] and Java RMI [Wollrath et al., 1996] are all based on a similar model. In this model, which we call the *remote-object model*, a distributed object is an object running on a remote machine, but which is presented to clients as a local object by means of proxies. A middleware layer mediates between the clients and the object and, in particular, takes care of the transport of requests and replies between the client and the object over the network. This model is illustrated in Figure 1.1.

The implication of the remote-object view is that all distribution aspects of the distributed object are managed by the middleware layer. This approach implies, in turn, that to support large-scale distributed applications this layer should be flexi-

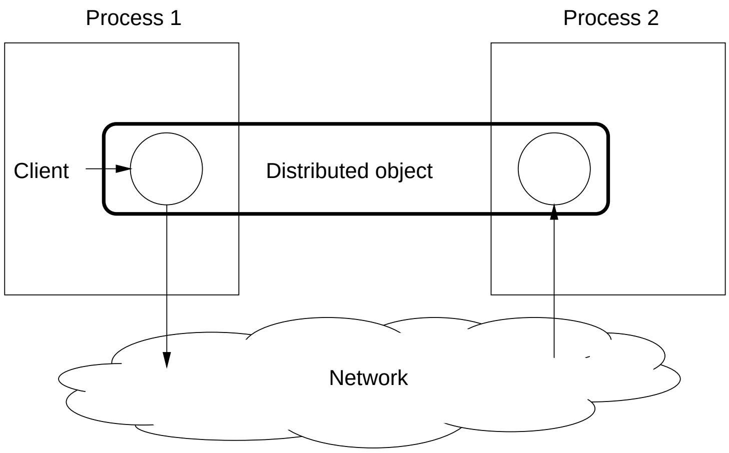


Figure 1.2: The distributed shared object model.

ble enough to allow an application developer to plug-in and configure application-optimal protocols and policies. It is not yet clear how such a middleware layer can be made flexible. A number of efforts at improving the flexibility of these middleware layers are, however, underway [Hayton et al., 1998], [Eliassen et al., 1999; Wang et al., 2001; Bea Systems et al., 1999]; many of them are based on reflective techniques [Kiczales et al., 1991].

In the Globe project, an alternative way of looking at distributed objects was developed. This model of distributed objects is called the *distributed shared object (DSO) model*. In this model, a distributed shared object is a distributed entity, an object physically distributed over multiple machines with the object's *local representatives* (proxies and replicas) cooperating to make the object's functionality available to local clients. In other words, a distributed shared object is a wrapper encompassing all the object's proxies and replicas, rather than a remotely accessible object implementation. This model is illustrated in Figure 1.2.

The advantage of this model over the remote-object model is that the distributed object itself is in charge of its distribution aspects. This results in flexibility for the application developer because he is no longer bound by the facilities of a middleware layer, but can apply any technique, protocol, and policy to any object and thus customize the nonfunctional behavior of his application. We claim that this model is a better basis for a development platform for large-scale applications because it provides a more natural, flexible, and extensible way for handling the complex nonfunctional aspects of such applications, in isolation from functional aspects.

To show the validity of the project's claim we have developed a middleware platform based on this object model and a number of large-scale management services in support of this platform. We have currently built two large applications: *GlobeDoc*, a more efficient replacement of the World Wide Web (as a hypertext system) [Van Steen et al., 1999b], and the *Globe Distribution Network (GDN)*, which is the topic of this dissertation.

1.2. THE GLOBE DISTRIBUTION NETWORK

With the rise of the home and personal computers came the ability for people to write their own software. The willingness of the authors to share these programs with others, and the interest of others led to widespread sharing of these programs. Sharing via transportable media (i.e., tapes or floppies) evolved into sharing via bulletin board systems (BBSes) with the advent of modems. In recent years, the Internet has greatly accelerated the sharing and development of this so-called *freely redistributable software*, to a point where millions of people are using the software, hundreds of thousands are participating in its development, and corporations are built around packaging the software for users.

The Globe Distribution Network is an application for the efficient, worldwide distribution of freely redistributable software packages, such as the GNU C compiler, the GIMP graphics package, Linux distributions, and shareware [Bakker et al., 2000, 2001a, b, 2002]. In the future we may extend it to distribute also other free content, such as free digital music. Distribution of software via the Globe Distribution Network is made efficient by encapsulating the software into distributed shared objects employing a replication protocol that efficiently replicates them in areas with many downloading clients, based on past and present usage patterns. The Globe Distribution Network takes a novel, optimistic approach to stop the illegal distribution of copyrighted and illicit material via the network. Stopping illegal distribution is important, because otherwise people and organizations making resources available to the GDN run the risk of being prosecuted for enabling illegal publication. The GDN is designed to be high available and behaves in a well-defined manner when failures can no longer be masked.

For its basic functionality the GDN can be compared with a number of systems. It is similar to the commercial content distribution networks, such as Akamai's Freeflow/EdgeSuite [Akamai Technologies, Inc., 2002], Exodus' 2Deliver Web Service [Exodus, 2002] (formerly Digital Island), and Speedera's Content Delivery [Speedera Networks, Inc., 2002]. It can be compared with wide-area file systems such as AFS [Howard et al., 1988], or to the rapidly growing class of peer-to-peer networks [Oram, 2001], such as Freenet [Clarke et al., 2002], CFS [Dabek et al., 2001b] and OceanStore [Rhea et al., 2001]. It is not targeted at cooperative development of free software over the Internet and is therefore not related to distributed source-code control systems such as (remote) CVS [Cederqvist et al., 2001], GTE [Goa et al., 1999] or Visual SourceSafe [Microsoft Corporation, 2001b; Source Gear Corporation, 2001].

We chose the distribution of freely redistributable software as an example application for a number of reasons. The most important reason is that the application itself has many interesting aspects. Many people are interested in free software, and many people are creating *free software*[2], resulting in an application that is large in terms of numbers of users and reasonably large in the amount of data that needs to be handled. To give an example, the SourceForge Web site aimed at supporting the development of free software over the Internet has over 450,000 registered users, and hosts close to a Terabyte of software (July 2002). The inherent worldwide nature of the application implies it has a large scale geographically. The application also has interesting security aspects. Unauthorized modification of the software being distributed must be impossible and malicious persons should not be able to use the GDN to illegally distribute copyrighted or illicit material. Furthermore it is our intention to let the GDN use spare server capacity provided by many different people and organizations, resulting in a large administrative scale for the application. This design goal also allows us to study the impact of untrusted servers on application design.

A second reason for choosing software distribution is that the current Internet applications for distributing free software are in need of an update. FTP and HTTP have proven to scale quite well, but replication and security have been added onto, instead of integrated into the applications. As a result, a lot of things in particular with respect to replication still have to be done by the user. These include finding out which mirror sites exist, dealing with site failures and handling inconsistencies between mirrors (caused by the periodic pull model applied in many mirroring solutions). The Globe Distribution Network provides an integrated solution where failures only very rarely require human intervention. Modern content delivery networks also offer an easy-to-use, integrated solution, but are not yet (widely) available to free-software publishers. The source code of the GDN is made freely available.

Note, however, that the primary purpose of designing the Globe Distribution Network is to show how a distribution network can be built using the Globe middleware, as a validation of the middleware, rather than designing the best distribution network that solves the problems of FTP and HTTP.

[2] The term “free software” should be read as “freely redistributable software” in this dissertation, although the former term usually denotes open-source software (a subset of freely redistributable software as it does not include shareware).

1.3. CONTRIBUTIONS

This dissertation makes the following contributions:

- It shows how a distribution network for free software can be and has been built using the Globe middleware platform. The architectural principles and design are discussed in detail in this dissertation.
- It shows how the illegal distribution of copyrighted works and illicit content in a (software) distribution network can be prevented using an optimistic cease-and-desist method.
- It demonstrates how these ideas can be applied in practice. As of December 2001, our prototype implementation is running on five hosts located around the world, hosting up to 20 GB of data.

1.4. STRUCTURE OF THE DISSERTATION

Chapter 2 outlines the requirements for a worldwide distribution network for freely redistributable software packages and offers brief analyses of the problems involved in implementing these requirements. Chapter 3 gives an overview of the Globe middleware platform, providing the necessary background for the rest of the dissertation. In Chapter 4, I discuss how software is encapsulated in distributed shared objects, how these objects replicate themselves over a network of object servers, and how this leads to an efficient system for distributing software. The measures necessary to ensure secure and legal operation of the Globe Distribution Network are identified in Chapter 5, which also describes our initial implementation of these measures. In Chapter 6, I address the application's requirements with respect to fault tolerance and durability and our initial approach to satisfying these requirements. Chapter 7 presents some performance figures for the initial implementation running on a number of sites around the world. Chapter 8 discusses related work. Finally, conclusions of this dissertation are drawn in Chapter 9.

CHAPTER 2

General Requirements

In this chapter, we will identify general requirements for a worldwide distribution network for freely redistributable software and briefly analyze the problems associated with implementing these requirements. Due to the complexity in their implementation, a number of requirements will not be investigated further in this dissertation as they warrant a separate investigation.

To be able to better describe the desired functionality, we first introduce some terminology in Sec. 2.1. This terminology is based on definitions by Conradi and Westfechtel [1998]. Sec. 2.2 identifies requirements and implementation issues regarding basic distribution functionality. Sec. 2.3 deals with query functionality and Sec. 2.4 discusses desired functionality and issues with respect to notifying users of new releases of software packages and automatic updates of installed software. Security requirements, fault-tolerance and application-management requirements are discussed in Sec. 2.5, 2.6 and 2.7, respectively. Sec. 2.8 lists the selection of requirements that are investigated in this dissertation.

2.1. TERMINOLOGY

A *software package* is an application, a library, or any piece of software that is published as a separately named entity. Examples are the GNU C compiler, the Ghostscript Postscript interpreter and the Linux kernel. A software package continuously evolves as bugs are fixed, new functionality is added, or when it is adapted to changes in other software packages on which it depends (e.g. libraries). This evolution results in a string of *revisions*; that is, versions that are meant to replace other, earlier versions. Often software development is not linear but proceeds along a number of parallel paths. For example, in one line of revisions new functionality and new designs are tried out, while in another line no new func-

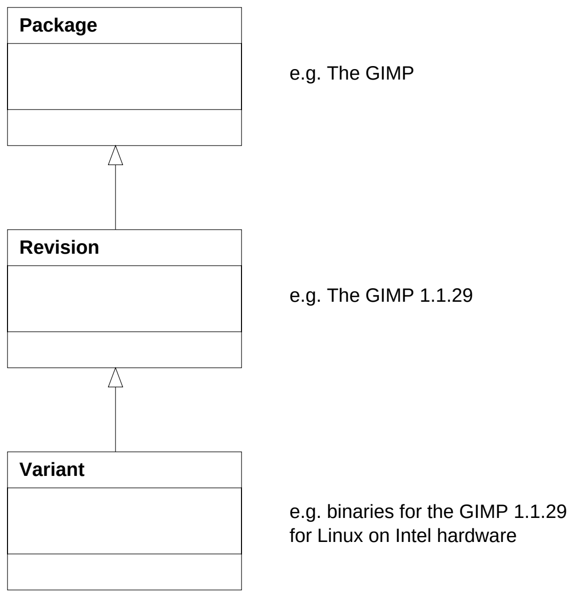

Package
e.g. The GIMP
Revision
e.g. The GIMP 1.1.29
Variant
e.g. binaries for the GIMP 1.1.29
for Linux on Intel hardware

Figure 2.1: The relationship between packages, revision and variants in pseudo-UML notation[Rumbaugh et al., 1999]. The open arrows represent a generalization relationship.

tionality is added and the code is changed only to fix bugs and to handle changed application programming interfaces (APIs).

Each revision of a package can have a number of *variants*; that is, versions somehow derived from that revision which are not meant to replace it, but instead coexist with that revision. An example of variants is formed by compiled binaries for different platforms. However, a revision can also have multiple source-code variants specifically targeted towards a particular platform when the code cannot be or is intentionally (e.g. for performance reasons) not made platform independent. Multiple source-code variants may also be found in cases where the software package needs to support a user interface in widely differing languages (Western vs. Arabic or Asian). The term *version* is used to denote either a revision or variant of a software package. The relationship between packages, revisions and variants is shown in Figure 2.1.

A variant may be published in one or more *file formats*. These can be generic file archive formats (ZIP, GZIP-ed TAR) or specialized formats for packaging software, such as RPM [Bailey, 1998] (used in the RedHat Linux distribution) or the DEB format [Software in the Public Interest, Inc., 2001] as used in Debian's

Linux distributions.

The term *diff* or *patch* is used to denote the set of differences between the source code of two successive revisions of a software package. Diffs are generally distributed in specialized file formats readable by tools such as diff and patch on UNIX systems. These tools are able to change the source code of the older version to that of the new version based on the differences specified in the diff file.

As already mentioned, a software package may depend on several other software packages which provide part of its functionality, such as libraries. As a software package evolves it may come to depend on different packages at different points in time. Each *revision* of a software package generally depends on a specific set of other software packages. A revision of a software package does, however, not always depend on specific revisions of the required packages. A revision of software package *A* may be able to function with any revision from a set of compatible revisions of a specific software package *B*. A dependency where a revision of a software package depends on a specific revision of another software package is called a *specific dependency* (sometimes referred to as a *bound dependency*). The term *generic dependency* is used for a dependency of a revision on a set of revisions of another package, identified by some expression. An example of generic dependencies can be found in the RPM software distribution format [Bailey, 1998]. This package format allows the specification of generic dependencies of the form

$$<\textit{package name}> <\textit{operator}> <\textit{revision ID}>$$

where *operator* is a standard comparison operator ($<, <=, =, >, >=$.) The *revision ID* is a package-specific identifier (e.g. 2.4.2). When a package in RPM format is installed (using the associated rpm tool), the dependencies for the package being installed are matched against the database of packages already installed on the computer and if the dependencies are not satisfied (because either the installed packages are too old or too new) the installation is aborted. RPM also checks dependencies when installed packages are updated or removed.

A specific revision of a software package and the (sets of) revisions of other software packages it depends on is an example of a *configuration*. A configuration, in general, is a collection of revisions or variants of revisions that have certain properties. An example configuration is the collection consisting of the latest revisions of a software package and all the packages it depends on, which may be of interest to anyone wanting to use the bleeding edge of all software packages. Another example is a set of revisions which are known to be working together reliably. Such a tested configuration may be of interest to a system administrator striving for stability on his system. Orthogonal to properties related to revision histories are variant and file-format properties. For example, one user may be interested in the source-code variant of the packages in GZIP-ed TAR format and while another might prefer compiled Linux-i386 binaries in RPM format.

A *distribution* is a named collection of software packages. A distribution has its own revision history (unlike a configuration), and can be published in multiple variants (i386, Alpha, SPARC). Well-known examples of distributions are the free operating systems, such as RedHat Linux, Debian Linux, and FreeBSD. However, a distribution does not necessarily encompass a complete operating system. Other collections of packages, such as the GNOME and KDE desktop environments, are also considered distributions since they are separately numbered (e.g. GNOME 1.4, KDE 2.0).

A *software producer* is a person who develops or currently maintains one or more software packages. Some packages have more than one producer at a time and the group of producers of a package may vary over time. In the free-software community such a person is often referred to as the *maintainer* of a software package.

A *local software management system (LSMS)* is an application for managing the installation and deinstallation of software on a particular computer. Examples are RedHat's RPM [Bailey, 1998], Debian's APT-GET [McCarty, 1999], Sun Microsystems' pkg suite, and Microsoft's Windows Update. For the purpose of this dissertation this term also applies to site-wide software management systems such as Microsoft's SMS [Microsoft Corporation, 2001a].

2.2. BASIC DISTRIBUTION FUNCTIONALITY

The basic functionality of a software distribution network is to allow producers of free software to make available different variants, new revisions and diffs of their software packages in various file formats to anyone who is interested. The challenge in implementing this functionality is how to deal with the large amount of work for servers and network being created by the large number of interested users and the amount of the free software being made available.

Currently, many people are interested in utilizing free software, although their number is hard to quantify as reliable statistics are not readily available. Free software has traditionally been popular at educational institutions, and more and more people are using free Web browsers, audio players and such at home. Looking at free operating systems, the number of Linux users is estimated at 5 to 15 million [Penfield Jackson, 1998; IDC, 2001], and Linux is growing rapidly in the $0–$100K server market [IDC, 2000]. Many people are still installing the software from a free or low-cost CD, but I expect that as people get access to more and cheaper bandwidth, more will start using the Internet to download the software.

The current volume of free software being published is unknown, but evidence suggests it is considerable. We can get an impression of this volume by looking at two Internet sites prominent in the open-source community: sourceforge.net and ibiblio.org (formerly sunsite.unc.edu). In March 2001, the SourceForge site hosted Web pages for 17000 open-source projects, corresponding to at least 17000 software packages.[1] The total amount of freely redistributable software made available through their FTP servers at that time was 774 Gigabytes (measured using a directory listing of the complete site). This collection consists of various revisions and variants of software packages, as well as various revisions and variants of Linux and *BSD distributions. There is considerable overlap because, for example, distributions are offered in a number of ways, for example, both as an ISO9660 Compact-Disc image and as a "live" file system (i.e., as it would appear on a user's hard disk after installation). More than half of the data is in a compressed data format. The ibiblio.org site hosts an archive containing most software that runs on the Linux operating system. Measured from a directory listing of this archive, it currently holds 67 GB of data, which includes a number of distributions. Looking at these two sites, the current volume of free software appears to be in the order of several hundred Gigabytes.

How much work the servers and network have to do when running a free software distribution network depends on

1. the number of downloaders,
2. how often they download software, and
3. the size of software packages being downloaded.

The frequency of downloads is, in turn, determined by (1) how often new packages and new revisions appear, (2) how often people download software packages they did not use before and (3) how often people update already installed packages. The amount of work to be done by servers and network can be characterized as large, given that there are potentially many downloaders and their number is likely to increase when they get access to more bandwidth. Moreover, there are many free applications available for various (free) operating systems.

The actual amount of work for a free software distribution network at a particular point in time depends, of course, on the number of simultaneous downloaders and how much software is being downloaded. It is hard to estimate what the size of this work load would currently be, and how it will develop in the future. What is known is that, at present, it exhibits peaks, generated by so-called *flash crowds* [Nielsen, 1995]. When a new revision of a popular software package is released,

[1] In June 2002, the number of projects on SourceForge was 43000, a considerable increase.

it frequently happens that a very large group of users tries to download the newly released version at the same time.

In summary, the challenge to implementing the basic functionality of a software distribution network is to distribute the download traffic such that efficient use is made of the available servers and network, and overload is avoided. Preferably, the implementation should be scalable to accommodate longer-term increases (and decreases) in work load that may occur in the future. In the past people have partially solved this problem using crude mirroring solutions. More recently, content distribution networks have provided a more elegant solution, but they are not (widely) available to free software publishers. The Globe Distribution Network will have to solve this problem using the facilities of the Globe middleware.

2.3. BASIC QUERY FUNCTIONALITY

It is important that a software distribution network enables the interested users to find the software they are looking for. A software distribution network should enable various ways of finding the desired software.

A basic requirement is that a user is able to retrieve the software he is looking for based on the unique name for the package and the identification of the specific revision and variant, as assigned by the software producer (e.g. "gcc 2.95.2"). The challenge here is to provide location-transparent naming of the software. Because of the load balancing required for efficient distribution, software may be replicated in many locations. This set of locations is dynamic and changes due to failures, deliberate migration and introduction and removal of servers. A location-transparent naming facility is therefore necessary to provide users with long-lived identifiers for software which can be freely stored and communicated (cf. Uniform Resource Names [Sollins and Masinter, 1994]). Pitoura and Samaras [2001] provide a survey of large-scale location-independent naming systems.

A software distribution network should also allow users to browse the set of available revisions and variants of a package to find the desired one. To this extent, a structured overview of the revision history, which includes the relationships between revisions and which names the different development paths is helpful. Furthermore, a software distribution network should allow a user to find packages based on their properties. Based on a specification of the desired properties of the package sought, both in terms of functionality and in terms of supported platform, a user should be able to find the package in the software distribution network that best matches these properties. An example of a Web site that provides these facilities is http://www.download.com/ (in May 2002).

Implementing this functionality appears to be relatively straightforward: create a database containing the required information and configure a query-processing engine that can scale to a large number of simultaneous users. Apart from query handling, the challenge is to prevent malicious persons from polluting the database, which would diminish its usefulness. Possible solutions for the latter problem are censorship by moderators or reputation systems [Lethin, 2001]. We discuss security matters in more detail below.

Related to attribute-based retrieval of software packages is the retrieval of a software package along with the packages it depends on. The ability to retrieve and install a consistent configuration of software packages (i.e., a configuration that matches the specified dependencies) in a specific variant is highly valued. This functionality is, for example, provided by Ximian's RedCarpet software management system [Ximian, Inc., 2002]. Implementation issues here lie again in setting up a database, keeping this database up-to-date and accommodating the query load. This type of database is very much akin to design and engineering databases, in which component dependencies, and multiple configurations (tested/untested) are also part of the application domain [Katz, 1990]. Important in the context of software distribution is the relationship of the query software to the local software management system (LSMS). The LSMS has information about already installed packages and their internal dependencies. Service to the user is greatly improved if this information is used as additional input to their queries. It enables the query processor to determine which required packages are missing and signal conflicts with already installed software. Information about what revisions of packages are installed on a (networked) computer is security sensitive information. We return to the relationship between a software distribution network and local software management systems in the next section, and to the security issue in Sec. 2.5.

2.4. NOTIFICATION AND AUTOMATIC UPDATE

Other useful functionality is the ability to be notified of the publication of new variants or new revisions of a specific software package. This functionality is useful for both regular users and software producers. Users should be able to subscribe to notifications for their favorite software packages, relieving them of the burden of regularly checking (i.e., give them the advantage of automatic push of notifications vs. periodic, manual pull). Producers can register to be notified of new versions of packages their own software depends on, allowing them to check for changes in those packages that may affect them.

To implement this functionality we need a publish/subscribe system that can keep track of the subscriptions of, and send notifications to, large numbers of users. The publish/subscribe system should support asynchronous notification;

that is, store notifications for people not currently online. To support changes in usage patterns the system should also be scalable. The challenge in building such a system lies in distributing the notification to the potentially millions of interested parties without generating excessive duplicate network traffic. Large-scale event notification is a well-researched subject, see, for example, PIM [Deering et al., 1996], the Cambridge Event Architecture [Bacon et al., 2000], or Siena [Carzaniga et al., 2000]. The SourceForge free-software site currently supports notifications of updates via e-mail.

Related to notifications about new versions is auto-update functionality; that is, support for the automated retrieval and possibly installation of updates (i.e., new revisions) of software packages already installed on your system. This requires a coupling between the local software management system and the distribution network, in particular to make sure that the set of installed packages remains internally consistent (i.e., no dependencies are broken), as outlined in the previous section. Various degrees of coupling are possible, ranging from a management system using the dependency, query and/or notification facilities of the distribution network, to a solution where the publisher or vendor of a particular operating-system distribution only uses the distribution facilities of the network. Such flexibility is necessary to allow distribution publishers and vendors to retain full control over (the collection of) installed packages if they wish to do so. Hence, many Linux distributions today support this functionality (e.g. RedHat). An important issue here is authentication of the source of the updates, which we discuss next.

2.5. SECURITY REQUIREMENTS

Security is of particular importance to distributed applications operating in a public network such as the Internet. This statement holds even stronger for a software distribution network because of the sensitive nature of the content it distributes and the fact that is permits large-scale data sharing. Four (classes of) security requirements are identified:

1. ensuring authenticity and integrity of the distributed software
2. preventing illegal distribution of copyrighted or illicit material
3. anonymity of up- and downloads, and
4. high availability and protecting the integrity of the machines running the distribution network

We examine each of these requirements in turn.

2.5.1. Ensuring Authenticity and Integrity of Content

From a security perspective, download and execution of binaries or source code, without considering their origin, constitutes a considerable risk to the integrity of the computer the code is run on. The literature says little about the occurrence of backdoors or purposely destructive code in freely redistributable software. Dailey Paulson [2001] reports on an incident where a hacker published a malicious patch to BIND, a popular DNS-server, on a well-known security site. Many people applied the patch enabling the hacker to launch a denial-of-service attack against a network-security software company. A well-known example of a freely available application acting unethically is the case of Real Network's RealJukebox. This audio player sent back information about its user's listening habits which could be tied to the user's personal information without making this known to its users [Robinson, 1999]. According to Martin et al. [2001], freely downloadable software components from commercial organizations frequently show insufficient concern for users' privacy. The risk involved in using free software therefore, currently, appears to be disclosure of private information, which may include credit-card numbers and such.

Obviously, trust plays an important role in the domain of free software, and will continue to do so. Verifying that source code is benign is time-consuming, complex and beyond most users' capabilities. People often trade security for convenience and download a binary instead of compiling the source themselves, thus throwing away a possibility for checking the extent of the damage after an incident. Although the use of digital signatures [Schneier, 1996] to provide end-to-end authenticity and integrity guarantees cancels the risk of malicious modification of a package after publication, it does not provide guarantees about the good intentions of the original publisher.

A minimal requirement is therefore that a software distribution network allows its users to (easily) assure themselves of the origin and authenticity of the software downloaded and that it has not been modified while in the distribution network. It should preferably enforce the use of these facilities, because, at present, these facilities are not used by all software producers. Moreover, many of the downloading users do not perform the required associated checks, despite the fact that the current distribution infrastructure (DNS, FTP servers) is known to have security holes. Fortunately, authenticity checks are increasingly becoming part of development environments and run-time systems [Chappell, 1996; Gong, 1999], or are provided by publishers of (e.g. Linux) distributions.

The challenge in implementing this requirement first of all lies in devising a security process that allows these authenticity and integrity guarantees to be made with a sufficient level of certainty. A *security process* is defined as the procedures to be followed by the persons involved and the software measures to support and ensure safety of these procedures.

2.5.2. Preventing Illegal Distribution

One of the most pressing legal problems concerning the Internet today is the illegal distribution of copyrighted or illicit works, which it enables on a large scale [Samuelson, 1997; Macedonia, 2000; Davis, 2001]. With the transition from research to public network and the advances in bandwidth and compression methods over the last few years, the Internet has become the primary medium via which illegal copies of copyrighted works, such as software, music and videos, are being distributed. In addition, certain groups are using the Internet to share and publish materials such as child pornography and racist texts which may not be distributed or legally owned in most countries. Also considered potentially illicit content is *controversial free software*. Controversial free software is defined as software that uses patented technology, software that can be used to circumvent copyright-protection measures, software that employs strong cryptography, or software that contains (potentially) offensive material. The legality of such software varies from country to country.

A software distribution network should incorporate measures which make it impossible or at least difficult for malicious persons to illegally distribute these types of content via the distribution network. Or, more precisely, a software distribution network should adhere to all requirements imposed by international and local law to prevent illegal distribution. Otherwise, the persons or organizations participating in the software distribution network (i.e., running a component of the network on their machines) may be prosecuted for copyright infringements or illegal ownership or distribution of illicit content.

The challenging issue here is (again) finding a security process that, in this case, prevents the illegal distribution to the extent required by law. This process should encompass the differences that exist between countries with respect to what constitutes illicit content and illegal distribution. The process will impose restrictions on where (in which country) what software may be stored and thus influences the basic distribution functionality, as discussed in Sec. 2.3. It may even require access control to prevent people from downloading material illegal in their country from another country where it is legal. An important additional goal here is to protect software producers against false allegations. For example, it should not be possible for a malicious system administrator participating in the distribution network to accuse a software producer of doing illegal uploads.

This security process is likely to depend heavily on current regulations. A particularly complex problem is how to allow a software distribution to support different security processes over time, as regulation evolves, without radical changes to the application. I call this *legal evolvability*. We return to general evolvability of a distribution network in Sec. 2.7.

2.5.3. Providing Anonymity

The third class of security requirements concerns anonymity of the actions performed by users and producers. A software distribution network should allow people to download software from the distribution network anonymously. What software people like to use can be considered private information and should therefore be kept secret. Implementation complexity depends on the level of privacy required. If users simply do not want the distribution network to know who is downloading what, a simple mediator service such as the Anonymizer [Anonymizer.com, 2001] can be used, which adds a single layer of indirection and is aware of the hosts the users access. Systems such as Crowds [Reiter and Rubin, 1999] and Onion Routing [Goldschlag et al., 1999] allow a higher level of anonymity. They hide the origin and destination of traffic by forwarding it through multiple servers. A challenge for a large-scale implementation is therefore to ensure efficient use of network resources, while making sure the system still provides the expected anonymity guarantees.

In addition to keeping their downloads private, people will want to keep hidden which software is already installed on their machine while downloading. It is of interest to hackers intending to gain access to a certain site to know which revisions of which software packages are installed. By matching this information against a list of revisions and the known security bugs that exist in those revisions, they can easily find vulnerabilities in the system and exploit those. An implementation issue is how to keep the information about which revisions are installed private while using it to determine which software packages can be safely downloaded and installed (i.e., without breaking dependencies, see also Sec. 2.3).

The complementary requirement to anonymous downloads is allowing anonymous upload of software. The reason for providing this functionality is that for some controversial free software packages it is not clear whether or not they constitute illicit material, since the software has both legitimate and illegitimate uses. To resolve the ambiguity, the creators of the software would have to go to court. To not run the risk of a conviction and subsequent legal penalties, some creators prefer to publish the software anonymously. A well-known example of ambiguous software is the DeCSS code [Simons, 2000], which can be used both for playing Digital Versatile Discs (DVDs) on open-source operating systems and for obtaining the unprotected video material from the disc, thus allowing lossless and unrestricted further copying. An example of a system providing anonymous publication to ensure riskless free speech[2] is Publius [Waldman et al., 2000]. A number

[2] Source code has been recognized as a form of free speech in the United States [Burk, 2001].

of peer-to-peer networks also provide anonymous uploads [Oram, 2001].

The desire for publishing software without revealing your identity conflicts with some of the possible measures for preventing illegal distribution of content. For example, when the publisher is anonymous it is hard to block that person from the software distribution network when it is ruled that he illegally published the code (as in the DeCSS case at this time (March 2001)).

2.5.4. Availability and Integrity of Servers

A global software distribution network should be highly available; that is, it must be up and running most of the time. One reason for imposing this requirement, other than convenience, is the global nature of the application: when people in one time zone switch off their computer to go to bed, the people a few time zones later are fully awake and, as a consequence, the distribution network should be available 24 hours a day. Two factors can threaten the availability of a distributed application: deliberate attacks and hardware or software failures. The latter factor is discussed in the next section.

So, in addition to attempting to gain access to or destroy data on user's computers via malicious software, or use the distribution network for illegal distribution, we can expect that malicious persons may try to launch *denial-of-service (DoS)* attacks against the application. Countering denial-of-service attacks, such as flooding network connections to servers, is outside the scope of this dissertation because they can only be dealt with at the network/operating-system level. Nonetheless, a number of things can be done at the application level. Three types of DoS attacks are distinguished:

1. attempts to overload servers running the application
2. attempts to crash the application or servers, and
3. attempts to impede the operation of the application by corrupting its data

We briefly summarize how these type of attacks are generally dealt with.

Overloading: An important observation to make is that a denial-of-service attack poses, to a large degree, the same challenge as a crowd of people all trying to download a new package immediately. Hence, the facilities for handling flash crowds also help in countering DoS attacks. In general, the way to counter DoS attacks is to have the clients accessing the service do more work than the server(s) offering the service. One example is for a server to create an easy to generate but hard to solve cryptographic puzzle for each new client and accept the service request from the client only when the client has shown it solved the cryptographic puzzle [Dwork and Noar, 1992]. Because the puzzle is easy to generate but hard to solve the client has to do more work than the server, making it harder for the client to overload that server.

It is clear that this method affects performance, in particular, it increases the latency of operations. The challenge lies in providing a scalable mechanism which increases the workload for the client as the load on the service increases. For a software distribution network, this protective measure should be activated only when the available server resources are nearly exhausted to prevent severe degradation of performance during flash-crowd traffic. There is evidence that some denial-of-service attacks can be distinguished from flash crowds, in which case the countermeasures can be activated during attacks only [Jung et al., 2002].

Crashing and corrupting: A particular class of denial-of-service attacks is denial-of-service by resource destruction; that is, hackers trying to crash servers by exploiting programming errors in the application. The risk of this type of attacks occuring is larger when the source code of the application is available. These attacks can be countered by employing sound programming practices (e.g. rigorous input checking), authenticating users and logging their actions such that when they act maliciously they can be blocked, and enable fast upgrade of the application to deploy fixes quickly. For a software distribution network, requiring that users authenticate themselves not just for uploads but also for read operations such as downloads may be considered too strict by its prospective users and could conflict with the anonymity requirements just discussed. As with denial-of-service attacks by overloading, the challenge is to balance the counter measures with the other requirements of the application.

Attempts at exploiting programming errors to corrupt the application's data can be countered using the same methods as attempts to crash the application. Sound programming practices ensure that little vulnerabilities exist. User authentication limits the number of possible attackers, and action logging ensures that when vulnerabilities are exploited the time frame and extent of the damage are known. Requiring user authentication may not be possible for download operations in a free software distribution network, as just mentioned. An important new dimension is containment; counter measures should be introduced to prevent the spreading of corruption. Containment is required to keep a large-scale application manageable. We come back to manageability in the next section when discussing failure requirements.

Server Integrity: Programming errors cannot only be exploited to crash servers or corrupt data, they are also frequently used to gain access to the computers hosting the application. Again, what is needed are sound programming practices, authentication and logging, enabling rapid distribution of fixes and limiting what an attacker can do, for example, by assigning only minimal permissions to the application, such as enabled by the Java 2 platform [Gong, 1999].

2.6. FAULT-TOLERANCE REQUIREMENTS

In this section we identity fault-tolerance requirements for a software distribution network. As already mentioned in the previous section, an important requirement is high availability. Providing near-continuous service despite hardware and software failures in a wide-area network with many computers is nontrivial, to say the least.

Preferably, a software distribution should also be *reliable*; that is, provide uninterrupted service for long periods of time.[3] Reliability is important because it keeps a large-scale distributed application manageable. Operations on a software distribution network may involve a large number of computers (e.g. uploading a new version of a popular software package to a large set of hosts in anticipation of a flash crowd). This property makes the application more susceptible to partial failures, while at the same time making it hard for an application's administrators to correct (the effects of) these failures. What adds to the complexity of recovery is the fact that the distribution network processes many simultaneous operations. To keep the distribution network manageable it is therefore important that the application itself tries to recover from failures whenever possible.

Making a large-scale distributed application operate reliably is complex. First, reliability is a systemwide property and implementing this requirement therefore affects all components of the application. Second, redundancy is a powerful technique for coping with failures, but fault tolerant replication protocols capable of handling all possible failure scenarios are complex to design, verify and implement. This statement holds in particular for protocols to be used over the Internet, given the Internet occasionally suffers from network partitions. The ISIS toolkit [Birman and Van Renesse, 1994] offers the most successful protocol suite for building reliable applications. Third, combining fault tolerant replication protocols with distributed object-based programming can raise a number of complex issues [Bakker et al., 1998]. Fourth, it is unclear how to accurately and securely detect failure of an application component in the Internet. There is evidence to suggest failure detection is practically feasible with a sufficient degree of reliability [Stelling et al., 1998]. Finally, a software distribution will, most likely, have to be made reliable by means of software. Hardware fault tolerance, for example,

[3] Availability of an application concerns the total uptime of the application in a given period (e.g. a year), the reliability of an application is measured by the mean time between failures.

fault-tolerant servers, costs money and given that most freely redistributable software is also free in monetary sense, cannot be afforded unless donated or some business model for generating revenue exists.

As an additional requirement to reliability, a software distribution network should exhibit *strong failure semantics* [Cristian, 1991] in case a failure cannot be compensated for. In other words, when a failure can no longer be masked, the application should at least attempt to restore the application to a clearly defined state. This requirement is imposed, again, to keep the application manageable in the presence of failures. For a global software distribution network, operations must either succeed or fail and in the latter case restore the application to its state before the operation was started. These failure semantics are referred to as *atomic with respect to exceptions (AWE)* semantics.

Exhibiting strong failure semantics, like reliability, is also a systemwide property and equally complex to implement. The complexity lies in analyzing the application to determine all single and multiple-failure scenarios, devising the appropriate measures and verifying that they indeed provide the desired guarantees in all scenarios. This process is particularly complicated if the application has a large number of components.

2.7. MANAGEMENT REQUIREMENTS

A software distribution network may need a resource management system for managing the available server and network resources, at both the global and the local level. If there are limited resources available to the application, a global management system is required to prevent popular or large software packages from taking up too many resources, and thus hindering the efficient distribution of other software packages. As this functionality may be required by multiple applications it should be provided by the middleware platform. Fair allocation of resources on a global scale is a relatively new field of research. Current efforts concentrate on resource allocation and accounting in computational grids [Foster and Kesselman, 1998], as discussed in [Wolski et al., 2000; Grid Forum, 2001].

The distribution network may also need to support a local, that is, per-server resource management system. When a person or organization provides server capacity they should be able to specify how many resources (bandwidth, memory, disk space) the distribution network is allowed to use. This functionality should be provided either by the operating system or the middleware platform. Some operating systems allow servers to impose quotas on resources, but these facilities are not always used and other operating systems do not have these facilities. If these facilities are missing or turned off because they are too expensive to use, the middleware should provide application-level facilities to limit resource usage. WebOS [Vahdat et al., 1998] is an example of a middleware platform that provides its own local resource management system. JRes extends the Java run-time environment to provide resource management [Czajkowski and von Eicken, 1998].

It is important that a distribution network can be adapted to new functional requirements and changes in usage patterns and in its operating environment, that is, the Internet. I call this *evolvability*. At the lower end of the evolvability spectrum is, for example, the ability to support new, more efficient, replication protocols; at the other end is the ability to dynamically update the application while it is running. Scalability of an application can be seen as the ability to adapt to new usage patterns without modification to its design. If we consider high availability as the ability of the application to survive in space, we can consider evolvability the ability of the application to survive in time.

2.8. FOCUS OF THIS DISSERTATION

Unfortunately, it is beyond the scope of this research to develop a software distribution network that satisfies all the requirements identified above, in particular, because of the complexity involved in implementing some of these requirements. I therefore confined myself to investigate the following requirements:

- Basic upload and download facilities,
- All security requirements,
- All fault-tolerance requirements,
- Global and local resource management.

This selection means I will not investigate (1) how to specify, store and resolve dependencies between software packages and downloading consistent configurations of required packages, (2) querying functionality more advanced than downloading a software package using location-independent identifiers, (3) the ability to receive notifications of new software releases and auto-update facilities, and (4) evolvability of the application.

CHAPTER 3

The Globe Middleware Platform

This chapter gives an overview of the Globe middleware platform. It provides the background necessary to understand the description of the design and implementation of our worldwide software distribution network, the *Globe Distribution Network* presented in the following chapters. The Globe middleware platform is sometimes referred to as a worldwide distributed system. These designations are considered to be the same for the purpose of this dissertation. The Globe middleware platform is designed to support 1 billion (10^9) users. Homburg [2001] provides a more complete description of Globe and its relationship to other worldwide distributed systems such as Globus [Foster and Kesselman, 1997] and Legion [Grimshaw et al., 1999].

Sec. 3.1 describes Globe's distributed shared object model. The implementation of this model is explained in Sec. 3.2. Sec. 3.3 discusses the various naming services of Globe and how they can be used to gain access to a distributed shared object. Sec. 3.4 describes the steps involved in a method invocation on a distributed shared object, and Sec. 3.5 explains how a DSO is created. Sec. 3.6 gives a more detailed explanation of Globe's object-location service. The basic functionality of our special server software for hosting local representatives and our services for locating available servers willing to host local representatives are discussed in Sec. 3.7 and Sec. 3.8, respectively.

3.1. DISTRIBUTED SHARED OBJECTS

Globe, like other object-based or object-oriented[1] middleware platforms, is based on the concept of distributed shared memory (DSM) [Li and Hudak, 1989].

[1] Object-based systems, unlike object-oriented systems, do not support inheritance [Wegner, 1987].

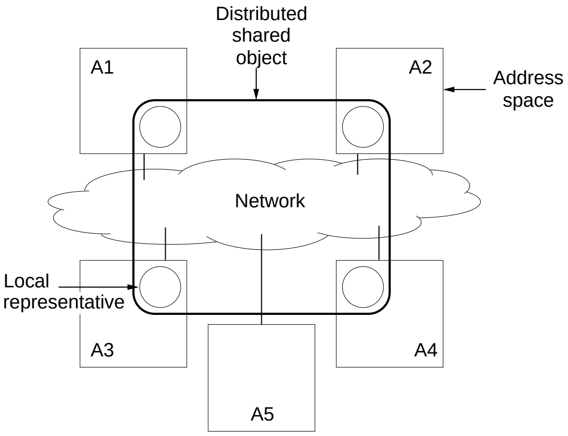


Figure 3.1: A distributed shared object distributed over four address spaces (A1-A4). In each address space, the DSO is represented by a local representative. Address space A5 does not currently participate in the distributed shared object. This figure is reproduced from [Homburg, 2001].

In DSM systems, processes communicate by reading and writing a (logically) shared address space. In the case of object-based shared memory systems, this address space consists of a collection of objects combining a piece of data with user-defined methods for manipulating that data. Remote processes communicate indirectly by invoking methods on these *distributed objects*, thus reading and writing the encapsulated data. In general, the distributed-object concept allows for a uniform representation of both information and services and provides implementation flexibility by decoupling interface and implementation.

As briefly described in the first chapter, a design goal for the Globe middleware is supporting many different implementations and enabling application-specific solutions when necessary. This idea is reflected in its model of objects. Globe differs from other object-based or object-oriented middleware platforms in its view of what constitutes a distributed object. In our view, a distributed object should have complete control over its (distributed) implementation. As a result, in Globe, a distributed object manages all its nonfunctional aspects itself, such as transport of method invocations, location and replication of its state, and security, using only a minimum of supporting services. We have called this model of distributed objects the *distributed shared object (DSO)* model [Van Steen et al., 1999a; Homburg, 2001].

Figure 3.1 illustrates the distributed shared object model. We view a distributed object as a logical unit that is physically distributed over multiple address spaces. The distributed shared object in Figure 3.1 is distributed over four address spaces (A1–A4). In each of the address spaces the distributed shared object is represented by a *local representative* (address space A5 in Figure 3.1 currently does not participate in the DSO and therefore does not contain a local representative). The power of the DSO model is that the implementation of a local representative can differ not only from address space to address space (cf. proxies and replicas) but also from DSO to DSO. By allowing each DSO to have different implementations of its local representatives, we are enabling per-object and per-application specialization of a DSO's implementation. The freedom to implement local representatives differently allows a distributed shared object to use the communication, replication, and security protocols best suited for that particular DSO or the application it is part of. In other words, the flexibility we argued is necessary for successfully developing large-scale distributed applications in Chapter 1 is provided by enabling each DSO to select the implementation of its local representatives.

Globe's philosophy and object model can be best illustrated by focusing on the management of the location and replication of a distributed shared object's state. Different applications have different nonfunctional requirements with respect to security, fault tolerance and performance. As a result, different applications have different requirements with respect to how, where, and to what degree the state of its objects should be replicated and with which consistency model [Adve and Gharachorloo, 1996]. Globe supports this multiplicity of replication protocols and policies as follows. Because replication and location of the state of a DSO is an aspect of a DSO's implementation, they are under control of the DSO itself. Concretely, they are under control of the object's local representatives. In other words, the local representatives of a distributed shared object contain all code for doing replication and communication. Combined with the freedom to select the implementation of local representatives on a per-object basis, we are able to use different replication protocols and policies for different objects and therefore different applications.

The implementation of local representatives is not completely left to the application programmer, however. The structure of local representatives (described in the next section) separates application, replication and communication code. Furthermore, all replication protocols are accessible via a standardized interface. This separation and standardization enables the creation of a library of reusable replication and communication-protocol implementations from which a programmer can select the ones appropriate for each object in his application (cf. Horus [Van Renesse et al., 1996]). Of course, it remains possible to implement a new application-specific protocol if no appropriate protocol is available.

The Globe philosophy is not just applied to the replication of an object's state. Research is currently also concentrating on the design and implementation of a

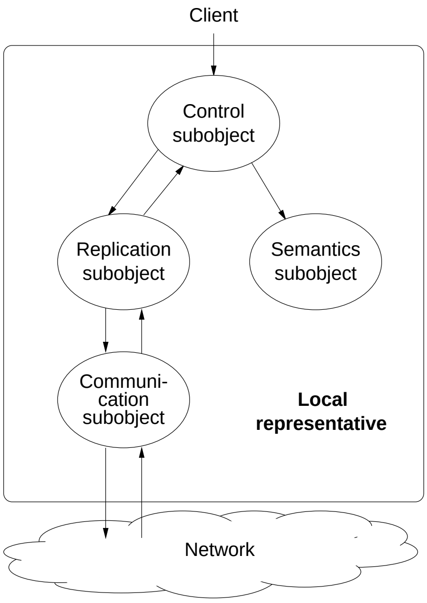


Figure 3.2: A local representative composed of four subobjects. Arrows indicate only an invocation, not the return of the invocation. Upward pointing arrows therefore represent upcalls.

framework to easily incorporate security [Leiwo et al., 2000; Hänle and Tanenbaum, 2000; Leiwo et al., 1999], and, in the future, fault tolerance. This framework will enable application programmers to choose the security and fault tolerance measures most suitable for their application and to introduce application-specific measures when desired.

3.2. IMPLEMENTATION OF A DSO

Physically, a distributed shared object consists of multiple local representatives. Each local representative resides in a single address space and communicates with local representatives in other address spaces. A local representative is composed of several so-called *subobjects* as shown in Figure 3.2. A typical composition consists of the following four subobjects:

Semantics subobject: This subobject contains the actual implementation of the distributed object's methods and logically holds the state of the object. The semantics subobject is written by the application programmer in a programming language such as Java, C, or C++. It can be developed without having to take many distribution or replication issues into account, but some issues an application programmer cannot ignore, as will be shown in Chap. 4. Accesses to a semantics subobject are serialized: at most one thread is active in a semantics subobject at a particular point in time.

Replication subobject: A DSO may have semantics subobjects in multiple local representatives for reasons of fault tolerance or performance. The replication subobject is responsible for keeping the state of these replicas consistent according to the consistency model for the distributed shared object. To this extent, the subobject communicates with its peers in other local representatives using an object-specific replication protocol. Different local representatives may contain different replication subobjects, implementing different roles of the replication protocol (e.g. proxy, master, slave). Different distributed shared objects will use different (sets of) replication subobjects depending on their needs, as just discussed. Also, as already mentioned, an important aspect of replication subobjects is that there is one standardized interface for all replication subobjects. Sec. 3.4 describes this interface in more detail.

Communication subobject: This subobject is responsible for handling communication between the local representatives of the distributed object residing in different address spaces, usually on different machines. Depending on what is needed by the other subobjects, a communication subobject may offer (reliable or unreliable) primitives for point–to–point communication, group communication, or both. This is generally a system-provided subobject (i.e., taken from a library).

Control subobject: The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. This subobject is needed to bridge the gap between the programmer-defined interfaces of the semantics subobject, and the standardized interface of the replication subobject. For example, the control subobject marshalls and unmarshalls method invocations and replies. Because of its bridging function a control subobject has both a standardized interface used by the replication subobject and programmer-defined interfaces used by client processes (in Globe, as in Java, a (sub)object can have multiple interfaces.) In general, the control object is generated using a stub compiler.

A local representative acting as replica for the distributed shared object contains these basic four subobjects. A local representative that only forwards method invocations and return replies (a proxy) does not contain a semantics subobject.

In the initial design of Globe, other subobjects have been defined, for example, for handling security functions. This design is currently being refined to proper security and fault-tolerance frameworks and will therefore not be discussed here. It is important to realize that, in principle, an application programmer can even define his own structure for local representatives, because the implementations of local representatives are loaded dynamically from an *implementation repository*, as discussed below. From a software engineering and reuse point of view, however, utilizing our structure is to be preferred.

3.3. NAMING AND BINDING

To access a distributed shared object (i.e., to invoke its methods), a client process first needs to install a local representative of the object in its address space. The process of installing a local representative in an address space is called *binding*. Before explaining binding, however, we first have to describe how naming is done in the Globe middleware [Ballintijn et al., 2001].

Each DSO in Globe is identified by a worldwide unique *object handle*. This object handle, consisting of an *object identifier* and some additional information, never changes during the lifetime of the object and, most importantly, is location independent. The actual locations of the DSO, that is, *where* (network address, port number) its local representatives are located, and *how* (which replication and communication protocol) they can be contacted is maintained by a special service, the *Globe Location Service (GLS)* [Van Steen et al., 1998a]. The information that identifies the location of a local representative and how to communicate with it is called a *contact address* of the distributed shared object. The Globe Location Service is designed to store contact addresses for 10^{12} objects.

Object handles are long strings of bits and therefore not human friendly. To improve usability, an additional name service has been introduced that maps symbolic names to object handles, the *Globe Name Service (GNS)*. This design results in a two-level naming scheme: symbolic object names are mapped to object handles by the Globe Name Service which are, in turn, mapped to one or more contact addresses for the object by the Globe Location Service. This process is illustrated in Figure 3.3.

Distributed shared objects are free to decide how much use they make of the Globe Location Service. Typically, an object will register (only) its local representatives acting as replicas in the GLS. But, following the Globe philosophy, an object may, for example, decide to register only a single central contact point and keep track of the location of its local representatives itself, when it can do so more efficiently.

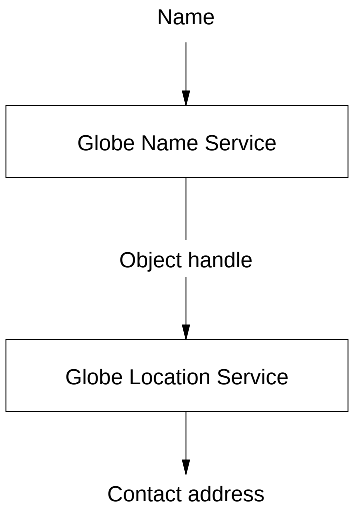


Figure 3.3: Globe's two level naming scheme.

Binding to a DSO (i.e., loading a local representative) now works as follows. For brevity let us assume that the client process has already acquired an object handle of the DSO whose methods it wants to invoke. The client calls a special function in the run-time system, named bind, passing it the object handle. The run-time system takes the object handle and asks the Globe Location Service to map it to one or more contact addresses. The returned contact addresses will identify the most convenient (e.g. geographically nearest) replica of the DSO. Using the information in the contact addresses, the local run-time system then creates a new local representative in the client's address space and integrates this new representative into the DSO. Creating a new local representative involves loading the implementation of the local representative from a nearby (trusted) *implementation repository*. The implementation consists of several *local class objects* which are loaded and then instantiated to create the local representative and the appropriate set of subobjects. Local class objects in a repository are identified by so-called *implementation handles*. The subobjects are initialized with information from the contact address. This initialization mechanism is used, for example, to supply the replication subobject with the details of the replication policy to be used. This procedure is similar to remote class loading in Java [Liang and Bracha, 1998].

Binding is described in more detail in [Homburg, 2001]. The design of the Globe Name Service can be found in [Ballintijn et al., 2000]. The inner workings of the Globe Location Service are described in Sec. 3.6.

We conclude this section by introducing some additional terminology. The

```
interface integer
{
    int set( in int i );
    int get();
};
```

Figure 3.4: Interface of our integer DSO. For didactic purposes the set method is made to return a result, namely the value of its in parameter.

replication policy of an item (either a distributed shared object, data item, or service part), is defined as an abstract specification detailing *how* (using which protocol and following which consistency model) an item is distributed or replicated and *where* (a specification of a set of hosts) the item or a replica of the item may be located. An item's *replication scenario* is the concrete implementation of that policy at a particular point in time; that is, it describes on which actual hosts copies of the item are currently located and (for DSOs) which replication subobjects are currently used. The set of hosts concretely hosting replicas of the item are called the item's *home set*, after Jalote [1989]. Roughly speaking, the replication scenario of a DSO is described by the set of contact addresses stored in the Globe Location Service at a particular point in time.

3.4. REPLICATION INTERFACES

This section describes the standardized interfaces of the replication and control subobjects in more detail. The interfaces are explained using a simple example distributed shared object encapsulating a single integer. Its interface, called integer, is shown in Figure 3.4. Globe has its own Interface Definition Language, but for clarity OMG/CORBA IDL [Object Management Group, 2001] is used throughout this dissertation.

For this example it is assumed that the integer DSO uses a master/slave replication protocol and currently has two replica local representatives: one master and one slave. Furthermore, it is assumed that a client process has bound to the distributed shared object and thus already has a proxy local representative in its address space. This situation is illustrated in Figure 3.5.

To explain the replication interfaces we will go over the steps which are taken by the various subobjects during an invocation of the object's set method by the client process, in the following manner. The series of steps is split into four stages. For each stage, we first describe one or more of the interfaces used and then show how it is used in this stage of the method invocation. For a more detailed explanation of the interfaces and the rationale behind them, see [Homburg, 2001].

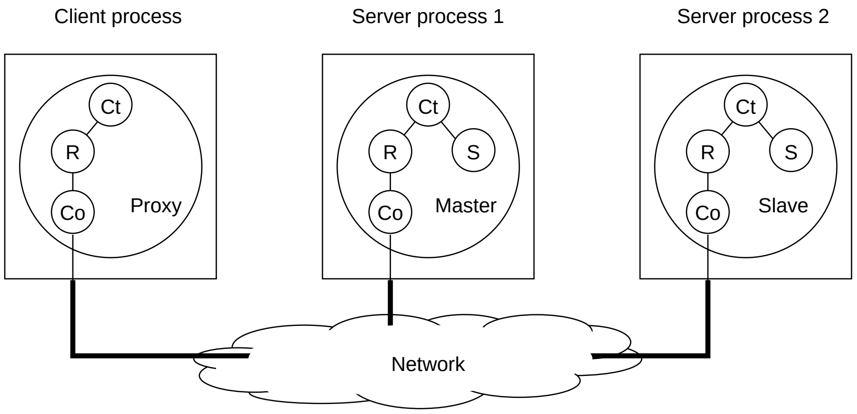


Figure 3.5: Our integer object has three local representatives, a proxy running in a client process and two replicas. The object uses master/slave replication, with the replica on server process 1 acting as master. Ct, R, S and Co, are abbreviations for control subobject, replication subobject, semantics subobject and communication subobject, respectively.

3.4.1. Stage 1: Shipping the Invocation to the Master Replica

The replication subobject's repl interface is shown in Figure 3.6. As can be seen from the figure, the code of a replication subobject is application independent and operates on opaque invocation messages.

The first series of steps involved in the invocation of the integer object's set method is depicted in Figure 3.7. The CORBA-IDL syntax

<interface>::*<method>*

is used to denote a method in a specific interface. The invocation will set the integer in the integer object to the value 481.

1. The invocation on the integer DSO starts by the client process invoking integer::set(481) on the proxy local representative. The local representative delegates this call to its control subobject by making the same invocation on it.

2. To signal the start of the method invocation to the replication subobject, the control subobject invokes the start method in the replication subobject's repl interface. An argument to this call is a flag indicating that the method to be invoked on the integer object will change its state. Knowledge about which methods modify the state and which do not is encoded in the control subobject and would, for example, be derived from programmer annotations

```
enum   action_t { SEND, INVOKE, RETURN };


interface repl
{
    action_t start( in boolean ModifiesState );
    action_t send( in boolean ModifiesState,
                   in sequence<octet> MarshalledRequest,
                   out sequence<octet> MarshalledReply );
    action_t invoked( in boolean ModifiesState );
};
```

Figure 3.6: A replication object's repl interface. Error parameters have been left out for simplicity.

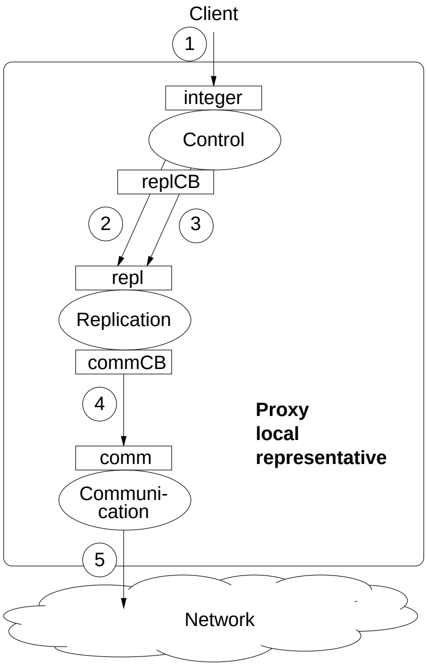


Figure 3.7: First stage of the invocation of the integer object's integer::set method. The client process invokes the set method on the proxy local representative in its address space. Rectangles represent interfaces; arrows indicate only an invocation, not the return of the invocation.

to the integer interface by the stub generator creating the control subobject. These annotations are not shown in Figure 3.4.

3. The invocation of repl::start returns the value SEND, by which the replication subobject instructs the control subobject to marshall the integer::set(481) invocation and pass the marshalled invocation to the replication subobject using the repl::send method.

4. The replication subobject creates a replication-protocol message containing the marshalled method invocation and its own additional headers. The replication subobject implements the client part of a master/slave protocol and, following this protocol, ships the invocation message to its peer in the master local representative using the communication subobject. This subobject's comm interface contains standard communication primitives and will not be discussed further. The location of the master local representative was contained in the contact address used to construct this proxy representative, obtained from the Globe Location Service during binding.

5. The communication subobject interfaces with the operating system's network layer to send the message to the master replica.

3.4.2. Stage 2: Performing the Invocation on the Master Replica

To understand the sequence of steps that occur when the replication-protocol message carrying the method invocation is received at the master replica, the concept of a *callback interface* has to be introduced first. A callback interface is an interface of a subobject which is used only by threads created as part of an upcall from the network layer. For the purpose of this dissertation a callback interface can be considered a regular interface.

There are two callback interfaces in a local representative: the commCB interface used by a communication object to pass an incoming replication-protocol message to the replication subobject, and the replCB interface of the control subobject which is used by the replication subobject. These interfaces are shown in Figure 3.8 and Figure 3.9, respectively.

The next stage in the method invocation is depicted in Figure 3.10.

6. The invocation message is delivered to the communication subobject by the operating system.

7. The communication subobject removes its own headers and forwards the resulting replication-protocol message to the replication subobject by calling commCB::msgArrived.

```
interface commCB
{
    void msgArrived( in NetworkAddress SourceAddress,
                     in sequence<octet> ReplProtoMessage,
                     out sequence<octet> ReplProtoReplyMessage );
};
```

Figure 3.8: A replication object's commCB interface. "NetworkAddress" is an opaque type for, for example, IP addresses (cf. struct sockaddr in UNIX networking [Stevens, 1998].) Error parameters have been left out for simplicity.

```
interface replCB
{
    void handleRequest( in sequence<octet> MarshalledRequest,
                        out sequence<octet> MarshalledReply );
    void getState( out sequence<octet> MarshalledState );
    void setState( in sequence<octet> MarshalledState );
};
```

Figure 3.9: A control object's replCB interface. Error parameters have been left out for simplicity.

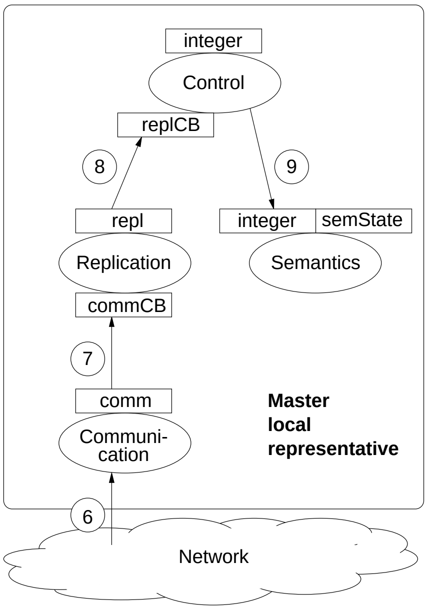


Figure 3.10: Second stage of the invocation of the integer DSO's set method. The replication-protocol message originating in the client process arrives at the master replica and is processed, resulting in the actual execution of the integer::set method.

8. The replication subobject decodes the message's headers and finds the incoming message is a state-modifying method invocation. If there are no other invocations currently underway, the replication subobject asks the control subobject to unmarshall the method invocation and perform the actual invocation on the semantics subobject by calling the handleRequest method in the control subobject's replCB callback interface. An argument to this call is the remainder of the replication-protocol message, that is, the marshalled method invocation.

9. The control subobject invokes integer::set(481) on the semantics subobject. The semantics subobject executes the method, setting the encapsulated integer (its state) to its new value. Next, the control subobject marshalls the result of the invocation, in this case just the return value of the set method (the integer 481). Finally, it leaves the replCB::handleRequest method, returning the marshalled reply as an out parameter to the replication subobject.

```
interface semState
{
    void getState( out sequence<octet> MarshalledState );
    void setState( in sequence<octet> MarshalledState );
};
```

Figure 3.11: A semantics object's semState interface. Error parameters have been left out for simplicity.

3.4.3. Stage 3: Updating the Slave Replica

Since the integer::set method modifies the state of the integer DSO, the replication subobject should now update or invalidate the copy of the state stored in the slave local representative (see Figure 3.5). The update variation is described here.

In Globe, a semantics subobject is currently responsible for creating marshalled versions of its state (and thus that of the distributed shared object) and update its state from a marshalled version. To this extent, each semantics object has a semState interface that the application programmer must implement. This interface is depicted in Figure 3.11. Observe the overlap with the replCB interface in Figure 3.9. The reason that a control object implements the replCB interface, and not the semState interface and an additional interface containing just the handleRequest method, is due to historical reasons.

The description of this stage of the method invocation is split into two parts. The first part is illustrated in Figure 3.12.

10. The replication subobject, knowing that the method it asked the control subobject to perform changed the state, calls the control subobject's replCB::getState method.

11. This invocation is directly transformed into a semState::getState call to the semantics subobject. The semantics subobject creates a marshalled version of its state and returns this to the control subobject which, in turn, returns it to the replication subobject as a result of the invocation of replCB::getState.

12. The replication subobject creates a "state-update" message from the marshalled state by adding its own headers and passes this message to the communication subobject for transportation to the replication subobject's peers in the slave local representatives.

13. The communication subobject forwards the message to the network layer.

The second part of stage 3 of the invocation is illustrated in Figure 3.13.

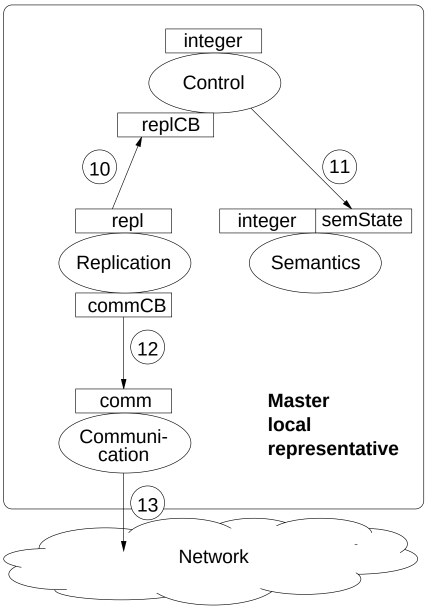


Figure 3.12: The first part of stage 3 of the invocation of the DSO's set method. The master local representative creates a marshalled version of the new state and forwards it to the slave representative such that it can update its semantics subobject and make the two replicas consistent again.

14. The replication-protocol message containing the updated state is delivered to the communication subobject of the slave by the operating system.

15. The communication subobject, in turn, delivers the message to the replication subobject using that object's commCB::msgArrived method.

16. The replication subobject determines that the message contains a state update and calls the control subobject's replCB::setState method to instruct it to install this new state in the semantics subobject.

17. The control subobject does so by invoking the semState::setState method on the semantics subobject. Note that the semantics subobject unmarshalls a state that was marshalled by its peer semantics subobject in the master in step 11.

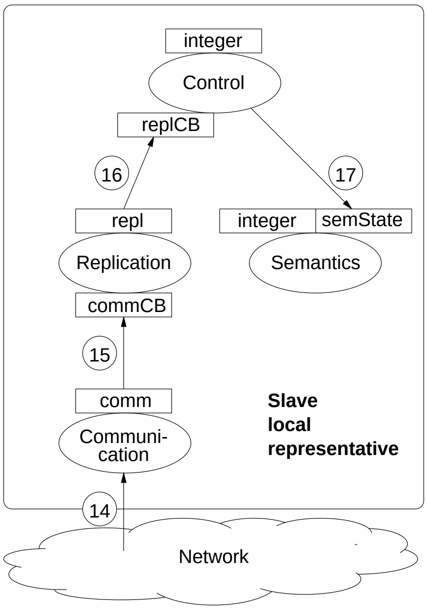


Figure 3.13: The second part of stage 3 of the invocation of the DSO's set method. The marshalled version of the updated state is received at the slave local representative and installed in its semantics subobject, making the master and slave replica consistent again.

3.4.4. Stage 4: Returning the Invocation's Result to the Client

18. After the successful installation of the new state in the semantics subobject, the slave's replication subobject returns an "OK" message to its peer in the master.

19. When the master replica's replication subobject receives the "OK" message it converts the marshalled reply of the invocation it received from the control subobject in step 8 (page 37) into a invocation-reply message. It passes this reply message to the communication subobject by returning from the commCB::msgArrived invocation. The communication subobject returns the reply message to the proxy local representative in the client process (see Figure 3.5).

20. The proxy's replication subobject was blocked waiting for a reply to its invocation message from the master replica. The receipt of the invocation-reply message unblocks it. The replication subobject was blocked in the

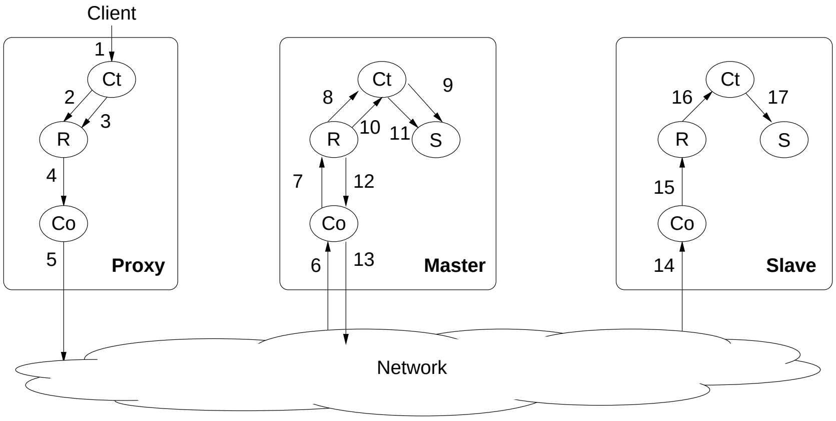


Figure 3.14: The steps involved in an invocation of a state-modifying method on a master/slave replicated distributed shared object. Circles represent subobjects, rounded boxes represent local representatives. Ct, R, S and Co, are abbreviations for control subobject, replication subobject, semantics subobject and communication subobject, respectively.

control subobject's call of its repl::send method. The replication subobject extracts the marshalled reply from the protocol message and returns it to the control subobject via the MarshalledReply parameter of the repl::send method (see Figure 3.6), and returns RETURN as action value.

21. As before, this value signals to the control subobject that it should return the result of the invocation to its caller. The control subobject unmarshalls the reply and returns the value to the client process by returning from the integer::set method, thus completing the method invocation on the integer DSO.

What is not discussed in this example is the use of the INVOKE replication action shown in the definition of the repl interface (see Figure 3.6). This value is returned by a replication subobject of a local representative in a client's address space when that local representative is stateful. In other words, when a client has a replica local representative in its address space and the replication protocol allows the method invocation to be carried out in the local address space (e.g. because it is a read operation) the return value of the repl::start method is INVOKE instead of SEND, signaling that the control subobject should perform the method invocation on the local semantics subobject.

The whole sequence of steps is summarized in Figure 3.14.

3.5. CREATING DSOS

We have not yet discussed how DSOs are created, and, in general, this is still a topic of research. We therefore briefly look at how DSOs are created in our current implementation of the Globe middleware. To create a distributed shared object at least one local representative for the object should be created. In the current Globe implementation, this initial LR is constructed in two steps.

1. A client wanting to create a distributed shared object first creates a description of the implementation of a local representative. This description specifies which subobjects the LR be composed of, and some initialization data. This description is similar to those found in a contact address of a DSO stored in the Globe Location Service.

2. Next, the client sends a "create first replica" request to a Globe object server (see Sec. 3.7) that contains the description. Based on this description the object server creates the initial local representative for the DSO. Generally, this local representative will register itself in the Globe Location Service.

3.6. THE GLOBE LOCATION SERVICE

To efficiently map object handles to contact addresses on a worldwide scale, the Internet is organized into a hierarchy of *domains*. The domains at the bottom of the hierarchy represent moderate-sized networks, such as a university's campus network or the office network of a corporation's branch in a certain city. The next level in the hierarchy is formed by combining these leaf domains into larger domains (e.g. representing the city's metropolitan-area network). This procedure is applied recursively up to the root domain, which encompasses the whole Internet. Note that domains in this hierarchy do not correspond to domains in the Internet's Domain Name System (DNS) [Mockapetris, 1987] and the subdivision can be based on metrics other than geographical distance (e.g. routing domains).

With each domain in the hierarchy we associate a *directory node*, as shown in Figure 3.15. Each directory node keeps track of the locations of the distributed shared objects in its associated domain, as follows. For each DSO that has local representatives in the node's domain, a directory node stores either the actual contact address (network address and protocol information for contacting the representative) or a set of *forwarding pointers*. A forwarding pointer points to a child directory node and indicates that a contact address can be found somewhere in the subtree rooted at that child node. Because a DSO may consist of multiple replicas located in different child domains, a directory node may store more than one forwarding pointer per DSO. Normally, the contact addresses are stored in the leaf

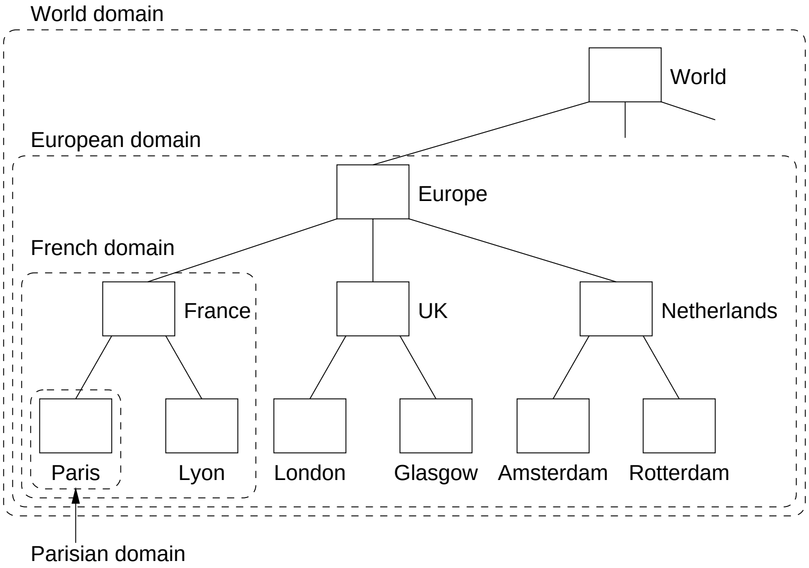


Figure 3.15: The Globe Location Service divides the Internet into a hierarchy of domains, represented by dashed, rounded rectangles in the figure. Associated with each domain is a directory node, represented by a small rectangle. For simplicity the domains for cities and countries other than Paris and France have been left out.

directory nodes. However, storing the addresses at intermediate nodes may, in the case of highly mobile objects, lead to considerably more efficient lookup operations, as explained in [Van Steen et al., 1998a]. This design has some (seemingly) radical consequences. For each distributed shared object on the Internet, there is a tree of forwarding pointers from the root node to the directory nodes that contain the actual contact addresses. Before explaining that this, in fact, does not create a single point of failure or bottleneck, we first look at how object handles are resolved.

During binding (see Sec. 3.3), the client sends a look-up request to the directory node of the leaf domain the client is located in. The leaf node checks to see if it has a contact address for that DSO in its tables (i.e., it checks if the DSO has a representative in this (leaf) domain). If not, it forwards the request to its parent node, which, in turn, checks its tables. This process is repeated until either (1) a contact address for the object is found or (2) a forwarding pointer is discovered or (3) the root of the tree is reached, in which case the object has no registrations in the GLS. In case a forwarding pointer is found, the lookup operation continues down into the subtree pointed to by the forwarding pointer and follows the tree of forwarding pointers to a node in that subtree that stores an actual contact address. If multiple forwarding pointers are found, one is selected at random.

The advantage of this design is, that if a distributed shared object has a representative near to the client, the Globe Location Service will find that representative using only "local" communication. In other words, the cost of a lookup is proportional to the distance between the client and the nearest local representative of the distributed shared object.

An apparent problem with this design is that the root node, or in general, the higher-level nodes in the hierarchy have to store a lot of forwarding pointers and handle a lot of requests (if representatives of the DSO are not located near their prospective clients). Our solution to this problem is to partition a directory node into one or more *directory subnodes*. Each subnode is made responsible for a specific part of the object-identifier space via a special hashing technique and can run on a separate machine. For further details, see [Van Steen and Ballintijn, 2002].

3.7. THE GLOBE OBJECT SERVER

We have developed special server software for hosting local representatives of (long-running) distributed shared objects: the *Globe Object Server (GOS)*. It provides facilities for local representatives running in the GOS to handle reboots and crash failures of the host computer and support their passivation and activation for server resource management. A GOS is application independent; it, and the local representatives running in it can be managed remotely.

This section describes the initial implementation of the Globe Object Server, which allows local representatives to survive just a reboot, not failures. Chap. 6 will describe extensions that allows an object server and its contents to also survive ungraceful shutdowns. The Globe object server is currently implemented as single user-level process and consists of three components: the *server manager*, the *persistence manager* and the *communication-object manager*.

The server manager controls the operation of the object server. It processes the object-management commands the server receives over the network, such as "create replica" and "destroy replica." At present, a simple RPC protocol is used. The communication-object manager and the persistence manager are resource managers. The persistence manager provides an operating-system independent interface to the persistent storage of the host machine. Replicas that need large amounts of storage use this interface to store and retrieve these parts of their state. The persistence manager keeps track of the persistent resources a replica uses such that when a replica crashes it can free those resources. The communication-object manager manages the communication resources of an object server. In particular, the communication-object manager provides transparent multiplexing of communication streams to the same hosts, reducing the number of TCP connections required.

As already mentioned, the current Globe object server provides facilities for local representatives (generally replicas) to survive the graceful shutdown and restart of a server. When an object server is shut down, it signals this fact to the running replicas. Based on their own (object specific) policy, the replicas then decide to stay or remove themselves from the object server. When a replica decides to stay it marshalls its internal state (which includes the state of the object) and writes it to persistent storage using the persistence manager. The server manager records which replicas are staying and need to be restored after rebooting and stores this information in a persistent log. When the object server restarts it reads this log and recreates the replicas which then, in turn, recreate their internal state from disk and contact their peers to check for missed updates. Special measures are in place to make sure the network contact points (i.e., TCP port numbers) of the replicas remain the same. If these contact points changed after a reboot, the contact addresses for these replicas in the Globe Location Service would have to be updated. By making sure the same contact points are used again the impact of a reboot is minimized.

3.8. THE GLOBE INFRASTRUCTURE DIRECTORY SERVICE

A large-scale distributed application makes use of multiple object servers. The set of object servers are used varies with time under the influence of failures, deliberate migrations and changes in clients' usage patterns. The set of object servers potentially available to a particular application also varies over time, as servers are decommissioned or new servers are introduced. To allow applications to easily discover which object servers are available to them we introduced a new middleware service.

The *Globe Infrastructure Directory Service (GIDS)* [Kuz et al., 2001] keeps track of all object servers that are currently available worldwide. The GIDS allows applications to discover suitable object servers based on a specification of desired properties, related to technical capabilities (amount of memory, available bandwidth, operating system and hardware platform), security attributes (which person or organization operates this object server), and the location of the object server. Once a matching server has been found the application and server enter a negotiation phase to determine whether or not they want and can cooperate, and negotiate the exact details of that cooperation.

Typically, an application developer would specify the required properties of the object servers when the application is started. When an application, or rather the distributed shared objects making up the application, discover they need a new object server (to maintain their required fault-tolerance degree or to optimize server load or network usage by creating a new replica somewhere), they contact the GIDS to supply them with the contact information for a new object server that matches the specified properties. As shown in the next chapter, this service is in particular useful for applications with varying usage patterns which also exhibit peaks and dynamically changing server pools such as the Globe Distribution Network.

The current implementation of the GIDS uses the Light-weight Directory Access Protocol (LDAP) and standard LDAP servers [Loshin, 2000]. The GIDS divides the world into a set of base regions (generally subdivisions of the leaf domains identified for the Globe Location Service.) Per base region there is an LDAP server, called the *Regional Service Directory* that keeps track of the available object servers and their properties. The base regions are organized into a hierarchy, currently based on their geographical location, which allows clients (i.e., objects looking to create a new replica somewhere) in other base regions to locate the appropriate Regional Service Directories.

CHAPTER 4

Distributed Revision Objects

As explained in the previous chapter, Globe uses a uniform model to represent information and services: distributed shared objects. Any Globe application, including the Globe Distribution Network (GDN), can therefore be thought of as a group of processes communicating through a collection of distributed shared objects. Concretely, the GDN is modeled as a group processes inserting software into distributed shared objects (programmers producing software) and a (large) number of processes retrieving software from those distributed shared objects (people downloading the software).

This chapter explains how software is encapsulated in distributed shared objects, that is, the mapping from software packages, revisions and variants to distributed shared objects, and how these "software DSOs" are implemented such that the distribution of software from producer to user is done efficiently. Sec. 4.1 explains what is meant by efficient distribution and how it can be achieved. Next, the mapping from software packages, revisions and variants to distributed shared objects is discussed in Sec. 4.2. The interface and semantics of the "software DSOs" are discussed in Sec. 4.3. Sec. 4.4 describes how packages, revisions and variants can be referred to now that they are encapsulated in DSOs. Sec. 4.5 describes the basic implementation of the DSOs. Their replication protocol is discussed in detail in Sec. 4.6, in particular, how it allows the Globe Distribution Network to handle flash crowds.

Designing mappings from the application domain to Globe DSOs is an aspect of application design that has not been studied until now, and therefore represents a valuable contribution of this dissertation. The practical problems of interface design, and how the Globe middleware handles objects with large state are also aspects that are described for the first time in this dissertation. In the areas of persistence and replication protocols I borrow from existing work by others, which are not my contributions, although I was involved in the design of the persistence facilities. The key ideas of the replication protocol are borrowed from Pierre et al. [2000, 2001]. The details of the scenario reevaluation and load balancing aspects are my extensions. I have provided a detailed description of the replication protocol used to give a complete and concrete picture of how the GDN makes distribution efficient, and the roles of the Globe middleware services in this process.

4.1. EFFICIENT DISTRIBUTION OF SOFTWARE

A software distribution network *efficiently* distributes its content when it places the content such that optimal use is made of the underlying data network. What is meant by optimal use is explained in the following paragraphs.

The Internet currently suffers from temporarily or permanently overloaded network links that act as bottlenecks. To remove these bottlenecks there are two solutions. The hardware solution is to increase the available bandwidth on the overloaded links. However, instead of increasing bandwidth the problem can also be solved by reshaping communication in such a way that the overloaded links are used less. This solution can be applied to situations where the network is used by many people to retrieve the same data or use the same services from the same parties.

The tools for reshaping communication are distribution, replication and caching (DRC). By *caching* I mean temporarily storing results of service requests. The use of these three tools is illustrated in Figure 4.1. Figure 4.1(a) shows a group of users in Network 1 communicating with a shared object in Network 2. The size of the communication has caused the link between Network 1 and 2 to become overloaded. Replication can be used to create another instance of the shared object in Network 1, reducing the load on the link, as illustrated in Figure 4.1(b). Distribution, or more precisely, migration can be used instead of replication to relocate the data or service in the overloading part, when this move does not simply shift the direction of the overload on the link (see Figure 4.1(c)). Caching, like replication, can prevent having to go over the network link altogether.

A prerequisite for being able to use DRC is that computing resources are available to migrate to or install a replica on. To a large extent the choice between using DRC or upgrading a network link is determined by the price of server capacity compared to network bandwidth. Historically, the price of (wide-area) bandwidth has been considerable, making it more cost effective to setup caching proxy servers or mirror sites. In many cases setting up a caching server or replica can be done autonomously, enabling organizations to work around network problems that exist outside of their realm of influence.

Another prerequisite is the ability to identify *hot spots* in the network. Hot

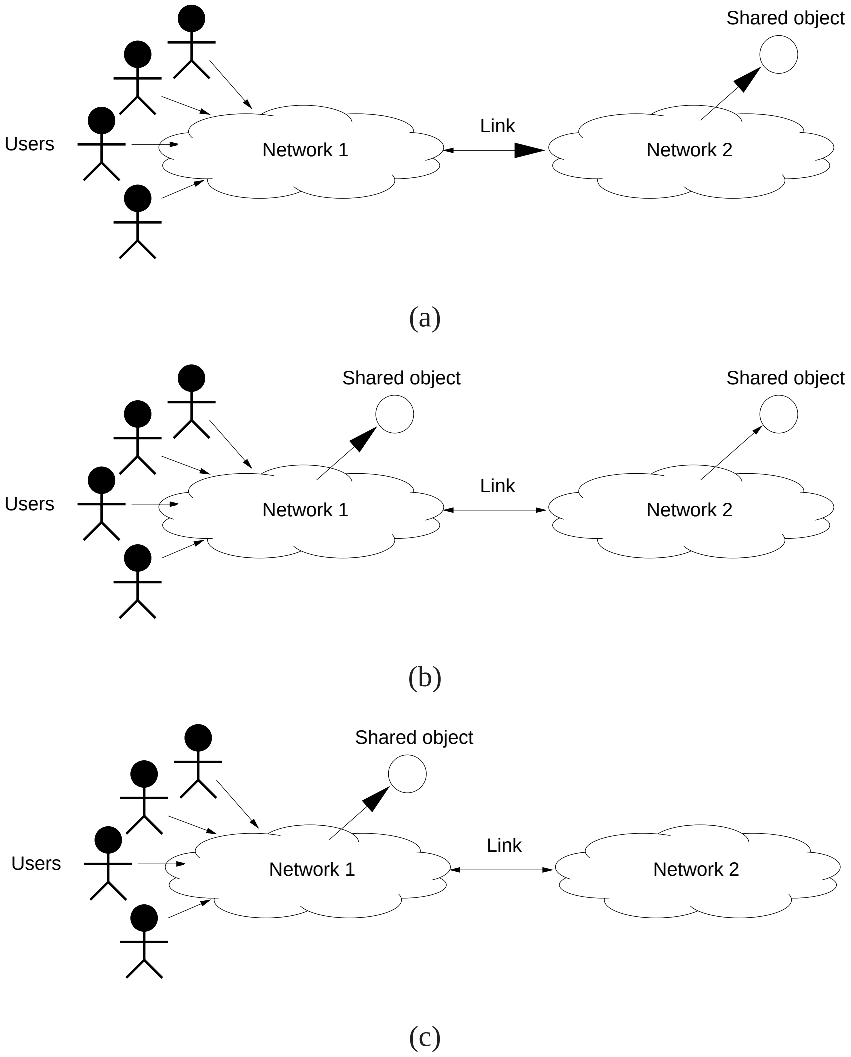


Figure 4.1: (a) A shared link connecting Network 1 and Network 2 is overloaded (illustrated by the large arrow head going into Network 2) because many users in Network 1 are accessing the shared data/service located in Network 2 (the shared object is assumed to be able to handle to the load). (b) Overload of the link is avoided by replicating the shared data/service in Network 1. (c) Overload of the link is avoided by migrating the shared data/service to Network 1.

spots are congested links or overloaded routers. Some hot spots are permanent, such as, at present, the transcontinental network links, whereas others are just temporary. Effective use of DRC by an application requires short-term (by the application itself) and long-term (e.g. by its administrators) analysis of usage patterns, of both the application itself and the data network it is running on. From these analyses the application or its administrators should infer where to best place replicas, and whether it is useful to do migration or caching.

Besides the availability of servers and knowledge of usage patterns, there are other factors that may restrict the applicability of DRC. In general, security constraints may prohibit data or services from being replicated or migrated. Furthermore, when data or services are more frequently updated than read, replication may not reduce traffic over the shared link. For example, when the read/write ratio is low, more network traffic may be generated by the replication protocol keeping the replicas consistent than is saved by creating a replica near to the users. In a software distribution network content is more read than it is written and there are no strong security requirements prohibiting replication.

Distribution, replication and caching, as tools for optimizing network usage, will continue to play a role in the Internet in the near and mid-term future. Networking technology is progressing rapidly both in terms of link and switching capacity [Stix, 2001; Edwards, 2000]. However, it may be some time until these advanced technologies have been deployed and bandwidth has become cheaper than server capacity. Furthermore, the required bandwidth in the backbone of the network is proportional to the number of people simultaneously accessing the network times the (average) amount of bandwidth they use. As more and more people get access to more and more bandwidth, this required capacity will increase many times. It is therefore likely that links in the backbone of the network become overloaded, either temporarily or for longer periods of time.

Distribution, replication and caching are not used only for optimizing network usage. They are also used for server load balancing (e.g. handling flash crowds) and achieving fault tolerance. Hence, whatever role DRC may play in the future for optimizing network usage, they will always be an important part of a software distribution network's implementation.

A software distribution network, such as the Globe Distribution Network, should support distribution, replication and caching of its contents. Given that in the GDN content is encapsulated in Globe distributed shared objects, these objects will have to be implemented such that their state is distributed and replicated in a way which optimizes the use of the Internet. The following sections explain the interface and implementation of these distributed shared "software" objects. Note that in this chapter we look only at using replication for network and server load balancing; replication for fault tolerance is discussed in Chap. 6.

4.2. MAPPING SOFTWARE PACKAGES TO DSOS

This section describes how software is encapsulated in distributed shared objects, that is, the mapping from entities and relationships in the free-software domain to distributed shared objects. Recall that in this dissertation a *software package* is an application, a library, or any piece of software that is published as a separately named entity. A software package evolves and as a result, multiple *revisions* are created, that is, versions that are meant to replace other, earlier versions. Each revision can have a number of *variants*, such as source code and binaries for different hardware platforms. Variants, in turn, can be stored in archive files with different file formats (e.g. .gz, .rpm). A *distribution* is a named collection of software packages A distribution has its own revision history, and can, like a software package, be published in multiple variants (i386, Alpha, SPARC). Examples of distributions are RedHat Linux and FreeBSD, but also more general collections of packages, such as the GNOME desktop environment.

4.2.1. Revision and DistributionArchive DSOs

In the Globe Distribution Network, a software package is published as a set of *revision DSOs* (also called *revision objects*). Each revision DSO contains archive files that contain all available variants of that revision. The same variant may be offered in multiple archive files with different file formats. So a revision DSO is basically a collection of archive files, containing the different variants of a particular revision of a software package in various file formats.

The mapping scheme for software packages is illustrated in Figure 4.2 using the GIMP application as an example (GIMP manipulates images in various image formats). Each revision of the GIMP package is turned into a separate distributed shared object. The revision DSO encapsulating revision 1.1.29 would, for example, consist of the source code in tar.gz format and binaries for Linux on i386 and Alpha processors in .deb and .rpm package formats.

A distribution is published in the GDN as a set of *DistributionArchive DSOs*. Each DistributionArchive DSO contains a variant of a revision of the distribution in a specific file format, for example, the ISO9660 CD-image of the i386 variant of revision 7.1 of the distribution, or the collection of archive files in .rpm format, containing the i386 binaries of the packages that make up revision 7.1 of the distribution. This mapping scheme is illustrated in Figure 4.3.

For distributions that contain packages developed by others than the publisher of the distribution, we assume that that a distribution is published separately from its individual packages; that is, even though a package may already be published via the GDN by its original producer, the publisher of a distribution will publish his own copy, possibly with his own modifications. For example, if revision

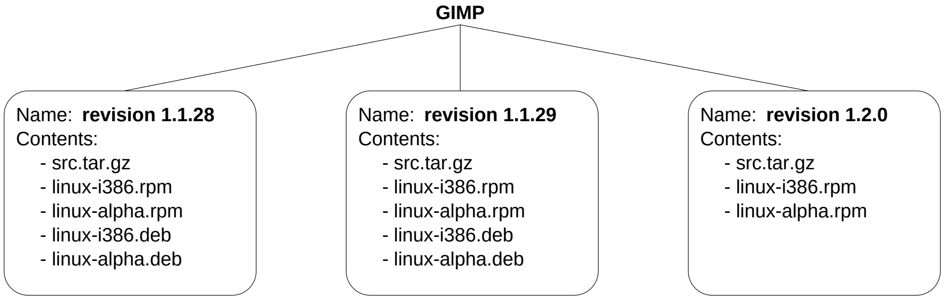


Figure 4.2: (Rounded boxes represent distributed shared objects) The GIMP package is published as a set of revision DSOs in the Globe Distribution Network. For brevity, only three revision objects are shown in the figure. Each revision DSO contains the different variants in which the revision is published, in multiple file formats.

7.1 of RedHat for i386 hardware includes revision 1.2.1 of the GIMP package (compiled for the i386 platform), the publisher of RedHat 7.1 will not link to the revision DSO containing v1.2.1 published by the maintainer of GIMP, but instead publish its own copy of the 1.2.1 binaries for Intel. This separate publishing is standard practice because distribution publishers want to keep control over their distribution.

4.2.2. Discussion

Before discussing the choice for revision DSOs and DistributionArchive DSOs, we first look at the general considerations to make when designing a mapping from application domain to DSOs.

From a technical viewpoint, two data items *A* and *B* should be turned into two separate distributed shared objects if they have differing nonfunctional requirements. For example, item *A* may be much more popular than *B*; that is, *A* may be accessed many times more. To provide fast and network-efficient access to item *A* it has to be replicated many times, whereas it would be wasteful from a resource and consistency-management perspective to also replicate item *B* to such a large degree. In this case it would therefore be more efficient to encapsulate *A* and *B* in different distributed shared objects, following perhaps the same replication policy but with different replication scenarios (item *A* being replicated on more hosts than *B*). The terms replication policy and scenario were defined in Sec. 3.3.

There are, however, costs associated with turning something into a DSO. These costs preclude fine-grained mappings where large numbers of small data

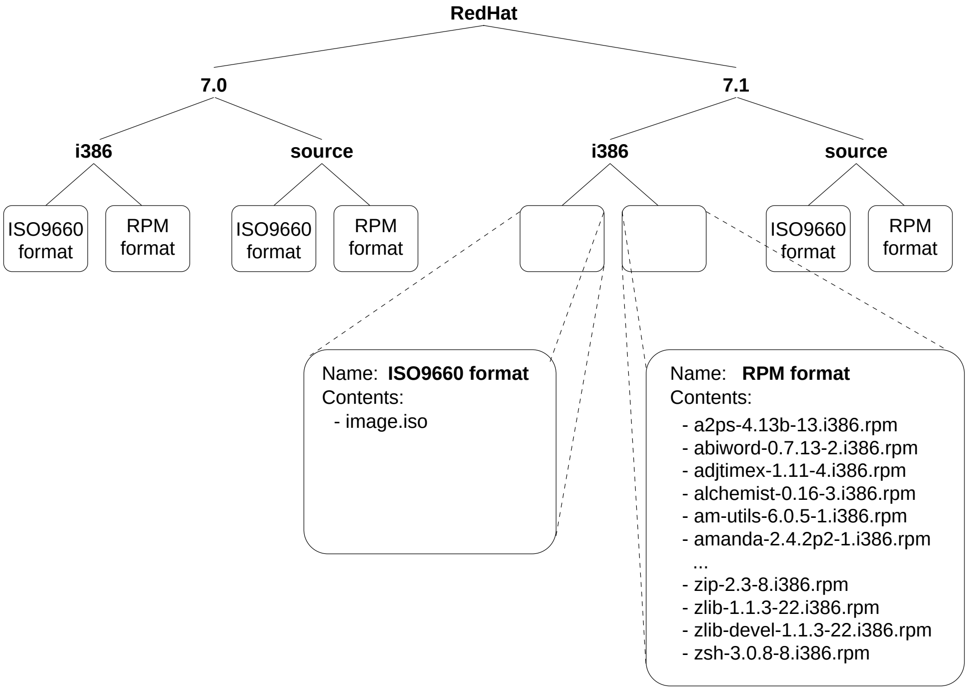


Figure 4.3: (Rounded boxes represent distributed shared objects) The mapping of the RedHat Linux distribution to DSOs. For brevity, only two revisions of the distribution (7.0 and 7.1) are shown. The objects are organized into a tree for presentation purposes. Following the tree from root to leaf together with the label of the object gives a description of the DSO's contents.

items are encapsulated in separate DSOs, as the combined costs become too high. We focus on four factors:

1. Memory footprint of local representatives
2. Use of scarce communication resources (i.e., multicast groups)
3. The costs of using the Globe Location Service to register and locate replicas of the DSO (when the object cannot efficiently keep track of its replicas itself).
4. Replication-protocol overhead (which can be high in intelligent/adaptive protocols)

We discuss each of these four factors in turn.

The *state of a replica local representative* is defined as the state of the object as contained in the semantics subobject and information used by the other subobjects, such as replication-protocol state, to function as a local representative of the DSO. The memory footprint of a replica local representative depends on how much of its state it keeps in virtual memory and how much it keeps on disk. Although local representatives with large state will store most of it on disk (see Sec. 4.5.1), a local representative will always consume a certain amount of virtual memory. This amount can be rather small, but if the state of the object (i.e., the data item it encapsulates) is rather small the overhead can be considerable, as is the case for the GDN (see below).

In general, the costs of using multicast are the costs of maintaining the multicast trees in the Internet's routers, and the relative scarceness of IP version 4 multicast addresses (250 million maximum) compared to the number of distributed shared objects the Globe middleware is being designed for (10^{12} [Van Steen et al., 1998a]). The latter implies that multicast can be used in DSOs only sparingly, or, in other words, it can be used only in those DSOs where applying multicast saves large amounts of bandwidth.

The costs of using the Globe Location Service are not yet known. A factor to consider is that the Globe middleware is to support 10^9 users[1] and 10^{12} objects, implying the GLS has a capacity of 1000 objects per user. This number is not extremely high. On the other hand, it is an average and objects are not required to use the GLS when they can handle replica location and registration efficiently themselves. The capacity of the GLS in terms of the number of objects it can register is, however, something that should be kept in mind.

[1] More precisely, the set of applications running on the Globe middleware can have 10^9 users in total.

The costs of using an adaptive replication protocol depends on how much data needs to be shared between replicas to make decisions about the most efficient locations for replicas and their number based on the clients' access patterns. This amount of data, in turn, depends on the number of replicas of the DSO.

Finally, a Globe application developer may also have to consider the time required to bind to a DSO when designing a mapping from application domain to DSOs [Bakker et al., 1999]. In particular, binding is expected to require a significant amount of time, as it involves contacting one or more middleware services over the network. These services, in particular the GLS, internally also may have to communicate over the network. If fast response time is necessary, an application developer should therefore take care that the mapping supports this requirement. In particular, the mapping must preclude the application from binding to (many) random DSOs; that is, preclude the application from making accesses to DSOs which cannot be predicted in advance. For, if the set of DSOs that will be accessed cannot be identified in advance, the application developer cannot make sure that binding to these objects is started in advance, and thus cannot ensure that these objects are directly accessible by the application client when they need to be.

Revision-sized DSOs

We now look at how these factors affected the choice for revision-sized DSOs for distributing software packages. For the GDN, there are expected different nonfunctional requirements and thus replication scenarios for packages, revisions and variants and archive files:

1. Some packages may be more popular than others
2. Some revisions of a package may be more popular than other revisions
3. Some variants of a revision of a package may be more popular than other variants
4. A specific file format may be more popular than others, meaning that a variant stored in a file in format X may be more more popular than the same variant stored in a file in format Y.

These properties suggest that for maximum efficiency each archive file should be turned into a distributed shared object. Such a fine-grained mapping is, however, considered too expensive. We discuss three of the four factors mentioned above to show where the costs are (the costs of using the Globe Location Service are not discussed as they are currently unknown.)

Memory footprint In the initial Java implementation of Globe an empty local representative of a master/slave replicated DSO with the initial security measures enabled (see Chap. 5) takes up approximately 30 Kilobytes of (virtual) memory. This value was measured by creating 1000 local representatives and measuring the increase in the amount of memory used as reported through the Java Virtual Machine's Runtime.freeMemory() and Runtime.totalMemory() methods.

I assume that for each Kilobyte of virtual memory the machine hosting the local representative needs more or less 1 Kilobyte of disk space. This number could possibly be reduced by using an implementation of the Globe run-time system that swaps local representatives in and out of virtual memory itself (i.e., at the application level rather than operating-system level), as explained in Sec. 4.5.2. Such an implementation might namely reduce the disk-space requirements by compressing the swapped-out local representatives. The gain depends on the achievable compression factor.

An overhead of 30 Kilobyte of disk space per local representative is considerable if we compare it to the average size of archive files. The average size of an archive file containing an i386 variant of a software package in the RedHat Linux distribution is 116 Kilobyte (measured using revisions 4.1 to 6.2β of the distribution). This average is the median of the file sizes rather than the mean (which is 622 KB), as the distribution is skewed, as shown in Figure 4.4. For all archive files (i.e., all .tar.gz, .tgz, .tar.bz2, .rpm, and .deb files) in the pub/Linux directory of ibiblio.org on 10 July 2002, the median is 144 KB and the mean 990 KB, as shown in Figure 4.5.

These median file sizes imply that the overhead of using a DSO per archive file in terms of disk space is, at present, 20–25%. This is viewed as considerable overhead and implies that placing each archive file in a separate DSO is too costly at the moment.

Multicast Whether or not DSOs used for free software distribution belong to the class of DSOs that needs multicast to be efficient is unclear. This depends on (1) the amount of data such an object sends to groups, and (2) the size of those groups. The first factor can be roughly estimated for the GDN. Group communication in a DSO consists of sending updates or invalidates to groups of replicas, both of which are the physical result of logical updates to the DSO. Although the popularity of software packages largely differs, the rate at which new revisions, variants and archive files in different file formats of a package are published is fairly stable. No more than a few new instances of the package are published per day, implying that the read/write ratio and thus the number of updates to a DSO containing software (of whichever granularity) is low. The second factor, the size of the groups being sent to, that is, the number of replicas of a free-software DSO

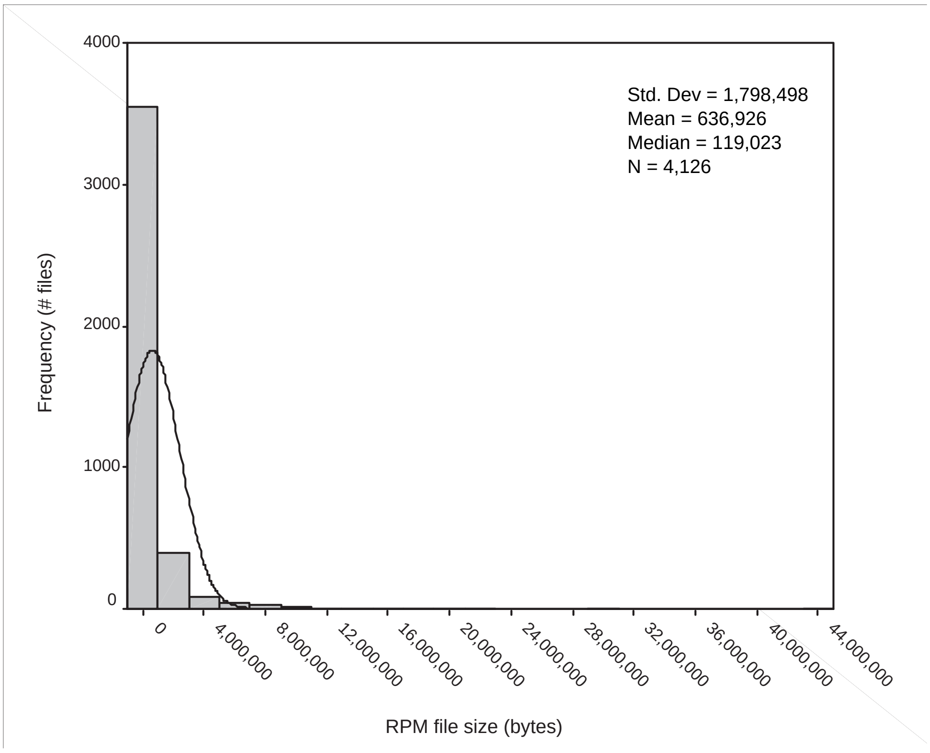


Figure 4.4: Histogram and normal curve of file sizes of i386-RPMs for RedHat 4.1–6.2β. The highest number on the X-axis indicates the size of the largest file in the set of files, rounded to the axis unit.

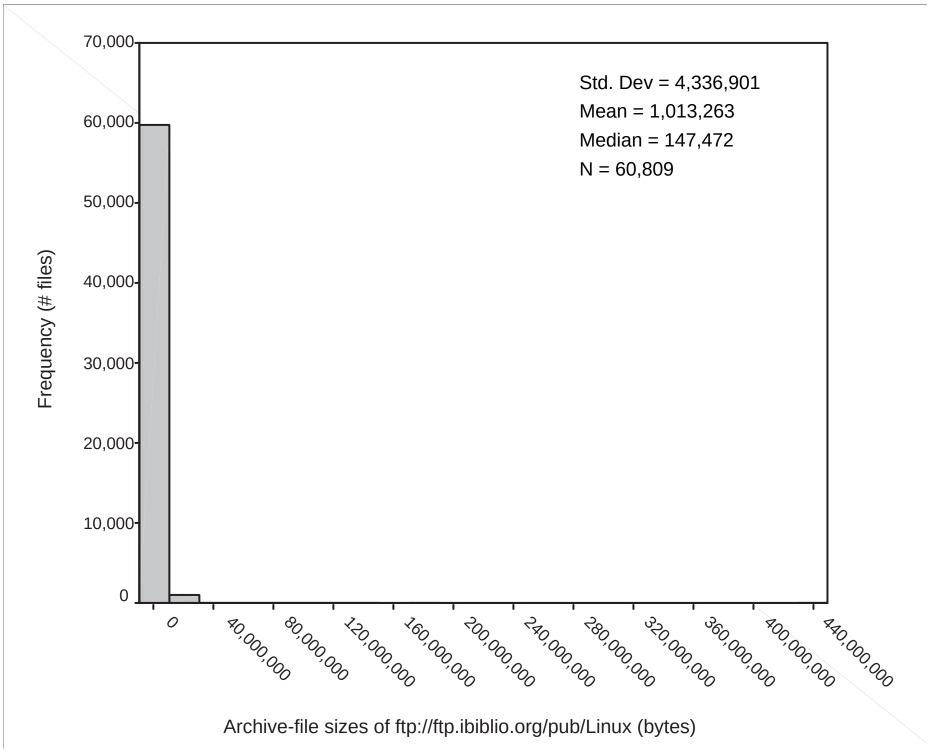


Figure 4.5: Histogram of archive-file sizes for ibiblio.org. The highest number on the X-axis indicates the size of the largest file in the set of files, rounded to the axis unit.

is hard to estimate. The number of replicas depends on the popularity of the software item, the total capacity of the system (i.e., if there are lots of resources, objects can be aggressively replicated), and the capacity of the average machine (i.e., if the average machine offers only limited capacity more replicas are needed for a certain capacity than if the average machine has high capacity). As a result, the average number of replicas is hard to estimate, making it, in turn, hard to assess whether free-software DSOs need multicast. If they need multicast, large number of objects and thus fine-grained mappings become infeasible.

Adaptive Replication The average number of replicas for a free-software DSO is hard to estimate, as noted above, implying that the cost of adaptive replication is not easily estimated. For small numbers of replicas the adaptive protocol used for free-software DSOs described in Sec. 4.6.2 is, however, relatively cheap to use.

Total number of objects While designing the GDN only a rough estimate of the number of objects required for each mapping was made, based on the size of the free software offering at that time. The total number of packages available was estimated at 50,000, a rough doubling of the number of projects registered at the SourceForge free-software site (see Sec. 2.2) at that time.

The average number of revisions per package was estimated around 5. An analysis of the FTP site of the Free Software Foundation which hosts many of the early free-software packages (i.e., the GNU software) shows that the number of revisions for a such package has median 5 , viewed over almost 11 years (see Figure 4.6). This number appears to contradict Raymond [2000] who argues that the development process of free software is characterized by the fact that developers "release early and release often". The actual average viewed over the complete set of free software may therefore be higher. For example, the Globe project has published 16 revisions to date, and the Linux kernel had 381 revisions in February 2000.

The average number of variants and number of file formats were estimated as follows. The volume of the free software in March 2001 was estimated to be several hundred Gigabytes in Sec. 2.2. The part of this volume taken up by individual packages is assumed to be around 200 Gigabyte for the calculation at hand (the various distributions make up the rest). With 50,000 projects and 5 revisions per project, an average revision is then 839 Kilobyte in size. If we assume the median file size of an archive file to be 130 Kilobyte (the mean of the median size of an i386 RPM in RedHat distributions and that of an archive file on ibiblio.org, see above) the number of archive files per revision is approximately 6. In other words, the average number of variants of a revision multiplied by the average number of file formats in which a variant is published is 6. If we assume

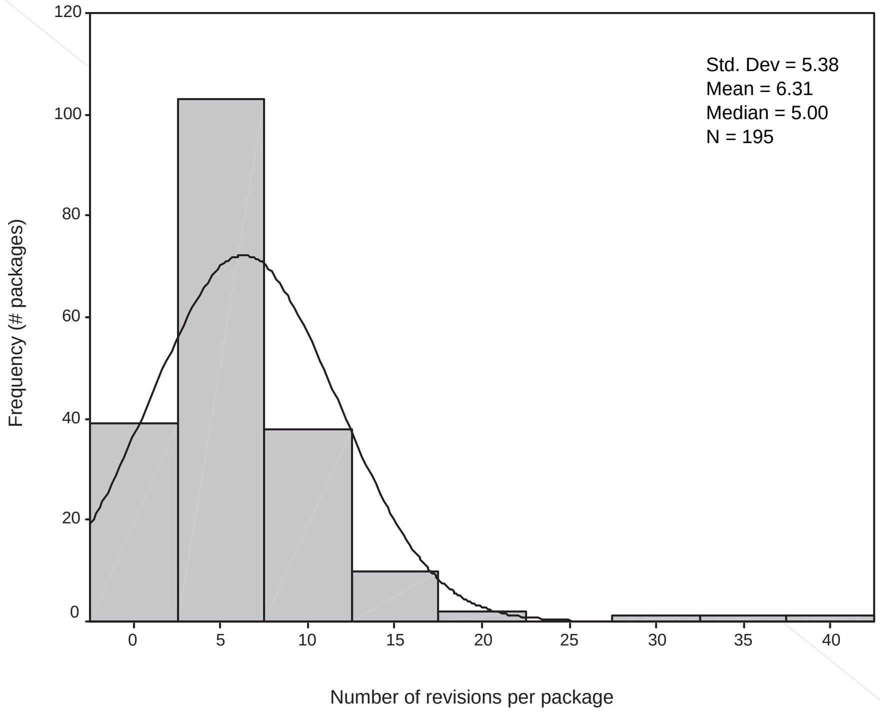


Figure 4.6: Histogram and normal curve of the number of revisions per package of ftp://ftp.gnu.org/gnu measured over the period April 1991–July 2002.

Table 4.1: Total number of objects in the GDN for different object granularities for the March 2001 volume of free software.

Mapping	Number of objects
Package-sized	50,000
Revision-sized	250,000
Variant-sized	750,000
Archive file-sized	1,500,000

two different file formats (e.g. .tar.gz and .rpm), the average number of variants is approximately 3.

The total number of objects for each mapping resulting from these estimates is shown in Table 4.1. The values in this table represent the minimum number of objects in the GDN, as they are based on the estimates of the current volume of free software. For a long-running distribution network that acts as a permanent store for the software (i.e., old packages and revisions are not removed), the number of packages will continuously increase, as new packages are created and old packages do not disappear. Hence, the number of DSOs in the GDN will also grow. It is not clear what happens to the average number of revisions per package, as this depends on the development activity and longevity of the package. The average number of variants and file formats depends on the number of popular hardware platforms and distribution formats, and is also hard to predict.

The coarsest-grained mapping is the one in which a complete package with all its revisions and variants is put into a single DSO. This mapping is also not a good idea. The reason is that it is the release of a new *revision* (or variant) of a popular package that generally attracts a flash crowd (see Chap. 2). As a result, replicating more than that revision (or variant) when handling a flash crowd is a waste of storage space and bandwidth (when transferring the copy to the new replica site). Some packages may have many revisions, leading to considerable overhead, in particular, when the package and thus its revisions are large.

One can argue that this granularity argument holds against revision-sized DSOs as well: if a particular variant of a revision is more popular than the other variants it is not useful to replicate those other variants as well. A mapping with a DSO per variant may be possible if the costs are acceptable. However, I currently do not have sufficient insight into the overhead of turning something into a DSO to warrant such an alternative.

In conclusion, the choice for revision-sized DSOs has been primarily determined by the overhead associated with more fine-grained mappings and the fact that replicating more than a single revision would be inefficient during flash crowds because of the resulting size of the DSO's state. The choice for using revision-sized distributed shared objects is therefore a trade-off between the cost of turning something into a distributed shared object and application requirements. We briefly return to the "DSO per archive-file" mapping in Sec. 4.2.3.

Mapping Distributions to DistributionArchive DSOs

The choice of representing a distribution as a set of DistributionArchive DSOs, each containing a variant of a revision in a particular file format, is simple. The size of a distribution like RedHat 7.1 for the i386 platform in ISO9660 format is so large that it should be placed in a distributed shared object by itself. Aggregating multiple file formats, let alone multiple variants or revisions, of a distribution into a distributed shared object leads to DSOs with such a large state that they are expensive to replicate. A replica of such a large DSO would require large amounts of disk space and it would take much time to transfer the state to the new replica location, making it difficult to quickly increase the number of replicas. Furthermore, it prevents us from exploiting differences in popularity of file formats by replicating only those formats widely, thus reducing overhead.

Consider the example of the current RedHat Linux distributions. Revision 7.3 of the RedHat Linux distribution is published in two language variants: English and Japanese. The English variant is, in turn, published as source code and as binaries for the i386 architecture. The source code variant is published in RPM file format and as ISO9660 CD images and is already 2.5 Gigabyte in size. The i386 variant in all file formats is 3.3 Gigabytes. This implies that publishing all or even a single RedHat revision in a single distributed shared object leads to objects that consume considerable amounts of disk space.

A DistributionArchive DSO has the same interface and implementation as a revision DSO. Therefore, in the following sections and chapters, revision DSOs should be read as "revisions DSOs and DistributionArchive DSOs". The former are used to distribute all variants of a specific revision of a package in one or more archive-file formats, the latter to distribute a single variant of a distribution in a single archive-file format.

4.2.3. Alternative Mappings

There are not many other ways in which software can be encapsulated in distributed shared objects. We briefly return to the "DSO per archive file" mapping in this section and discuss one other alternative mapping.

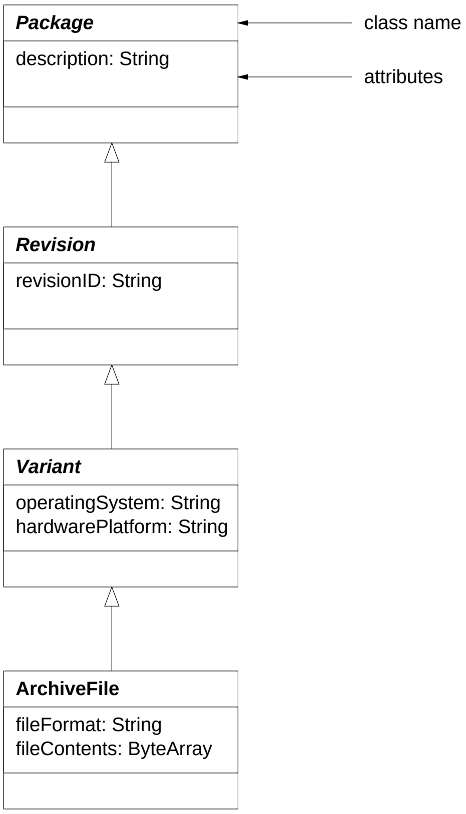


Figure 4.7: Static view of the Globe Distribution Network in UML notation [Rumbaugh et al., 1999]. The open arrows represent a generalization relationship. A class name in italics represents an abstract class.

A Direct Mapping

The previous section discussed a mapping from application domain to distributed shared objects based on technical arguments rather than using data-modeling principles. Data modeling principles prescribe that each entity and relationship in the application domain should be modeled as an object in the actual application. The application domain of published software modeled in UML [Rumbaugh et al., 1999] is illustrated in Figure 4.7. Taking this model, each archive file (containing a variant of a specific revision of a software package in some file format) should be encapsulated in an *ArchiveFile DSO*. *Packages*, *Revisions* and *Variants* are abstract classes and therefore have no concrete representation other than in the lowest level instances: the ArchiveFile DSOs.

The advantage of this mapping is that it is based purely on functional considerations. Archive files are separate distributed shared objects and not aggregated into a revision DSO because they are a separate entity in the model. This mapping also has advantages at the nonfunctional level. In particular, this mapping implies that ArchiveFile DSOs are read-only objects; that is, the software is put in the DSO at the time the DSO is created and is never changed afterwards. This property simplifies the implementation of replication protocols and access control mechanisms. The disadvantage of this mapping and the reason for not choosing it is the overhead it introduces, as already indicated.

Generic Objects

An alternative approach is to not define an explicit mapping between software packages, revisions, variants, archives and distributed shared objects at all. The idea would be to provide the producer of a software package with a generic DSO in which he could store archive files. The choice of what to store in these generic DSOs and therefore the granularity of replication is left to the producer of the software. For example, for a package whose variants (e.g. binaries) are small, he could create a single DSO in which to publish all revisions, variants and file formats.

The disadvantage of this approach is that the producer of the software package is made responsible for choosing the most efficient granularity. This choice should not be his to make, as it is the job of the Globe Distribution Network to make distribution of the software from producer to users efficient, not the producer of the software. Not all software publishers are experts in efficient distribution, and a bad choice by the producer can negatively impact the GDN. Consider, for example, the producer of a large package who puts all revisions, variants and archive files in a single DSO. If a new revision of the package becomes popular in a short period of time (i.e., attracts a flash crowd) the GDN would have to create many replicas of a large DSO of which only a small part (the new revision) would be of interest to the flash crowd, thus wasting resources.

4.3. INTERFACE AND SEMANTICS OF A REVISION OBJECT

This section explains the interface of a revision DSO, the semantics of the methods, and the rationale behind it. The implementation of revision DSOs is discussed below in Sec. 4.5. The interface is called the package interface. Two factors significantly influenced the design of this interface: (1) the need for sup-

```
interface package // upload methods only
{
    void startFileAddition( in string FileName,
                            in long long FileSize,
                            in traceCert traceCertificate );
    void putFileContent( in string FileName,
                         in long long Offset,
                         in sequence<octet> Block );
    void endFileAddition( in string FileName,
                          in traceSig traceSignature );
    ...
};
```

Figure 4.8: First part of the revision's DSO interface: the methods for uploading a file into the DSO. Error parameters have been left out for clarity.

porting uploads and downloads of large files (e.g. 650 Megabyte[2] CD-images in ISO9660 format) into a revision DSO, and (2) statelessness as a basic design principle for DSO interfaces. We discuss each of these factors and their influence on the interface in turn in the two following subsections. At the end of the section we discuss advantages and disadvantages of the chosen interface and alternative interfaces.

4.3.1. Handling Large Up– and Downloads

As explained in the previous section, a revision DSO is a collection of files. These files may range in size from 0 bytes to 650 MB or more. The interface and implementation of a revision DSO have to be such that files in the upper parts of the size range can be distributed efficiently. The issue of a producer loading too many large files into a single revision DSOis discussed in Sec. 5.4.3.

The time-tested solution for handling a large amount of data is to chop it up into blocks and process the blocks one at a time. Traditionally, to prevent a user from reading an only partially uploaded file there are methods such as open and close for signaling the start and end of an upload. For a revision DSO's interface, this solution results in the interface shown in Figure 4.8. Only the methods related to uploading a file are shown; download methods require more consideration and are discussed in the next section.

The usage of this part of the package interface is straightforward. To add a file, a client first calls startFileAddition passing the intended name for the file

[2] Quantifiers are capitalized to indicate a base of 1024, instead of 1000. E.g. Megabyte = 1024 times 1024 bytes.

and its size.[3] Next, the client makes repeated calls to putFileContent to place the contents of the file in the distributed shared object. Finally, it calls endFileAddition which makes the file accessible to other clients. During the upload the file is not accessible to other clients, although it is visible. In particular, a client trying to upload a file with the same name will receive a "name already in use" error.

Discussion

A block interface was chosen because it is the only way to pass or retrieve large amounts of data to or from a distributed shared object in the current Globe design. The current Globe design has no efficient support for methods that take large arguments. As described in Sec. 3.4.1, a method invocation must be marshalled by the control subobject, and must then be passed to the replication subobject for transmission as a single block of data. This choice of internal interfaces implies that if we pass a 650 Megabyte file as an argument to a file-upload method, the control subobject will have to marshall the 650 MB in a single block of virtual memory, which is very inefficient. The internal interfaces of a distributed shared object therefore require large amounts of data to be passed to the DSO by means of multiple method invocations. In other words, the DSO must have a block-based interface.

An important issue is the size of the blocks used to upload a file and who controls this size. The optimal block size depends on two factors, which we discuss in turn:

1. The available bandwidth of the network connections to the replica(s).

2. (to a lesser extent) The capacity of an object server.

Block size and bandwidth Optimal block size depends on the bandwidth and communication latency to the replicas as follows. An optimal block size allows local representatives to fully utilize the network connections between each other when the connections are available to them. For example, an upload from the local proxy local representative to a set of remote replica local representatives should be able to use all available bandwidth when possible to minimize upload time. The same statement holds for downloads from a replica local representative to a proxy.

To achieve maximum utilization when possible, the block size used should at least be larger than the network connections' *delay-bandwidth product* [Partridge, 1994], because of the properties of Globe distributed shared objects. Method invocations on a Globe distributed shared object are, at this point in time, all syn-

[3] The "why" of this method's FileSize and traceCert parameters and endFileAddition's traceSig parameter are explained in Chap. 5 and Chap. 6.

chronous, which means that control is not returned to the invoking client by the local representative until a reply has been received from its remote peer(s).

This property implies that, to fully exploit the network, the size of the blocks uploaded by the client should be large enough to allow the local representative to fill the network links to its peers. The higher the capacity and the latency, the larger the block size should be to get full utilization. For example, to fill a 1 Gigabit per second connection to a host 25 milliseconds away, the block size should be at least 3 Megabytes. Furthermore, the block size cannot be static, as it will have to grow when bandwidth becomes more abundant.

The risk of choosing a block size that is too small is that the proxy local representative may not be able to use the available network links to their maximum. As a result, up and download times become high (when link delay is high) or people may feel the application is not making full use of their network connection (when delay is low).

The obvious solution is therefore to simply choose a large block size. The communication subobject will prevent congestion due to the large amount of data being sent and can do flow control when necessary. In general, these facilities will be provided by the TCP protocol. When using UDP or group communication they will have to be implemented by the local representatives themselves in their communication subobjects.

Block size and server capacity The problem with this solution is, however, that block sizes must not be too large, as they impact memory usage at the object servers hosting the replicas of the revision DSO. A large block size can have a negative impact on the performance of the object servers because of the way DSOs are implemented, as follows. The performance of an object server, in terms of the number of simultaneous clients that can be supported, is bounded by the available hardware resources and how the object-server software makes use of them. Performance can be limited by either the capacity of the network connection, CPU power, memory size, I/O bandwidth to persistent storage, or storage capacity itself.

In the current Globe design the block size used by the revision DSO's interface determines directly the amount of memory needed at a replica local representative to process the method invocations for uploading or downloading a block to or from a DSO. As can be seen in Sec. 3.4, the control subobject of a replica makes the same method invocation with the same parameters on the semantics subobject as the client made on the proxy local representative. Therefore, if the block size is large, there is a small risk of available memory becoming the bottleneck of the object server before any of the other resources, preventing it from supporting more clients at the same time. Our clients currently use a block size of 1 Megabyte.

In the proposed interface the client is in charge of the size of the blocks, which is a disadvantage of this interface. Globe adheres to a separation-of-concerns principle in which the nonfunctional aspects of the application are handled by the middleware, and the functional aspects are handled by the client and the semantics subobject. As just shown, the block size must not be to small and not be too large, and depends on network and server capacities. This conclusion implies that block size is a nonfunctional aspect of the application, and should therefore, in principle, be controlled by the middleware, rather than the client of the revision DSO.

A block interface in Globe does not, however, allow the middleware to control the block size. When uploading a file via the block interface, the middleware, in particular, the control subobject has access only to the block of data given to it by the client. To get access to more data at a time the middleware would have to give feedback to the client, which is currently not possible or desirable in Globe. A similar argument holds for downloading files. When a client requests a block from a revision DSO, this method invocation is shipped to a replica. Next, the control subobject of the replica invokes the "download block" method on the semantics subobject which then returns a block of data. If the middleware wants to control the size of the block returned by the semantics subobject, it will have to communicate with this semantics subobject. The Globe design currently does not allow such communication, as it also represents a violation of the separation-of-concerns principle.

As mentioned above the block size problems are introduced by the fact that Globe has synchronous method invocations. We can work around the synchronous nature of method invocations by using multiple threads and doing parallel invocations. Consider the following example. A client creates a small number of threads, say 3. The client makes thread 1 invoke the method to download the first block, makes thread 2 download the second block and makes thread 3 download the third block, with a small delay between each invocation to prevent them from competing for resources. As soon as a block is returned the thread returning from the invocation starts downloading the next block the client does not have yet. This solution, sometimes called *prefetching*, increases throughput. Unfortunately, it complicates the client code considerably and is not an elegant solution to the problem. An alternative interface for revision DSOs that does provide an elegant solution but is complex to implement is discussed in Sec. 4.3.3.

The advantage of the block interface is that it is easy to implement. In addition, these methods are *pseudo-idempotent*; that is, their implementation is idempotent or their implementation can detect duplicate calls (such as, for example, the startFileAddition method). Pseudo-idempotent methods can make some fault-tolerance measures easier to implement. On the other hand, an upload using this interface is no longer an atomic operation that can either fail or succeed, the operation now consists of several method invocations. The loss of atomicity makes failure handling harder because a multi-method operation is generally harder to rollback when it fails halfway through and requires application-specific knowledge. We return to this issue in Chap. 6.

4.3.2. Stateless Downloads

In the previous section we discussed the part of the revision DSO's package interface related to uploading files, and how to handle large arguments to methods (related to both upload and download of files) in general. This section explains the methods in the package interface related to downloading.

The design of the download methods has been dictated by two principles. The first principle is that methods should be made read-only (i.e., not modify the state of the DSO) whenever possible. In most replication protocols, doing a read operation is cheaper than doing a write operation, because the latter type of operation generally requires that all replicas are (eventually) made consistent, requiring interreplica communication and administration.

The second principle concerns scalability: a distributed shared object should not store any per-client state whenever possible, as storing this information may cause troubles when the DSO is used by large numbers of clients. Moreover, keeping per-client state turns a method that might otherwise be implemented as read-only into a write method (the client-related entries in the DSO's state have to be updated, implying a write on the DSO), and thus violates the first principle. Furthermore, if a DSO has many clients that fail permanently or misbehave, a garbage collection mechanism is needed to purge their stale entries, complicating the implementation. Note that letting an individual replica keep track of the state of its clients is not an option since it does not allow a client to easily fail over to another replica.

The application of these two design principles has led to the two methods for downloading shown in Figure 4.9, both of which are read-only methods. A client downloads a file by repeatedly calling the getFileContent method. The client itself is responsible for adjusting the offset into the file, thus avoiding per-client state in the distributed shared object. The last block of the file has been downloaded when the amount of data is returned is less than the requested block size.

Not having a revision DSO keep track of per-client state introduces a concurrency issue, however. The package interface as defined above provides protection against simultaneous upload of two files under the same name, but does not entirely protect a client downloading a file against replacement of that file. In other words, the GDN does not support *immutable files*. To replace an existing file, the uploader first deletes the old file (using the deleteFile method discussed below) and then uploads the new file under the same name using the regular upload methods

```
interface package // up- and download methods
{
        void startFileAddition( in string FileName,
                                in long long FileSize,
                                in traceCert traceCertificate );
        void putFileContent( in string FileName,
                             in long long Offset,
                             in sequence<octet> Block );
        void endFileAddition( in string FileName,
                              in traceSig traceSignature );

        void getFileContent( in string FileName,
                             in long long Offset,
                             in long long MaxBlockSize,
                             out sequence<octet> Block );
        long long getFileSize( in string FileName );
        ...
};
```

Figure 4.9: Second part of the revision's DSO interface: the methods for downloading a file from the DSO. Error parameters have been left out for clarity.

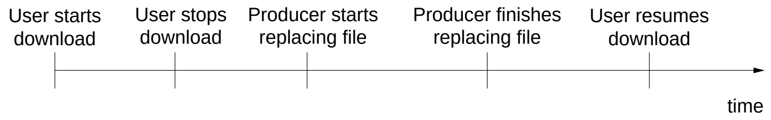


Figure 4.10: A user downloads a file from a revision DSO in two sessions. In between the sessions, the producer managing the DSO replaces the file with another.

(startFileAddition, putFileContent and endFileAddition).

In most cases, a client receives an error when a file it is downloading is being replaced. The client either receives a “file not found” when calling getFileContent after the deletion of the file and before the upload of the new file has started or a “file in use” error, when calling getFileContent after the upload started. However, when the file replacement takes place entirely between two consecutive invocations of getFileContent, the client receives no error and does not immediately notice the replacement. Only at the end of the download the client is likely to notice that something went wrong in most cases, when the size of the file downloaded turns out to be smaller or larger than indicated by getFileSize at the start of the download. If the DSO were aware of the download it could signal the problem to the downloading client.

Although this replacement problem is not likely to occur frequently in a software distribution network where the files are versioned, some software producers might, however, not practice correct publishing procedures and replace existing files. In addition, the problem is not entirely theoretical given that the GDN should support discontinuous downloads (and uploads). In today’s Internet the average bandwidth available to a user is such that it requires a long time to download a large file. It is therefore useful to allow a user to stop an up or download and restart the operation at a time convenient to the user. This functionality requires support in the up- and download tools and does not affect the package interface, except for the exceptional case indicated above.

Because downloads may be paused by the user, the scenario of a file being replaced between two calls to getFileContent becomes more likely, as illustrated in Figure 4.10. As it is annoying to find out at the end of a discontinuous download that the large file downloaded is corrupt, the package interface is changed to allow the replacement to be detected, as shown in Figure 4.11.

The interface is changed as follows. We basically make files in a revision DSO immutable by assigning them a so-called *incarnation ID*, a 64-bit random number, when they are uploaded. The incarnation ID changes when a new file is uploaded under the same name. As a result, when a file is deleted (using the deleteFile

```
typedef long long      incarnationID_t;

interface package // final
{
    void startFileAddition( in string FileName,
                            in long long FileSize,
                            in traceCert traceCertificate );
    void putFileContent( in string FileName,
                         in long long Offset,
                         in sequence<octet> Block );
    void endFileAddition( in string FileName,
                          in traceSig traceSignature );

    incarnationID_t getIncarnationID( in string FileName );

    getFileContent( in string FileName,
                    in incarnationID_t IncarnationID,
                    in long long Offset,
                    in long long MaxBlockSize,
                    out sequence<octet> Block );

    long long getFileSize( in string FileName,
                           in incarnationID_t IncarnationID );

    void getTraceInfo( in name fileName,
                       in incarnationID_t IncarnationID,
                       out traceSig traceSignature,
                       out traceCert traceCertificate );

    void deleteFile( in string FileName );
    void allFiles( out sequence<string> FileNames );
};
```

Figure 4.11: The complete package interface of a revision DSO. Error parameters have been left out for clarity.

method, explained below) and a new file under the same name is uploaded into the revision DSO, these files have different incarnation IDs. A replacement-safe download proceeds as follows: at the start of the download the client obtains the incarnation ID of the file by calling getIncarnationID and stores this value locally. The signature of the getFileContent method for downloading the contents of a file is changed such that it now takes an incarnation ID as an additional parameter. If the incarnation ID passed as parameter and the incarnation ID stored in the distributed shared object do not match, the getFileContent returns a "stale incarnation" error. These changes allow a download tool to detect the replacement of the file immediately after resuming the download.

Shown in Figure 4.11 are the three methods of the package interface which we did not yet discuss. The getTraceInfo method is explained in Chap. 5, the deleteFile method can be used to remove files from the DSO and the allFiles method can be used to obtain the list of names of all the files in the DSO.

In the next section we discuss alternative interfaces before going into the semantics and implementation of revision DSOs.

4.3.3. Alternative Interfaces

The disadvantage of the block interface is that it puts the client in control of the size of the blocks in which a file is transferred. This section shows two alternative interfaces that put control over the block-size parameter in the hands of the network-aware parts of a revision DSO. Another alternative is to extend Globe with asynchronous method invocations which we discuss first.

Alternative 0: Asynchronous Method Invocations

A solution that achieves full network utilization (largely) irrespective of the block size chosen is to introduce asynchronous method invocations to Globe. With asynchronous method invocations, during an upload, the proxy local representative returns control to a client after an invocation to the "upload block" method without waiting for the reply from the replica(s). A client can then immediately reinvoke the method to upload the next block, allowing the proxy local representative to utilize the network to its full potential and thus reducing upload time for the client. The disadvantages are that asynchronous method invocations not only make client code more complex, but implementing them also require significant changes to the generic implementation of a local representative, or the introduction of a completely application-specialized implementation.

Alternative 1: Transferring Files via Serializable Local Objects

To understand this alternative interface we first have to take a look at a particular aspect of Java's Remote Method Invocation (RMI) mechanism [Wollrath et al., 1996; Riggs et al., 1996]. Java RMI supports the passing of regular Java objects as arguments in calls to remote Java objects via a pass-by-value mechanism. This mechanism copies the regular Java objects to the remote host and is based on the standard Java marshalling facilities.

The mechanism works as follows. When a client invokes a method on a remote Java object that takes a regular (local) object as an "in" parameter, a reference to the local object is passed to the remote object's proxy in the local address space. The proxy uses this reference to create a marshalled copy of the local object by calling the object's standard serialization method writeObject. This marshalled copy is sent, along with the rest of the method invocation, to the host running the remote object. At the remote host, the marshalled object is unmarshalled by creating a new instance and calling its readObject method, thus creating a copy of the local object on the remote machine. When the run-time system makes the actual invocation on the remote object, a reference to the copy of the local object is passed. A similar procedure of marshalling and unmarshalling is used for local objects that are "out" parameters of a remote object's methods, only the local object is initially created at the server side and then copied to the client side.

By introducing a similar pass-by-value mechanism for local objects in Globe we can upload and download large files into a Globe distributed shared object without putting clients in charge of block sizes, as follows. The basic idea is that files are encapsulated in local Globe objects[4] that are subsequently copied to remote hosts via the pass-by-value mechanism. Since, in this mechanism, the local representatives (proxies and replicas) of the remote object are in charge of marshalling the local objects, control over the block size in which the file is transferred is in their hands, and not in the clients'. To illustrate how the object-by-value mechanism works in Globe, we look at how a file is uploaded into and how a file can be downloaded from a revision DSO in turn.

Uploading files The upload is illustrated in Figure 4.12. A client wishing to upload a large file first creates a local object that logically encapsulates the file; that is, it holds a reference to the file but not the file's contents, which are kept on disk (arrow 1 in Figure 4.12). This local file object (labeled F in the figure) imple-

[4] Globe, in addition to having its own distributed-object model, also has a specific model for local objects, where objects are accessed via *binary interfaces* to achieve programming language independence. The model is similar to the model adopted by Microsoft's COM [Rogerson, 1997]. For the purpose of this dissertation, however, Globe local objects can be considered regular programming-language objects.

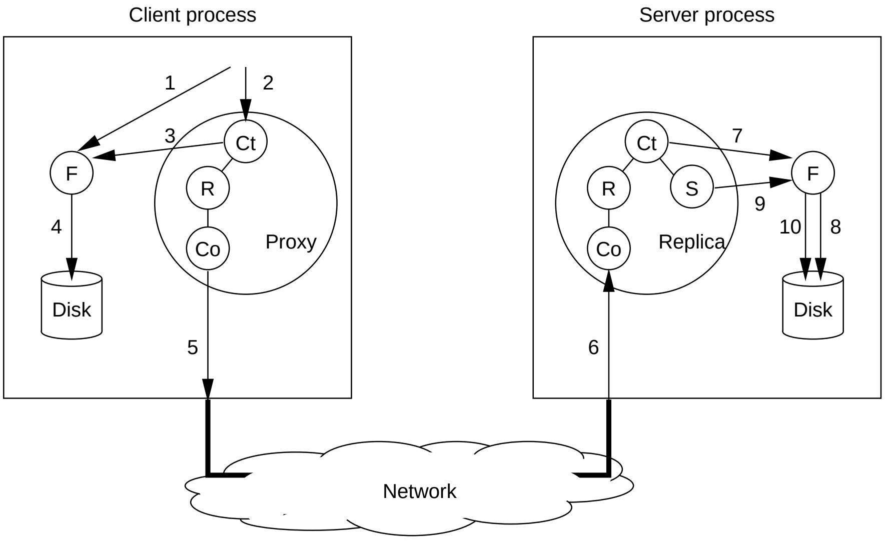


Figure 4.12: The file to be uploaded is encapsulated in a local object which can marshall itself and can thus be transmitted by the control subobject to its peers, similar to marshalling a semantics subobject during slave updates or the installation of a new replica. Arrows indicate only an invocation, not the return of the invocation.

ments a modified version of the semState interface that is normally used for marshalling and unmarshalling the semantics subobject, called semLargeState. This semLargeState interface allows the state of a local object to be marshalled in parts. To upload the file into the distributed shared object, the client invokes a method on the local representative called, for example, addFile (arrow 2 in Figure 4.12). This method takes as parameters a global identifier for the file and a pointer to the local file object. When the invocation is being marshalled, the control subobject uses the new semLargeState interface of the local file object to read out its contents (arrow 3 in the figure). The local object implements this interface by reading a block of the file from disk (arrow 4), adding some administrative information, and passing this data on, until the whole file has been marshalled. The local object can optimize disk access by reading ahead if such facilities are not provided by the operating system.

The control subobject of the proxy sends the marshalled blocks to its peers in the replica local representatives (arrows 5 and 6 in Figure 4.12). The control subobject of each replica creates an empty local file object and starts filling it with the marshalled contents received from the sending control object (arrow 7). This local file object creates a new file on the local file system and writes the unmarshalled blocks into the file, thus creating a local copy of the original file (arrow 8 in the figure). When the transfer is complete, the normal invocation procedure is resumed and the addFile invocation is made on the local semantics subobject, with a reference to the newly created local object encapsulating the local copy of the file as second parameter. The semantics subobject can now access the file via other, potentially application-defined, interfaces of the local file object (arrows 9 and 10 in Figure 4.12).

Downloading files Consider a getFile method for downloading files from a revision DSO that takes two arguments, the (global) identifier for the file to be downloaded and an “inout” parameter, in particular, a pointer to an empty local file object. The idea is that this empty file object will be copied to the machine hosting the replica of the revision DSO, where it will be filled with the contents of the desired file and then copied back to the client machine, using the pass-by-value mechanism.

In detail, this procedure works as follows. The local file object is initialized with a target local filename to which it writes its state, which will be the downloaded file. The target file name is specified by the user of the download tool. To download the file, the client invokes the getFile method on the revision DSO’s local representative. Because getFile’s file-object parameter is an “inout” parameter the local representative marshalls the (empty) file object (the target filename is ignored) and sends it, along with the rest of the marshalled method invocation, to a replica local representative. The replica's control subobject, before making the getFile invocation on the semantics subobject, creates an empty local object and unmarshalls the empty state of the original local object in it. A pointer to this local object is passed as the actual parameter of the getFile invocation on the semantics subobject.

The semantics subobject getFile method now logically copies the file from its state into the local file object. Physically, the semantics subobject simply invokes a method on the local file object instructing it, when asked to marshall itself, to read file X from local storage and present that as its state. When the semantics subobject returns control to the control subobject of the replica, the control subobject marshalls the local file object and ships it as the result of the getFile method invocation to the proxy local representative in the download tool. The control subobject in the proxy unmarshalls the local object into the empty local file object supplied by the client, using the file object's semLargeState interface. The file object writes the file data to the local file system under the target file name with which the file object was initialized at the beginning, thus completing the download operation.

Using an inout parameter instead of an out parameter to pass the local file object, and setting the target filename beforehand avoids an extra copying step at the client side. With an inout parameter, the file object can directly write the file to the location specified by the user of the download tool beforehand. Limiting the number of copy operations is important, in particular for large files, for copying can negatively affect performance and increases disk-space requirements.

Discussion As already mentioned, the advantage of this alternative over the block interface is that it puts the whole contents of the file at the disposal of the local representative in one action, enabling it to send the file out in blocks as large as optimal. Furthermore, it makes the up- and download of a file an atomic operation again, simplifying failure handling.

Extending Globe with a pass-by-value mechanism for local objects is, however, nontrivial, as it complicates the implementation of the control and replication subobjects. In particular, the implementation of these subobjects should be such that multiple method invocations can be processed in parallel. Processing in this case refers to the handling of the method-invocation requests and the replies, and not the execution of the methods which is always done serially (as Globe requires that all method invocations on a semantics subobject are serialized, see Sec. 3.2). Parallel processing is necessary to achieve good up- and download times. In the block-based interface, methods do not take a lot of time to process, and this can therefore be done serially. In the pass-by-value solution, however, the up- and download methods operate on the whole file, requiring longer processing. If these methods cannot be handled in parallel, clients have to wait for all previous clients to finish their operations before theirs is executed, which means that the time between the start of an upload or download and its completion, as perceived by the user, will increase.

Allowing parallel processing complicates the subobjects' code and has significant impact on the standardized interfaces of the subobjects. Significant adjustments have to be made to the repl interface (discussed in Sec. 3.4.1). In particular, the repl::send should be changed to allow a control subobject to send large marshalled method invocations and invocation replies. The replCB interface should be changed to allow the replication subobject to pass a large marshalled method invocation to the control subobject in parts. Furthermore, the old semState marshalling interface has to be upgraded to allow marshalling and unmarshalling of objects with large state. As we will see below, this upgrade of the semState interface is also necessary to support semantics subobjects (i.e., distributed shared objects) with very large states. The exact changes to the interface and the resulting semLargeState interface can be found in Sec. 4.5.1.

The pass-by-value solution presented here requires more concurrency measures. In case of a download, the file is not accessed by the semantics subobject but by the control subobject. As a result it is possible that the file is deleted by a subsequent method invocation while the control subobject is still reading the contents. The Globe run-time system will have to be changed to prevent such concurrency. Another implementation issue is how the local representatives access the local file system. To keep its implementation operating-system independent, a local representative should use a standardized interface provided by the local run-time system that hides the operating-system specific features. Filenames, or persistent IDs in general, are valid only on the local machine and care should be taken that these identifiers are not transmitted. For example, when these names are stored in the state of the local semantics subobject, the marshalling code should marshall the contents of the file rather than their names.

Alternative 2: Encapsulating The File in a Distributed Shared Object

An alternative to turning files into local Globe objects and passing them by value is to use temporary distributed shared objects to transfer files between hosts. The idea is that the host containing the file to be transferred encapsulates the file in a temporary distributed shared object, thus making the file remotely accessible. The hosts that need the file bind to the temporary DSO and read the file using its methods. The temporary DSO can be specialized for one-to-many and one-to-one communication patterns (i.e., uploads or downloads, respectively) thus optimizing the transfers. We discuss upload and download of files via temporary DSOs in turn.

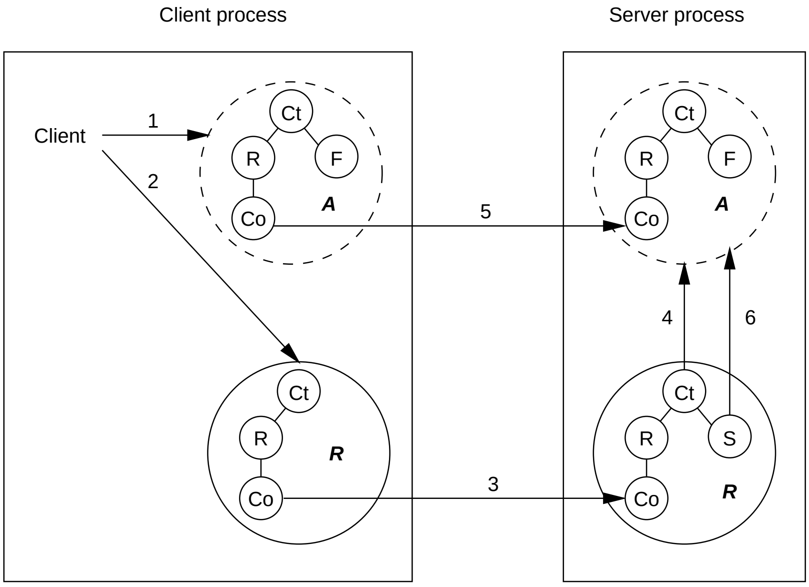


Figure 4.13: Uploading a file into DSO *R* using DSO *A*. The local representatives of DSO *A* are drawn dashed to indicate it is a dynamically created and temporary DSO.

Uploading files The use of temporary DSOs to upload files into a revision DSO is illustrated in Figure 4.13 for a revision DSO *R* using a master/slave replication protocol. A client wishing to upload a file into the revision DSO creates a temporary distributed shared object, called the *argument DSO*. This argument DSO (labeled *A* in Figure 4.13) has a single replica local representative on the client's machine and the state of object is the file to be uploaded. The file is put into the argument DSO via method invocations, or is passed as initialization data at creation time, after which the DSO becomes read-only. The replica of the argument DSO does not register itself with the Globe Location Service. These steps are represented by arrow 1 in Figure 4.13.

To upload the file, the client invokes a method called addFile on the revision DSO (arrow 2 in Figure 4.13). This method takes two arguments: the global identifier for the file and a reference to the argument DSO, more precisely, a pointer to its local representative. When marshalling the addFile method invocation, the control subobject of the revision DSO replaces the pointer to the argument DSO's local representative with the contact address of the argument DSO. The marshalled method invocation is then transmitted to all relevant replicas of the revision DSO (arrow 3 in the figure).

The replicas, when receiving the method invocation, extract the marshalled contact address and use it to bind to the local representative of the argument DSO on the client machine (arrow 4 in the figure). During binding, a new local representative of the argument DSO is created in the local address space, in particular, the run-time system creates a replica local representative which means that a copy of the state of the argument DSO (the file to be uploaded) is transferred from client to replica host (arrow 5).

When binding is finished, the control subobject of the revision DSO invokes the addFile method of the revision DSO's semantics subobject, passing the global identifier of the file and a pointer to the argument DSO's newly created local representative as arguments. The addFile of the semantics subobject can now read the file and add it to its state by invoking methods on the argument DSO (arrow 6 in Figure 4.13.

Downloading files Files can be downloaded from a revision DSO using temporary DSOs as follows. Consider a getFile method for downloading files that takes just the global identifier for the file as a parameter and that returns the contact address of a distributed shared object. When the client invokes the getFile method, the invocation is marshalled and shipped to the nearest replica in the usual way. The getFile method of the replica's semantics subobject, when invoked by the replica's control subobject, creates a new temporary distributed shared object. This argument DSO has one local representative, co-located with the replica and has the desired file to be downloaded as its state. After the argument DSO has been created, the semantics subobject obtains its contact address and returns this as the result of the getFile method. The contact address is marshalled and sent back to the proxy local representative of the revision DSO in the client's address space, which, in turn, returns it to the client.

Using this contact address, the client binds to the argument DSO, thus creating a local representative of the argument DSO in its address space. The client then reads the contents of the file by invoking methods on this local representative. The local representatives of the argument DSO are specialized for file transfer and use a prefetching mechanism to make sure the network connection between source and destination hosts is used efficiently. Their implementation is therefore different from the argument DSOs used for uploads.

Discussion For downloads, I choose prefetching over creating a slave replica at the destination host, as is done during uploads. The reason for doing so is that prefetching avoids an extra copying step for large files. When creating a slave replica, the state of the argument DSO has to be transferred to the client as a whole, unmarshalled and written to disk by the semantics subobject when the state is large (see Sec. 4.5.1). This copying step is not necessary when doing prefetching since the specialized local representatives can prefetch blocks of the file small enough to be kept in core.

The argument DSO's local representatives (both for up and for downloading) cannot be registered in the Globe Location Service, since this would result in an insert operation on the Location Service for each invocation of the addFile and getFile method, thus increasing the load the Globe Distribution Network generates on the GLS considerably.

The advantage of using temporary distributed shared objects is that the argument DSO can control how the state and thus the file is transmitted to its local representatives. Unfortunately, this solution also has three disadvantages. First, the implementation of argument DSOs may have to be tuned to the replication protocol used by the revision DSO. For example, if the revision DSO is actively replicated [Schneider, 1990] and the argument DSO is not aware of this, there is a risk of overloading the client machine, as follows. During an addFile method invocation (i.e., an upload) on an actively replicated revision DSO, the client machine will receive many bind requests for the argument DSO (one for each replica of the revision DSO). As a result, it will have to marshall and transmit its state many times, which may overload the machine. To prevent this problem the argument DSO should be specialized to be used in conjunction with an actively replicated revision DSO. In particular, the argument DSO's state-transfer mechanism, which normally pulls the state from a replica, would have to be replaced with a simultaneous push to all binding hosts, which is more efficient for an actively replicated revision DSO.

Second, the argument-DSO mechanism cannot be used in conjunction with all replication protocols, in particular, not with protocols that use a lazy form of replication. When doing lazy replication, a write-method invocation on the revision DSO may return before the method has been executed at all replicas. This form of asynchrony works as long as a method invocation is self-contained. This is not the case when using temporary DSOs to upload files, because executing the addFile method requires the existence of the argument DSO. There is no simple solution to this problem, because it is difficult to determine when a method invocation has been processed by all replicas of a lazily replicated revision DSO.

The third disadvantage of this solution is that it also has a negative impact on upload and download times. Recall that all method invocations on a semantics subobject are serialized (see Sec. 3.2). This property, and the fact that in this solution the up- and download methods operate on the whole file, imply that all uploads and downloads are now strictly serialized, whereas in case of a block-based interface multiple up and downloads can proceed in parallel. As a result, clients have to wait for all previous clients to finish their operations before theirs is executed, which means that the time between the start of an upload or download and its completion, as perceived by the user, will increase for large files.

4.3.4. Semantics of a Revision Object

The *semantics of an object* is the way an object behaves observed by means of the effect of method invocations on the (state of the) object and the possible values returned by each invocation and the meaning of those values. Knowing the semantics of an object is important for the clients of the object. In most object-oriented programming languages today there is no formal definition of an object's semantics because creating this definition is generally considered not worth the effort, and an informal definition generally suffices.

I believe that for distributed objects it is important to define one part of the object's semantics. In particular, every replicated object should make explicit the *consistency model* used. The consistency model describes the effects of write operations on a replicated data item as observed by different clients. The consistency model of a distributed object therefore describes the results of read-only methods and the effects of state-modifying methods on the (replicated) state of the object.

The consistency model determines to a large extent the semantics of the distributed object. For example, when a replicated object uses the single-copy consistency model, every client of the object will get the same result when invoking the same read-only method with the same parameters at the same time. However, when a replicated object uses a weak consistency model, two clients may get different results when the object has multiple replicas that are not (yet) consistent and each client talks to a different replica. In other words, the observed behavior of the object is different for each client.

For most applications, a distributed object should behave as if it were an object running on a single machine. Sticking to strict single-copy semantics (i.e., *sequential consistency* [Lamport, 1979]), however, can seriously impede performance and is not required for all applications. If possible and when necessary from a performance perspective, an application developer should therefore select a weaker consistency model. Selecting a different consistency model is relatively easy and merely consists of selecting a replication protocol for the object that implements the desired consistency model. We return to the process of selecting a replication protocol for an object in Sec. 4.6.

I require for the Globe Distribution Network that revision DSOs have single-copy semantics, as a software distribution network that does not provide each user with an accurate listing of its contents is not useful. People accessing the GDN expect to see all revisions and variants that are published by the producer at that point in time, otherwise they will use other distribution channels that are more up-to-date. This means that a revision DSO should contain the archive files of all available variants and should keep its replicas consistent at all times, such that when a user binds to the revision DSO and lists its contents he will always see all available archive files.

4.4. REFERRING TO PACKAGES, REVISIONS AND VARIANTS

This section describes how packages, revisions and variants are named in the Globe Distribution Network. Recall that in the naming scheme of the Globe middleware (explained in Sec. 3.3), distributed shared objects are identified by a symbolic name which is mapped to an object handle using the Globe Name Service. The object handle is, in turn, mapped to contact addresses using the Globe Location Service.

The symbolic names for revision DSOs are structured as follows:

globe:<*prefix*>/<*package name*>/<*revision ID*>

The *prefix* part of the name denotes the name space in which the revision DSO's name is registered. At present there is a single global name space for all Globe distributed objects, and the prefix denotes the part of the global name space in which the registrar of the object is allowed to register (name, object handle) pairs. We return to the issue of who registers names for objects in the chapter on security (Chap. 5). For the purpose of this dissertation, a prefix describes a path through a hierarchy of domains in which the Internet is divided, as in the Domain Name System [Mockapetris, 1987]. An example prefix is /nl/vu/cs/globe/arno. Package name and revision IDs, the other components of the object name, are currently free-form, but this might change in the future when more advanced querying facilities, such as those described in Chap. 2, are added to the GDN. Examples of full Globe object names are:

- globe:/nl/vu/cs/globe/arno/lstreeviz/1.0
- globe:/org/gimp/gimp/1.2.1

Archive files in a revision DSO can be referred to by concatenating their file name to the revision DSO's object name. For example,

globe:/org/gimp/gimp/1.2.1/sparc-solaris.tar.gz

Distributions and DistributionArchive DSOs are named in a similar fashion. The DSO containing the ISO9660 image of RedHat 7.1 for i386 processors is named:

globe:/com/redhat/distribution/7.1/i386/ISO9660

4.5. IMPLEMENTATION OF A REVISION OBJECT

Sec. 4.3 discussed the interface and semantics of a revision DSO. This section describes the essentials of the implementation of a revision DSO. Security and fault-tolerance aspects are discussed in Chap. 5 and Chap. 6, respectively. There are three important implementation issues for a revision DSO: (1) how to deal with a semantics subobject with large state, (2) how to make a revision DSO persistent, and (3) what replication protocol should be used to achieve efficient distribution. The first two issues are discussed in this section. The latter issue warrants a separate discussion in Sec. 4.6. This section and the following chapters also include a description of our initial implementation of the Globe Distribution Network.

4.5.1. Handling Large State

The state of a revision DSO, or rather, the states of the individual semantics subobjects in the DSO's various local representatives, can grow large because of large files being uploaded into the DSO. As a result it becomes undesirable and infeasible to keep these states in memory. The solution to this problem is to either partition the state over multiple machines or to use secondary storage. Given the low cost of hard-disk space compared to memory we discuss only using hard disks for storing large states.

To keep memory footprint small, the revision DSO's semantics subobject directly stores files that are uploaded on disk. The files are written to disk using an operating-system independent interface to the local file system offered by the Globe run-time system, which uses a numerical *persistence IDs* to identify files. The semantics subobject keeps only the mapping from global file identifier (i.e., the filename passed by the uploader in the call to startFileAddition) to persistence ID in core. When the object's users store many files in the revision DSO, this mapping becomes large, in which case it is also stored on disk. The operating system is assumed to cache files in memory to optimize performance.

A potential disadvantage of having semantics subobjects access persistent storage is that local representatives are always kept in virtual memory. Sec. 4.5.2 describes a passivation/activation mechanism for local representatives that allows an object server to swap out local representatives and thus support many local representatives without an infinite amount of virtual memory. How failures such as running out of disk space and media failures are handled is discussed in Chap. 6.

Transferring Large States

The current replication interfaces replCB and semState (shown in Figure 3.9 on page 36 and Figure 3.11 on page 38, respectively), can marshall, transmit and unmarshall the state of a DSO only atomically. Atomic operations are no longer desirable or feasible if the size of the state becomes large: they are not feasible when the state is larger than the host's virtual memory and not desirable from a performance viewpoint.

Before explaining the modifications made to these interfaces we first briefly recapitulate when state transfers take place. In the current Globe implementation state updates or retrievals are always initiated by the replication subobject. State transfers occur in two cases: (1) when a new replica is created and (2) in some replication protocols (e.g., master/slave) after a state-modifying method has been executed. In the former case, the replication subobject of the new replica sends a GETSTATE request to a nearby replica. Following this request, the replication subobject at the existing replica calls replCB::getState on the control object, which is turned into a semState::getState invocation on the semantics subobject. The same procedure is followed in the latter case in a master/slave protocol that uses invalidates. In the case of a master/slave replication protocol where the master's replication subobject sends out the new state after each update, the state is transmitted to the slaves in a NEWSTATE message which is turned into calls to replCB::setState and semState::setState on, respectively, the control and the semantics subobject, as described in Sec. 3.4.3.

The replCB and semState interfaces are adapted to accommodate large state as follows. The idea is to enable transfer of states in blocks. To this extent, the getState methods in both interfaces are extended with two additional arguments: a MaxBlockSize "in" parameter and a boolean "out" parameter named MoreBlocks. The MaxBlockSize is used by the client (replication or control subobject) to indicate the maximum size of a marshalled block of state it is able to process. The MoreBlocks parameter is used by the control or semantics subobject to signal to the client that not all of the state has been marshalled. A client will continue requesting blocks of marshalled state until the object sets this flag to false. The setState method is extended with one "in" parameter: a boolean MoreBlocks parameter, used by the client to signal more blocks are forthcoming. This parameter is in a sense superfluous, because the object can probably also derive the fact that more blocks are coming from the marshalled data, but is added for convenience. The updated interfaces, called replLargeStateCB and semLargeState, respectively, are shown in Figure 4.14.

The disadvantage of this solution is that it makes the application programmer's job (the implementor of the semantics subobject) harder. Implementing marshalling and unmarshalling methods that use blocks is tedious. The advantage of this solution is that it does not require major changes to the current Globe implementation.

```
interface replLargeStateCB
{
    void handleRequest( in sequence<octet> MarshalledRequest,
                out sequence<octet> MarshalledReply );
    void getState( out sequence<octet> MarshalledStateBlock
            in long long MaxBlockSize, out boolean MoreBlocks );
    void setState( in sequence<octet> MarshalledStateBlock,
            in boolean MoreBlocks );
};
```

(a)

```
interface semLargeState
{
    void getState( out sequence<octet> MarshalledStateBlock,
            in long long MaxBlockSize,
            out boolean MoreBlocks );
    void setState( in sequence<octet> MarshalledStateBlock,
            in boolean MoreBlocks );
};
```

(b)

Figure 4.14: The (a) replCB and (b) semState interfaces adjusted to handle distributed shared objects with large state.

Alternative Solutions

Using Virtual Memory to Handle Large State An alternative way for handling semantics subobjects with large state is to use the operating system's virtual memory facilities. The idea is to configure the host running the local representative in such a way that the disk space that is normally used to store files is available for swapping. Such a configuration allows the semantics subobject to ignore I/O issues and just store the state in virtual memory.

Requiring a specific host configuration conflicts with the design goal of the GDN to allow many different people and organizations to contribute server capacity, which makes this alternative less interesting. Furthermore, the files in the local representatives have to be stored on disk persistently to allow server reboots and handling server failures without requiring a re-fetching of all files from another replica. Most operating systems do not provide persistent and fault-tolerant virtual memory, implying that a host needs double the disk space. Persistent local representatives are discussed in the next section.

To circumvent the special configuration and double disk space problems and to improve performance over the direct file I/O solution used above, the semantics subobject could use *memory-mapped files*. Most modern operating systems have facilities to allow a process to access files on persistent storage as if they were regions in memory by clever use of the virtual memory subsystem. The advantage of this approach is performance: memory-mapped files are not cached in the kernel, nor does the data need to be copied from kernel to user space for a user process to access it. This optimization might require operating-system specific semantics subobjects, but these are supported via our binding mechanism. Another solution is to use the facilities of the language environment. For example, Jaguar is an extension to Java which supports memory-mapped files presented as Java objects [Welsh and Culler, 1999]. To use these facilities in Globe, one has to replace only the implementation of the operating-system independent interface to persistent storage offered by the Globe run-time system.

Using Stream-Based Interfaces for Transferring Large State An alternative way to transfer large states is to use stream-based interfaces such as used in Java's serialization/deserialization model. The methods for marshalling and unmarshalling a Java object use output, respectively, input streams to which the application programmer can directly write all of the marshalled state, or read out the marshalled state without having to take into account externally imposed boundaries. Furthermore, using streams relieves him of the burden of keeping administrative information between (un)marshalling calls.

Using streams in Globe local representatives is possible, but requires considerable reworking of the subobjects' interfaces. We therefore only briefly discuss how the subobjects would interact with stream-based interfaces. A replication subobject wanting to transfer a state to a peer local representative asks the communication object for an output stream for sending data to that peer. The replication subobject passes this stream to the control subobject via the replCB::getState call, which, in turn, passes it to the semantics subobject. The semantics subobject, when invoked, simply marshalls the state and writes it to the output stream, thus transmitting the data to the peer local representative via the communication subobject. At the receiving end, the replication subobject passes the control subobject an input stream logically connected to the output stream at the source local representative, via the setState methods. The semantics subobject uses this input stream to read the marshalled state. At both sides, the communication subobjects handles communication failures and make sure communication proceeds efficiently.

This alternative has conceptual advantages but complicates the implementations of subobjects. Consider the example of a replication subobject implementing a master/slave replication protocol with state shipping (i.e., after an update the master sends the new state to the slaves). In the proposed block-transfer method, the master replication subobject interacts with the communication subobject directly to send out the state during an outbound state transfer. When a slave crashes during the transfer (signaled by a SIGPIPE signal from the TCP connection), this error can be directly reported to the master replication subobject, which can then appropriate action (e.g. update its list of slaves). In the stream-based solution, special measures would have to be taken to ensure that such errors are reported to the replication subobject and not to the semantics subobject that is writing to the outbound stream. Another example is when a semantics subobject is receiving new state from a replica and this replica's host machine crashes during the transfer. Special care must be taken that the thread active in the semantics object reading the inbound state stops attempting to read the state and returns to the replication subobject, such that the replication subobject can select another replica to retrieve the state from. This can be complex if failure detection is done by other subobjects in the local representative. Streams may also require complex buffer management in the communication subobject.

As the design of distributed shared objects continues to evolve, it is preferable to keep the old interfaces until either the interactions between the existing subobjects and possibly new ones have stabilized or if it turns out that a stream-based interaction model would facilitate the integration of new subobjects. Furthermore, this alternative offers no immediate performance or scalability gains.

4.5.2. Persistent Revision Objects

Revision DSOs should be made persistent as this makes it easier to implement fault tolerance and allows reboots of individual or all object servers (i.e., supporting longevity). In this section we discuss only how revision objects are able to survive server reboots by storing their state on hard disk. Fault-tolerance aspects and the use of persistent storage for this purpose are discussed in Chap. 6. The use of persistent storage to handle distributed shared objects with large state, which we discussed above is orthogonal to this issue. It can, however, be exploited to achieve a more efficient implementation, as explained below.

Persistence Support of the Globe Object Server

The initial implementation of the Globe Distribution Network uses the persistence facilities offered by the Globe Object Server, which were discussed in Sec. 3.7. These persistence facilities are currently used for graceful reboots, which work as follows.

When the administrator of an object server instructs the server to shutdown for a reboot, the *server manager* part of the object server signals this fact to the *persistence manager*. The persistence manager, in turn, tells all local representatives running in the server to passivate themselves. A local representative that wants to stay on this object server then starts marshalling its state, which consists of data private to the local representative and the states of its various subobjects. The local representative first creates a new file using the run-time system's operating-system independent interface (which is implemented by the persistence manager). Next, it asks each of the subobjects for a marshalled version of their state using a special interface implemented by all subobjects, called lrSubobject. The semantics subobject creates a marshalled version of its state that can be used (unmarshalled) only locally, as opposed to creating a globally usable version when invoked by the control subobject during state shipping. Concretely, the semantics subobject of a revision DSO marshalls only the table mapping global filenames to the persistence IDs of the files in its state (or the persistence ID of the file containing the table, if the table was too large too keep in memory).

The replication subobject marshalls replication protocol state and the contact addresses it registered in the Globe Location Service. The latter are stored such that it can reallocate the same network contact points (e.g. TCP ports) when reactivated. The local representative completes its passivation by writing its own private (administrative) data and returning the persistence ID of the file containing all marshalled state to the persistence manager. The persistence manager stores the identifiers for all local representatives in a file, after which the server manager shuts down the object-server process.

Discussion

The advantage of the solution is that it avoids double storage which is important when the state of the semantics subobject is very large, as it is here. This solution also avoids extra copying steps (from disk to memory and from memory to disk), which can impact performance of the object server. In particular, not doing extra copying can speed up the reboot of an object server.

Fast passivation and activation is also useful for resource management, in particular, for managing the object server's use of virtual memory. In the initial implementation an object server always keeps all local representatives in memory. Keeping them in memory does not pose a problem, since the local representatives of revision DSOs are small, given most of their state is on disk. However, if a machine has more free regular disk space than virtual memory an object server can use passivation and activation to increase the total number of local representatives it can support (the number of local representatives that can run simultaneously, i.e., the server's *working set* is still bounded by virtual memory). It would passivate least-used local representatives and reactivate passivated local representatives when requests come in, based on some replacement policy. Our initial implementation of the object server does not yet support passivation and activation for these resource management purposes.

The disadvantage of this solution is that the programmer of the semantics subobject has to implement two sets of state marshalling and unmarshalling methods, one for local and one for global serialization/deserialization.

Alternatives

Homburg [2001] describes another solution where the replication subobject interfaces with a new *persistence subobject* to persistently store the state of the semantics subobject. The replication subobject keeps the persistence subobject informed about updates and supplies the persistence subobject with the marshalled state of the semantics subobject via the control subobject's replCB::getState interface.

The conceptual advantage of this solution over our solution is that the application programmer has to implement only one marshalling interface. An implementation disadvantage is that it does not allow the optimization just described, where state of the semantics subobject already on disk is not copied and stored on disk again.

4.5.3. Downloading over Multiple TCP Connections

According to The Globus Project [2000], using multiple TCP connections for a transfer can improve throughput on wide-area links, even between the same source and destination. This property is exploited in the GridFTP protocol [The Globus Project, 2000; Allock et al., 2001], an extension of the FTP protocol that, in addition to having improved performance, allows random file access, strong centralized authentication and access control, and restarts of transfers that failed due to communication or server errors.

The GDN can, unfortunately, make only limited use of this property of TCP on wide-area links. In particular, the GDN can use *parallel transfers*, that is, a transfer from a single server using multiple connections, but not *striped transfers*: a transfer of a single file from multiple (replica) servers [The Globus Project, 2000]. Parallel transfers can be supported by implementing a replication subobject that sends large requests and replies in blocks over multiple TCP connections, using a non-multiplexing communication subobject.

Striped transfers are difficult to implement in a distributed shared object. In a striped transfer different parts of the same file should be retrieved from different servers, which is incompatible with the distributed object model. Basically, *n* getFileContent method invocations should be sent to *n* different servers, and each method invocation should retrieve a different partition of the file (controlled by the Offset parameter). This behavior cannot be implemented inside a revision DSO's local representative given its current structure (i.e., the current decomposition in subobjects and their interfaces), because of the separation between application-specific and application-independent aspects. To allow striped transfers a revision DSO's local representative would have to be optimized specifically for the GDN. Striped transfers can be implemented by a GDN client, but only if it can create *n* different proxies (i.e., bindings) for the same DSO that each forward a method invocation to a different replica. Allowing a client such fine-grained control over binding behavior is considered undesirable from a transparency viewpoint.

4.6. IMPLEMENTATION: THE REPLICATION PROTOCOL

Given that in the GDN content is encapsulated in Globe distributed shared objects, these objects will have to be implemented such that their state is distributed and replicated in a way that optimizes the use of the Internet and the GDN servers. This section describes the replication protocol used and how it achieves this goal. The replication protocol is application specific, but is useful for other applications as well.

I propose to use a simplified version of the adaptive replication mechanism for World Wide Web documents developed by Pierre et al. [2000, 2001]. In this scheme a distributed shared object monitors its own read- and write-access patterns, and based on this information periodically adapts its *replication policy* and *replication scenario* (for the definition of these terms see Sec. 3.3). In other words, the object re-evaluates which replication protocol is optimal for its current access patterns and on which hosts replicas should be placed. This scheme has similarities with the SPREAD architecture [Rodriquez and Sibal, 2000], but operates at a higher level (i.e., SPREAD optimizes HTTP traffic by intercepting messages at the IP level).

In the adaptive replication mechanism, the distributed object takes a number of evaluation criteria into account to determine the best scenario, in particular Pierre et al. [2000]:

1. The download time perceived by the client
2. The number of times a client reads from an inconsistent replica
3. The amount of wide-area traffic generated

By applying weights to these three criteria, a distributed object is able to select a replication scenario that optimizes one or more criteria [Pierre et al., 2000, 2001].

To simplify matters, only one protocol is considered, but one that can be tuned on a per-object basis. Tuning concerns the number and placement of replicas (i.e., the object's replication scenario). The replication protocol chosen and the rationale for choosing this protocol is discussed in Sec. 4.6.1. Sec. 4.6.2 discusses how the replication protocol determines where to place replicas and how it balances network load. Server load balancing is discussed in Sec. 4.6.3. Handling flash crowds involves both network and load balancing and is discussed in Sec. 4.6.4. The influence of security and fault-tolerance requirements on the replication protocol are discussed in Chap. 5 and Chap. 6, respectively.

4.6.1. Basic Replication Protocol

The replication protocol used for all revision DSOs uses *active replication*; that is, when a method is invoked that changes the state of the object, the (marshalled) method invocation is sent to all replicas, and each of the replicas carries out the method invocation individually (in contrast, when *passive replication* is used, only a single replica (the master) executes the write method). A client invoking a state-modifying method sends it to a central replica, called the *core replica*. The core replica imposes a total order on these method invocations, forwards them to all other replicas, and also carries out the invocation itself. The core replica sends the reply of the invocation to the invoking client. The protocol is illustrated in Figure 4.15.

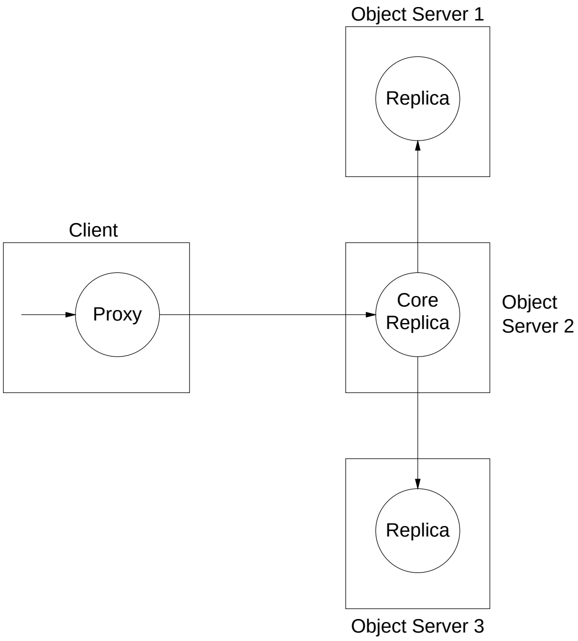


Figure 4.15: The basic replication protocol for revision DSOs. Squares represent processes, circles local representatives, and arrows (marshalled) method invocations. Replies are not shown.

Discussion

I arrived at an active replication protocol by a process of elimination. We can eliminate protocols from the set of replication protocols currently used in Pierre et al.'s adaptive replication scheme by looking at which of the three evaluation criteria are important for revision DSOs. For revision DSOs, all three evaluation criteria are important, as well as scalability. Download time and consistency are important for user satisfaction, as argued in Sec. 9 and Sec. 4.3.4, respectively. Low wide-area traffic is important for efficient distribution, as discussed in Sec. 4.1.

The requirements for strong consistency and scalability reduces the set of candidate protocols to four: (1) client/server (i.e., no replication), (2) master/slave replication with a check on each read to see if the slave is still consistent with the master, (3) master/slave with a master invalidating a slave after updates, and (4) active replication. The first two protocols I dismiss because they do not scale to large numbers of readers. This leaves active replication and master/slave with invalidation.

Active replication was chosen over master/slave with invalidation because, in most cases, the former uses less bandwidth when updates are made to a revision DSO. In master/slave with invalidation, slaves that are asked to perform a read method after an update always have to retrieve the new state from the master. This procedure uses more bandwidth than active replication if two conditions hold:

1. After an update to the state, replicas are asked to execute a read-method invocation at least once.

2. The (marshalled) state of the object is large compared to the updates on the object (i.e., the marshalled write-method invocations).

The latter condition holds for revision DSOs, which generally consist of multiple files and whose updates are small because of the block-based interface. Even if we count an upload of a whole file as an update, the state of the DSO is larger in most cases, as it is likely to already contain one or more files. Assuming that the former condition also holds for replicas of revision DSOs, active replication is therefore to be preferred over an invalidation-based protocol. The rationale for this choice was taken from Bal et al. [1992] who describe a number criteria for selecting replication protocols for large-grain parallel applications, which apply well to distributed shared objects.

The risk of the core replica becoming a bottleneck is low. A revision DSO is used to publish only a particular revision of a software package. As a result, the number of clients simultaneously uploading files into the DSO is low, and thus the amount of work for the core replica is small.

The GDN does not make use of caching in the implementation of revision DSOs. Recall that in this dissertation the term caching exclusively denotes the caching of replies of method invocations, which can be done at both proxies (caching for the local client) and at replicas (caching for a collection of remote proxy local representatives).

Caching in proxies is useful only when a client downloads the same file or asks the object to list its contents at least twice. In our initial implementation of the GDN we have an HTTP-to-Globe gateway that allows a user to access distributed shared objects via a standard Web browser. Because it serves multiple users, caching of replies in the proxy local representative in the HTTP-to-Globe gateway may be beneficial. However, replacing the proxy with a (temporary) replica local representative may achieve the same network-traffic reducing gains. Furthermore, using a replica instead of a caching proxy reduces the complexity of the replication protocol (i.e., the protocol does not have to keep track of the consistency of cached replies).

Caching of replies at a *replica* is useful only when storing the reply in the replication subobject is cheaper than regenerating the reply by reexecuting the method on the semantics subobject. For a revision DSO storing the reply is more expensive: keeping blocks of downloaded files in core adds to the local representatives' memory footprint, and keeping the cached replies on disk will make caching as expensive as redoing the method invocation.

4.6.2. Mid– to Long-Term Network Optimization

The complexity of applying replication to achieve optimal network usage for a software distribution network primarily lies in how to determine where the replicas should be placed given the fact that network usage patterns are dynamic and are caused by many different Internet applications sharing the network. This section concentrates on how a software distribution network can optimize its own network traffic and performance, in particular, it focuses on replica placement in the mid- to long-term. Short-term placement (i.e., how to deal with flash crowds) is discussed in Sec. 4.6.4. The policy presented here should be considered a feasible, but not necessarily the best, solution.

Replica-Placement Policy

To explain replica placement the following terminology is introduced. The *Autonomous System (AS)* [Bates et al., 1995] of a machine connected to the Internet indicates the routing domain a machine belongs to. A routing domain is defined as a group of networks having a single routing policy. For example, all universities in the Netherlands are grouped together in a single Autonomous System. Autonomous Systems are identified by a small integer, called the *Autonomous System Number (ASN)*. The ASN for an IP address can be obtained by doing a query on

RIPE's WHOIS database [Réseaux IP Européens, 2001]. Pierre et al. [2000] define *wide-area traffic* as any traffic from a host in one AS to a host in another AS (i.e., all inter-AS traffic). This definition is based on the observation that most ASes contain a large number of routers and computers. The current [May 2001] estimate of the number of ASes making up the current 109 million host Internet [Internet Software Consortium, 2001] is 8000, resulting in an average of 13,000 hosts per Autonomous System.

Looking at this definition we can make two interesting observations:

- If clients have a replica in their Autonomous System, they do not generate any wide-area traffic, by definition,

- Under the assumption that intra-AS bandwidth is plentiful, a client having a replica in its Autonomous System has an optimal download time.

Observe that following the above definition of wide-area traffic, all network-load problems are solved by creating 8000 replicas, one in each of the 8000 ASes of the Internet. As this is currently not feasible, I define the following policy for the placement of replicas of revision DSOs:

1. Initially, a DSO consists of a single core replica. How the location of this core replica is chosen is discussed in Chap. 5.

2. The DSO creates additional replicas in the Autonomous Systems that have the most downloads (i.e., generate the most reads).

3. The DSO adjusts the placement of replicas from time to time. The number of replicas a DSO O is allowed to have is determined by how many read-method invocations the object received in the last measurement period, relative to the total number of reads on the GDN in that period, and denoted by the following formula:

$$\textit{\#replicas}(O, T_{n+1}) = \frac{\textit{\#reads}(O, T_n) \cdot \textit{total \#object servers in GDN}(T_n)}{\textit{total \#reads on GDN}(T_n)}$$

   where *#reads* denotes the number of read operations (i.e., invocations of methods which do not modify the state) on the object in the period T_n, *total # reads on GDN* denotes the estimated total number of reads on the GDN in that period, and *total #object servers in GDN* denotes the estimated number of object servers in the GDN in the period T_n.

For the remainder of this dissertation the total number of reads on the GDN divided by the number of object servers in the GDN is referred to as the *ObjectServerClientRatio (OCR)*. The above formula can therefore be written more concisely as:

$$\#replicas(O, T_{n+1}) = \#reads(O, T_n) \cdot OCR$$

4. The length of the measurement period, called the *scenario–re-evaluation interval* is the same for all revision DSOs, but they do not necessarily readjust the placement of their replicas at the same point in time.

5. To prevent a popular package from creating too many replicas, a maximum is imposed on the number of replicas. This maximum is not an absolute number, but is relative to the OCR parameter.

Implementing the Policy

The replica-placement policy is implemented as follows. Each proxy local representative includes its Autonomous System Number in the marshalled method invocations it sends out. The proxy's ASN can be retrieved from the Globe runtime system (RTS), which it obtained by doing a WHOIS query at the time the RTS software was installed on the client host. Each replica of the revision DSO collects a log of the clients that invoke its read methods, in particular, it records the client's Autonomous System Number. These logs are summarized and periodically (e.g. daily) sent to the core replica. These summaries simply list the number of read-method invocations the replica received from each AS. The summary message includes the replica's own Autonomous System Number. Summaries are stored on disk by the core replica's replication subobject.

The core replica, at fixed intervals, does a calculation based on these summaries to determine which replication scenario is optimal for this revision DSO. More precisely, the core replica determines on how many and which hosts it should place replicas. This calculation is done, for example, once a week (i.e., the scenario re-evaluation interval is a week). The output of the calculation is a list of the Autonomous Systems that generated the most reads in the previous period and the number of replicas to create in each AS.

The total number of replicas in the list (i.e., the number of replicas the DSO is allowed to have) is determined by the formula described above. Both the total number of reads on the GDN and the number of object servers in the GDN and thus the OCR parameter are estimated and adjusted periodically. An object always obtains the latest value of the OCR parameter, which is considered a global tuning parameter of the replica-placement policy, before re-adjusting its replication scenario.

When this list differs from that of the previous period, the core replica adjusts the revision DSO's replication scenario in five steps.

1. The core replica determines which ASes do not yet host sufficient replicas.

2. The core replica maps these new entries in the list to geographical locations. The geographical location of an Autonomous Systems can either be obtained from the WHOIS database directly, or from a service such as NetGeo [Moore et al., 2000; CAIDA, 2001] that builds on the WHOIS database and can map an ASN to a latitude and longitude pair.

3. The geographic locations of the Autonomous Systems are used to find appropriate object servers by doing lookups on the Globe Infrastructure Directory Service (GIDS), as discussed in Sec. 3.8.

4. The core replica of the revision DSO asks these object servers to create a replica of the DSO.

5. Existing replicas no longer on the list are asked to delete themselves.

Discussion

This replica-placement policy optimizes network usage and performance for the Globe Distribution Network, as it reduces wide-area traffic and provides optimal download time for a large number of a DSO's clients. Network usage and performance are further optimized by using the Globe Location Service, which, for this dissertation, is assumed to connect clients that do not have a replica in their Autonomous System to the nearest Autonomous System that does.[5] The advantage of this policy is therefore that it does not explicitly require information about the topology of the network. In particular, no knowledge is required about the bandwidth of the links connecting two Autonomous Systems. Topological information is used implicitly by using the Globe Location Service. The disadvantage is that clients may wind up using a bad link.

This policy does not take into account the cost of installing replicas and updates. Revision DSOs will be read more than written—otherwise someone is rapidly publishing software that no one is interested in—and we can therefore use replication without worrying too much about the overhead of maintaining consistency. Given this high read/write ratio, the consistency overhead is therefore

[5] The GLS currently returns the contact addresses of replicas which are geographically nearest, which does not necessarily correspond to the nearest replica in terms of network distance.

largely determined by the number of replicas that need to be kept consistent. The cost of installing a new replica can also be ignored here, as a DSO's replication scenario is updated only periodically (e.g. once a week). The cost of creating a new replica are primarily determined by the size of the state of the object that must be transferred to the new server.

The OCR parameter plays an important role in the policy. It governs the macroscopic behavior of the Globe Distribution Network via the microscopic behavior of its many revision DSOs. At first glance, it may appear that the effectiveness of this policy depends on the value of the OCR parameter being accurate, which would require the estimates of the total number of object servers and the total number of reads on the GDN to be precise. Fortunately, however, a wide margin of error is permitted in these estimates. More precisely, the problems introduced by under- or overestimation of the OCR parameter are handled by the server load balancing and flash-crowd control mechanisms of the GDN, which we discuss in detail in Sec. 4.6.3 and Sec. 4.6.4, respectively. Here we briefly describe how these two mechanisms neutralize the effects of under- and overestimation. In the following, it is assumed that there are enough object servers available to handle the actual load and that only the OCR estimate is wrong.

OCR estimated too low As a result, a revision DSO creates too few replicas compared to the actual number of reads it receives. This under allocation might cause object servers to become overloaded.

There are, however, two factors that prevent overloading from occurring. First, the GDN evenly distributes the load over the available servers. In particular, when the core replica maps the list of ASes to a concrete set of object servers using the GIDS, it uses the load of the object server as a selection criterion, thus avoiding heavily loaded servers.

Second, when a server does become overloaded due to a sharp increase in accesses on the replicas it hosts, the revision DSOs' mechanism for handling flash crowds is activated and the replicas start reducing the workload by creating new replicas on other object servers. Given the assumption there are, in fact, enough object servers available, the overload is quickly removed.

OCR estimated too high When the OCR is set too high, a DSO creates more replicas than is necessary for serving its actual number of clients. This does not create a problem as the costs for creating and maintaining a replica of a revision DSO are relatively low. The cost of a replica is dominated by the amount of persistent storage it requires and the bandwidth required to transfer the state to the new object servers. Memory footprint is small as the state of the DSO is kept on disk. The costs in terms of CPU usage are also low in this case, because there are more replicas than required, which causes the load on the revision DSOs to be more evenly distributed over the set of object servers.

The risk of an estimate that is too high is therefore waste of secondary storage and networking resources. The GDN has two mechanisms to prevent excessive overallocation of these resources. First, there is the maximum that the policy imposes on the number of replicas a DSO may create. This maximum, which is relative to the OCR parameter, prevents any DSO to replicate to more than, for example, 20% of the available object servers. Therefore, if the OCR estimate is too high, the maximum prevents a DSO with only a limited number of actual clients from creating excessively many replicas.

Second, the server load balancing mechanism allows object servers to impose a limit on the number of replicas it is willing to host. When this limit, called the object server's *LR limit*, is reached and the object server is asked to create another replica by a core replica of a DSO, the object server refuses. To keep DSOs from continuously trying to create replicas on "full" object servers (i.e., to prevent thrashing behavior), when a core replica of a DSO has received a number of refusals from an object server it refrains from asking other object servers.

There is another problem associated with estimating the OCR parameter too high. The problem is that revision DSOs that re-evaluate their replication scenario soon after the OCR parameter has been set too high will allocate too much object-server capacity. As a result, there may be a shortage of capacity for the DSOs that re-evaluate their scenarios later.

Consider the following micro example. In this example, there are two DSOs (*A* and *B*) each receiving a fair number of accesses, a DSO *C* with a large number of accesses, and the number of available object servers is too low to accommodate all accesses. We assume the OCR parameter has not yet stabilized and has just been set to a value which is too high, and that the two moderately popular DSOs *A* and *B* calculate their optimal replication scenario before highly popular DSO *C*.

Because of the large OCR value, *A* and *B* create more replicas than they should given their load, resulting in a situation where the two DSOs take up a significant part of the available object servers. We assume the creation of the replicas of *A* and *B* causes these servers to reach their limits. As a result, later, when *C* starts to re-evaluate its replication scenario, it encounters overloaded servers, causing it to forcibly limit its replication degree. The popular DSO *C* now has too few replicas to serve its clients, resulting in increased wide-area bandwidth consumption and increased download times for these clients.

In other words, the problem is that the interval between the introduction of the high OCR parameter and the time when a DSO does its replication-scenario calculation determines how fair a share of the available object servers a DSO can use. It is not clear whether this problem occurs in practice, and further investigation into this issue is necessary. It is clear, however, from all the issues described above that an application such as the Globe Distribution Network would benefit from a global resource management system, as already indicated in Chap. 2.

In the above policy, Autonomous Systems are mapped to a geographical location using the WHOIS database or a service such as CAIDA's NetGeo which builds on the WHOIS database. There does not seem to be a need to offload this task to a special service in the Globe Distribution Network to reduce the load on WHOIS and/or NetGeo servers. If there are 50.000 software packages with with an average number of 25 revisions, these packages will occupy 1.250.000 distributed shared objects. For a scenario–re-evaluation interval of one week and an even distribution of scenario re-evaluations over this interval, in theory, 2 revision DSOs are doing a re-evaluation each second. This load can easily be handled by existing services.

Taking Into Account Network Conditions

To use replication to achieve efficient distribution, we need the ability to determine current and potential hot spots in the underlying network. We just discussed how the Globe Distribution Network can optimize its own usage of the Internet. Further optimization will be possible if the GDN takes into account actual network conditions. With knowledge about which links are more-or-less permanently overloaded, revision DSOs can adjust their replication scenario to reduce any load they might create on those links. Information about current networking conditions can be obtain from services such as the Network Weather Service [Wolski et al., 1999]. As incorporating network conditions is a large research topic in itself this issue will not be addressed further in this dissertation.

4.6.3. Server Load and Replication

This section presents the details of the Globe Distribution Network's server load balancing scheme. Server load balancing in the GDN comes after network load balancing, that is, network load balancing dictates in which region replicas should be placed and server load balancing determines on which object server(s) in that region the replicas should be hosted. Server load balancing has two aspects: balancing the revision DSOs over the available servers and balancing the clients over the available replicas. These aspects are discussed in turn.

Balancing Replicas over Regional Object Servers

The basic idea of our load-balancing scheme is that the load of the server is taken into account by the core replica when readjusting a revision DSO's replication scenario. In particular, object-server load is taken into account when mapping the list of Autonomous Systems with the most clients to a concrete set of object servers. As explained above, the identifiers (ASNs) of the Autonomous Systems with the most clients are first mapped to a geographical location and then to concrete object servers using the Globe Infrastructure Directory Service (GIDS).

The mapping procedure is as follows. To enable load balancing, object servers register whether they are overloaded or not as one of their properties in the GIDS. The core replica doing the mapping selects the servers that are not overloaded. When an underloaded object server cannot be found in the desired geographical location, the core replica attempts to find one in the general vicinity. By moving up in the hierarchy of GIDS regions (see Sec. 3.8), the core replica can discover the sibling regions of the geographical location it is interested in and look for servers there.

As already mentioned in Sec. 4.6.2, if this procedure fails to return a usable object server for a certain percentage of the ASes in the list, the core replica refrains from asking other object servers. The reason for stopping the search is to prevent DSOs from endlessly trying to find underloaded object servers when there are none available, for example, because the OCR parameter was set too high in relation to the actual number object servers available, and many objects allocated more replicas than they should.

The object servers in an AS may not be able to handle all clients in the servers' and neighboring ASes that are interested in the revision DSO . This situation is handled by the flash-crowd control mechanism (as discussed in Sec. 4.6.4). Basically, each replica monitors client traffic and when this suddenly increases tries to find object servers in its clients' ASes and asks them to also create a replica. The client-over-server load balancing scheme, discussed below, then takes care that the clients in the AS are balanced over the new and existing replicas.

An issue not yet addressed is how an object server determines whether it is overloaded or not. I propose to extend the Globe run-time system with a method called rts::overloaded that returns a boolean value indicating whether the object server as a whole is overloaded. The method calculates the server's load using conventional methods, such as looking at relative CPU and memory utilization, and also takes into account limits imposed by its owner, for example, the *LR limit* introduced in Sec. 4.6.2 that simply limits the number of local representatives on the server, independent of their resource consumption. The object server periodically invokes this method and registers the return value in the GIDS as one of the server's properties.

Discussion A possible improvement over the proposed rts::overloaded method with a boolean return value is to have it load values in a standardized metric, which will make it easier for the GIDS to compare object servers.

An alternative to registering load information in the GIDS is to have an object server indicate it is overloaded to the core replica at a later stage. As described in Sec. 3.8, the GIDS is used only to make an initial selection from the available object servers. Whether or not object and server are willing to cooperate is determined in the negotiation phase that follows. This phase provides a natural opportunity to let the object server indicate whether or not it is willing to create a new replica. However, if we assume that the server decides autonomously whether or not it is willing to create another replica, and does not take into account who is making the request, indicating server load in this way may be too expensive. If no real negotiation between client and server is necessary, this alternative only unnecessarily increases the time it takes to find a suitable object server.

Balancing Clients over Regional Replicas

As described in Chap. 3, clients find a nearby replica of the revision DSO they are interested in via the Globe Location Service (GLS). The Globe Location Service as such helps to achieve network-load balancing, because it returns the contact addresses of nearby replicas and thus directs clients to replicas that exist in their Autonomous System or near to it. When there are two or more replicas in a region, the GLS, however, does not do load balancing over these replicas. The reason is that different objects may have different load balancing policies and load balancing is therefore not part of the GLS.

I propose to spread the clients in a region over the available replicas as follows. The idea is to introduce an explicit negotiation phase at bind time (i.e., when a client application tries to install a local representative of a revision DSO in its address space). To enable bind-time negotiation, the contact addresses of a DSO are extended with a reference to an implementation of a particular *binding-control protocol*. When binding starts, this protocol implementation is loaded and used by the run-time system to check whether the replica referred to in the contact address is willing to accept new clients, before actual binding begins. If so, binding continues as usual. If the replica is heavily loaded and refuses, the RTS tries other replicas. The idea of a negotiation phase before the actual binding is not new, Homburg [2001] introduced this phase and refers to it as the destination-selection phase. In addition to refusing new clients a replica can also proactively control its load by telling existing clients (i.e., the proxy local representatives they use) to rebind to another replica.

This mechanism allows replicas to make their own decisions about whether to accept a new client or not. For example, each replica could check its current resource consumption relative to the total capacity of the object server and accept clients only when this will not cause it to use more than its fair share of the resources. In principle, this mechanism can even be used to allow different objects to use different policies. However, as an interim solution I propose to have replicas look just at the overall load of the object server as indicated by the value of the rts::overloaded method.

Replicas can also tell existing clients to rebind to a replica on another server. This facility is necessary to handle clients that have long-lasting bindings to a replica. These clients can keep the load of the server above an acceptable level by doing many method invocations even after the replicas have started refusing new clients.

Discussion For the GDN the Globe Location Service is used as the service for locating nearby replicas. It is possible to use other location services, for example, some of those found in (hierarchical) Web caching systems. An example of this class of location services is the one used in the CRISP Web caching system [Gadde et al., 1998; Rabinovich et al., 1998]. In CRISP, Web caches can be organized into groups based on proximity to each other. Each group of nearby Web caches has its own location server that keeps track of the group's contents (i.e., which Web pages they cache). Clients are assumed to contact a nearby cache server, which, if it does not cache the page itself, contacts the local location server to see if the content is cached in the vicinity. If not, the origin Web server is contacted directly. This location service is based on the observation that retrieving content from far-away caches may be slower and more expensive than contacting the origin server. This observation allows the location servers to work autonomously, resulting in the decentralized architecture.

A issue that needs to be resolved when using location services from Web caching systems in the GDN is that of locating the origin server of a distributed shared object, when a nearby replica cannot be found. For Web pages, the URL directly identifies the origin server, that is, there is a one-to-one mapping from object identifier to origin server. To use these services in GDN, there should be a fast and scalable way of locating the origin server of a distributed shared object, which is the object server hosting the core replica of the object. Whether or not there is a scalable way of locating a DSO's origin server depends largely on the mobility of core replicas. The mobility of core replicas determines the number of changes to the mapping from object identifier to origin server, which, in turn, determines whether a one-to-one mapping scheme such as in the Web is possible and, for other schemes, whether the mapping is efficiently cachable and replicable.

The level of mobility of core replicas in the GDN is unlikely to be high. Core replicas are expected to be moved only for management reasons (e.g. when the owner of the host decides it no longer wants it there), and not, for example, to bring the content closer to the downloaders. Core replicas may be moved to a host with more capacity, although, in general, capacity will be increased by creating more replicas (as explained in Sec. 5.4.3). In addition, some forms of mobility can be handled by using DNS (i.e., by using DNS names instead of IP addresses a class of migrations can be supported without changes to the "DSO to core-replica host" mapping).

If mobility of core replicas were an issue the GDN would have to use a location service that supports mobile objects. The Globe Location Service already includes support for handling mobile objects (rather, it has been specifically designed with mobile objects in mind). Other location services that support replicated and mobile objects are described by Pitoura and Samaras [2001]. The GLS was chosen for the GDN as the purpose of this dissertation was to test the Globe middleware and its services.

The rts::overloaded method can be used as a poor man's approach to local resource management, by making the thresholds it uses to calculate the load configurable by the object-server owner. By allowing an object-server owner to, for example, set the overload threshold for CPU utilization to 50% he can prevent an object server from accepting new clients when the CPU is still only half utilized. An extension of this load-control scheme is to introduce client priorities. Client priorities can, for example, be used to increase the number of clients that is served by a replica. By assigning a certain class of clients (e.g. faraway clients) a lower priority (resulting in a lower transfer rate for those clients), the replica can keep its load at the same level but serve more clients simultaneously. This is an improvement over the scheme without client priorities where the extra clients would have been refused because the replica's server was full. Note that actually refusing accesses from faraway clients to benefit nearby clients is an incorrect policy, because it implies that no accesses by remote clients will be recorded and, as a result, no replicas will be created in these remote regions (see Sec. 4.6.2).

4.6.4. Handling Flash Crowds

The term *flash crowd* is used to denote a large group of people all trying to visit the same Web site or download the same item at the same time [Nielsen, 1995]. Due to such a large influx of traffic, servers, their network connections, and network links in the path from clients to servers may become overloaded.

I adopt Pierre et al. [2001]'s solution for handling flash crowds. Each replica monitors the rate at which it is accessed. If this rate suddenly increases, the replica autonomously finds other object servers in the ASes the traffic is predominantly coming from, using the same mechanisms as used by the core replica when re-evaluating the overall replication scenario. Each new object server then creates a new replica, which is registered in the Globe Location Service. By virtue of the binding-control protocol just introduced, a significant part of the new clients are then directed to the new replicas and no longer to the original replica. The original replica, in addition to creating new replicas, also signals the sudden influx to the core replica, allowing it to do more global optimization for dealing with the problem. To prevent uncontrolled allocation (i.e., not subject to the limits of the replica-placement policy), the number of additional replicas an overloaded replica can create autonomously is limited.

Discussion

A limit is imposed on the number of additional replicas a replica can create on its own authority to prevent unlimited growth of the number of replicas and interference with the replica-placement policy which attempts to achieve a fair allocation of the available object servers to the revision DSOs.

4.6.5. Alternative Replication Protocols and Policies

As argued in Sec. 4.6.1, there are no alternatives for the active replication protocol chosen. Our requirements with respect to consistency and scalability rule out a number of protocols and the characteristics of revision DSOs in terms of size and access patterns shrink the selection to one. Note that the choice for an active replication protocol and the replication policy is made on the assumption that server space is cheaper than wide-area bandwidth. Therefore, if this situation should change in the future, both policy and protocol will have to be chosen anew.

There are clearly many variations possible on the replica-placement policy. Unfortunately, there was not enough time to investigate alternative policies or experimentally validate the outlined policy, as the focus of this dissertation was on creating a complete design of the GDN and significant time was spent investigating, in particular, the security aspects of the GDN. Hence, the policy presented here should be considered one possible solution that highlights the issues that a replication protocol for a revision DSO in the Globe Distribution Network should address.

4.6.6. Initial Implementation

The initial implementation of the Globe Distribution Network uses a master/slave protocol with invalidation and does not support network or server load balancing. Replication of revision DSOs is therefore controlled by the owner of the object (a person) at this point in time, and not by the object itself.

Chapter 5

Security

In Chap. 2 we identified the major security requirements for a software distribution network. In this chapter we look at how the Globe Distribution Network meets these requirements. Four (classes of) security requirements were identified:

1. Ensuring authenticity and integrity of the distributed software
2. Preventing illegal distribution of copyrighted or illicit material
3. Anonymity of up- and downloads
4. High availability and protecting the integrity of the machines running the distribution network

We start our discussion in Sec. 5.1 and Sec. 5.2 with how the GDN handles illegal distribution, as this requirement affects the security design of the GDN most. Sec. 5.2 also discusses GDN's support for anonymity. Sec. 5.3 discusses how users can determine the authenticity and integrity of software distributed via the GDN. Sec. 5.4 describes the GDN's measures for ensuring availability of the distribution network and its hosts. Finally, in Sec. 5.5 the initial implementation of this security design is described. The security design for the Globe Distribution Network has been published in [Bakker et al., 2001b].

5.1. PREVENTING ILLEGAL DISTRIBUTION

A free software distribution network allows authors to make their free software available to a large audience. This ability to share information with a lot of people makes it an interesting target for people who want to illegally distribute copyrighted or illicit content. In this section we look at ways to prevent this type of abuse and how they can be used in the Globe Distribution Network.

We must take action to prevent the illegal distribution of copyrighted works or illicit content via the GDN so that the persons or organizations participating in the GDN (e.g., the owners of object servers storing and providing the GDN content) do not run the risk of being prosecuted for copyright infringement or the distribution of illicit material. In some countries, in particular in the United States, the computer owner himself is liable for copyright infringement if copyrighted content is served from his computer even if the owner did not place it there [United States Government, 1998]. The same risk of liability exists for pornography and other illicit materials.

Before looking at illegal distribution we first briefly examine the legal basis of the free distribution of software. Software is generally considered a literary work and hence protected by copyright [World Intellectual Property Organization, 1996].[1] Allowing free redistribution of software therefore requires the legal consent of the copyright holder, generally the author. For software, but also for other types of works, standard licenses, such as the GNU General Public License (GPL) [Free Software Foundation, Inc., 1991] have emerged that, amongst other things, permit free distribution by anyone who wishes to do so. For the purpose of this dissertation, *software* is defined as not just source or binary code but also includes the associated documentation, example files (images, audio, video), etc.

The above suggests that we could define legal distribution of software as the distribution of software that has been made freely redistributable by its copyright holder. This definition is, however, incorrect as not all types of software can be legally owned and distributed. In Chap. 2 we distinguished a class of software called *controversial free software*. Controversial free software was defined as software that

1. Uses patented technology,
2. Can be used to circumvent copyright-protection measures,
3. Employs strong cryptography, or
4. Contains racist or potentially offensive material (e.g. Nazi symbols, nudity).

Whether or not instances of this class of software can be legally owned and distributed differs per country. For the first type of controversial free software it depends on the geographical scope of the software patents whether or not the software is illicit in a particular country. The legality of the second type depends on the current copyright laws of the country in question. Software whose sole purpose is the circumvention of copyright protection is illegal in most countries, but

[1] For an interesting discussion about whether software should be protected under copyright law, patent law, or free speech legislation, see [Burk, 2001].

for software that has legitimate purposes as well (“dual use software”), different countries have different regulations. In many countries the use and ownership of software that uses strong cryptography is legal, and distribution is allowed except to certain nation states. Furthermore, the distribution of strong cryptographic software generally does not require the original publisher to follow special procedures. A notable exception is the United States [U.S. Department of Commerce, 2001]. Finally, for the fourth type of controversial free software it also depends on local legislation whether this software can be owned and distributed. In other words, what constitutes legal distribution and therefore what is illegal distribution differs per country, and a mechanism for preventing illegal distribution via a worldwide software distribution network should take this fact into account, which complicates matters considerably.

In addition, in some countries governments are imposing restrictions on what content may be *accessible* from within their country. A famous example is the case of a French court requiring the American company Yahoo to make their auction Web site, on which Nazi memorabilia were being sold, inaccessible to French citizens [Zacks, 2001]. If such regulations were extended to certain types of software, a worldwide distribution network might have to take these into account. The risk for the distribution network is that if it does comply with a country’s restrictions on remotely accessible content it may be forbidden to operate in that country, even if none of the disputed content is handled by the distribution network within the borders of the country. This particular issue will not be addressed in this dissertation.

In the following section we discuss the various solutions for preventing illegal distribution via a software distribution network, and how per-country differences can be taken into account with these solutions. Per-country differences will only be looked at briefly in this dissertation.

5.1.1. Content Moderation

The basic model of a software distribution network (SDN) is that one group of people uploads software into the SDN and another group downloads that software from the SDN. The most obvious solution to preventing illegal distribution is therefore to check content before it is uploaded. We call this solution *content moderation*. In content moderation, one or more people, called *moderators*, manually check all content before it is allowed onto the SDN. Manual checking is required because a computer cannot tell copyrighted from noncopyrighted content nor illicit from legal content.

For this solution to work the moderators have to do their job properly, making neither intentional nor accidental errors, otherwise illegal content will get onto the SDN. This requirement also represents the first disadvantage of content moderation. Manual checks are error prone and, furthermore, may be defeated by cleverly encoding illicit content into inconspicuous content. As a result, content moderation cannot always prevent illegal content from getting onto the network and therefore does not provide 100% protection against liability for the parties participating in the SDN.

The amount of work that has to be done by the group of moderators is determined by two factors:

1. The amount of new content published

2. The amount of checking that has to be done on a piece of content.

Moderators should always check whether the content is actually free software. What additional checks need to be done depends on the rules about what software constitutes legal software. Doing these checks is tedious and time consuming, in particular, the checks that require manual inspection of the source code, such as checking whether or not a software package infringes on a patent, uses strong encryption or can be used to circumvent copyright measures.

The second disadvantage of content moderation is that, when abuse is low, most of this work is done in vain. As a result, if there is little abuse, we expect moderators to perceive the moderation work as superfluous, become demotivated, and no longer volunteer their time for moderating the software distribution network's content.

The fact that the amount of work required is proportional to the amount of content published might imply that this solution is limited in scale. To accommodate a large volume of new content, a large group of moderators is necessary, assuming moderators have only a limited moderation capacity. The latter assumption can be safely made in light of the fact that we are talking about a distribution network for free software, making content moderation a voluntary job. Having a large number of moderators makes it harder to fulfill the requirement that moderators are trustworthy. Its limited scalability is considered the third disadvantage of content moderation.

The fourth disadvantage is that content moderation introduces a long delay between the initial submission for publication and the actual publication in the distribution network, especially if the moderator has to check many patents, court rulings, etc. We expect that software maintainers wanting to publish via the SDN will find this delay irritating.

The advantage of content moderation is that, in principle, it does catch all obvious attempts at illegal distribution, such as attempting to upload MP3-encoded Top 100 songs. Table 5.1 (page 116) summarizes the advantages and disadvantages of content moderation.

An example where content moderation is being used to prevent illegal distribution is ibiblio.org's free-software archive. Ibiblio's Metalab Archive (formerly sunsite.unc.edu) is the primary archive for free software that runs on Linux and is heavily mirrored by many sites around the world. Uploads to the archive are checked for obvious attempts at illegal distribution [Ibiblio, 2001].

Content Moderation and Per-Country Differences

There are basically three ways of using content moderation such that a software distribution network (SDN) adheres to local regulations.

1. The software distribution network is partitioned into countries/jurisdictions and any content to be stored and distributed in a country/jurisdiction via the SDN is first moderated by the moderators for that country/jurisdiction. To be able to enforce export regulations, each partition must be accessible only to the people of its jurisdiction.

2. There is one group of moderators for the whole worldwide SDN that accepts only software that is legal everywhere. In other words, out of all software legislation the lowest common denominator is established and only software that meets this standard is accepted.

3. There is a group of moderators for the whole worldwide SDN that label software in such a way that each country can configure its part of the SDN to accept only the content that is legal there using the information on these labels. An example label is "uses 128-bit cryptography". Labels such as "cannot leave United States" could be used to indicate export regulations.

5.1.2. Cease and Desist

Content moderation at upload time is the only solution that can prevent illegal content from getting onto a network. It is therefore the only solution that can catch all illegal distribution, with the restriction of the human ability to identify the content. If, however, some (temporary) illegal distribution is legally acceptable there are alternative solutions to *a priori* content moderation.

One such solution is the *cease and desist* scheme of my design [Bakker et al., 2001b]. In the cease-and-desist scheme, users of the software distribution network can freely publish content, but the content a user publishes remains traceable to that user. If content is suspected of having been published illegally, its presence in the distribution network is reported to a group of moderators. This group of moderators checks whether or not the content was published illegally, and if so, blocks the publishing user's access to the distribution network and has his publications removed. In other words, the cease-and-desist scheme tries to limit illegal distribution by banning (provably) malicious users from the distribution network. Abuse is proven by making uploads nonrepudiatable.

Intuitively, the software distribution network becomes similar to a world-writable directory on a UNIX operating system: everybody can place files in the directory but the files always remain traceable to the user that put them there because of the associated ownership information.

This scheme for handling the problem of illegal distribution of copyrighted or illicit content is in line with current legal developments. For example, in the United States legislators have recognized that it is often not feasible to moderate content beforehand. Hence, in the recent changes to copyright law "provider[s] of online services," such as Internet Service Providers can request legal protection from copyright infringements by their users [United States Government, 1998]. If a user publishes other people's copyrighted works on the ISP's servers, the ISP cannot be held liable and is required only to remove the copyrighted content once they have been notified by the copyright holders. France has similar legislation [Oram, 2001].

The cease-and-desist scheme does not prescribe who should report illegal distribution, as this depends on legislation. In the above example, making moderators aware of copyrighted works in the distribution network is the legal responsibility of the copyright holders. Likewise, it could be the legal responsibility of law enforcement agencies to report illicit content. In other cases it may be the joint responsibility of the software distribution network's users.

The amount of work for the moderators in the cease-and-desist scheme depends on two factors:

- The number of reports of illegal content
- The amount of checking that needs to be done on the allegedly illegal content.

As the amount of work depends on the number of reports, ideally, it is proportional to the level of abuse. If there are few users doing unlawful publication there will be few reports. If there are many abusers the number of reports will be high. As a result, the work of the moderators is always useful and will not be perceived as superfluous.

To achieve this ideal situation, however, there should be few false reports. False reports would be submitted by malicious people trying to undermine the protection scheme. Their number can be kept at a minimum only if there is a threshold for an accuser to make an allegation. The most effective threshold is one where the accuser stands to lose something when the allegation is false. The threshold I chose is to require that allegations are made by a user who will be blocked from the software distribution network himself if the allegation proves false (i.e, he will lose his right to publish on the SDN).

For the cease-and-desist scheme to work efficiently, more is necessary than preventing false allegations. It should also be impossible or at least difficult for a violator to regain access to the software distribution network after he has been blocked. The following method was chosen to satisfy this requirement. Candidate users are required to prove their real-world identity, which is published on a black list when the user is found guilty of illegal distribution. This black list is checked at each application for access to the software distribution network, thus keeping violators reapplying for access out. Alternative ways of preventing false allegations and regaining access are discussed in Sec. 5.2.2.

Cease-and-Desist and Per-Country Differences

There are basically three ways in which cease-and-desist can be used to have a software distribution network (SDN) take into account local regulations:

1. There is a group of moderators for each country/jurisdiction. Access to the distribution network is given at country/jurisdiction granularity, and object-server owners allow content on their machines only from publishers with access to the local part of the distribution network. For publishers this approach implies that they must obtain access to the SDN for each country they wish to distribute their software in. The moderator groups of different countries/jurisdictions could cooperate to block publishers who violate widespread laws (e.g. illegally publish copyrighted works or child pornography). To be able to enforce export regulations, each partition must be accessible only to downloaders from its jurisdiction.

2. There is one group of moderators for the whole worldwide SDN that accepts only software that is legal everywhere. In other words, out of all software legislation the lowest common denominator is established and only software that meets this standard is accepted.

3. A publisher determines in which countries his content is legal. He labels the content with the names of the countries in which it is allowed. If he mislabels intentionally, or unintentionally (e.g. his software violates a software patent in France which he didn't know about) he is blocked in those parts of the network where his content violates the law. Allegation checking and blocking is done by either a global group of moderators or a group per country.

5.1.3. Reputation

The reputation of a software publisher can also be used as a method for preventing illegal distribution. We can distinguish two variations on this idea:

Variant 1: Per-Site Access Currently, people who want to setup a mirror of free software often select a collection of software packages or distributions based on the good reputation of the author/publisher and configure their servers to directly mirror the primary publication sites. This method is applied, for example, for the Linux kernel and for well-known distributions such as RedHat and FreeBSD.

Abuse is nearly impossible with this method. To have software published by other sites, a malicious publisher not only has to establish a good reputation, but he has to also create a large user base such that site owners start considering the software as a candidate for replication. As establishing a reputation and creating an audience is time consuming and hard, the risk of abuse and thus illegal distribution in this method is low.

The need to establish a reputation is also the disadvantage of this approach. Ideally, a software distribution network should give equal access to all publishers such that each publisher can immediately benefit from the SDN's resources and facilities when demand for his software grows. This approach also requires replica site owners to monitor which producers are popular and trustworthy. Otherwise not enough resources will be available for replicating the producers' software. This is considered a disadvantage.

Variant 2: List of Trusted Producers The disadvantage of each site owner having to monitor the reputation of producers could be overcome by introducing a group of reviewers who maintain a central list of trusted producers. In other words, a group of reviewers monitors producers' reputations on behalf of all site owners and place trustworthy producers on a trusted-producer list. Site owners configure their sites to accept only content published by the producers on this list. Although a simple solution, it still suffers from the problem that resources are not available to all publishers of free software. The latter problem cannot be solved by lowering the reputation threshold as that will make it easier for malicious publishers to get access.

Reputation and Per-Country Differences

For taking into account per-country differences in which software may and may not be legally distributed, the fact that site owners have fine-grained control over what content is replicated on their site is an advantage. Site owners can choose the producers making software that is compatible with the laws of their country. On the other hand, this can also be considered a disadvantage as not everybody with some spare server capacity wants to get into which software is legal and thus might not bother with making that capacity available for free-software distribution.

Introducing reviewers who establish a list of trusted producers can again be used to reduce the individual site owner's burden. There are basically two ways in which this could be set up.

1. There is a group of reviewers per country/jurisdiction, and site owners use the list of trusted producers as defined by their country/jurisdiction's reviewers.

2. There is single group of reviewers for the whole software distribution network. Only producers that publish software that adheres to the lowest common denominator of all software legislation are put on the trusted-producer list.

5.1.4. Other Approaches

Several efforts are underway to prevent users from accessing (e.g. viewing) illegally copied works (e.g. copyrighted music or videos) at the hardware level. The general idea is that the computer hardware does not allow copying or viewing of content unless authorized by the copyright holder [Stefik, 1997; 4C Entity LLC, 2000]. Ignoring the feasibility and desirability of such measures, for the Globe Distribution Network it could mean that preventing illegal distribution of copyrighted works is no longer the designer's concern as it is now handled by the hardware. However, preventing the distribution of offensive material and illegal free software still is a matter for the GDN, and therefore these approaches do not offer a complete solution.

5.1.5. Comparison

Table 5.1 compares content moderation (CM), cease-and-desist (C&D) and the second reputation-based scheme, focusing on a number of criteria. The criterion "application procedure for users" means that to get access to the distribution network users have to go through a non-trivial application procedure. The criterion "reporting procedure for illegal content" means that to report illegal content a non-trivial reporting procedure has to be followed. As indicated above, the amount of work for moderators in the C&D scheme is lower than for CM if abuse is low, and assuming there are few false reports. This property is the prime advantage of C&D, as a distribution network for free software will depend on volunteers to make it work, and should therefore keep the amount of work for these volunteers

Table 5.1: Comparison of the content moderation and cease-and-desist schemes.

Scheme	Content moderation:		Cease and desist:		Reputation[*]:	
Abuse level	Low	High	Low	High	Low	High
Amount of work for moderator / reviewer	large		small[◇]	large	medium[†]	large[‡]
Amount of work for end user	none		tiny[♯]	small[♯]	none	
Publication delay	high		none		none	
Application procedure for producers	no		yes		yes[♭]	
Reporting procedure for illegal content	no		yes		no	
Open to all	yes		yes		no	
Applicable only if some illegal distribution allowed	no		yes		yes	

[*] = The information about the reputation-based scheme is true under the assumption that producers that have been given access because of their good reputation never turn rogue. Otherwise we have to introduce measures to catch these producers which may require extra work from either reviewers or end users.

[◇] = Only when measures are in place to prevent false reports.

[†] = Reputation reviewers have to actively look for candidate producers eligible for access.

[‡] = When most candidate producers are not to be trusted, the reputation reviewers will have to invest a lot of effort to find eligible ones.

[♯] = In the cease-and-desist and reputation schemes, end users or third parties have to report illegal content. In the end-user case, as the population of end users is many times larger than the group of moderators the amount of work required from an end user will, on average, be rather small.

[♭] = The “application procedure” in the reputation scheme is nontrivial as this consists of establishing a reputation and a considerable user base.

as low as possible. In a situation where there is a lot of abuse CM may be more effective at preventing illegal distribution, because it is proactive. The fact that there is an intrinsic delay between submitting the software for publication and the time of publication in the CM scheme makes CM less suitable for use in a software-publication context, where rapid publication of new releases is sometimes crucial.

In my research for this dissertation I did not look at the reputation-based schemes. In retrospect, a reputation-based approach, in particular the second variant (i.e., having a group of reviewers decide which users get access to global resources), might work reasonably well. However, as shown in Sec. 5.2.3 the way cease-and-desist is incorporated in the Globe Distribution Network is rather generic, and also supports a reputation-based scheme for preventing illegal distribution. Furthermore, the CM and C&D schemes are not compared with respect to their ability to handle per-country differences, as this aspect is not fully investigated in this dissertation.

5.2. THE GDN AND ILLEGAL DISTRIBUTION

To explain how cease-and-desist is used in the GDN we first recapitulate briefly the general model of the GDN, illustrated in Figure 5.1. The GDN consists of a collection of object servers running replicas of distributed shared revision DSOs, containing revisions of software packages. Software producers upload archive files containing software into the GDN's revision DSOs which are subsequently requested from the objects by downloaders.

It is our design goal to allow many different people and organizations to run object servers and make them available to the GDN. These operators are referred to as *object-server owners*. In this section we discuss only how the fact that the GDN spans multiple administrative domains is taken into account in the cease-and-desist design. Object-server owners are therefore assumed to be trustworthy in this section. An analysis of the problems that untrustworthy and malicious object-server owners can cause, and solutions to a number of these problems are given in Sec. 5.4.2.

What the design currently does not fully take into account are the differences between countries with respect to which content may be legally distributed in that country. Moreover, the GDN also does not currently provide measures to prevent people in a country with strict laws from downloading illegal content from countries where this content is legal. These issues require further investigation. In the meantime, we define our own global policy of what can be distributed via the GDN. Given that the GDN is to be used for the distribution of free software, we define *inappropriate content* as anything that is not freely redistributable software

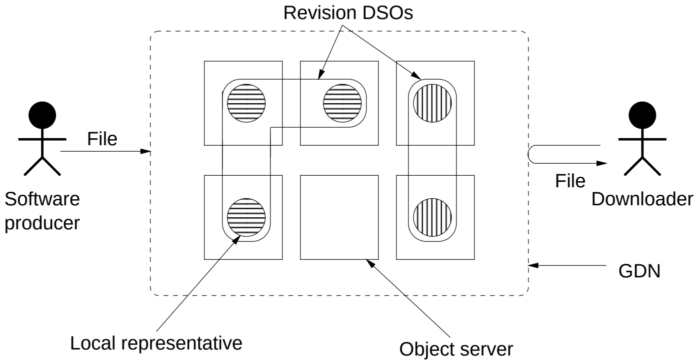


Figure 5.1: General model of the GDN. Stick men represent people, rectangles represent object servers, rounded boxes represent distributed shared revision DSOs, and circles represent local representatives of revision DSOs.

or part thereof.

5.2.1. Traceability via Digital Signatures

Content traceability is incorporated in the GDN as follows. When a software maintainer wants to start publishing his software through the GDN he has to contact one of the so-called *access-granting organizations*. An access-granting organization, or *AGO* for short, verifies the candidate's identity by checking his passport or other formal means of identification. In addition, the organization checks if this person has not been banned from the GDN by any of the other AGOs by consulting a central black list. If the candidate is clean, the access-granting organization creates a certificate linking the identity of the candidate to a candidate-supplied public key and digitally signs this certificate. This certificate is called the *trace certificate* and the key pair of which the public key on this certificate is one part is called the *trace key pair*. In addition to creating a trace certificate, the AGO supplies the producer with *Globe security credentials* that allow him to access the GDN.

An owner of an object server specifies which producers it wants to give access to his object server. In principle this is done at AGO-granularity: the server owner specifies which AGOs it trusts to do a proper identity and black-list check, and only producers that have credentials and certificates signed by those AGOs will be allowed to place content on that owner's object server. Owners can also give access to or block individual producers.

An upload now proceeds as follows. Assume the producer has created a re-

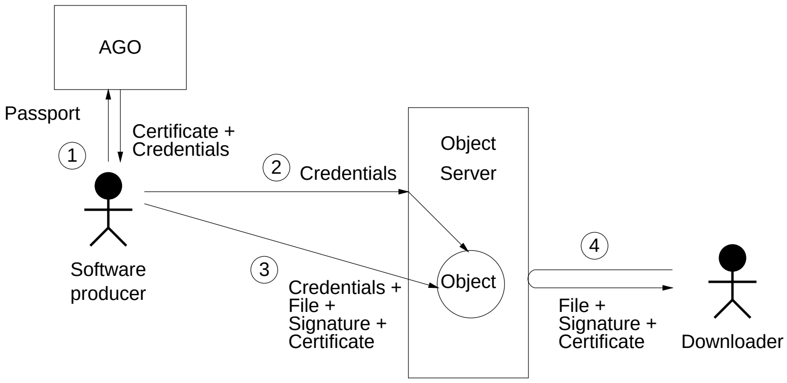

AGO
Passport
1
Certificate + Credentials
2
Credentials
Object Server
Software producer
3
Credentials + File + Signature + Certificate
Object
4
File + Signature + Certificate
Downloader

Figure 5.2: Basic operation of the GDN with traceable content.

vision DSO on one of the object servers that trusts the AGO the producer got his credentials from. Before uploading a file into the GDN, the producer creates a digital signature for this file using the trace key pair. This signature is referred to as the *trace signature*. The trace signature and associated trace certificate are uploaded into the revision DSO along with the file. To this extent, the startFileAddition and endFileAddition method signaling the begin and end of an upload (discussed in Sec. 4.3) were modified as follows. The startFileAddition method invoked at the beginning of an upload takes as extra parameter the trace certificate. The trace signature is passed as an argument to the endFileAddition method.

When the upload is finished, the revision DSO verifies the trace signature. If the signature is false, either because the certificate did not contain the right public key, the file did not match the digital signature, or the producer has been banned from the GDN, the object removes the uploaded file from its state. As only files that are provided by an active (i.e., nonblocked) producer and that carry a valid signature are allowed in an object, all content in the GDN is always traceable.

The whole procedure is summarized in Figure 5.2. To get access to the GDN a software publisher first identifies himself to an AGO and receives a trace certificate and Globe credentials in return (arrow 1 in Figure 5.2). Second, the producer requests an object server, using his Globe credentials, to create a revision DSO (arrow 2). Third, the producer creates a digital signature for the archive file to be published and uploads it along with the file and the trace certificate into the revision DSO (arrow 3). Finally, a user downloads the archive file, trace certificate and trace signature from the revision DSO and verifies that they match (arrow 4).

To ban a producer from the GDN when illicit content traceable to him is found, the following procedure is executed. When someone finds illicit content in the

GDN he contacts a GDN producer who will make the accusation on his behalf. The accusing producer notifies all object-server owners and the access-granting organization that gave the suspected producer access of the publication of illicit content by the suspect. The access-granting organization, in addition, receives a copy of the signed illicit content, and verifies that this content is indeed inappropriate and digitally signed by the violator. If this is the case, the producer's Globe credentials are revoked and the violator is placed on the central black list shared by all AGOs and is thus effectively banned from the GDN. The accusing producer is banned himself if the accusation he makes proves false.

The actions taken by the object-server owners upon notification depend on their *content-removal policy*. They may destroy their replicas of all objects that contain content signed by the violator, or delete the replicas of only the objects mentioned in the allegation. They may do so immediately upon notification by the accusing producer, or only after the allegation has been verified by the AGO. Object-server owners can also decide not to remove the content but instead temporarily block accused producers from their server.

What policy object-server owners will adopt depends on the requirements imposed by the law, the level of abuse and whether or not people report the abuse. In principle, object-server owners are autonomous and can decide for themselves which policy they adopt. However, the GDN may also impose a systemwide policy to guarantee certain systemwide properties with respect to illegal distribution. We currently require object servers to follow a systemwide policy where all content published by a violator is deleted, but only after verification of the evidence. This policy provides protection against malicious GDN producers trying to remove well-known software packages from the GDN.

5.2.2. Discussion

In Sec. 5.1.2 we identified two issues that need to be resolved to successfully use the cease-and-desist scheme. The first issue is how to make sure people who have been previously banned from the software distribution network cannot regain access. In the GDN this problem is solved by requiring candidate producers to prove their real-world identity, which is published on a (public) central black list shared by all access-granting organizations for the GDN if the producer is found guilty of violating GDN policy with respect to content. Formal means of identification are necessary as they cannot be easily faked. It is assumed that the candidate presents his passport or other identification means to a local representative of the access-granting organization, or sends a copy of the document to the AGO, certified by a mutually trusted third party.

Other methods of access control are also possible, for example, requiring the endorsement of a certain number of other GDN producers, or a refundable fee which is lost when the producer violates the rules. Which method for blocking access are permitted depends on the legal requirements, as some methods make it easier to regain access than others. For example, the law may require publishers of child pornography to be locked out permanently. Directly and permanently blocking violators, as I propose, is necessary if there are many violators, just to keep the amount of illicit content in the distribution network at a (legally) acceptable level.

At present, I do not allow for fair mistakes, for example, by blocking a violator only after *n* violations. This particular method can be used in the future if it meets the legal criteria just mentioned. A method for handling mistakes where a producer appeals against decisions may be too labor-intensive for the access-granting organizations.

The second issue is how to keep the amount of work for the moderators low when abuse is low. The moderator task is in this solution carried out by the AGOs. Keeping the amount of work low when abuse is low comes down to keeping the number of false allegations to a minimum. The approach chosen in the GDN is to require that allegations are made through an active GDN producer who will be blocked himself if the allegation proves false. Although blocking the accusing producer himself when the accusation is false may seem like a drastic measure, I believe it is necessary to have a threshold for an accuser in the form of a possible sanction in order to keep the amount of work for access-granting organizations low. It may, however, not be necessary to block a false accuser the first time round or block him permanently. One could allow for a few mistakes and revoke access just for a period of time, as long as the number of false reports made remains low.

It is in the interest of the accusing producer to make these accusations, as in the long run, not participating in banning malicious producers will result in the collapse of the GDN and deprive the accusing producer of a cheap distribution channel for his own software. In other words, making an accusation on behalf of other users is the price producers have to pay for access to the resources of the GDN. One can also imagine people specializing in the role of accuser, that is, people with access to the GDN acting as public prosecutors and explicitly requesting users to report illicit content to them.

Alternative methods for keeping false allegations low are similar to those for keeping violators out of the GDN, for example, a refundable fee which is lost when the accusation is false. Requiring that a group of users or producers make the allegation may not be a good alternative to blocking a false accuser. The effectiveness of the group method depends on the number of malicious persons in the user community. If there are many and they are organized it will be easy for them to swamp the AGOs with false reports. This method basically lacks a way to stop malicious users or producers from repeatedly making false allegations.

An interesting topic for future research is to see whether an effective method for limiting false allegations can be devised based on (end-)user reputation [Lethin, 2001], which would enable users to make allegations directly. An example of such a reputation system is the one used on the slashdot.org news and discussion Web site [Malda, 1999].

The correct operation of the GDN's scheme for limiting illegal distribution depends on two factors: (1) the goodwill of the GDN producers and (2) the correct functioning of the access-granting organizations. In theory, the scheme works even if the majority of Internet users want to abuse the GDN for illegal distribution. Eventually, all abusers will have been black listed and only truthful people will have access. However, by the time we have reached this situation no person with truthful intentions will be making object servers available anymore. This scheme therefore practically depends on the goodwill of the GDN producers. Given that their good name is at stake (the black list of GDN abusers is public), we expect most GDN producers will behave.

The scheme itself provides some protection against misbehaving access-granting organizations. When a truthful access-granting organization mistakenly gives a previously blocked producer access again, an object server ends up serving illicit content. However, as before, this illicit content will be removed and its uploader is blocked when it is detected. When an access-granting organization (purposely or not) does not respond to accusations of abuse by producers it gave access to or (purposely) gives blocked producers access again, the AGO will get a bad reputation. Object-server owners will start refusing any producers the AGO accredited and eventually the AGO will be ousted from the GDN.

The relationship between a producer and the AGOs that gave him access should be viewed as a contract. A producer is given access to the GDN in exchange for which he promises not to abuse this right. If he does, it is the other party in the contract, the AGO, that establishes this fact and terminates the contract on this ground. In principle, the contract requires a GDN producer not to violate the GDN's global policy on what content may or may not be allowed. An AGO could, however, also impose additional restrictions on content and revoke only the credential it gave the producer instead of blocking him at all AGOs. This per-AGO flexibility can be used to incorporate per-country differences, as outlined in Sec. 5.1.2. What can also be made to differ per access-granting organization is to what level a candidate has to prove his identity to the AGO. The policy specifying the AGO's requirements is called its *access-granting policy*. For example, a group of object-server owners could setup their own AGO and define their own access-granting policy and thus their own rules about who gets access to their servers. A minimum requirement for an access-granting policy is, however, that the AGO is able to reveal a producer's real name when that producer is proven to illegally distribute content.

When implementing cease-and-desist, attention should be paid to the cryptography used to create the digital trace signatures. Attackers may attempt to uncover a trace private key so that they can embarrass the publisher owning that key by illegally publishing content as if it originated from the publisher. A protection against such an attack is to use large key lengths [Schneier, 1996]. Large key lengths are particularly important in the GDN for two reasons. First, the archive files and their associated trace signatures will be stored in the Globe Distribution Network for long periods of time, giving an attacker ample time to perform his attacks. Second, the trace signatures are basically cryptographic digests of the archive file created using known digest algorithms, implying that both the ciphertext and the plaintext are available to the attacker, allowing a *known-plaintext attack* [Schneier, 1996]. Increasing the chances of a successful attack may be the fact that I plan to use composite trace signatures that contain a cryptographic digest for each block of a file published for fault tolerance (see Sec. 6.3.1), increasing the amount of plain- and ciphertext available to the attacker.

5.2.3. Using Other Protection Measures

Although designed for supporting the cease-and-desist scheme, this architecture is also able to support other schemes for preventing illegal distribution. Content moderation could be supported by having access-granting organizations act as moderators and requiring users to upload content via their AGO. A reputation-based approach is also supported. An AGO could grant access to users based on their reputation rather than their willingness to show their passport and not appearing on the central black list. (i.e., have an access-granting policy based on reputation).

5.2.4. Anonymity

The GDN was not explicitly designed to support anonymous uploads and downloads, but does offer support for weakly anonymous uploads. Anonymous downloads can be provided by setting up a HTTP-to-GDN gateway and using existing solutions such as the Anonymizer [Anonymizer.com, 2001], Crowds [Reiter and Rubin, 1999] and Onion Routing [Goldschlag et al., 1999].

The need for supporting anonymous uploads in a free software distribution network is small. The reason for supporting it would be to allow people who are unsure about the legal status of their free software to publish that software without immediate risk of prosecution. The GDN only supports weakly anonymous uploads as follows. It is not necessary for access-granting organizations to put the software publisher's real name on the trace certificate that it issues. As stated above, the minimum requirement is that the AGO is able to reveal a producer's real name when he is proven to illegally distribute content. This property opens the possibility for publication under a pseudonym. The minimum requirement still holds, however, so there is a risk for the publisher that when the content he uploads in the GDN cannot be legally published his name will be revealed. Again, this condition can be viewed as part of the contract between AGO and software publisher.

5.3. AUTHENTICITY AND INTEGRITY OF SOFTWARE

People downloading software from a software distribution network want to be assured of the authenticity and integrity of the software downloaded; that is, is the package that they just downloaded the actual GIMP application or does it contain a malicious Trojan horse?

In the GDN, establishing the authenticity of software is the responsibility of the downloading user. In principle, the GDN guarantees only the integrity of the distributed software. It provides no authenticity guarantees other than the verified identity of the uploader as it appears on the trace certificate (which may be a pseudonym), as discussed above. Guarantees concerning the authenticity of software should therefore come from mechanisms outside the GDN. The GDN does, however, provide hooks for such external verification.

Currently, free software distributed via HTTP or FTP is authenticated using public-key cryptography. Maintainers of software packages digitally sign the archive files with a private key and publish the associated public key on the well-known Web site of the software package (e.g. www.kernel.org for the Linux kernel). People that download the software obtain the public key from the well-known Web site and use it to check the digital signature, thus establishing the authenticity of the software. We refer to this signature as the *end-to-end signature*, to distinguish it from the trace signature introduced in the previous section. This authentication scheme requires that the associated public key is obtained from a trustworthy source that guarantees that the key actually belongs to the maintainer of the package. Note that even though Web sites currently do not meet this requirement they are nonetheless used for this purpose in practice.

Another disadvantage of this approach is that end-to-end signatures do not protect against the renaming of files. Renamed files pose a risk because a malicious person may, for example, change the name of an archive file containing an old version of a software package with a known bug or security hole to that of the most recent version of that package. Downloaders should therefore always check that they indeed have the correct version by looking at the contents of the archive file downloaded.

The GDN supports only the automatic verification of end-to-end signatures. The GDN makes it the responsibility of the downloading user to obtain the proper public key. Concretely, when downloading a file from the GDN the end-to-end signature is downloaded along with it. The GDN client software then does the end-to-end authenticity check, using a set of public keys supplied by the downloading user. If the set does not contain the required public key, the user is prompted to supply it.

Another result of the GDN's approach to authenticity is that the naming of files and revision DSOs in the GDN is not regulated. GDN producers are free to name their revision DSOs and files as they want. Downloaders should therefore not rely on names as an indication of what content they are downloading. They should make sure that they have the real name of the software they are looking for. The Globe Name Service is assumed to reliably translate Globe symbolic object names to the object handle of the object.

The reason for not having the GDN provide strong authenticity guarantees is that we expect users not to trust any statements a software distribution network makes about the authenticity of the software they download. We expect users will want to verify themselves that the software they downloaded and which they will be running on their systems is what they expect it to be. Furthermore, it is also difficult for a distribution network such as the GDN to provide strong authenticity guarantees. Consider the following example. To guarantee that the revision DSO named “GIMP 1.1.29” actually contains revision 1.1.29 of the GIMP application we would have to establish who is the maintainer of GIMP and make sure that only that person can create a revision DSO named “GIMP 1.1.29” in the GDN and can upload files into that object. Making sure only a certain person can use certain names and edit certain objects is relatively easy, but establishing who is the maintainer of a specific package is, in general, rather difficult.

5.4. AVAILABILITY

The Globe Distribution Network should have high availability; that is, it must be up and running most of the time. Two factors influence availability: deliberate attacks on the GDN, which we discuss here, and failures, discussed in Chap. 6. Two types of attackers are identified:

1. External attackers (e.g. crackers)

2. Internal attackers, in particular, malicious producers and malicious object servers.

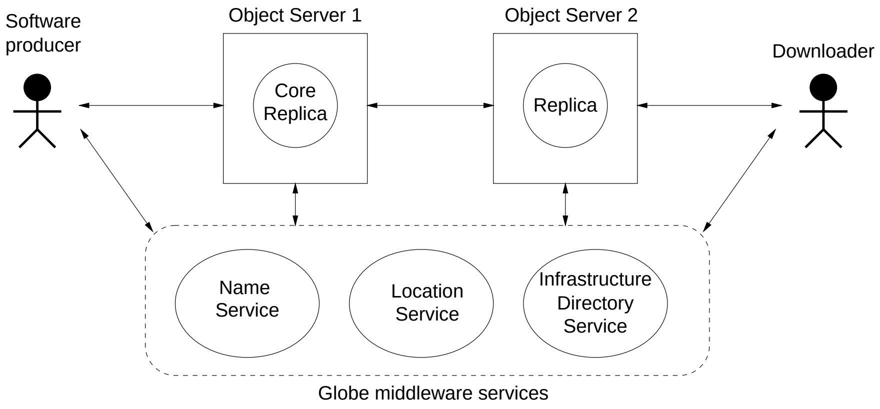


Figure 5.3: Principals in the GDN.

We do not consider denial-of-service attacks by network flooding.

5.4.1. Access Control

External attackers should not be able to delete content from the GDN or otherwise render the distribution network unavailable. Such a requirement is often better expressed positively by defining what legitimate users are allowed to do and to refuse all other actions. To this extent a simple (role-based) access-control model [Sandhu et al., 1996] was developed that describes the security requirements for the GDN. Producers and object servers are initially assumed not to misbehave. We drop this assumption and study the consequences and counter measures later in Sec. 5.4.2.

The basic model of the GDN and the parties involved as established in Chap. 4 are recapitulated in Figure 5.3. Producers upload software into revision DSOs. Each revision DSO consists of one core replica and a number of additional replicas, kept consistent via an active-replication protocol. In this protocol, the core replica is responsible for forwarding the state-modifying method invocations invoked by the producer (e.g. as part of an upload) to the other replicas. Users download the software by invoking read-only methods on the revision DSO. As the Globe Distribution Network is an application running on top of the Globe middleware, it uses the middleware's services, in particular, the Globe Location Service (for keeping track of the location of replicas), the Globe Name Service (for assigning symbolic names to revision DSOs) and the Globe Infrastructure Directory Service (used by revision DSOs to find suitable object servers to create replicas on).

For this dissertation, we assume that there is a trusted organization called the

GDN Administration. The GDN Administration is responsible for running the Globe middleware services, for allowing people or organizations to become an access-granting organization for the GDN, and has the right to update the GDN software itself. In addition, we assume that the middleware services are secure and have their own measures to protect against availability attacks.

The access-control model for the GDN is as follows. When a software producer is given access to the GDN he is given *object-creator* rights, that is, he is able to request object servers to create revision DSOs. When creating a revision DSO the producer becomes the *owner* of that DSO, which means he can

- Make updates to the state of the object (i.e., invoke state-modifying methods).
- Request object servers to create or delete replicas of the object.
- Register and deregister contact addresses for the object with the Globe Location Service.
- Delegate these rights to other Principals.

As described in Chap. 3, creating a DSO currently consists of creating an initial replica local representative. At creation time, the owner of the object assigns the initial replica the *core-replica* role, thus delegating the following rights to this replica:

- Authorize state updates to other replica local representatives. As described in Sec. 4.6.1 it is the responsibility of a core replica to forward state-modifying method invocations made by clients to all replicas.
- Request object servers to create or delete replicas of the object. As described in Sec. 4.6.2), the core replica coordinates replica placement for the whole object.
- Register and deregister a replica's contact addresses with the GLS.
- Delegate the latter two rights (minus replica deletion) to other replica local representatives.

When a core replica decides the revision DSO needs a new replica it contacts a suitable object server and, in the process, delegates *replica* rights to the new replica. An authorized replica is allowed to

- Register and deregister its own contact address for the object with the GLS.
- Register itself at the core replica such that it can receive updates.

- Retrieve a copy of the state of the object from the core replica.
- Report statistics about its client population to the core replica.
- Request object servers to create replicas of the object. This right is required to allow a replica to autonomously handle server overload (see Sec. 4.6.4).
- Delegate these rights to other replica local representatives.

The owner of a revision DSO may assign *uploader* privileges to other GDN producers, enabling them to update the state of the object. For example, if a group of people are cooperating to make binary variants of a software package available for different platforms (Alpha, Intel), the chief maintainer of the software package could assign them uploader privileges to the revision DSO, such that the others can autonomously upload new variants. When the owner of the object leaves the development team he either transfers his ownership rights to another member of the team or the remaining members of the team have to republish the software in new objects. Which method is chosen depends on the terms on which the developer leaves and whether or not the security system implementing this access control model allows transfer of ownership rights.

An issue not yet addressed by this model is the ownership of symbolic object names (see Sec. 3.3 and Sec. 4.4). It is assumed that (at least) the *owner* of the object owns a name space in a name or directory service in which he can register the object. More precisely, the owner has the right to register names in that name space, but he does not have the exclusive right to register names for that particular object; other people, for example, other developers can register other names for the object in their own name spaces.

Discussion

The access-control model ensures that external attackers cannot reduce the availability of the Globe Distribution Network by attacks other than network flooding and sending many download requests. The latter case is handled by the flash-crowd control mechanism of Sec. 4.6.4. It is assumed that the implementation of this access-control model also takes into account basic security risks, such as message replays [Schneier, 1996].

If symbolic object names from the owner's name space are frequently used to refer to the published software, a problem can occur when there are multiple developers. The problem occurs when the owner leaves the team on unfriendly terms, without transferring the right to register names or object ownerships. In that case the remaining developers should republish the software in other objects but are not able to register these new objects under their old, well-known names.

For example, assume a person *X* publishes revision 1.0 of his jointly developed software package in a revision DSO and names the object /org/x/revision-1.0/ in the /org/x name space which is owned by him. Furthermore, assume this name becomes the well-known name for this revision of the software package. If *X* leaves the development team without transferring any ownership, the remaining team members cannot republish the 1.0 revision in a new object and register the object under the well-known name /org/x/revision-1.0/. It is assumed that the remaining developers will reregister in a different name space when this problem occurs.

5.4.2. Internal Attackers

As mentioned in Sec. 5.2 one of the design goals of the GDN is to allow many different people and organizations to run object servers and participate in the GDN. Until now we have assumed that these object-server owners are trustworthy. However, people may attempt to undermine the availability of the GDN from the inside by running a modified and maliciously acting object server. Modifying object servers is easy as the source code of GDN is assumed to be publicly available. We, as GDN designers, do not have and can never have complete control over object-server machines and thus cannot prevent this malicious behavior.

Furthermore, although we have looked at how to prevent GDN producers from publishing illegal content we have not looked at attacks on the availability of the GDN they could mount. In this section we first analyze how the GDN's availability could be reduced by malicious producers, maliciously modified or maliciously operated object servers, or malicious clients. Next, we present a number of solutions for disarming these internal attackers.

Malicious Producers A GDN producer can play three roles: that of object creator, that of owner of an object, and that of uploader. The object-creator rights can be used to do a denial-of-service attack by means of resource allocation. An object creator could create many revision DSOs (i.e., initial replicas) on many object servers, thus reducing the total available object-server capacity, which, in turn, makes the GDN less available to other producers and objects. The owner rights can also be used to create superfluous replicas. Owner and uploader privileges can be used by a malicious producer to make the state of the revision DSOs large such that they take up large amounts of persistent storage. An owner can also make many registrations for his objects in the GLS. This latter problem is considered outside the scope of this dissertation.

Malicious Object Servers The most basic attack for a malicious object server is to serve downloaders different content. Doing so only hinders downloaders, as the integrity of the content is protected by the trace signature. However, if the content served is not what the user expects but still traceable (i.e., a malicious object server could serve the user the content of a totally different object), users will not notice a problem until they do the end-to-end authenticity check. The fact that object servers may not be trustworthy makes the end-to-end authenticity check absolutely vital to the secure downloading of software from the GDN.

Object servers can be delegated core-replica and replica rights. In general, a GDN producer will select an object server that he trusts as the host for the core replica of the revision DSO. At present, it is assumed that if a producer does not know which servers to trust he will consult other GDN software producers to discover trustworthy servers. In the future reputation systems may be used for this purpose, see Sec. 5.4.3. As the server hosting the core replica is trusted we can consequently assume that the core-replica rights will not be abused. However, an object server might turn out not to be so trustworthy, and, more seriously, a malicious producer may himself run malicious object servers, create core replicas on them and use those to attack the availability of the GDN. The core-replica rights can be used to overallocate resources similar to a producer's owner rights. A core replica can create many replicas on other object servers, with large states, thus making those servers' resources unavailable for others.

If all object servers except the one running the core replica cannot be trusted, the core replica becomes the only trustworthy source of the object's state, which introduces a capacity problem. A new replica when it is created needs to obtain a copy of the state in order to serve clients. This copy can now be safely obtained only from the core replica, significantly increasing its workload. This increased workload may overload the machine the core replica runs on. Other tasks of the core replica such as sending the state-modifying method invocations to replicas are not expected to generate much more work for the core replica when the number of replicas scales up, due to the low frequency of updates on a revision DSO.

When an object server is granted replica rights this includes the right to report statistics to the core replica about the number of clients that access it and where they are located (i.e., their autonomous system). This right can be abused to

1. Create more replicas than necessary
2. Trigger the core replica to delete replicas
3. Trigger the core replica to create replicas in the wrong location

The first can be achieved by reporting more accesses than actually seen. The second can be achieved by generating low-usage statistics for a replica and reporting those to the core replica as if they originated from that (well-behaved) target replica. These low-usage numbers may cause the core replica to delete the seemingly underutilized replica when it re-evaluates the revision DSOs' replication scenario (see Sec. 4.6.2).

In the third attack a malicious replica or group of replicas attempts to deprive a group of clients in a particular region from getting their own replica. This attack can be mounted when the clients use these malicious replicas as the object representatives most convenient for them. In that case, the replicas could refrain from reporting the clients' downloads to the core replica of the object, and it would not create a replica in the clients' region. This attack is only possible in this particular situation, and thus unlikely to occur. In all other situations, the clients are connected to well-behaved replicas that will allocate them a replica by means of the flash-crowd control/server load balancing mechanism (described in Sec. 4.6.4), a mechanism that cannot be interfered with by non-participating malicious replicas.

There are potentially also vulnerabilities in the replication protocol used by revision DSOs. For example, if state update messages need to be acknowledged by the receiving object servers and the core replica does not proceed until replies have been received from all replicas, malicious object servers could sabotage this update by not replying. In general, malicious object servers can sabotage collective decisions to be made by the object's local representatives.

Another (general) problem at the protocol level is that of principals sending maliciously modified protocol messages. In contrast to sending correct messages that have negative availability effects, these messages are purposely incorrect and are sent to unnecessarily allocate resources, cause a crash of the peer process, corrupt the peer's data, or enable the sender to gain access to the peer's host computer. As indicated in Chap. 2, the counter measures against this type of attack are sound programming practices, authentication of users and logging their actions, and enabling fast upgrade of the application when exploitable bugs are discovered. We will not discuss actual vulnerabilities of the replication protocol used, as describing the replication protocol at the level of detail required for such an analysis is outside the scope of this dissertation.

Malicious Downloaders The third class of internal attackers are downloaders running maliciously modified clients. Although the access-control mechanisms and replication protocol prevent them from doing real harm, they can easily mount a denial-of-service attack against a revision DSO by specifying a huge block size (e.g. 650 MB) when downloading or uploading a large file (e.g. an ISO9660 CD image). This specification will cause an object server to allocate huge amounts of memory, leading to out-of-memory errors and a slowdown of the server.

5.4.3. Countermeasures Taken by the GDN

In this section we describe the measures employed by the GDN to prevent or counter the internal attacks just identified.

Keeping Producers In Check

A GDN producer is kept from creating too many revision DSOs as follows. Although the popularity of software packages largely differs, the rate at which new revisions of a package are published is fairly stable. Rarely more than one new revision is published per day. In terms of GDN, a well-behaved software producer will not create more than a few revision DSOs per day. This property enables us to limit both the rate at which a malicious producer can create revision DSOs, and put an upper limit on the total number he is able to create. For this purpose the *GDN Quota Service (GDNQS)* is introduced.

To create a new revision DSO that is, create an initial replica (see Chap. 3), the producer now first has to contact the GDNQS to obtain an *object-creation ticket*. Object servers will only create an initial replica if the request is accompanied by such a ticket. The GDNQS keeps track of how many tickets have been granted to a producer in the last 24 hours and does not issue tickets to producers who exceed the limit, thus limiting the creation rate. The GDNQS also roughly imposes a limit on the total number of revision DSOs a producer is allowed to create. The total number of revisions of a software package depends largely on the age of the package, as the revision-publication rate of software is relatively stable and low. We therefore assign a producer an annual quota of revision DSOs he is allowed to create, enforced by the GDN Quota Service. The GDNQS can be considered the global resource management system for the GDN.

We keep a producer from allocating too many resources on a particular object server by introducing a local resource management system for object servers. The local resource management system keeps track of how many resources are used by each (replica) local representative and to which producer this local representative belongs, and denies allocation requests if a producer has already been allocated his fair share. In addition, we impose a limit on the size of the state of a revision DSO (e.g. 1 GB), enforced by the code of the object's local representative. The limit is set centrally for the whole GDN and is adjustable to allow growth in average file size.

Furthermore, the server's resource management system deletes local representatives which are not frequently used, thus providing protection against producers, core replicas, and regular replicas trying to reduce availability of the GDN by allocating useless additional replicas. An exception is made for initial replicas, to prevent revision DSOs containing old, and therefore unpopular revisions of software packages from being deleted when they should be kept for archive purposes. An attacker could counter this measure by setting up clients that access the superfluous “malicious” replicas, thus keeping up their replicas’ usage, but this requires a sustained effort from the attacker, and is therefore assumed unlikely.

These solutions for keeping a GDN producer in check require that each software publisher operates under a single identity such that his or her resource usage can be correctly recorded. Otherwise, if a software publisher could use multiple identities he could allocate *n* times the resources (where *n* is the number of separate identities he can assume), making the resource-management measures much less effective. Implementing this requirement in the GDN is complicated by the fact that a software publisher can become GDN producer via multiple access-granting organizations, and is identified by the trace certificate issued by an AGO. To meet the single-identity requirement AGOs must coordinate their actions such that they issue trace certificates all listing the same identity. This measure makes the GDN less suitable for anonymous publication (see Sec. 5.2.4), as this coordination between AGOs will require disclosing the publisher’s real-world identity to all AGOs.

Keeping Object Servers In Check

To prevent malicious object servers from affecting other object servers and hindering downloaders the following measures are taken. First, to give the object-server owners some control over which object servers his server cooperates with, object-server owners are allowed to specify preferred object servers and block servers they do not like (e.g. by specifying certain IP-address ranges). These rules are also used for evaluating “create replica” requests the object server receives.

As a second counter measure, the replication protocol for revision DSOs is changed as follows. Replicas no longer report statistics to the core replicas. Instead, replication is handled solely by the revision DSO’s flash crowd control mechanism discussed in Sec. 4.6.4, and now works as follows. Each replica monitors its own load. When this load becomes too high, the replica tries to find object servers in the regions the traffic is predominantly coming using the Globe Infrastructure Directory Service, and requests them to create extra replicas of the DSO. When client demand drops, the underutilized replicas are garbage collected by the resource management system of the object servers, freeing up resources for other objects, as explained above. As this mechanism also takes into account the location of clients, this change does not affect the capability of the GDN to do network load balancing. More research is needed to determine what the effects of losing a global coordinator for replication are on the efficiency of the replication protocol.

A result of not being able to trust object servers is an increased workload for the core replica, as it becomes the only trustworthy source for the state of the object. To handle the increased workload a revision DSO is now allowed to have multiple core replicas. They are assumed to run on trustworthy hosts and coordinate their actions. The use of multiple core replicas for fault tolerance is discussed in the next chapter.

The most important measures for protecting a downloader against a misbehaving object server are the end-to-end authenticity and integrity checks (see Sec. 5.3). If these checks are properly carried out, an object server can be only obnoxious to a downloading user since any malicious modifications to the software will be detected. As an additional, albeit limited protection against badly behaved object servers, the GDN allows users to black-list object servers in their client software or to specify preferences (e.g. preferably connect to object servers from the .edu domain).

An interesting new area of research is trust models, sometimes also referred to as reputation systems [Lethin, 2001; Cornelli et al., 2002]. These systems record the reputation of a server, as established from previous interactions with that server by clients and other servers. Using the reputation system, new clients and servers can then select a server with a good reputation to interact with or check a server's reputation before interacting. The application of reputation systems to Globe applications is currently under investigation [Pierre and Van Steen, 2001].

Keeping Clients In Check

The current implementation of Globe distributed shared objects requires that the problem of huge arguments in method invocations be handled at the replication-protocol level. To prevent replicas from allocating large amounts of memory to handle a large method invocation sent by a malicious client the replication subobject is changed to refuse method invocations larger than a certain value. The exact value is a configuration parameter of the replication protocol, allowing it to evolve as server and network capacity increases and to be configured according to application-specific requirements.

5.4.4. Alternative Countermeasures

A simple alternative to the GDN Quota Service for making sure that GDN producers cannot create too many revision DSOs, is as follows. Recall that creating a revision DSO consists of creating an initial replica on an object server (see Sec. 3.5). The alternative solution is to require that producers create and run the initial replicas on their own hosts. GDN producers without a permanent Internet connection should approach an object-server owner and negotiate permission to host their initial replicas on his server. I prefer the GDNQS solution as it never requires active participation from object-server owners, and thus allows the GDN to operate more automatically.

5.5. INITIAL IMPLEMENTATION

I have written an initial implementation of the design described above. The access-control part of the implementation is a temporary solution as a security subsystem for the Globe middleware is still being researched. In our implementation there is no uploader role, ownership of an object is nontransferable, and registration and deregistration of contact addresses in the Globe Location Service currently does not require special privileges. Furthermore, as indicated in Chap. 4, revision DSOs currently do not support automatic replication, but use a master/slave protocol where the object's owner has to create replicas manually. The initial implementation also does not support automatic end-to-end signature checking, nor does it take into account malicious object servers, or producers attacking the availability of the GDN. In the future I hope to replace this implementation by a more complete one that uses the Globe security framework currently being researched.

The initial implementation works as follows. For an overview of the basic operation of the GDN with content tracability, see Figure 5.2 on page 119. Part of the GDN Administration (i.e., the governing body of the GDN, see Sec. 5.4.1) is the *Certification Authority for Access-Granting Organizations (AGOCA)*. Organizations wanting to act as AGO for the GDN send an application to the AGOCA that includes a public key generated by the AGO. If approved, the AGOCA returns to the AGO a certificate signed by the AGOCA. This certificate is called the *AGO certificate*.

An author of free software wanting to publish it via the GDN first creates a public/private key pair, in particular, a 1024-bit RSA key pair. After applying for access at one of the GDN's AGOs and passing the identity tests, the producer is given an X.509 version 3 certificate [Housley et al., 1999] containing the public key he supplied signed by the AGO. This certificate acts both as trace certificate and as Globe security credential. The certificate hierarchy is shown in Figure 5.4.

The AGO creates a record for this producer in an LDAP server [Loshin, 2000]. This record is uniquely identified by the serial number of the producer's trace certificate and the name of the AGO, and contains a field stating whether the producer is currently *active*, *accused* of publishing illegal content, or *blocked* by this AGO. This field is referred to as the *status of a producer*. The AGO's LDAP server may be replicated around the world for performance. The collection of LDAP servers used by the GDN is collectively named the *GDN Access Control Service (GDN*

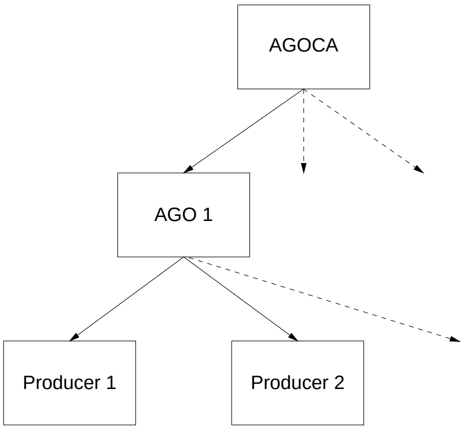


Figure 5.4: The certification hierarchy in the GDN's initial security implementation. Arrows represent "certified by" or "signed by" relationships.

ACS), as it is the authoritative source for determining whether or not a producer's access to the GDN has been revoked.

Next, the producer installs the trace certificate in his *GDN producer tool*, and is now ready to upload software into the GDN, as follows. First, he uses the producer tool to create an empty revision DSO. As explained in Chap. 3, creating a DSO currently consists of creating an initial replica of the object on an object server. The object server and producer tool use a Transport Layer Security (TLS) library for authentication (and secure communication) [Dierks and Allen, 1999]. As described above, each object-server owner can specify which AGOs he trusts and thus is willing to accept producers from. He does so by creating a key store containing the AGO certificates of the AGOs he trusts, and configuring the TLS library such that this key store is used as "Certification Authority key store." In other words, a client sending a request to the object server must authenticate itself with a certificate signed using one of the certificates in that key store.

Assuming the producer tool successfully authenticates itself to the chosen object server, the producer has now created an empty revision DSO. Ownership of the object is assigned to the producer creating it by encoding an SHA-1 digest [Schneier, 1996] of the producer's public key in the object's object handle. Recall that a DSO's object handle is its life-long, worldwide unique, location-independent identifier (see Chap. 3). The initial replica of the revision DSO configures itself to accept only state-modifying method invocations that are received from its owner over a TLS connection. In particular, the replica requires the peer at the other side of the connection to authenticate itself using the trace key pair that matches the SHA-1 digest in the DSO's object handle. A trace key pair matches the SHA-1 digest if the peer proves it knows the private key of the trace key pair, and an SHA-1 digest of the public key of the trace key pair is equal to the SHA-1 digest in the object handle.

In our current master/slave replication protocol the initial replica acts as the master. We make sure that slave replicas accept updates only from this replica as follows. Each object server has its own public/private key pair. When a producer asks an object server to create the initial replica, the object server executes the request and returns the contact address of the master replica, and the object server's public key. To create a slave replica, the producer sends this information to a candidate object server. This object server creates a slave local representative of the DSO using the master's contact address. In addition, the slave object server configures the slave replica using the master object server's public key such that the slave replica accepts updates only when received over a TLS connection from the master local representative of the DSO located on the master server.

Only the owner of a revision DSO is allowed to create or delete additional slave replicas. An object server asked to create or delete a slave replica will execute this request only when received from the object's owner over a TLS connection. To establish that it is the owner making the request, the object server uses the SHA-1 digest of the owner's public key in the object's object handle. Our security mechanism ensures crackers cannot modify the state of the revision DSO or alter the DSO's replication scenario, given that only the producer has access to the trace private key and the master replica's authorization is tied to the key pair of its own object server which is trusted.

Having created the empty revision DSO the producer can now upload a file into the GDN. The producer tool first binds to the revision DSO. Next, the tool calculates an MD5 signature for the file and uploads the file into the DSO by invoking the startFileAddition method (which takes the producer's trace certificate as parameter), a number of invocations of putFileContent and finally invoking the endFileAddition method that has the MD5 signature as parameter.

Contrary to what is described in Sec. 5.2.1, the revision DSO does not check itself whether the trace signature is correct. Instead, to simplify the implementation, the traceability of a file is checked by a *back-end traceability checker (BETC)* (pronounced "betsy"). A BETC is a user-level process that runs in parallel to each object server and checks the traceability of the files in the local representatives the object server hosts. If a local representative contains untraceable files, BETC instructs the object server to unbind from that object, thus deleting its replica of the content. In other words, when an untraceable file is uploaded into a revision DSO all object servers hosting a local representative of the object will delete that local representative.

The BETC works as follows. Periodically, its associated object server is asked to list which local representatives it hosts. The BETC selects from this list the local representatives that are new (i.e., created since the last check) and the local representatives whose state has been updated. The latter information is obtained from the local representatives which each record the time it was last notified or received an update of the state of the object.

The BETC checks each file in the selected local representatives. First, the BETC downloads the file to local storage, and retrieves the trace signature and trace certificate from the revision DSO. Second, it verifies that the trace certificate is issued by one of the access-granting organizations trusted by the owner of the object server, following the certificate-chain checking rules of [Housley et al., 1999]. Third, BETC verifies that the trace signature indeed matches the downloaded file by recalculating the MD5 digest and comparing it to the digest in the trace signature. Finally, the BETC contacts the GDN Access Control Service (GDN ACS) to determine whether the publisher of this file has been accused or is blocked from the GDN. If one of the first two tests fails the BETC instructs the object server to remove the local representative. If GDN ACS reports that the producer is accused, it depends on the configured content-removal policy (see Sec. 5.2.1) whether or not the replica is deleted. The replica is always deleted if the producer is blocked.

In addition to checking new or updated replicas, the BETC also periodically checks the status of the producers of every file stored in the object server. In other words, rather than receiving notification from accusing producers or the access-granting organization about a change in the producer's status (i.e., a push model) as described in Sec. 5.2.1, a pull model is used to see whether producers have been accused or blocked by their access-granting organization.

At present, accusations are made via e-mail for simplicity. A producer making an accusation sends a report containing the file with the disputed content, the associated trace signature and certificate to the access-granting organization of the violator. The report is digitally signed by the accuser. Upon receipt, the AGO changes the producer's status in the GDN Acces-Control Service to "accused." When the accusation proves correct, the accused producer's status is changed to "blocked" and his real name is placed on the central black list. This central black list is stored on the *root ACS server*, an LDAP server operated by the GDN Administration. The steps of a successful accusation and the resulting removal of content is illustrated in Figure 5.5.

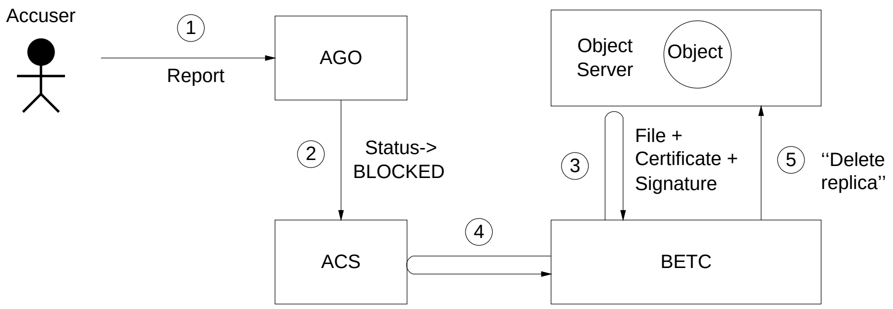


Figure 5.5: The steps of a successful accusation. (1) The accusing producer sends a report to the violator's AGO. (2) The AGO finds the accusation to be true and revokes the violator's access by contacting the Access Control Service. (3) Periodically, each BETC checks the content hosted by its associated object server. (4) Part of this check is obtaining the status of the content's producer from the ACS. (5) The status of the malicious producer is "blocked" so a BETC instructs the object server to destroy the replicas that contain content traceable to the violator.

CHAPTER 6

Fault Tolerance

This chapter describes how the Globe Distribution Network can be made fault tolerant. Fault tolerance has three aspects: availability, reliability and failure semantics. Availability indicates the probability of a system being available at any moment in time. The reliability of a system indicates how often it exhibits failure. Failure semantics define the state of the system after a failure. Sec. 6.1 reiterates the fault-tolerance requirements for the GDN, lists the assumptions about the type and frequency of failures the GDN must be able to handle, and presents a model of the dependencies between the different components of the GDN that helps in presenting and understanding our measures for making the GDN meet its requirements. Sec. 6.2 describes how the GDN is made highly available and reliable. Finally, in Sec. 6.3 we describe how the GDN is made to exhibit strong failure semantics when failures can no longer be masked. Unless stated otherwise, object servers in this chapter are assumed to be well trustworthy (i.e., are not setup to purposely attempt to disrupt the GDN by sending harmful protocol messages or return false replies).

6.1. REQUIREMENTS AND SYSTEM MODEL

Chapter 2 identified the fault tolerance requirements for the Globe Distribution Network. The GDN should be highly available, as it has large numbers of users that depend on it.The GDN should also be reliable and have strong failure semantics. Reliability implies that the application itself handles most failures and masks them from its users and administrators. Strong failure semantics ensure that when failures can no longer be masked, the application is at least brought into a well-defined state before reporting the error. The GDN is required to be *atomic with respect to exceptions (AWE)*, that is, its semantics for a failure during an operation

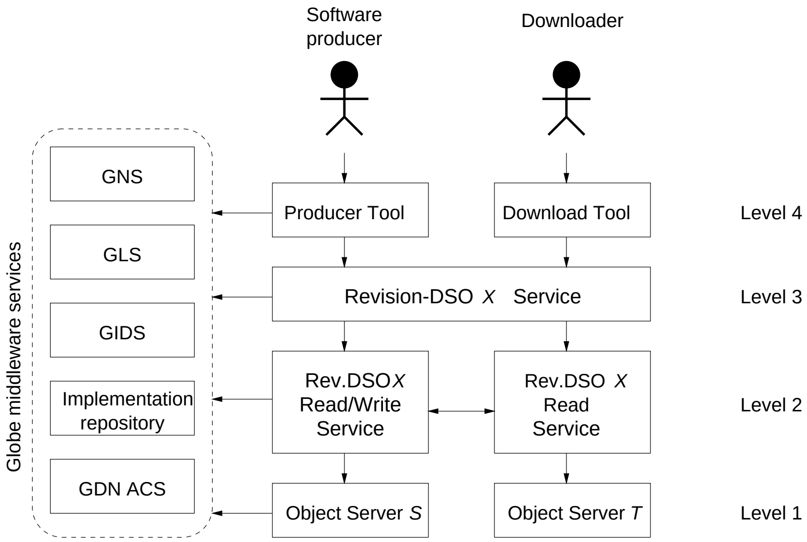


Figure 6.1: Simplified dependency model for the GDN, showing only a single revision DSO and two object servers. Arrows represent dependencies, boxes represent services, and a rounded dashed box represents a collection of services. Each GDN component/service is classified using a number of levels, shown on the right-hand side. The dependencies between the middleware services and the right-hand column of GDN components have been left out for simplicity, that is, all GDN components depend on the Globe middleware services

should be that either the operation is carried out, or it is not and the application is returned to the state it was in before the start of the operation [Cristian, 1991]. Both reliability and AWE failure semantics are necessary to make the GDN, an application with many components and many parallel operations, easy to manage.

In order to determine how an application can be made fault tolerant it is important to know how the components of the application interact. In particular, it is important to know which components depend on each other to see which component may be affected when a certain component fails. Cristian [1991] introduced the idea of defining a model of a distributed application in terms of services and dependencies between services to aid failure analysis. The model for the Globe Distribution Network is shown in Figure 6.1. It has been simplified to show only a single revision DSO *X* and two object servers, *S* and *T*.

The rationale behind the model is the following. Downloaders and software publishers depend on their tools to give them access to the GDN. These tools, in turn, depend on the collection of revision DSOs that hold the GDN's content to serve the requests from their users. A revision DSO X is modeled as three services: a *revision-DSO service*, a *read/write service* and a *read service*. The revision-DSO service is implemented by the proxy local representatives (see Chap. 3) of the revision DSO running in the up- and download tools, and depends on the read/write and read services to carry out its work. The read/write service is implemented by the revision DSO's core replica and the read service by the object's regular (i.e., noncore) replicas. The read service depends on the read/write service as the core replica is sometimes used as a source of state for a new regular replica (in particular, when the core replica is the object's only replica). The read/write service depends on the read service as the former service's task is to update the state of the object, which includes the copies in the object's regular replicas. The reason for modeling a DSO in terms of three services is that, in principle, different mechanisms could be used to make the read and read/write services meet their fault-tolerance requirements. The revision-DSO service is added to provide an extra layer between the client and the read and read/write services that can be used to mask failures in these latter two services, if they chose not to mask their failures from their clients.

Read/write and read services depend on a set of Globe object servers. Globe object servers host the revision DSO's local representatives implementing the read and read/write services and provide access to persistent storage. All services depend on the Globe middleware services: the Globe Name Service, the Globe Location Service, the Globe Infrastructure Directory Service, an implementation repository and the GDN Access Control Service. Not shown in Figure 6.1 (for simplicity) is the lowest level in the dependency hierarchy: the hosts running the tools and object servers and the network that connects them.

To complete the system model we need to make explicit the assumptions about what faults can occur in each GDN component. The hosts running the GDN components are assumed to exhibit only crash and performance failures, and no response or state-transition failures [Cristian, 1991]. This assumption implies that the hosts may crash or be slow, but always correctly execute the GDN components' code. The network can exhibit omission (i.e., packet loss) and performance failures (i.e., delay messages), although the former are assumed to be masked by the transport-level network protocols, in general. Partitions of the network are assumed not to occur. The GDN software is assumed to exhibit only crash failures, that is, the software either works or stops abruptly and does not exhibit Byzantine behavior such as response or state-transition failures.

The following sections describe how the GDN is made highly available, reliable and exhibits AWE failure semantics given the system model just presented. This dissertation does not discuss how Globe's middleware services are made fault tolerant, see, for example, Ballintijn et al. [1999].

6.2. AVAILABILITY AND RELIABILITY

Making sure a distributed application is highly available and reliable starts, in principle, at the host and network level. Hosts and network can be made highly dependable using hardware redundancy (e.g. processors with a hot backup, disk arrays [Chen et al., 1994], and multiple independent network connections). However, given the free nature of the Globe Distribution Network, we cannot employ hardware solutions to increase availability and reliability. Instead, we start one level higher: making sure Globe object servers are up and running most of the time.

6.2.1. Level 1: Fast Object-Server Recovery

Globe object servers can be made highly available by enabling them to quickly recover after a crash with most of their state intact. To this extent, Globe object servers currently support a simple checkpointing mechanism. Periodically, the object server creates a checkpoint by halting the processing of incoming requests, waiting until current requests have been processed, and then saving its state to disk. The state of an object server consists of the states of the local representatives it hosts and the administration the object server maintains about these local representatives. Once the object server's state is stable on disk, the previous checkpoint is deleted in an atomic disk operation. After a crash, the new object server reads the last complete checkpoint back from disk, recreates the local representatives, and passes them their marshalled state. Each local representative then reinitializes itself and synchronizes with its peers in an application- or even object-specific manner. How a local representative of a revision DSO synchronizes with its peers is discussed below.

Checkpointing the state in this fashion negatively affects the object server's availability as it does not process requests during a checkpoint. The assumption underlying this mechanism is therefore that checkpoints can be made quickly. In general, this assumption holds for an object server containing just GDN revision DSOs, as follows. First, none of the methods on a revision DSO take a lot of time to execute, so the checkpointing thread does not have to wait long before it can start checkpointing the server's state to disk after it has stopped the server from accepting new requests. Second, although the state of an object server used for GDN can be large, most of it is already stored on disk, in particular, the large states of the local representatives. Therefore, most of the object server's state need not be saved again, considerably decreasing checkpoint time. As described in Sec. 4.5.2, a local representative of a revision DSO stores the large parts of its state, in particular, the archive files containing the software, on disk, and implements the lrSubobject interface that allows the object server to quick marshall and unmarshall the local representatives' remaining in-core state. The parts of the object server's state that is already stored on disk at the time of a checkpoint is referred to as the server's *stored state*. So, in general, the amount of work to be done during a checkpoint of an object server hosting local representatives of revision DSOs is small and is not expected to significantly impact availability. This mechanism can be called user-directed checkpointing rather than incremental checkpointing [Plank et al., 1995], as we logically checkpoint everything, but use the fact that the application already has data stable on disk to reduce the actual amount of data to checkpoint.

A revision DSO's local representative (LR) recovers from a crash as follows. The revision DSO's replication protocol is changed such that each write-method invocation or copy of the state sent to a replica by the revision DSO's core replica now contains a *state-version number* that identifies the state. In case of a write-method invocation the state-version number identifies the version of the state that results from executing the method. There is a well-defined total order for state-version numbers as the core replica imposes a total order on all write-method invocations (see Sec. 4.6). To recover from a crash a local representative of a revision DSO contacts one of its peers and sends it its current state-version number. If trusted servers are used the recovering LR can contact any peer; in case of untrusted servers it must contact the core replica. If the peer has a higher state-version number, the recovering LR destroys itself (required to implement AWE failure semantics, see Sec. 6.3.2). What happens when the recovering local representative is still in sync with the peer is described below.

This recovery method for local representatives of revision DSOs assumes the stored states of the LRs (i.e., the archive files containing the software) are not damaged during a crash. To ensure that file data is not lost during a crash and that, when a file is damaged this is detected and appropriate action can be taken, the following measures are taken. First, before the actual checkpoint is made, the persistence manager of the object server, which handles disk I/O for local representatives (see Sec. 4.5.1) is asked to make sure all data are written to disk and not just to operating-system buffers. This measure ensures that the archive files are safe on disk even if the server crashes just after the checkpoint. Second, the integrity of the files making up the stored state is checked before they are used.

For the GDN, the local representatives of revision DSOs control and perform the integrity checks themselves. The checks are done after the LR has determined it still has the most recent version of the state (by exchanging state-version numbers with its peer). If this is the case, the LR checks the integrity of the archive files it stored on disk using the files' trace signatures and certificates (i.e., a trace signature of a file includes a cryptographic digest of the associated file which can be used to check its integrity, see Sec. 5.2.1 and Sec. 6.3.1). If this integrity check fails, the LR deregisters itself with the rest of the object and destroys itself.

A checkpointing and recovery mechanism improves not only the availability of the services it hosts, but also the manageability of the application. In particular, if there is an automatic recovery mechanism an application administrator does not have to concern himself with each host failure. This is useful, in particular, for applications where replication is done manually.

Alternative

An alternative way of checkpointing the local representatives in an object server is as follows. Instead of saving just the remaining in-core state of an LR, the checkpointing mechanism creates its own copy of the complete state of a local representative (i.e., including the archive files). To keep checkpointing time to a minimum (and thus maximize availability) the checkpointing mechanism checkpoints only local representatives whose state was updated since the last checkpoint (i.e., incremental checkpointing). This status information should be provided to the checkpointing mechanism by the local representatives.

The advantage of this approach is that the programmer of the local representative's semantics subobject has less to do with fault tolerance: he does not need to implement the lrSubobject quick marshalling interface (if we integrate this checkpointing mechanism with the current server-passivation facilities, see Sec. 4.5.2). The disadvantage of this approach is, however, that we need double the storage space as it effectively requires maintaining a complete backup copy of every local representative hosted by the object server. Furthermore, if a lot of large local representatives were updated since the last checkpoint, the checkpoint may take considerable time as it involves copying large amounts of data. For an application storing large amounts of data, such as the GDN this alternative is too expensive. This alternative represents a trade-off between ease-of-programming and disk space.

The present checkpointing scheme can provide the local representatives of DSOs with a number of choices with respect to who checks the integrity of their stored state. The integrity checks can be performed by the local representatives themselves, as in the GDN, or by the object server's persistence manager. In particular, a local representative can request the persistence manager to recover it only when its checkpointed state is intact. The advantage of this approach is that the programmer does not have to concern himself with integrity checking. The disadvantage is that the integrity checks are done always, even if the LR decides to destroy itself, which increases recovery time, as checking the integrity of the stored state may be time-consuming. Having the local representative itself control the integrity checking of its stored state does not have this disadvantage, and enables it to check the integrity of the files using application-specific methods (as in the GDN), although this is not necessary. The LR could still request the persistence manager to verify the integrity.

6.2.2. Level 2: Availability and Reliability of the Read Service

As discussed in Chap. 4, revision DSOs create replicas to optimize download speeds, and to balance network and server load. We use these extra replicas, created for performance reasons, to improve the availability and reliability of the read service of a revision DSO.

As shown in Figure 6.1, the read service of a revision DSO has two clients: the revision-DSO service and the read/write service. To make the read service highly available to the revision-DSO service, hierarchical masking of failures is used, as follows. Recall that the revision-DSO service is considered to be implemented by the proxy local representatives of the revision DSO. These proxy local representatives are connected to the nearest replica of the DSO. When this nearest replica does not respond quickly enough to a (read) method-invocation request, the proxy reconnects to a different replica, thus masking the failure from the download tool, and allowing the user to continue.

The other client that depends on the read service is the revision DSO's read/-write service. Its task is to update all copies of the state of the revision DSO, in particular, those in the replicas of the read service. As this operation involves all replicas that are operating correctly at the time of update, no special measures can be taken to improve availability or reliability of the read service to the core replica.

6.2.3. Level 2: Availability and Reliability of the Read/Write Service

In the replication protocol used by revision DSOs (described in Sec. 4.6.1) the core replica plays a central role. It keeps track of the replicas of the revision DSO. Proxies send all write-method invocations to the core replica which then forwards them to all other replicas. If there are no regular replicas (because the object is not popular or due to crashes) the core replica also services reads from clients. In addition, when we allow replicas on untrusted servers, the core also is the only trustworthy source for the state of the object. The availability and reliability of this read/write service is improved by introducing multiple core replicas. It is the responsibility of the revision DSO's owner to create these extra core replicas. The core replicas coordinate all actions previously taken by the single core (e.g. ordering updates, replication-scenario reevaluation) using a consensus protocol.

6.2.4. Level 3: Availability and Reliability of the Revision-DSO Service

The revision-DSO service is implemented by the proxy local representatives of the revision DSO. Proxy local representatives run inside the user's producer and download tools, and are created dynamically when binding to the revision DSO (see Sec. 3.3). The reliability of the service these local representatives provide to the tools can therefore not be improved using redundancy, only by sound programming. Availability can be said to be guaranteed by the Globe Location Service and the implementation repository containing the code of the proxies, as they ensure that if a user wants to use the revision-DSO service he can dynamically load an instance.

6.2.5. Level 4: End-to-End Integrity Protection

Chap. 5 describes how the GDN relies on digital signatures to guarantee the integrity of a file distributed through the GDN. This integrity check also detects any data corruption that has occured due to failures inside the GDN that may have gone unnoticed. In this sense, the trace signatures on files in the GDN (see Sec. 5.2.1) provide *end-to-end* integrity protection, a desirable property [Saltzer et al., 1984].

6.3. AWE FAILURE SEMANTICS

This section describes how the GDN is made to exhibit strong failure semantics when failures cannot be masked. Operations on the GDN should be atomic with respect to exceptions (AWE). More specifically, our goal is to make not only individual method invocations on revision DSOs exhibit AWE semantics, but also uploads and downloads, which are higher-level operations consisting of multiple method invocations. Recall that files are uploaded and downloaded in blocks (see Sec. 4.3). We discuss how method invocations, downloads and uploads can be given AWE failure semantics in the following sections.

6.3.1. Well-behaved Downloads

Making downloads exhibit AWE failure semantics is easy. In most cases a download will be successful given that the download tool can fail-over to other replicas. This statement holds for read-method invocations in general. There are two cases where a download may need to be rolled back: when all replicas have become unavailable and when the download tool itself crashes. In both cases rollback is simple because revision objects do not keep track of the state of a download (i.e., which parts of the files have been downloaded by the client). Rolling back the operation therefore involves only deleting the incomplete file from the downloading user's disk. This can be done by the download tool, immediately (all replicas crashed) or when it is restarted (tool crash).

For downloads the GDN actually provides semantics stronger than AWE. As a convenience to the GDN user, special measures are taken to allow a user to continue a download after a crash of his download tool or temporary unavailability of the revision DSO. In other words, the download tool allows the user to choose between aborting (AWE semantics) or restarting the download. Restartable downloads are supported as follows. The download tool starts a download by retrieving the desired file's trace signature (see Sec. 5.2.1) from the revision DSO and storing it on disk. Our trace signatures are special signatures that can be used to check not only the integrity of the whole file, but also of its individual blocks. In particular, a trace signature is a record consisting of the cryptographic digests [Schneier, 1996] of the individual blocks and a cryptographic digest of the whole file encrypted with the producer's trace private key (required for traceability).

If the tool crashes during this first step, its reincarnation simply downloads the trace signature again. Next, the tool starts downloading the file from the object in blocks. When the tool or the machine it is running on crashes at this point, the reincarnation reads the trace signature from the local disk. The trace signature contains a checksum, allowing the tool to detect if the signature has been damaged. Using the trace signature, the tool checks if any of the already downloaded blocks of the file were damaged during the crash. In particular, it recalculates the digest of each block and compares it to the block's digest in the trace signature. If any blocks are damaged, the tool downloads these blocks again.

After the integrity checks on the downloaded data, the tool resumes the download at the point where its previous incarnation crashed. The final step in the download procedure is verification of the complete file and downloading the end-to-end signature, both of which can be repeated after a nonhalting crash of the tool.

6.3.2. Well-behaved Uploads

Making write-method invocations and file uploads atomic with respect to exceptions is complex. Implementing these semantics basically means developing a transaction mechanism that allows single writes and sequences of writes to be executed atomically, as Globe currently lacks such a mechanism. Developing a transaction mechanism for Globe is an extensive research topic and therefore outside the scope of this dissertation. I therefore resort to an *ad hoc* solution with weaker semantics that strives to make writes and uploads succeed whenever possible. This solution sacrifices replicas in order to prevent having to report failure,

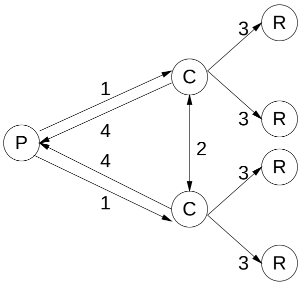


Figure 6.2: A successful invocation of a write method. Circled Ps denote proxies, circled Cs denote core replicas and circled Rs regular replicas.

assuming that replicas will be recreated by the object if client demand requires it. This solution is considered sufficient for the time being, given the number of uploads into a revision DSO is expected to be low.

As an upload consists of a number of write-method invocations we first explain how individual writes are handled. A successful write-method invocation proceeds as shown in Figure 6.2. When a client invokes a write method on a proxy local representative, the proxy forwards the marshalled invocation to all core replicas (the arrows labeled 1 in Figure 6.2). The core replicas then execute a consensus protocol to uniquely order the write relative to other writes (i.e., impose a total order on writes) (arrows labeled 2 in the figure). Next, each core replica forwards the request to the regular replicas it knows about (i.e., each regular replica is known at only one core replica at a time) (arrows labeled 3 in the figure). All core and regular replicas carry out the method, after which all core replicas return the results of the method invocation to the proxy (arrow 4 in Figure 6.2).

We distinguish three types of failures that can happen during a write: crash failure of a replica (halting or nonhalting), noncrash-failure of a replica (e.g. out of disk space), and crash failure of the upload tool. To handle the first type of failure we take the following measures. When a replica, either core or regular, crashes during the upload it is pronounced dead and no longer considered part of the object even if the object server recovers. When a core replica crashes, the regular replicas connected to it contact the Globe Location Service to discover another core replica, retrieving a new version of the state if necessary. When all core replicas fail, this is detected by the remaining regular replicas which destroy themselves thereby destroying the object.

Replica failures other than crashes are handled as follows. A regular replica that fails to execute the method because of a local failure destroys itself. Core replicas report the result (failure or success) of the method execution to their peers.

When at least one core replica reports success, the core replicas that failed to execute the method destroy themselves. Core-replica failures are reported to the upload tool along with the results of the method invocation. It is the object owner's responsibility to recreate failed core replicas. We take no special measures to handle crash failure of the upload tool.

With this strategy for executing write methods, an upload consisting of multiple writes is successful if at least one of the core replicas succeeds in carrying out all method invocations. The only case where our measures are not sufficient is when all core replicas fail to execute one of the methods. In this case, the core replicas instruct the remaining regular replicas to destroy themselves. The core replicas will not destroy themselves, instead they report a failure of the method invocation to the upload tool. It now is the responsibility of the upload tool to rollback the upload by deleting the partially uploaded file from the revision DSO. Should the delete method fail the only option is to recreate the object.

To reduce the chance that an upload fails due to a lack of disk space at a particular replica (i.e., a noncrash failure), we reserve the required disk space at the start of the upload. As explained in Chap. 4, to prevent clients from seeing a partially uploaded file, uploads of a file start and end with special method invocations (i.e., startFileAddition and endFileAddition). By having the startFileAddition method reserve the required disk space we reduce the chance of an upload failing half-way through. This explains the FileSize parameter of the startFileAddition method (see Figure 4.11).

This approach to achieving AWE failure semantics for uploads also works if servers are untrusted. As untrusted servers are assumed to run only regular replicas and not core replicas, there is no opportunity for them to sabotage the upload as they are not involved in making the decision to abort.

Alternative

An alternative protocol for write-method invocations (see Figure 6.2) is to have the proxy send the method invocation to just one core replica and have that core replica distribute the invocation to the other core replicas. This approach requires extra measures to make sure the proxy receives a reply when the core replica to which it sent the request fails. These extra measures may make this protocol harder to implement when TCP (i.e., a connection-oriented protocol) is used as transport protocol. In particular, the reply would have to be cached at the other core replicas to be retrieved by the proxy when it reconnects to another core replica after the crash, or the other core replicas would setup a connection back to the proxy to deliver the reply, complicating the proxy's implementation. The latter would also reverse the normal direction of connection setup which may present problems when firewalls are used.

6.3.3. Side Effects of Failures

Failures of object servers can cause the Globe Location Service (GLS) and Globe Infrastructure Directory Service (GIDS) to be out of sync with the actual situation. In particular, they could list replicas or object servers that are no longer available.

To handle inconsistencies in the GLS, proxy local representatives are made to fail over to another replica if the one identified by the contact address returned by the GLS does not respond quickly. To handle inconsistencies in the GIDS (which keeps track of the available object servers) the revision DSO's replication subobject is adjusted such that it will ask for another candidate object server if it does not get a reply to its "create replica" request.

In addition, measures are taken to avoid inconsistencies from occurring. When considering the GLS, inconsistencies due to halting crashes of object servers (i.e., they do not recover) are avoided by having object servers periodically register contact addresses again. Object servers that suffered just nonhalting crashes deregister the contact addresses of replicas that could not be recovered after a crash. Periodic reregistration is also used to prevent inconsistencies between GIDS and which object servers are actually up and running.

Object-server failures can have other side effects. In specific circumstances replica failure can cause overload on the GLS and remaining replicas of the object, but they are too specific to discuss in this dissertation, see [Ketema, 2000].

CHAPTER 7

Performance

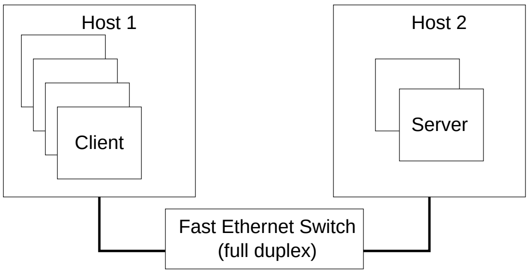
This chapter describes a number of small performance tests that were conducted with the initial implementation of the Globe Distribution Network. Sec. 7.1 compares the performance of a single Globe object server hosting revision DSOs to that of an Apache HTTP server [The Apache Software Foundation, 2002]. The second set of tests, described in Sec. 7.2, looks at the performance of the initial implementation in a wide-area environment.

7.1. SERVER PERFORMANCE

The goal of the first set of experiments is to compare the performance of a Globe object server to that of the Apache HTTP server. In particular, we are interested in the average throughput per client when large numbers of clients are accessing a server, and the number of downloads per second each server can support.

The experimental setup is shown in Figure 7.1. All clients are run on a single host (Host 1) and access server processes running on Host 2. The hosts have identical hardware and software. For the tests with Apache, we used the version that comes with RedHat 7.1 (Apache 1.3.19), and the wget HTTP client (version 1.6). The Apache server is configured following advice from RedHat [Likins, 2002]. Its configuration is shown in Table 7.1. For the tests with GDN, we used version 1.0 of the GDN implementation (available from http://www.cs.vu.nl/globe) which is written in Java. To execute the Java code we used the IBM Developer Kit for Linux, Java 2 Technology Edition, version 1.3-2001-09-25, which includes a high-performance just-in-time (JIT) compiler and is available from http://www-106.ibm.com/developerworks/java/jdk/linux130/.

In the first two experiments we compare the throughput achieved to that of the



Configuration of both Host 1 and Host 2:
Dual PIII 933 MHz, 2 GB memory, 100 Mb/s Ethernet (Intel Ether Pro 100 onboard),
Adaptec 29160 Ultra-160 SCSI controller, 2x Seagate 73 GB 10.000 rpm disks,
RedHat 7.1, Custom configured 2.4.9 kernel, with SCSI driver from Adaptec.

Figure 7.1: Setup for the server-performance experiments.

Table 7.1: Apache configuration.

Apache parameter	Value
MinSpareServers	20
MaxSpareServers	80
StartServers	32
MaxClients	256
MaxRequestsPerChild	10000

maximum theoretical throughput of TCP over 100 Mb/s Ethernet, which is 11.3 MB/s[1].

7.1.1. Experiment 1

The first experiment is aimed at measuring the average throughput per client for a large number of clients simultaneously downloading the same file. In this experiment with the Globe object server, the 30 Megabyte file to download is stored in

[1] We arrive at this number as follows. The maximum size of an Ethernet frame is 1526 bytes, and the interframe gap is 12 bytes, giving a total of 12304 bits per frame [The Institute of Electrical and Electronics Engineers, Inc., 2000]. The bit time for FastEthernet is 1 bit/100 Mb/s = 10 nanoseconds, implying that $1 / (12304 \cdot 10 \cdot 10^{-9}) = 8127$ frames can be sent over a FastEthernet link in a second. Of a 1526-byte frame, 1460 bytes can be used for data when TCP/IP is used, resulting in a maximum theoretical bandwidth of 1460 bytes/frame $\cdot$8127 frames = 11.3 MB/s.

just one revision DSO. This revision DSO consists of a single replica local representative running in the object server. For the Apache server we start *n* HTTP clients in parallel. These clients immediately begin downloading the file from the server.

For the GDN we also start *n* GDN clients in parallel, but these clients do not start downloading immediately, for the following reason. When starting a large number of GDN clients, which are Java programs, the time between when the client that was started first commences downloading, and the time the client that was started last commences its download is considerable. This large gap is due to the fact that Java programs take more resources, and can thus affect the test results. Hence, the GDN clients wait until all other clients are running and ready before starting to download. As a result, the tests with GDN clients do not take into account connection-setup time, whereas the tests with HTTP clients do. Modifying the HTTP clients such that connection time is also not taken into account was dismissed, as it would change the client behavior as observed by the Apache server. Instead of connecting and immediately receiving the request there would be a (small) delay between connection and reception. In this experiment, the GDN client does not check the trace signature of the file after download, as we are measuring server performance. Both the HTTP client and the GDN client discard the downloaded data by writing it to /dev/null, so there is no real disk I/O at the client side. GDN clients download the file in blocks of 1000 KB.

The results of this experiment are shown in Figure 7.2. We repeated the experiment 3 times for both types of servers. The numbers used are the average of the 3 runs. As the figure shows the performance difference between Apache and the Globe object server is less than 10 percentage points for large numbers of clients. The performance bottleneck is the 100 Mb/s Ethernet network.

7.1.2. Experiment 2

In Experiment 1, all clients downloaded the file from a single revision DSO with a single replica. In Experiment 2, each client downloads the file from a different (single replica) revision DSO. The reason for conducting this particular experiment is as follows. Globe serializes all method invocations on a replica (see Sec. 3.2). As a result, the single revision DSO of Experiment 1 may be a bottleneck for server performance as clients have to wait for method invocations by others to complete. The hypothesis is therefore that the average throughput per client for a Globe object server is better if the clients are not reading from a single revision DSO.

The results of the experiment, shown in Figure 7.3, shows that the hypothesis is false. For 10–100 concurrent clients the average throughput per client when downloading from different objects is practically the same as when using a single revision DSO. The maximum difference between the throughputs is 3%. In other

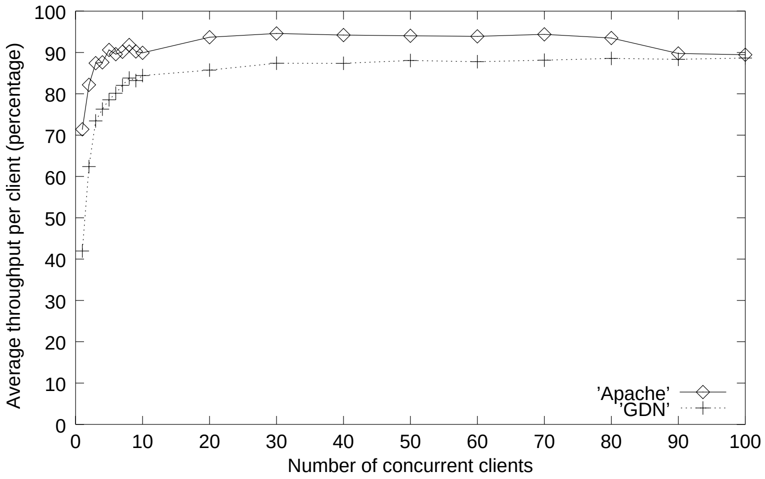


Figure 7.2: Average throughput per client for Apache and the GDN as a percentage of TCP's maximum theoretical throughput over 100 Mb/s Ethernet. This theoretical throughput of TCP is 11.3 Megabyte per second divided by the number of concurrent clients.

words, local serialization of method invocations does not appear to hamper server performance.

7.1.3. Experiment 3

In the previous two experiments the clients always downloaded large files from the servers. To see how the servers compare with a heterogeneous workload, the following experiment was conducted. Both servers were loaded with the 50 most popular files on the SourceForge free-software site in October 2001. In the GDN case each file was placed in a separate object. The size of these files ranged from 21 KB to 15 MB (average 1.5 MB). At the client side 50 clients were started. Each client continuously downloaded the same file from the server. After 30 minutes we killed all clients and count the total number of successful downloads.

Both the GDN and the HTTP client setup a new TCP connection for each download. The GDN client did not use the Globe Name Service or Globe Location Service during binding, instead it read the contact addresses of the replicas on the object server from a file. Both the HTTP client and the GDN client discarded the downloaded data by writing it to /dev/null, so there was no real disk I/O at the client side. GDN clients downloaded the file in blocks of 1000 KB.

The results of Experiment 3 are shown in Table 7.2. The experiment was

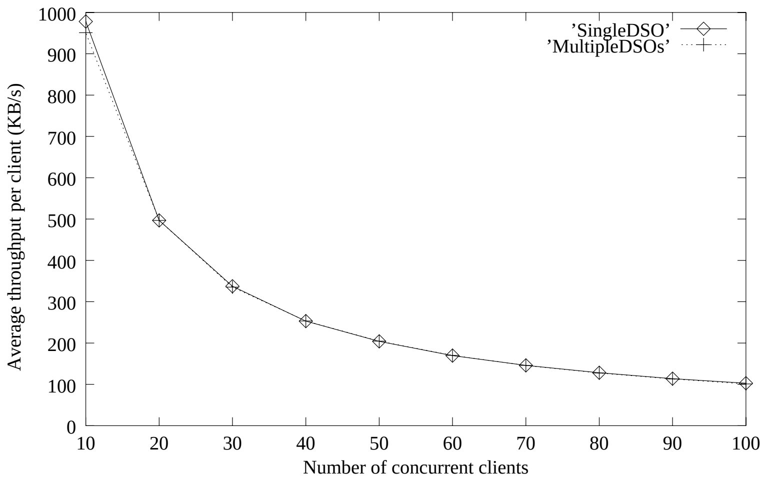


Figure 7.3: Average throughput per client for a single DSO and for multiple DSOs in Kilobytes/second.

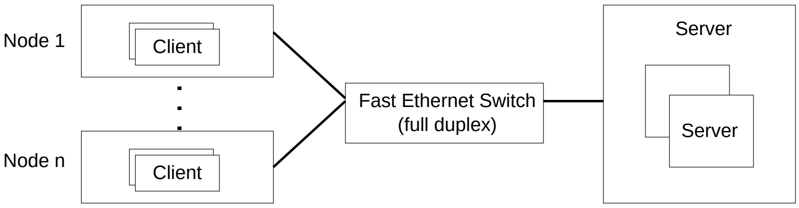
Table 7.2: Results of Experiment 3.

Server	Downloads per 30 minutes	Downloads per second
GDN	29306	16.28
Apache	30522	16.96

repeated three times, and the numbers shown are the average of the three runs. In this test GDN is only slightly slower than Apache. The bottleneck is also not processing at the servers, but the 100 Mb/s Ethernet network. It would therefore be interesting to repeat the set of experiments on Gigabit Ethernet hardware.

7.1.4. Experiment 4

Experiments 1–3 were conducted with a single machine hosting all client programs. To make sure this single machine did not create an artificial bottleneck, Experiment 1 was repeated using multiple client machines. The setup of this experiment is shown in Figure 7.4, and uses 1–50 dual-processor client machines. For the measurements with 1–50 concurrent clients each node ran a single GDN or HTTP client. For the measurements with 60–100 concurrent clients, each node ran 2 client programs, one on each CPU. The GDN clients do not wait with the download until all clients are running, as the times between the start of the first



Configuration of Server
Dual PIII 933 MHz, 2 GB memory, 100 Mb/s Ethernet (Intel Ether Pro 100 onboard),
Adaptec 29160 Ultra-160 SCSI controller, 2x Seagate 73 GB 10.000 rpm disks,
RedHat 7.1, Custom configured 2.4.9 kernel, with SCSI driver from Adaptec.

Configuration of Nodes
Dual PIII 1 GHz, 1 GB memory, 100 Mb/s Ethernet (IBM Mobile onboard)
RedHat 7.2, Custom configured 2.4.19-pre10 kernel.

Figure 7.4: Setup for the server-performance experiment using multiple client machines.

and the last client do not differ significantly, unlike in Experiment 1. As a result, in this experiment the time to setup a connection to the replica of the DSO on the server machine (i.e., binding time) is taken into account. The GDN client does not, however, use the Globe Name Service or Globe Location Service during binding, which results in small bind times. Another difference with Experiment 1 is that the Ethernet switch used was different (Lucent vs. Extreme Networks).

The results of the experiment compared to those of Experiment 1 are depicted in Figure 7.5. The curves with the '-1' prefix indicate the results of Experiment 1 with the single client machine. The curves with the '-n' prefix indicate the results of the experiment with many client machines. The curves are the average of three runs. Surprisingly, for larger numbers of clients the Apache server has better performance when the clients are on a single machine. The performance of GDN is better when multiple client machines are used. As a result, Apache and GDN are closer together in terms of performance when multiple client hosts are used. Furthermore, it makes it unclear whether or not the single client machine constitutes a bottleneck, which was the purpose of this experiment, although performance is close to TCP's theoretical maximum in both cases.

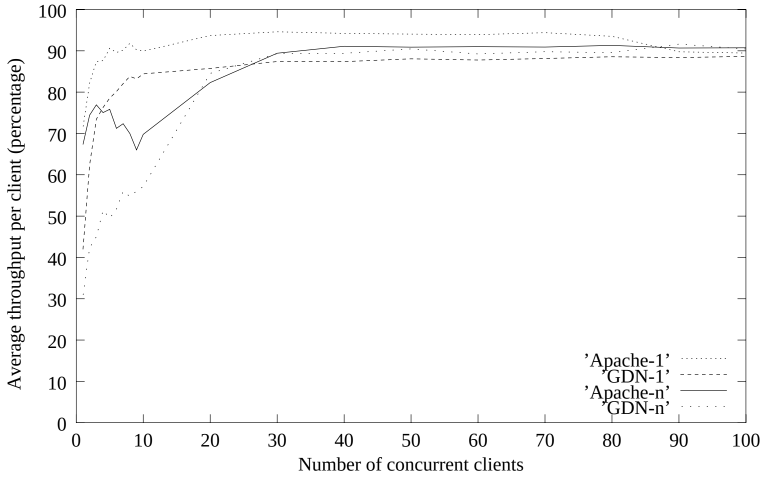


Figure 7.5: Average throughput per client for Apache and the GDN as a percentage of TCP's maximum theoretical throughput over 100 Mb/s Ethernet, measured using 1 or $n = 1$–50 dual-processor client machines.

7.2. WIDE-AREA PERFORMANCE

The final experiment is aimed at gaining insight into the performance of the current GDN implementation when used in an actual wide-area environment. In particular, this experiment measures in which parts of the implementation the GDN spends its time when downloading files over the Internet (i.e., the experiment provides an execution profile for the GDN on the Internet). In the future, I hope to do more extensive wide-area experiments (e.g. multiple clients, different network loads) for which time and resources lacked at present.

In this experiment we simulate a single user downloading a single file from a revision DSO. The user's machine is located in San Diego (California, USA). The revision DSO has its master replica on a machine in Amsterdam (The Netherlands), and a slave replica on a machine in Ithaca (New York, USA). The geographic locations of the sites involved in the experiment are shown in Figure 7.6. Table 7.3 lists the hardware and software configuration of the machines.

The experiment consists of the following two tests.

1. The client in San Diego downloads a 1 Megabyte file from the master replica in Amsterdam, which involves transfer over a transatlantic link.

2. The client in San Diego downloads the 1 Megabyte file from the slave replica in Ithaca, which is on the same continent.

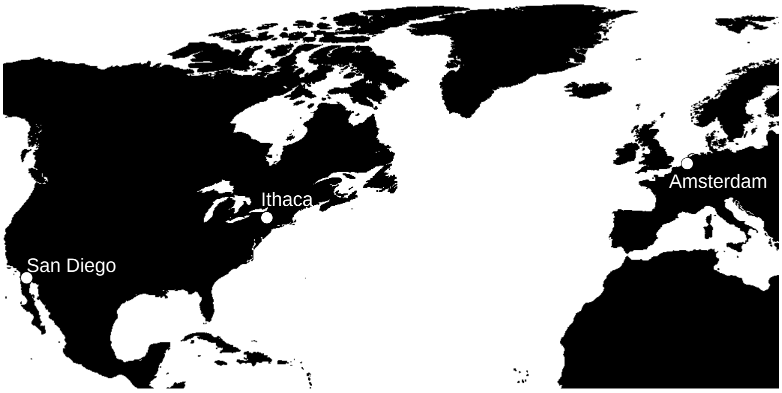


Figure 7.6: Geographic location of the test sites.

Table 7.3: Hardware and software configuration of the test machines.

Site	Processor	Memory	Network	OS	JDK
San Diego	Celeron 367.5 MHz	256 MB	10 Mb/s	FreeBSD 4.2	Sun JDK 1.2.2
Cornell	UltraSPARC-IIi 440 MHz	256 MB	10 Mb/s	Solaris 2.8.0	Sun JDK 1.4.0
Amsterdam	UltraSPARC-IIe 502 MHz	640 MB	100 Mb/s	Solaris 2.8.0	Sun JDK 1.3.1

Table 7.4: Comparison of end-to-end performance for GDN downloads from Amsterdam and Ithaca.

GDN Test	Average download time (min–max) in μs	Average throughput (min–max) in KB/s
1 MB download from Amsterdam	14,707,742 (14,259,718–15,128,651)	69.6 (67.7–71.8)
1 MB download from Ithaca	7,952,904 (7,598,283–8,421,981)	128.8 (121.6–134.8)

The one Megabyte file size was chosen as to not interfere with operations at the San Diego test site (the San Diego machine is located at an Internet measurement center). The results of the experiment are presented as follows. We first look at the end-to-end performance in terms of download time as perceived by the user. Next, we analyze the differences between the download times by looking at the steps involved in a GDN download and the time each step consumed during the downloads. Finally, we compare the download times of GDN to those of an HTTP client and server running on the same test machines.

7.2.1. End-to-End Performance

Table 7.4 shows the download times and throughput for the two tests (commas are just used as separators). The download times are the average of 99 downloads. The tests were conducted on a weekday in March 2002 at 13:00 GMT. At those times there was little activity on the machines used. The blocksize used is 1 Megabyte in this experiment. As the table shows, a download from Ithaca is faster than downloading from Amsterdam.

7.2.2. Performance Analysis

A download via GDN consists of the following steps (for a description of the methods of a revision DSO, see Sec. 4.3), illustrated in Figure 7.7.

1. The client binds to the revision DSO. Binding consists of four steps:

   (a) Mapping the symbolic object name to the object's object handle using the Globe Name Service

   (b) Mapping the object handle to the contact address of the replica using the Globe Location Service

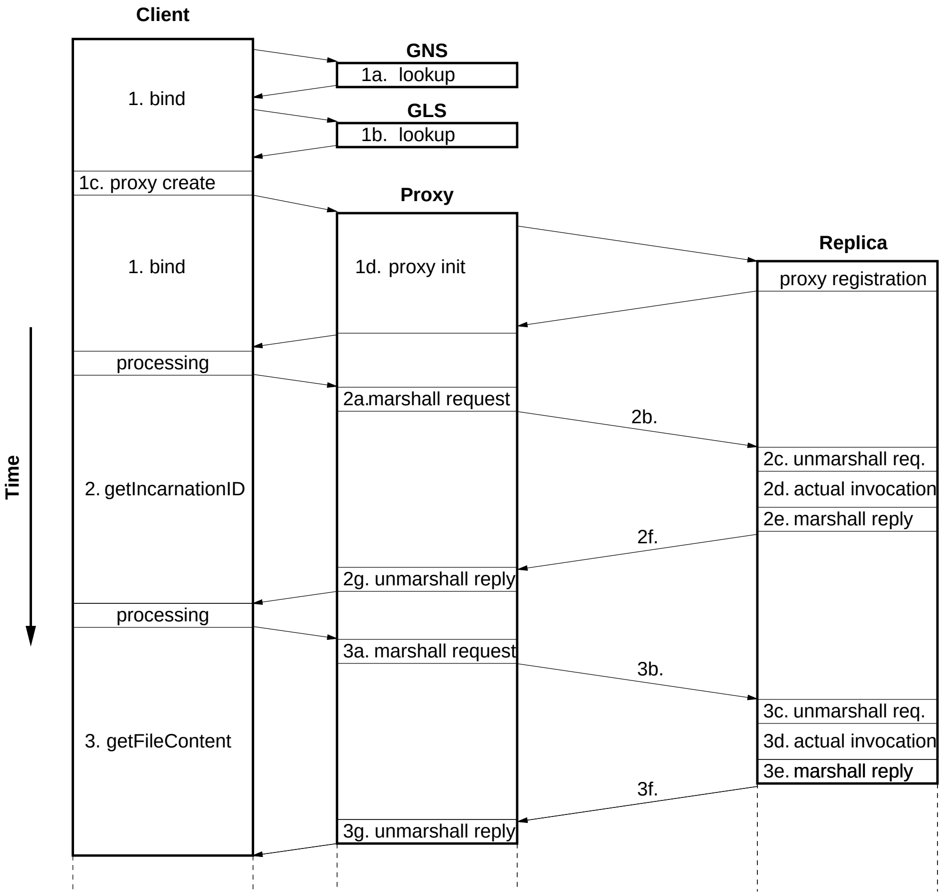


Figure 7.7: Steps involved in a GDN download. Boxes represent processes or services. Arrows represent inter-process or network communication. The last method invocation of the download is not shown here for simplicity.

   (c) Creating a proxy local representative from the contact address. This proxy local representative is referred to as “the proxy” in the remainder of this description
   (d) Initializing the proxy from the contact address. This step includes setting up a TCP connection with the remote replica

2. The client invokes the package::getIncarnationID method on the proxy of the revision DSO just created, to find out the current incarnation ID of the file to be downloaded. A method invocation consists of the following steps:
   (a) The proxy in the client marshalls the method invocation
   (b) The proxy sends the marshalled invocation over the TCP connection to the replica
   (c) The replica receives the marshalled invocation and unmarshalls it
   (d) The replica makes the requested method invocation on its semantics subobject
   (e) The replica marshalls the result of the method invocation
   (f) The replica sends the marshalled result back to the proxy
   (g) The proxy receives and unmarshalls the result and returns it to the client

3. The client repeatedly invokes the package::getFileContent method to download the file in blocks of 1 Megabyte (MB). As the file is 1 MB in size in the tests, the transfer of the file’s contents takes two getFileContent invocations: one to transfer the complete contents, and one to find out that the whole file was downloaded in the first invocation

We measured the following times. The measurement points referred to in the text are shown in Figure 7.8.

Bind total The time spent binding, defined as T7 minus T0. This time is further subdivided into

GNS-lookup the time spent in the Globe Name Service, defined as T2 minus T1.

GLS-lookup the time spent in the Globe Location Service, defined as T4 minus T3.

Proxy-create the time spent creating the proxy local representative, defined as T6 minus T5.

Proxy-init the time spent initializing the proxy, defined as T7 minus T6.

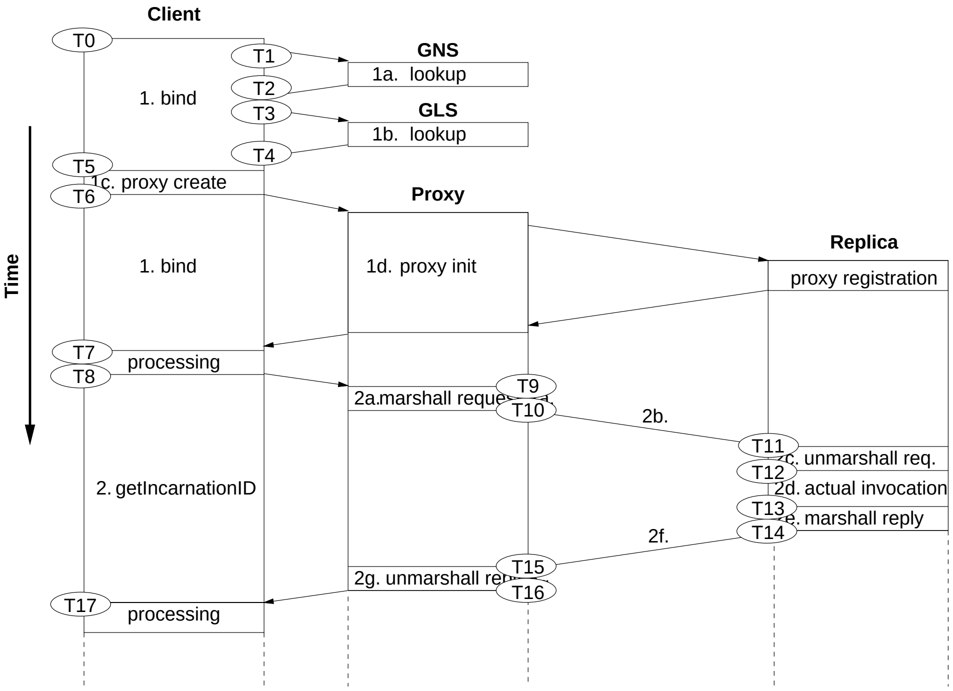


Figure 7.8: Measurement points for the GDN download. A measurement point is represented by a labeled ellipse. This figure shows only the binding steps and the first method invocation of the download for simplicity.

***MethodX* total** The time between the invocation of method *MethodX* by the client and the client receiving the result of the invocation. *MethodX* is one of:

- getIncarnationID,
- getFileContent(first block), or
- getFileContent(last block).

For getIncarnationID, getIncarnationID **total** is equal to T17 minus T8.

This ***MethodX* total** time is subdivided into the following three times:

Proxy *MethodX* The time spent in the proxy while preparing to send the *MethodX* method invocation request, and while processing the received result of the method invocation. This time therefore includes both marshalling of the request and the unmarshalling of the reply. For getIncarnationID, **proxy** getIncarnationID is equal to T10-T9 + T16-T15.

Replica total *MethodX* The total time spent in the replica local representative. For getIncarnationID, **replica total** getIncarnationID is equal to T14 minus T11. This time is subdivided into:

Actual invocation the time spent executing the actual method in the semantics subobject. For getIncarnationID the actual invocation time is T13 minus T12.

Pre/post invocation the time spent passing the request from the communication subobject to the control subobject, unmarshalling the request, marshalling the result after the invocation and passing it to the communication subobject for transmission. For getIncarnationID the pre/post invocation time is T12-T11 + T14-T13.

Transmission *MethodX* The time required to transmit the marshalled *MethodX* method invocation from the proxy to the replica, plus the time required to transmit the marshalled result of the method invocation from the replica to the proxy. In other words, the transmission times for request and reply are combined. The reason for not measuring request and reply times separately is simplicity. The combined time can be measured simply by subtracting the time spent in the replica from the time that passes between the proxy sending the invocation request and the proxy receiving the result of the invocation. For example, for getIncarnationID we calculate transmission time by subtracting T14-T11 from T15-T10.

The breakdown of the first test (i.e., client in San Diego downloads from master replica in Amsterdam) and the second test (i.e., client in San Diego downloads

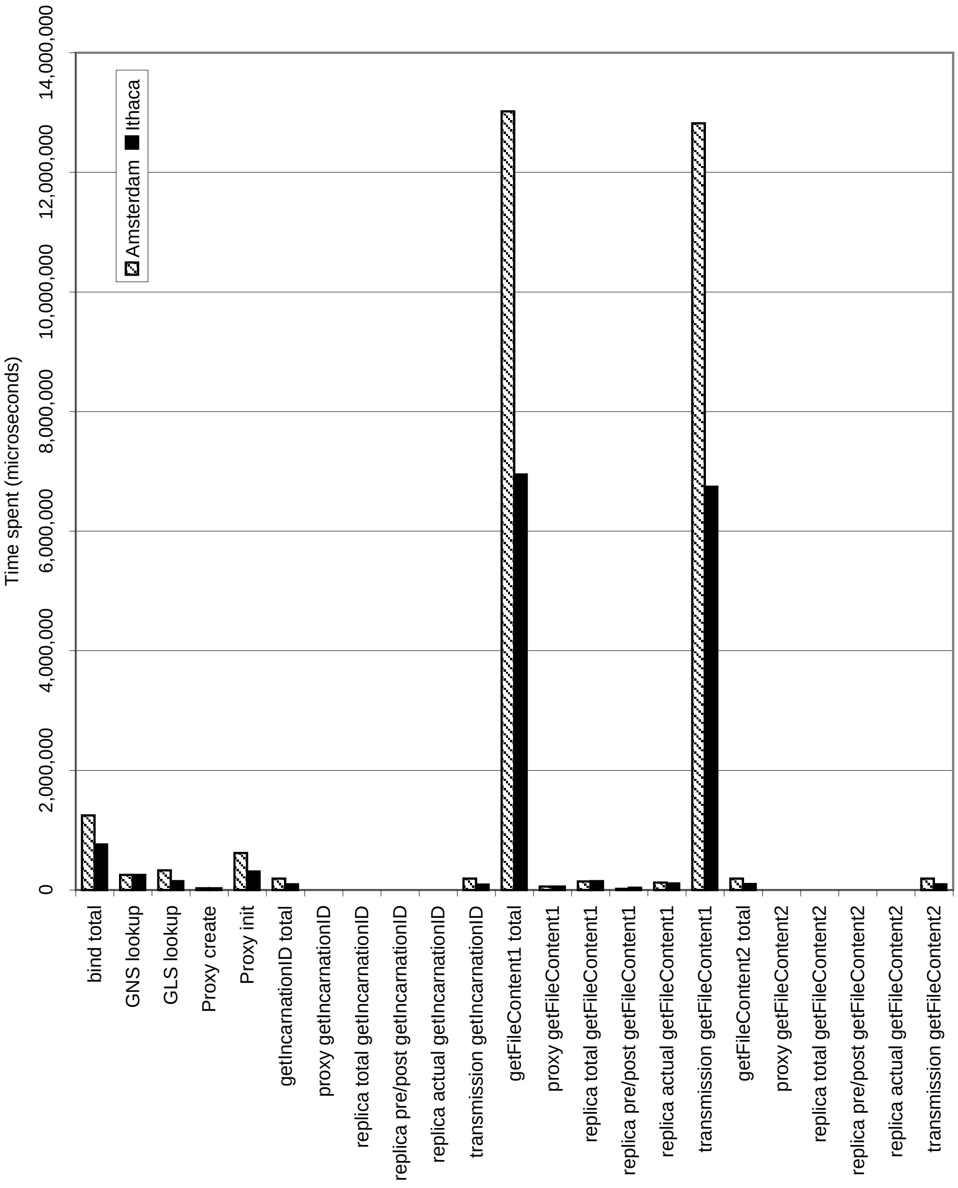


Figure 7.9: Comparison of the Amsterdam and the Ithaca download.

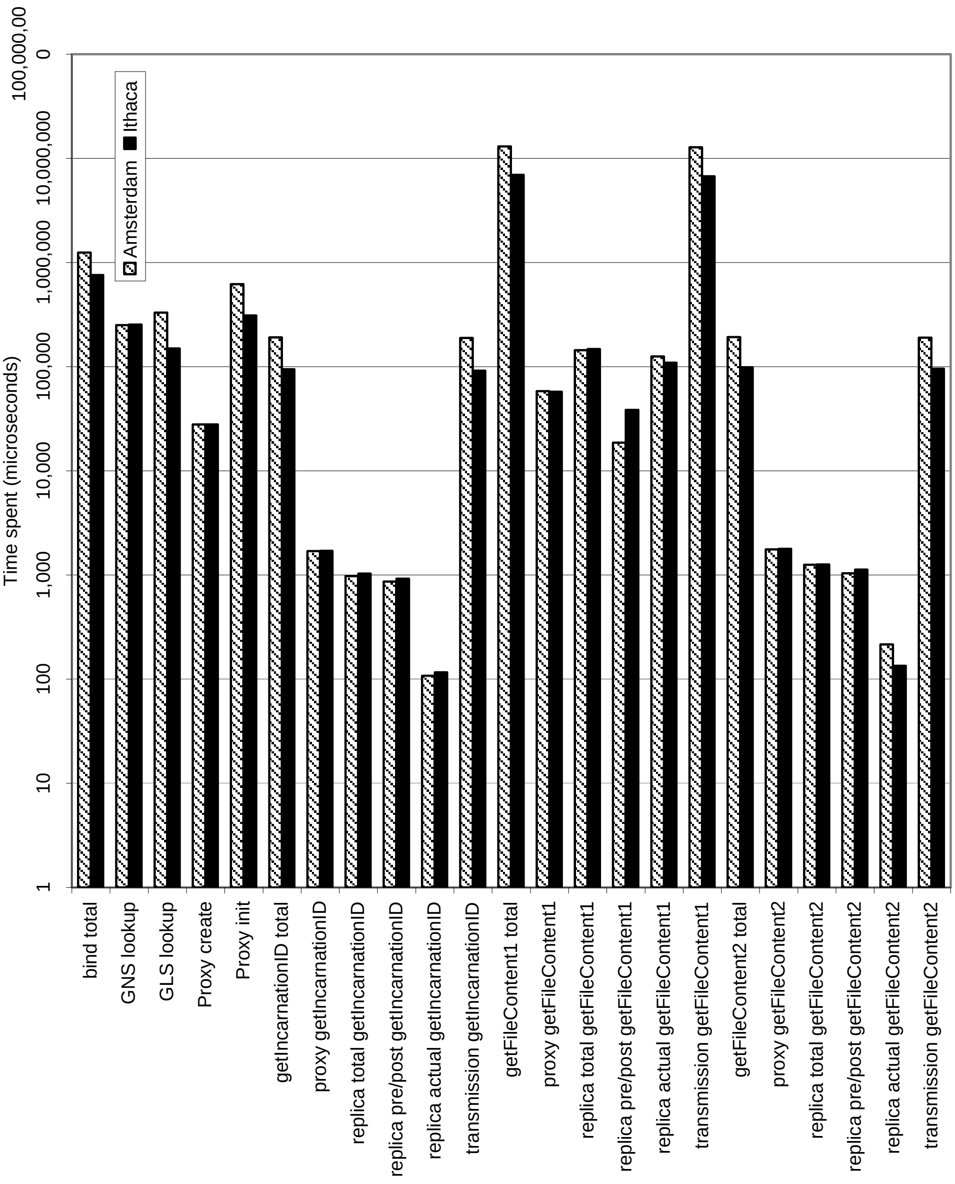


Figure 7.10: Comparison of the Amsterdam and the Ithaca download on a logarithmic scale.

from slave replica in Ithaca) are shown in Figure 7.9. To better compare the fast steps in a GDN download Figure 7.10 shows the same data as Figure 7.9 but with the times plotted on a logarithmic scale.

As can be seen in Figure 7.10, binding time is considerable for the download from Amsterdam. The lookup in the Globe Name Service is high, in this experiment, as it involves contacting a machine in Amsterdam. The lookup in the Globe Location Service also requires communication over the transatlantic link. Finally, inter-continental communication is also required to initialize the proxy, as initialization of the proxy involves setting up a TCP connection with the replica and exchanging a number of messages, which adds up significantly when using a high latency link.

The binding time when downloading from Ithaca is lower, as expected. This is mainly due to proxy-initialization time being lower, as the replica to connect to is on the same continent. Furthermore, because of the property of the Globe Location Service that lookup times are proportional to the distance between client and nearest replica, the GLS lookup time is also significantly lower; in fact, it is reduced by 55%.

The method-invocation times for both Amsterdam and Ithaca are dominated by transmission times. These transmission times, in turn, are dominated by the amount of data to be transferred. For the getIncarnationID method of the revision DSO there are few data to be sent, as the method does not have large input or output arguments. As a result, the transmission time for the method is almost equal to the round-trip times measured using ICMP ping (see Table 7.5). The transmission time for the first invocation of getFileContent is much higher, as it involves transferring the 1 MB file from the replica to the proxy. The transmission time of the second getFileContent invocation is small, as it does not transfer any file contents, but merely signals that the whole file has been transferred.

If we ignore transmission times, Amsterdam and Ithaca differ in the times used for marshalling and passing the 1 MB block between subobjects inside the replica, as well as the time required to execute the actual method invocation. An inspection of the measurements reveals that the average pre/post invocation time of Ithaca is affected by a few outliers (11 measurements larger than 50,000 μs), caused by other processes being started on the Ithaca machine during the course of the experiment. It is therefore better to look at the medians which are 18,372 μs for Amsterdam and 9,194 μs for Ithaca. The difference between these pre/post invocation times is due to the different versions of the Java Development Kit (JDK) being used (1.3.1 vs. 1.4.0), as established by repeating part of the experiment with JDK 1.3.1 on the Ithaca machine. Local scheduling also affected the average actual invocation time in Amsterdam and, to a lesser extent, in Ithaca. The median values are 78,536 μs for Amsterdam and 99,234 μs for Ithaca. The difference is

Table 7.5: ICMP Ping times between the test sites. These numbers are the average of 1000+ pings.

From	ping to Cornell (μs)	ping to Amsterdam (μs)
San Diego	87,779	189,009

most likely due to the different versions of the JDK and Amsterdam's faster processor (502 vs. 440 MHz). The disks used in both machines are comparable (both 7200 RPM ATA/66 disks), although Amsterdam's disk has a faster seek time (1 ms faster).

The time spent in the proxy of the revision DSO is rather high for the first getFileContent method invocation, compared to the pre/post invocation time in the replica. The high value is due to more frequent copying of the 1 MB block inside the proxy during marshalling, which takes considerable time on Sun JDKs.

For completeness, Table 7.6 shows the actual measurement data. Table 7.5 lists the round-trip times between the test sites, measured using ICMP Ping messages, for reference.

Table 7.6: Results for GDN with blocksize 1 MB.

Interval name	Download from Amsterdam (μs)	Download from Ithaca (μs)
Bind total	1,245,384	759,059
GNS-lookup	250,247	254,070
GLS-lookup	330,223	149,763
Proxy-create	27,867	27,819
Proxy-init	620,330	310,633
getIncarnationID total	191,718	94,442
Proxy getIncarnationID	1,696	1,702
Replica total getIncarnationID	977	1,034
Replica pre/post getIncarnationID	868	919
Replica actual getIncarnationID	108	116
Transmission getIncarnationID	189,045	91,705
getFileContent(first block) total	13,020,547	6,945,754
Proxy getFileContent(first block)	58,251	57,365
Replica total getFileContent(first block)	143,963	147,341
Replica pre/post getFileContent(first block)	18,585	38,349
Replica actual getFileContent(first block)	125,378	108,992
Transmission getFileContent(first block)	12,818,332	6,741,047

Table 7.7: Summary of GDN test results.

Interval name	DL from Amsterdam (microseconds)	DL from Ithaca (microseconds)
client end-to-end	14,707,742	7,952,904
bind total	1,245,384	759,059
total transmission	13,196,793	6,928,361
total processing	265,565	265,484

Table 7.6: Results for GDN with blocksize 1 MB.

Interval name	Download from Amsterdam (μs)	Download from Ithaca (μs)
getFileContent(last block) total	192,436	98,654
Proxy getFileContent(last block)	1,767	1,784
Replica total getFileContent(last block)	1,252	1,261
Replica pre/post getFileContent(last block)	1,038	1,127
Replica actual getFileContent(last block)	215	134
Transmission getFileContent(last block)	189,416	95,609

Summary

Table 7.7 summarizes the measured transmission and processing times. The client end-to-end time is the end-to-end download time as perceived by the user (taken from Table 7.4). The bind times are taken from Table 7.6. The total transmission time is the aggregate of all transmission times of the method invocations on the revision DSO required for the complete download. The total processing time is the sum of the time spent in the client itself, in the proxy of the revision DSO, and in the replica of the revision DSO. This time is calculated by subtracting the total transmission and bind times from the end-to-end download time.

This table shows that transmission time is the largest factor in both downloads. Processing times are the same for Amsterdam and Ithaca, which is not surprising, as the sites use rather similar hardware. As expected, binding time is significantly lower for a download from the slave replica in Ithaca. There may be room for improvement of binding times in general, as it may be possible to reduce the number of messages exchanged between proxy and replica at proxy-initialization time.

Table 7.8: Comparison of end-to-end performance for HTTP downloads from Amsterdam and Ithaca.

HTTP Test	Average download time (min–max) in μs	Average throughput (min–max)in KB/s
1 MB download from Amsterdam	13,181,869 (12,988,514–13,472,358)	77.7 (76.0–78.8)
1 MB download from Ithaca	6,739,778 (6,512,681–7,068,944)	151.9 (144.9–157.2)

Table 7.9: Comparison of end-to-end performance for HTTP and GDN with blocksize 1 MB.

Test	HTTP	GDN	GDN/HTTP
1 MB download from Amsterdam	13,181,869 μs = 77.7 KB/s	14,707,742 μs = 69.6 KB/s	+ 11.6%
1 MB download from Ithaca	6,739,778 μs = 151.9 KB/s	7,952,904 μs = 128.8 KB/s	+ 18.0%

7.2.3. Comparison to HTTP

To compare GDN to HTTP in a wide-area environment, Apache servers were set up on the machines in Amsterdam and Ithaca and 99 downloads of a 1 MB file were done from each server from the machine in San Diego using the wget HTTP client. These tests were conducted interleaved with the GDN tests on the weekday in March 2002. Table 7.8 shows the results.

Table 7.9 compares the results of the HTTP tests to the GDN tests. HTTP outperforms GDN in both the Amsterdam and Ithaca case. The difference between HTTP and GDN is mainly due to HTTP not having a binding step before the actual download, in both cases. If we ignore the binding step the difference is 2.1% (280,489 μs) in the Amsterdam case, and 6.7% (454,067 μs) in the Ithaca case.

These differences, referred to as the *end-to-end differences*, can be explained as follows. Recall that a GDN client contacts the object server three times: once to retrieve the file's incarnation ID, once to retrieve the 1 MB file, and once to read the (empty) remainder of the file (the TCP connection over which these request travel is setup in the binding step and therefore not counted). The HTTP client contacts the server only twice: once to setup a TCP connection and once to retrieve the 1 MB file. The differences can now be attributed to four factors. The calculation of the difference is shown in Table 7.10.

First, before connecting to the HTTP server the HTTP client does a DNS

Table 7.10: Explanation of the difference between HTTP and GDN.

Factor	Amsterdam (μs)	Ithaca (μs)
HTTP does DNS lookup	–14,027	–12,947
TCP connect faster than getIncarnationID	+4,983	+5,607
GDN downloads and writes file slower than HTTP	+98,500	+364,413
GDN does more network communication: getFileContent(last block)	+192,436	+98,654
Total	281,892	455,727
End-to-end difference	280,489	454,067
Unexplained	1,403	1,660

lookup to resolve the name of the server to an IP address, which takes 14,027 μs in the Amsterdam case, and 12,947 μs in the Ithaca case. The GDN does not do such a lookup, so this factor increases the differences.

Second, an HTTP client sets up a TCP connection slightly faster than a GDN client retrieves a file's incarnation ID. This factor explains 4983, respectively, 5607 μs of the difference between GDN and HTTP.

Third, GDN is slower in the time it takes to send the download request to the server, read the 1 Megabyte from disk, transfer it to the proxy and write it to disk there. More specifically, GDN is 98,500 μs slower in the Amsterdam case, and 364,413 μs slower in the Ithaca case. This factor explains why the difference between GDN and HTTP is larger in the Ithaca case (18.0% vs. 11.6%). The explanation for GDN being slower is two-fold. (1) GDN has more overhead as it does marshalling and unmarshalling, does more (inefficient) copying of the 1 MB block, and uses slower disk I/O (the Apache server on Solaris uses memory-mapped files). (2) in case of Ithaca, the experiment most likely suffered from congestion in the network, which caused TCP to limit its throughput.

The fourth factor explaining the differences is the fact that a GDN client contacts its server one additional time compared to HTTP, in particular, to read the (empty) remainder of the file. The extra round trip to the server makes GDN 192,436 μs slower in the Amsterdam case, and 98,654 μs in the Ithaca case.

These four factors explain most of the differences between HTTP and GDN. The small remaining discrepancies are attributed to the HTTP client being a little slower in pre- or post-download processing.

CHAPTER 8

Related Work

This chapter discusses the designs of a number of systems for distributing data to a worldwide audience, and compares them to the design of the Globe Distribution Network. For each system we give a general introduction, explain how the system achieves scalability and efficient distribution, and describe its security and fault tolerance features. The systems discussed are:

- The early file-sharing systems, such as AFS, Coda, FTP and HTTP (Sec. 8.1).
- Freeflow/EdgeSuite, a content distribution network operated by Akamai Technologies, Inc. (Sec. 8.2).
- RaDaR, an architecture for content distribution networks developed by AT&T (Sec. 8.3).
- Freenet, a peer-to-peer file-sharing system for anonymous publication and retrieval of information (Sec. 8.4).
- CFS, a fully decentralized peer-to-peer file-sharing system developed at MIT (Sec. 8.5).
- PAST, a largely decentralized peer-to-peer file-sharing system (Sec. 8.6).
- OceanStore, a worldwide distributed storage system developed at the University of California at Berkeley (Sec. 8.7).

8.1. EARLY SYSTEMS

The Andrew File System and Coda

One of the first systems to allow transparent file sharing over a wide-area network was the *Andrew File System (AFS)* [Howard et al., 1988; Satyanarayanan, 1990; Zayas, 1991]. File sharing is transparent in AFS as file names do not contain information about the location of the file and are valid globally (i.e., there is a single global name space for files). The architecture of AFS is as follows. At the highest level, AFS consists of a collection of administrative domains called *cells*. Each cell consists of a number of file servers and client machines. To read or write a file a client first creates a temporary copy of the (parts of the) file it needs, which are then read or written locally. The file server holding the master copy of the file records some information about the client such that it can invalidate the client's copy when an update is made by another client. For administrative purposes, files are grouped into *volumes* which are collections of related files (e.g. a user's home directory). Volumes may be replicated to improve access performance and availability of frequently-used files. Replicated volumes can, however, not be updated by clients. Clients cannot add, modify or delete files in a replicated volume; updating a replicated volume requires running a separate program and system administrator privileges.

AFS cannot easily be used for large-scale free-software distribution. To efficiently distribute software to a large user community, the files must be available from machines near to these users. The only mechanism in AFS suited for this is volume replication. Volume replication in AFS is, however, rather inflexible as it hard to update a replicated volume. Using this mechanism to do software distribution therefore requires extra administration to keep the number of updates low. For example, all new revisions and variants could be first placed on a read/write volume and moved to a replicated read-only volume once a day. The file caching mechanism in AFS does not scale to large numbers of users. Each user is assumed to have his own machine that caches files individually. The file server holding the master copy may therefore become overloaded if too many users access the file concurrently.

The Coda distributed file system is based on AFS and is more resilient to server and network failures [Satyanarayanan, 1990]. Its replication facilities are more suited for software distribution, as it allows replicated volumes which are mutable and lets a client use a preferred replica server which is near. Coda, however, assumes that the location and number of replicas of a volume do not change frequently. This assumption may restrict Coda's performance when used for large-scale software distribution. In particular, Coda's staticness means it can handle long-term changes in access patterns, but it may not be able to handle short, local peaks in load.

FTP and HTTP

The *File Transfer Protocol (FTP)* is the protocol that is traditionally used to distribute free software. To improve download performance and availability the software is placed on one or more servers, where the number of servers depends on the popularity of the software. Selection of these servers is done manually. As FTP does not support automatic replication of files over multiple servers, several protocols and programs have been developed for this purpose, for example, rsync [Tridgell, 2000]. Server and network load balancing is done by the users. A user will select the server that he expects or knows to have good performance and select another if that one happens to be down or performing badly.

The three largest problems of FTP are (1) no location transparency or automatic fail over, (2) inconsistency between replicas and (3) vulnerability to flash crowds. Users themselves are responsible for selecting the closest server and failing over to another server if that server is unavailable, which is not very user friendly. After a new version of a software package has been uploaded, it takes some time (e.g. several hours) before all replica servers have a copy, which is annoying for users. Flash crowds occur when a new version of a popular package is published and lots of users try to get at it. Often the capacity of the set of FTP servers hosting the package is insufficient to handle this peak load, and there are no facilities for temporarily allocating more server resources. Inconsistency of replicas can augment the flash crowd problem: if the new version is not yet available from all servers, users will focus on the servers that do have them, concentrating their load on this subset of the servers.

With the rise of the World Wide Web, people also started to use the *Hyper Text Transfer Protocol (HTTP)*, as a download protocol for free software. Although HTTP has some facilities for replication (i.e., caching HTTP proxies) and load balancing (i.e., HTTP redirects), these are not adequately used, if at all. As a result, software distribution via HTTP suffers from the same problems as FTP-based distribution. This situation is different for the distribution of content that generates revenue for its publishers, as discussed in the following sections.

8.2. AKAMAI'S FREEFLOW

In recent years, *Content Distribution Networks (CDNs)* have emerged as a way for publishers to deliver up-to-date digital content to their users quickly and reliably, and at lower monetary cost. The best-known content distribution network is offered by *Akamai* under the product names *Freeflow* or *EdgeSuite*

[Akamai Technologies, Inc., 2002; Leighton and Lewin, 2000]. The CDN is commonly referred to as just Akamai. Akamai builds on the Internet's DNS system to direct users' Web browsers to a nearby copy of the requested content stored on servers not operated by the content provider itself but by Akamai. Akamai uses knowledge about the Internet to select a server that is network topologically near to the user, optimizing for access latency.

Publishing of content via Akamai proceeds as follows. A content provider wishing to publish content via Akamai first *akamaizes* his Web pages. A Web page consists of a base HTML document that contains hyperlinks to other pages and references to images, audio or video. Akamaizing a Web page consists of replacing the references (e.g. local file names) in the base document that refer to static content by Akamai URLs. By replacing the references to this static content by Akamai URLs causes it to be delivered not by the content provider's own Web server but by the Akamai content distribution network. For many Web pages, this static content represents the bulk of the content to be transferred to the user when the page is requested, which implies that the content provider itself needs less infrastructure and system administration. The original akamaizing process as described in [Leighton and Lewin, 2000] has been simplified by the use of DNS delegation [Rabinovich and Spatscheck, 2001].

Akamaized content is accessed as follows. Assume the user's browser has an Akamai URL and needs to retrieve the content it identifies. To this extent, the browser starts resolving the server name in the URL to an IP address via DNS. The user's local DNS server delivers this lookup request to a DNS server operated by Akamai. Akamai's DNS server determines the region of the network the user is in by looking at the IP address of the user's DNS server and redirects the request to another Akamai DNS server in that region. This lower-level DNS server knows about the Akamai Web servers operating in its region, and returns the IP address of the server that is close to the user, not overloaded, and likely to already have the required content. Having resolved the server name, the user's browser establishes a connection with the designated Akamai Web server and sends it the remainder of the URL. If the Web server already has the requested content it first verifies that it still holds the latest version using a checksum of this latest version encoded in the URL. If it does not have the content or the content is out-of-date it retrieves the data from the content provider's Web site or another Akamai Web server, and, finally, returns it to the browser.

The scalability and speed of this retrieval procedure is due to two reasons: First, caching of DNS query results by the user's machine and the DNS server of its ISP makes it unnecessary for browsers to contact the Akamai DNS servers frequently. Second, Akamai selects a server that is network-topologically near, ideally located at the user's ISP, thus avoiding shared network links and decreasing access latency.

Akamai has limited facilities for handling flash crowds. It can return a smaller piece of content asking the user to try again later, or provide a ticket (presumably an HTTP cookie) that will allow a client to retrieve the content at a later time, implementing a form of priority queuing for clients. To prevent flash crowds from consuming too many resources, Akamai can impose limits on how many resources specific content is allowed to consume.

Publishers have to pay to put material on the Akamai content distribution network. How access control is done is not disclosed. Akamai does not provide special guarantees with respect to integrity and authenticity of content, as Akamai DNS and Web servers are trusted.

Fault tolerance of Akamai is ensured by using replicated DNS servers and letting the low-level DNS servers return the addresses of multiple suitable Web servers. In addition, each Web server has a buddy, another Web server that takes over its work by taking over its IP address when the former server fails. The available publications do not explain how Akamai guarantees data availability.

Discussion

Akamai is a commercial system and hence not all details about it are public. For example, it is not clear how Akamai ensures that most clients will be directed to a nearby server that already has the desired content. There is evidence that Akamai is generally able to select a server from the set of available servers that delivers the content to the user quickly [Johnson et al., 2001], and that using Akamai instead of the content provider's servers improves access latency [Krishnamurthy et al., 2001] (for small files). On the other hand, Akamai depends on the IP address of the user's local DNS server rather than the IP address of the user's machine directly to determine the best Web server for that user. A recent study shows that, in many cases, the DNS servers that people use are not located nearby, making a server's IP address an inaccurate approximation for the user's location, and thus a less suitable metric for selecting an Akamai Web server [Mao et al., 2002].

Akamai's techniques are patented, and using them requires obtaining a license from the Massachusetts Institute of Technology (MIT). Apparently, the license agreement between Akamai and MIT does not exclude MIT from licensing the technology "for non-commercial [...] purposes" [Akamai Technologies, Inc., 2000]. Obtaining a license is, unfortunately, likely to be only the first step in building a free-software distribution network on Akamai technology, as many details about the actual implementation of Akamai are kept secret.

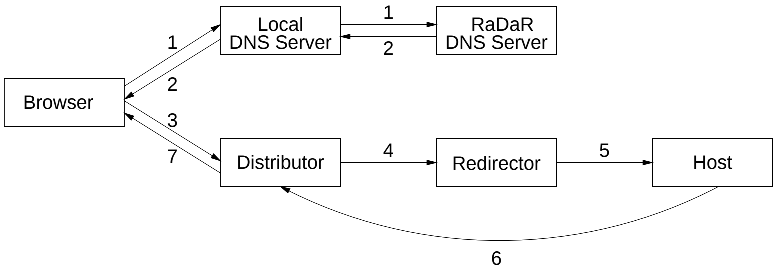


Figure 8.1: Content retrieval in RaDaR (the steps are explained in the text).

8.3. RADAR

Rabinovich and Aggarwal [1999] developed an architecture for content distribution networks called *RaDaR*. The RaDaR architecture aims to reduce the system administration effort required to run the CDN, and to prevent overload of servers and network backbone. To achieve the latter goal it exploits information from the Internet's routing protocols to serve content from a server that is lightly loaded and network topologically near [Rabinovich et al., 1999].

RaDaR does not modify URLs, so a URL of a site that uses a RaDaR-based content distribution network contains no special information. The process of retrieving a file from RaDaR given its URL is illustrated in Figure 8.1. The client machine running the user's browser contacts its local DNS server to resolve the server part of the URL to an IP address (arrow 1). The DNS server for the domain, operated by RaDaR, uses the client's IP address to return the address of a RaDaR *distributor* near to the client (arrow 2). The client connects to the distributor, which acts as an HTTP server and sends it the URL (arrow 3). The distributor determines which RaDaR *redirector* is responsible for keeping track of which hosts have replicas of the file identified by the URL using a hash function mapping URLs to redirector IP addresses. The distributor forwards the URL to the responsible redirector (arrow 4). The redirector, in turn, determines which host is closest to the client machine using information from the Internet's routing protocols, and forwards the URL to that host (arrow 5). The host returns the content to the distributor initially contacted by the client (arrow 6). The distributor returns the content to the browser (arrow 7), completing the file retrieval.

Publishing a Web site via RaDaR is simple from a publisher's perspective. The Web site's contents are placed on a RaDaR server and are subsequently replicated through the content distribution network based on client demand and server loads. Each RaDaR server has a limited view of the paths client requests take through the network. A server uses this information to detect if many clients for a particular file are located far away. If so, it contacts RaDaR's *replication service* to find underutilized servers in the clients' regions and asks them to create replicas of the file. Servers register the fact that they hold a particular file with the RaDaR redirector responsible for the file.

The replication service consists of an hierarchy of *replicators*. There is a leaf replicator in each OSPF area (an Autonomous System usually consists of one or more OSPF areas) that monitors the load of the RaDaR servers in that area. This information is digested and propagated up the hierarchy. This hierarchy of replicators offers a quick and scalable way for finding a lightly-loaded server in a particular network region. In addition to monitoring where clients come from, servers also monitor their own load, and start offloading work by replicating and migrating files to other servers when it gets too high, again using the replication service.

Discussion

RaDaR and the GDN architecture are similar in structure. Both RaDaR and GDN have dedicated services for mapping the location-independent name for a resource to its network topologically nearest replica, and for finding an appropriate server in a specific region (i.e., the replication service and the Globe Infrastructure Directory Service, respectively). As scalable resource location and discovery is a complex research topic, a detailed comparison of the RaDaR services to the Globe services is left to the principal investigators of the latter services (Ballintijn et al. [2001] and Kuz et al. [2002]).

The replication protocols used have similar goals: reducing the load on shared links (i.e., the backbone) and preventing server overload. In principle, RaDaR bases its decisions on detailed knowledge about where in the network requests come from. The current replication protocol for GDN uses a larger granularity (i.e., Autonomous Systems), suggesting that RaDaR will be able to make better decisions about where to place replicas. Simulations with a workload from a Web hosting service show that RaDaR is able to significantly reduce the amount of bandwidth consumed and improves client response times compared to a situation where there is no replication [Rabinovich and Aggarwal, 1999]. However, like Akamai, RaDaR depends on the IP address of the user's local DNS server to determine the user's location, which often is not an accurate measure [Mao et al., 2002]. As a result, RaDaR may place replicas in suboptimal locations.

8.4. FREENET

Freenet is a storage system that enables anonymous publication and retrieval of information [Clarke et al., 2002; Oram, 2001; Clarke et al., 2001]. It is largely decentralized and makes clever use of replication to make it resistant to censorship and to avoid performance bottlenecks.

Information in Freenet is stored as files. The way files are identified leads to two types of files: read-only files identified by a cryptographic digest of their contents, and writable files identified by a *signed-subspace key*. The signed-subspace keys allow files to be linked to a public/private key pair, which can be used to provide authenticity. The cryptographic digest that identifies a read-only file is called a *content-hash key*. The suggested method for publishing information that is subject to change is to create a subspace-signed file that contains the key of the read-only file containing the current version of the information.

Files are retrieved from the network of Freenet servers as follows. To retrieve a file a client sends a lookup request containing the file's key to a nearby server. This server looks in its cache of files to see if it already has a copy of the requested file. If this is not the case, the server forwards the request to another Freenet server based on the file's key. The lookup is forwarded to another server until the file is found or the time-to-live value of the lookup has been exceeded. If a server has already seen this request it returns an error, which causes the forwarding server to try another server. The same happens when a chosen server is down. When the file is found it is returned to the client along the path of servers traveled by the lookup request. Servers along the path place the file in their own cache.

Files are uploaded into Freenet in a similar fashion. After the user has chosen a file type (read-only or subspace-signed) and computed the file key, he sends an insert request to a nearby server. The insert request consists of just the file key and the number of servers the file should be stored on. Servers decrement this number and route the insert request to other servers based on the file's key until the required number of servers has been reached, at which point an OK message is returned to the requester along the path of servers. In response to this message the uploading client sends the actual data which are propagated along the established path and stored in each server's cache. File caches are the only storage space in Freenet, that is, data does not have a permanent storage location.

Freenet's routing and replication mechanism ensure that files are usually found after querying only a few servers. The routing mechanism is based on a list of (file key, server network address) pairs that each server maintains locally. Lookups are forwarded to the server whose associated file key is lexicographically nearest to the key of the file requested. The list of (file key, server address) pairs is continuously updated as a server processes requests in such a way that the server chosen as the next step in the path is the server that is able to locate the file in a short number of forwarding steps. The details of this procedure are outside the scope of this dissertation, see [Clarke et al., 2001].

Freenet takes the following measures against attackers trying to modify or delete files. Writable files identified by subspace-signed keys can be updated only by the owner of the associated public/private key pair. In particular, a server will modify a file only if the new version is signed and is more recent. Read-only files are protected against overwrites by Freenet's upload mechanism. If during the initial phase of an upload a server discovers it already has an entry for the file in its database it changes the insert request into a lookup and returns the contents of the old file to the uploading client. Freenet's current design is, however, not completely protected against outside attackers trying to remove content. There is a chance that uploading lots of junk content into Freenet can cause already stored content to be lost, as a Freenet server may discard unpopular files to make room for new uploads. Malicious modification of content by internal attackers (i.e., malicious servers) is possible but always detected as the integrity of read-only files is protected by their naming scheme, and that of writable files is protected by means of digital signatures created with the file's associated public/private key pair.

Anonymity of uploaders and downloaders is provided by Freenet's hop-by-hop forwarding. Hop-by-hop forwarding makes it hard to determine the original source or ultimate destination.

Freenet is able to handle server and network failures. Its routing mechanism ensures that other servers are tried when the most optimal server is down or cannot be reached. Availability of files is ensured by Freenet's aggressive replication strategy.

Discussion

At first glance, Freenet appears to be a suitable system for distributing free software because it has a number of interesting properties. First, Freenet is self maintaining. Self maintenance means there is less system administration to be done, which means the system is easier to use and, important for free-software distribution, cheaper to operate. Second, Freenet uses untrusted servers which makes it easier to set up a large-scale software distribution network. Third, Freenet aggressively replicates content which enables it to handle flash crowds.

What makes Freenet less useful for free-software distribution is its performance. In theory, performance is good as Freenet lets users find files after consulting only a few servers. However, Freenet routing is based on file keys and not on network performance. As a result, it is unlikely that Freenet will make optimal use of the underlying network. For example, a server that is logically only a few hops away can be located on the other side of the world. Although the choice of routing mechanism is justified by Freenet's design goals, it does make Freenet less attractive for large-scale software distribution.

As argued in Chap. 2, Freenet's facilities for anonymous retrieval and publication of software are useful for a software distribution network. Anonymous retrieval protects the privacy of users and anonymous publication enables authors to publish controversial software without risk of personal prosecution. However, in Freenet publication is anonymous by default, which I consider unsuitable for publicly accessible distribution network, as it fosters large-scale abuse (see Sec. 5.1).

8.5. THE COOPERATIVE FILE SYSTEM

The *Cooperative File System (CFS)* is a file storage and sharing system that is fully decentralized yet offers provable guarantees with respect to efficiency of file retrieval and storage, server load balancing and robustness [Dabek et al., 2001b].

Users perceive CFS as a collection of read-only file systems each identified by a public key. Users can add new file systems, and replace or remove an existing file system if they know the private key associated with its public key. The abstraction of a collection of file systems is implemented on top of a storage layer called DHash. DHash implements a distributed hash table that reliably stores blocks of data and allows for fast retrieval of those blocks. This latter property is primarily due to DHash's tight cooperation with a decentralized location service called Chord [Stoica et al., 2001; Dabek et al., 2001a]. Chord allows DHash to quickly locate servers hosting the persistent replicas of the desired block and has the property that lookups for a particular block travel a similar path of CFS servers. The tight cooperation of DHash and Chord allows DHash to exploit this property and cache the data block on servers on the lookup path, speeding up future retrievals by other clients. The architecture of CFS is shown in Figure 8.2.

Files are retrieved from CFS as follows. Assume the CFS client knows the identifiers of the blocks that make up the desired file. The CFS client starts fetching these blocks from DHash asynchronously starting with a small number of blocks and requesting the next block whenever a requested block comes in (i.e., the CFS client does prefetching to speed up retrieval). The DHash client code issues a Chord lookup for each block. Chord starts routing this lookup request towards the servers hosting the persistent replicas of the block based on the block's identifier. At each routing step, a check is made to see if the desired block is cached on the local server. If so, the block is returned to the client; otherwise the lookup continues until a server hosting a replica is reached. After receiving the block, the client sends a copy of the block to (at present) each server on the lookup

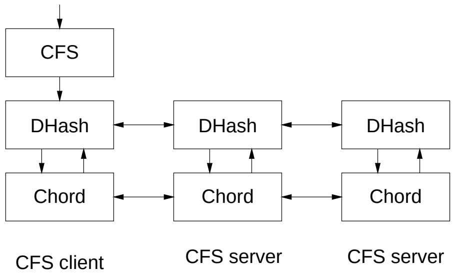


Figure 8.2: The architecture of CFS, taken from [Dabek et al., 2001b].

path just traveled, which then cache the block.

Chord implements fast lookups by combining a scalable routing mechanism based on fixed-length identifiers with measured network latency between servers. Without going into further detail, Chord allows a server storing a particular block to be reached in a number of steps that is in the order of the logarithm of the total number of servers in the system (i.e., the number of steps is $O(\log N)$ where N is the total number of servers). In addition, at each routing step, Chord can choose from a number of servers as the next step in the lookup. Chord will select the server that results in the fastest lookup in terms of network latency. In short, Chord will find one of the servers storing the desired block in a small number of steps, where "small" is relative to the total number of servers in the system.

DHash stores blocks as follows. Each block is identified by an SHA-1 hash of its contents. Using Chord, this identifier is mapped to a specific set of servers on which the block is subsequently stored. Servers can control the number of blocks they may be asked to store and thus their load. However, they cannot cause Chord to assign them a particular set of blocks, providing protection against malicious servers trying to make certain content unavailable. Chord currently has no protection against malicious servers misrouting lookups, but its designers are confident Chord can be made to deal with this kind of attack. To protect CFS from users trying to make the system unavailable by loading large amounts of junk content, it imposes quotas. In particular, a CFS server will allow each client, identified by its IP address, to use only a small percentage of its storage space.

CFS builds on earlier work to provide authenticity and integrity of the file systems it stores [Fu et al., 2000]. The integrity of blocks in CFS is protected against malicious servers by the fact that blocks are identified by an SHA-1 hash of their contents. As file systems are completely stored as blocks, this mechanism recursively ensures the integrity and consistency of the whole file system. The origin (authenticity) of a file in a file system can be established using the public key identifying the file system. In other words, any file in CFS can be traced to a particular public/private key pair.

To improve data availability, DHash replicates each block on a set of servers. This set is chosen such that the Chord lookup algorithm will home in on this set of servers when asked to locate that particular block. The DHash/Chord cooperation ensures that a nonfaulty server is selected automatically. If a server fails, DHash creates a replacement replica on a server with similar properties to maintain the degree of replication for the block. Because of the way Chord identifies servers, the set of servers storing a particular block are not likely to be physically close, and thus not likely to fail, leave the system or become unreachable at the same moment. The Chord location service itself is extremely fault tolerant and retains its fast performance even if there is massive server failure (e.g. 50% of the servers failing).

Discussion

The designers of CFS explicitly mention software distribution as one of its applications. Looking at its replication and caching facilities and location service it appears well suited for the job. Downloads will use the copies of the blocks near to the client in terms of network latency, and dynamic caching ensures that it can handle flash crowds.

CFS is, however, limited and inflexible when it comes to replica placement. The replication and caching policy of CFS is strongly tied to the properties of its location service, and hardwired into the system. GDN is more flexible and can gather more information about client location, access patterns, and network conditions. As a result, GDN may be able to make better decisions about where to create copies of the data. In addition, GDN's replication policy can be changed without significant impact on the implementation, as GDN is built on top of Globe. As it is yet unclear what the best policy is for worldwide replication of free software, the flexibility of GDN is considered an advantage.

What is currently lacking in CFS is support for preventing illegal distribution of copyrighted or illicit material, which is present in GDN. Adding such functionality, in particular my cease-and-desist scheme appears relatively simple, as follows. Assume there is a single access-granting organization that certifies public keys for publishers and revokes these certificates when the publisher publishes illegal content. The publisher of a CFS file system uses his certified key pair to sign every block he publishes through CFS, in such a way that a server can verify the origin of a single block in the file system without having to retrieve other blocks. When a server is asked to store or cache a block, it not only verifies the integrity of the block by computing the SHA-1 hash of its contents and comparing it to the block's ID (as is done now), but also checks that the block is signed by a publisher certified by the CFS' access-granting organization, and that this publisher is not blocked at present. The latter can be done, for example, by reading a certificate-revocation list published by the CFS AGO via CFS. Servers should also periodically perform this check for blocks they already store or cache to make sure they don't serve content of publishers that have been blocked after the server accepted the block.

The advantages of CFS over GDN are its decentralized architecture, making the application easier to maintain (GDN requires an organization to maintain Globe's middleware services, such as GLS and GIDS), and its support for untrusted servers.

8.6. PAST

PAST is a largely decentralized storage and data sharing system similar to CFS in architecture and techniques used to achieve scalability and good performance [Druschel and Rowstron, 2001; Rowstron and Druschel, 2001b]

The PAST API offers three operations: upload a file, retrieve a file and stop persistent storage of a file. The last operation is basically a delete operation with weak semantics, as users may still be able to retrieve the file for some time after this operation. Files are immutable and each one is owned by a single user. A file is identified by the cryptographic hash of the file's textual name, the owner's public key and a random value. Associated with each file is a *file certificate* that can be used to verify the integrity and authenticity of the file.

Like CFS, PAST works in close cooperation with a fault tolerant, decentralized location and routing service for fast file retrieval, in this case called Pastry [Rowstron and Druschel, 2001a]. Pastry has properties similar to CFS's Chord: it is able to locate a server hosting a replica quickly, and lookups for a particular file partially visit the same servers. Like CFS, PAST uses its cooperation with its location service to cache the requested file on the commonly visited servers to speed up future retrievals.

As PAST and CFS are so similar in architecture and techniques used we describe only their differences here. PAST stores, replicates and caches complete files instead of blocks. This is likely to affect performance, although it is not clear how [Rowstron and Druschel, 2001b]. It does require PAST to have a more sophisticated storage management system, as the distribution of files over available servers is uniform, but the sizes of the files differ greatly, implying that some servers may have to store much more data than others. PAST servers can therefore offload file replicas to servers nearby, or refuse to store a file altogether when storage is already low. In the latter case, PAST will automatically select a new group of servers to create replicas on. Simulations suggest the storage management system is very effective as it allows almost all available storage to be used. Furthermore, PAST uses trustworthy smart cards to impose storage quotas on users instead of restricting storage to a certain percentage of the available storage per IP address.

Security in PAST, in general, currently revolves around the use of smart cards. One must obtain a smart card from a smart card issuer to use PAST for publishing or to host a PAST node. A smart card contains a public/private key pair that is certified by the card issuer, and, when used by a publisher, a specific storage quota. The smart card is used to securely generate file certificates and server identifiers, and to administer a user's storage quota.

In particular, when a publisher's smart card creates a file certificate it is signed with the public/private key pair, and the storage required by the file and its replicas is subtracted from the user's quota as stored on the card. A server asked to replicate the file has its smart card check that the signature on the file certificate was made by a trusted smart card, and thus establishes that the publisher is allowed to use PAST (i.e., the smart cards do access control). In addition, if the signature is from a trusted smart card (e.g. by the same issuer) the server also knows that the storage it provides for the file was debited against the publisher's quota. When deleting a file, the user's smart card creates a *reclaim certificate* for the file that is also signed with the key pair, and which is sent to the servers storing the persistent replicas. These servers' smart cards compare the signature on reclaim certificate against that on the file certificate, and release the file only if they match. When releasing a file they return a *reclaim receipt* to the user which he can use to increase the quota on his smart card. Smart cards are assumed to be tamper proof.

The stated advantages of using smart cards is that

- they prevent certain denial of service attacks by malicious users and servers,
- they maintain storage quota efficiently and securely, and
- they allow users to anonymously join PAST as publisher or storage provider [Druschel and Rowstron, 2001], for example, by selling the cards for cash in stores.

Discussion

PAST's storage management system allows for high utilization of the available persistent storage. This property means that PAST is able to operate with less storage than others. It is unclear if the current GDN design will achieve high storage utilization. In GDN, the publishers of software select the location of the revision DSO's core replicas which may or may not cause a reasonable distribution over the available servers. Furthermore, for regular replicas, GDN first selects the network location to create the replica in and then looks for a server in that location. It is also not clear to what level of utilization this procedure leads.

As mentioned above, there are several advantages associated with using smart cards. These advantages, however, will not greatly benefit the GDN, and therefore employing smart cards in GDN would not be a large improvement. The denial of service attacks by malicious servers that are prevented by the use of smart cards are specific to PAST and Pastry. In particular, the use of a smart card prevents malicious servers from becoming the preferred server for storing a particular piece of content which would then enable them to deny access to that content. GDN needs to impose quotas only on the number of revision DSOs a publisher can create; fair allocation of storage and other resources is done automatically. For this small task it can use a centralized service without creating a scalability problem, and does not require a decentralized smart card-based solution. Finally, GDN does not support the fully anonymous publication enabled by anonymously bought smart cards. GDN forbids fully anonymous publication as this conflicts with its measures for preventing illegal distribution of content.

Like CFS, PAST is limited and inflexible when it comes to replica placement, as its replication and caching policy is also strongly tied to the properties of the Pastry location service, and hardwired into the system.

8.7. OCEANSTORE

OceanStore is a worldwide distributed storage system developed at the University of California at Berkeley [Kubiatowicz et al., 2000; Rhea et al., 2001]. Its purpose is to provide persistent storage and data sharing to 10 billion users connected to the Internet through various sorts of wired and wireless devices. To keep a storage system of this scale manageable it has been designed to be self maintaining, that is, tasks that generally require intervention by a system administrator have been automated. In particular, OceanStore automatically incorporates new servers into the system, is able to recover from server and network failures without intervention and adjusts itself to changing usage patterns. In addition, OceanStore has been designed to run on untrusted servers and be highly available.

Clients access OceanStore indirectly through applications built on the basic OceanStore API. This API is based on a model of objects consisting of blocks of data that can be read or written. The implementation of these objects is, however, highly sophisticated to meet the self maintenance and high availability requirements and to deal with the fact that servers cannot be fully trusted.

The model of an object that can be both read and written is implemented by means of immutable objects. Writing to an object creates a new immutable object. Each object has a group of servers, called its *inner ring* that records which immutable object represents the current version of the object. Immutable objects make it easier to recover from failures. The immutable objects are stored persistently across a large number of servers via *erasure coding techniques*. Erasure coding techniques, such as interleaved Reed-Solomon codes, split a data block into n pieces and transform these into $2n$ or $4n$ fragments [Plank, 1997]. The interesting property of these techniques is that any n of these fragments are sufficient to reconstruct the original data block. The use of these techniques and distributing the fragments over a large number of servers yields an extremely durable storage system.

Reconstructing an immutable object from n fragments is, however, computationally intensive and thus time consuming. To provide fast access to data, OceanStore allows servers to host complete replicas of immutable objects. The servers that form the inner ring of an object each host a *primary replica*. Other servers can host so-called *secondary replicas*. OceanStore's message routing and location service, called Tapestry, ensures that clients access the nearest replica (in terms of network latency).

OceanStore actively manages the degree of replication and the location of replicas. First, when a replica receives too many requests OceanStore creates additional replicas nearby. Second, OceanStore can detect if clients are accessing replicas that are far away and create replicas nearer to these clients. Finally, it has a mechanism for discovering which objects are often accessed together. This information can be used to improve performance by replicating or migrating objects in clusters. OceanStore's automatic replication mechanism also enables OceanStore to handle replica failures autonomously.

The integrity of data stored and shared via OceanStore is guaranteed by means of cryptographic digests. In particular, immutable objects are identified by globally unique identifiers consisting of a cryptographic digest of its data. This form of identification allows clients to do an end-to-end integrity check on the data retrieved from OceanStore.

Updates to an object are performed as follows. The client sends the update to the object's inner ring and to a number of secondary replicas. The inner ring checks for write permission and perform the update on the primary replicas if the update is allowed. An interesting property of the update mechanism is that it also operates on ciphertext (as servers may be untrustworthy, all client data stored in OceanStore is generally encrypted). Next, the inner-ring servers execute a Byzantine agreement protocol to decide whether or not to commit the update. A Byzantine agreement protocol is necessary to deal with untrusted servers inside the object's inner ring. When the commit is approved, the servers send out an update command to all secondary replicas, which are organized in an application-level multicast tree. This update command is signed by the inner- ring servers using a special cryptographic technique that produces a valid signature if sufficiently many servers are trustworthy and functioning. In addition to sending out committed updates, the inner ring erasure-codes the data and distributes the fragments to a large number of servers.

In parallel to the inner ring's activities the secondary replicas that also received the update from the client distribute this update to the other secondary replicas via the application-level multicast tree. How these replicas react to this message depends on the consistency requirements that were specified for the client. OceanStore objects can provide different consistency guarantees to clients, ranging from weak to single-copy semantics, similar to session guarantees in the Bayou system [Terry et al., 1994]. What consistency is required can be specified by the application programmer via the OceanStore API. For some consistency models a secondary replica will tentatively perform updates instead of waiting for the committed update from the inner ring.

Users get access to OceanStore via an *OceanStore Service Provider (OSP)*. Each OSP contributes a number of servers to OceanStore. Access to OceanStore is not free. Users pay, for example, a monthly fee to their OSP. OceanStore does not explicitly support anonymous publication or retrieval at present.

Fault tolerance is built into OceanStore from the ground up. OceanStore has its own communication and routing layer which allows communication between servers even when many network links have failed and normal Internet routing would not allow communication (in simulations). This routing layer is part of Tapestry, which also acts as OceanStore's location service. The location service is used to locate the inner ring of an object, its replicas and its fragments. This location information is also stored such that it can survive server failures. As explained above, user data is made durable and highly available via erasure codes and replication. OceanStore has additional mechanisms to ensure that data remains available even over a long period of time and many server failures. The Byzantine agreement protocol and special signature scheme for authenticating commits ensure that updates on an object can progress despite failures of inner-ring servers.

Discussion

Like the Globe Distribution Network, OceanStore optimizes network usage by connecting clients to nearby replicas and creating replicas near to clients when there are none. They both employ a location service that is able to find the nearest replica at cost proportional to the distance between the client and this nearest replica [Kubiatowicz et al., 2000].

OceanStore is, however, much more ambitious than the GDN. It has robust routing that can deliver messages in cases where normal IP routing fails. It provides extremely durable storage of information, even checking disks for signs of imminent failure. It is designed from the ground up to be self-maintaining and works around any malicious servers. What the OceanStore design, unfortunately, does not address is measures for preventing illegal distribution, one of the most pressing problems of today's worldwide file sharing systems. It should be possible to apply my cease-and-desist scheme in OceanStore, similar to how it can be applied in CFS (see Sec. 8.5).

CHAPTER 9

Summary and Conclusions

This final chapter consists of three sections. Sec. 9.1 summarizes the previous chapters. Sec. 9.2 presents my observations from the research conducted. Finally, Sec. 9.3 discusses future directions for the research.

9.1. SUMMARY

Chapter 1 argues that developing a new large-scale application for the present Internet is a daunting task, because, in addition to the complexity of the functionality of the application, developers of large-scale Internet applications have to deal with complex nonfunctional requirements. These requirements with respect to performance, security, and fault tolerance are introduced by the scale of the application and the properties of the Internet.

The way to make Internet application development less daunting is therefore to provide developers with comprehensive means to meet these complex nonfunctional requirements. This is the approach taken by the Globe middleware. In the Globe middleware, the comprehensive means consist of: (1) a model of distributed objects, called *distributed shared objects*, which allows a developer to separate functional from nonfunctional aspects, and which offers the flexibility to apply any protocol, technique or policy to meet the nonfunctional requirements; and (2) a large library of implemented protocols and techniques. Flexibility with respect to protocols, techniques and policies is considered paramount, as meeting the difficult nonfunctional requirements of Internet-scale applications will require that the solutions best suited to each application are used.

To validate that the Globe middleware is indeed a good platform for developing large-scale distributed applications, several applications have been built on top of it. One of these is the *Globe Distribution Network (GDN)*, which is an application for the efficient, worldwide distribution of freely redistributable software packages, such as the GNU C compiler, the Apache HTTP server, Linux distributions and shareware. In other words, the GDN allows Internet users to efficiently download a copy of the free software packages they are interested in, whenever and wherever they are located. This dissertation describes the design of the Globe Distribution Network, providing insight into how large-scale applications can be designed and built using Globe, and aiming to validate the approach taken by the Globe middleware.

Chapter 2 identifies the functional and nonfunctional requirements for a worldwide free-software distribution network. The fact that many people are using and producing free software, and that this interest is worldwide, requires that the application makes efficient use of the network and is available 24 hours a day. The latter requirement implies that the application must be fault tolerant and resistant to denial-of-service attacks. The domain and nature of the application also make the GDN an attractive target for other types of attack. Malicious users will try to use the GDN as a convenient channel for illegally distributing copyrighted works, such as commercial software or digital music. Furthermore, malicious users may attempt to alter the software being distributed in order to gain access to the machines that run that software. A distribution network such as the GDN should take measures against these attacks.

After providing the necessary background on the Globe middleware in Chapter 3, we go into how the Globe Distribution Network meets the identified requirements in Chapters 4, 5, and 6. Chapter 4 describes how software distribution is made efficient, that is, how the GDN makes sure most software downloads do not use shared wide-area network links on which bandwidth is generally scarce. To achieve this goal, the GDN encapsulates the free software in Globe distributed shared objects (DSOs) that use a replication protocol that replicates the DSO near to the clients. In particular, each revision of a software package is placed in its own *Revision DSO*. To publish large collections of software, such as Linux distributions, *DistributionArchive DSOs* are used that contain a specific revision and variant of the whole software collection in a specific file format.

The replication protocol of the Revision and DistributionArchive DSOs monitors client accesses. For each download, the free-software DSOs record the *autonomous system* the downloading client is located in. The DSOs use this information to periodically evaluate their replication scenario, that is, the number of replicas of the DSO and their location in the network. When the DSO's current replication scenario is not efficient, that is, the downloads from the object consume too much bandwidth on shared wide-area links, additional replicas are created in autonomous systems with many downloads, and underutilized replicas are deleted. A DSO also automatically adapts its replication scenario when a sudden sharp increase in the number of downloads is detected, a common phenomenon with new releases of popular software packages (the so-called *flash crowds* phenomenon).

How the GDN meets its security requirements is discussed in Chapter 5. To prevent illegal distribution of copyrighted works or illicit content, the GDN takes a novel approach. Instead of checking the content of files before allowing them onto the network (which is nearly impossible anyway), files are made permanently traceable to their publisher. When illegal content is found in a file, its publisher's access to the network is permanently revoked and the content removed. The advantage of this approach is that when there are few illegal publications, the amount of work for the administrators of the distribution network is small, an important property for a distribution network that does not generate any revenue. The GDN's architecture also supports other approaches to preventing illegal distribution. Authenticity and integrity of the software distributed via the GDN are considered properties that should be checked by the downloader in cooperation with the publisher, without intervention from the GDN. The GDN protects itself against outside denial-of-service attacks by enforcing a simple access-control model. Chapter 5 also identifies possible attacks on the GDN's availability by insiders (software publishers and server operators), and how these attacks can be countered. The chapter also describes an initial implementation of the security measures for preventing illegal distribution and outside interference.

Chapter 6 addresses how the GDN maintains high availability, is made reliable, and remains manageable despite failures of machines, networks and GDN components. High availability and reliability are achieved by a combination of fault-tolerant servers, exploiting the fact that the distributed shared objects containing the software are replicated for performance, and explicitly introducing extra DSO replicas to guarantee availability and to avoid having to report an error. Manageability is primarily achieved by making strong guarantees about the result of a GDN operation when failures occur. In particular, GDN operations should be atomic with respect to exceptions, that is, an operation should either succeed or fail without altering the state of the application.

To demonstrate the feasibility of the design of the GDN, an initial implementation has been built. Chapter 7 describes a number of small experiments that were conducted with this implementation, aimed at measuring the performance of the GDN compared to software distribution via the HyperText Transfer Protocol (HTTP). In these experiments, the performance of a GDN server was only 10% worse than that of an Apache HTTP server, or less, for large numbers of clients. In an experiment with downloading software via the GDN and HTTP on the Internet, downloads via the GDN were slower mainly due the considerable time required to map a distributed shared object's symbolic name to a contact address of the nearest replica, and creating a proxy of the DSO that connects to this replica. These actions took considerable time, either intrinsically or because the initial implementations of the application and the middleware were not yet complete and fully optimized. The advantages of the GDN over a pure HTTP-based solution are that the GDN automatically locates the nearest replica, automatically handles fail-over to others replicas, and ensures strong consistency for replicas.

Chapter 8 discusses the designs of other systems that are or can be used for worldwide software distribution. In particular, it looks at older systems such as HTTP, FTP, and AFS, modern content distribution networks such as Akamai, and state-of-the-art file sharing systems, such as CFS (a peer-to-peer network) and OceanStore. The older systems are not well-suited for large-scale software distribution, either from a technical or a user-friendliness point of view. Content distribution networks are much better suited as they have solved the problems of the older systems. The latest file sharing systems appear well-suited although some may trade in some efficiency in exchange for a decentralized architecture.

9.2. OBSERVATIONS

I make the following observations from the research described in this dissertation:

1. When designing the interfaces of distributed shared objects one cannot ignore the fact the objects will generally be replicated, nor can one ignore client-perceived performance and scalability.

2. The availability of services such as the the Globe Location Service and the Globe Infrastructure Directory Service considerably facilitates the development of replication protocols for distributed shared objects.

3. If a large-scale distributed application is to be operated by volunteers, the amount of system administration work for most volunteers should be low, and there should be no legal risks to these volunteers for participating in the application.

4. Distributed shared objects can be easily made secure using a TLS library.

5. A large-scale distributed application with many users may need to protect itself against abuse by these users as thoroughly as it protects itself against outside abuse.

6. Modeling an application in terms of services and dependencies between services is valuable while designing the measures to make a Globe-based application fault tolerant.

I will present these observations in more detail by going over the four steps that constituted the design process of the Globe Distribution Network:

1. Making software distribution fast and efficient.

2. Preventing illegal distribution, ensuring the integrity and authenticity of the software distributed, and anonymous publication and download (i.e., meeting the application-level security requirements).

3. Adding measures against external or internal denial-of-service attacks to the mechanisms for efficient distribution.

4. Ensuring the software distribution mechanism is available and reliable, and easy to manage despite failures.

9.2.1. Step 1: Making Distribution Fast and Efficient

Step 1 in the design process of the GDN consisted of the following substeps:

1. Defining a mapping from application domain (free software distribution) to DSOs. This step is prescribed by the Globe middleware, as all services and shared data must be implemented by Globe distributed shared objects.

2. Defining the interfaces for each type of DSO which includes defining the semantics of the methods in these interfaces.

3. Investigating how to efficiently deal with DSOs with large state.

4. Selecting a replication protocol for each type of DSO from the set of known protocols.

5. As no suitable one was available, a replication protocol was designed.

Observation 1 The design of the GDN provides more evidence that programming with distributed objects is intrinsically different from programming with objects in a single address space, as most famously noted by Waldo et al. [1994]. In particular, the design shows that the programmer should explicitly take into account the fact that the objects will be replicated for performance. If this fact is ignored, it may be impossible to make the application perform as required.

Replication, caching and partitioning are the means by which to achieve good performance in a distributed system. However, whether or not replication and caching can be applied successfully depends on the read/write ratio on the replicated or cached data item, as only data items that are updated infrequently can be efficiently replicated or cached. For the GDN this requirement implies that the majority of operations on the the distributed shared objects used for distributing the software should just read the state of the object and not modify it.

The read/write ratio of a distributed shared object depends on the nature of the object and its interface. The nature of the DSO, that is, roughly speaking, which entities or relationships in the application domain the DSO represents is determined by the mapping from application domain to DSOs. As some entities and relationships in the application domain change more frequently than others, the read/write ratio of a DSO depends on which entities or relationships it presents. This implies that the read/write ratios of the DSOs in the application, and thus their achievable performance, are in part determined in the mapping stage of application development.

The other factor determining the read/write ratio of a DSO is its interface. The interface of a DSO is defined in the interface-definition stage of development, the stage that follows the mapping stage. An interface specifies the functionality of the DSO. There are different sets of methods and method signatures that provide the same functionality. However, some sets of methods will result in more updates to the state of the object than others. Consider, for example, the interface of a hash-table object. If this interface allows only single elements to be inserted into the table but frequently multiple elements are inserted in sequence, an interface with an insert method that allows batches of elements to be inserted results in less updates to the state of the hash-table object. The choices made in the interface-definition stage can thus also affect the read/write ratio of DSOs and therefore their ultimate performance.

In addition to affecting the read/write ratio of objects, the interfaces of a DSO can also influence end-user performance and scalability. A badly chosen interface can hamper end-user performance, which translates to download speed in the GDN, as illustrated by the discussion on block sizes in Sec. 9. An interface that requires a DSO to keep per-client state, for example, about the progress of a download, affects not only the read/write ratio of the DSO, but may also limit its scalability in terms of the maximum number of clients supported. The performance of DSOs is also determined by their semantics, in particular, as specified by their consistency model (see Sec. 4.3.4). The stage in which the DSOs' consistency models are selected are therefore also important for the overall performance of the distributed application.

Observation 2 In Step 1 we can make a favorable observation about the claim that Globe provides flexible and comprehensive means to deal with the complex nonfunctional aspects of large-scale applications. Not only does it allow new replication protocols to be added, it also makes developing new replication protocols easier by providing two useful building blocks. The availability of the functionality provided by the Globe Location Service and Globe Infrastructure Directory Service (directing clients to the nearest replica, and discovering available object servers in a particular network region, respectively) greatly simplified the design of the replication protocol for the GDN's DSOs.

9.2.2. Step 2: Meeting Application-Level Security Requirements

It is not easy to ensure that a free-software distribution network is not abused for the publication of copyrighted or illicit content. The difficulty lies in designing an efficient security process that prevents or limits this type of abuse. The term security process was defined as a set of procedures to be followed by the different parties involved and software measures that support and ensure safety of these procedures. For the GDN, it was necessary that the security process was efficient, that is, did not require a lot of work from the parties involved, such as software publishers and object-server owners. For example, to attract publishers to use the GDN it is important that software publication via the GDN is relatively easy.

Observation 3 More important, however, for the GDN is that the total effort involved in administering the GDN is low, as the application does not generate revenue and will therefore have to be operated by volunteers. I expect that volunteers will not join in running the application if it requires them to invest much time and effort. This assumption implies that large-scale Internet applications that depend on volunteers for infrastructure and administration, such as the GDN, cannot be successfully deployed unless maintenance costs are low. As operational costs are also an important factor for commercial large-scale applications, a lesson (re)learned is therefore that a good middleware platform should not only facilitate development, but should also deliver applications that are manageable at low cost.

Another factor that can influence the success or failure of deploying a large-scale application on the Internet is legal responsibility. For the GDN, preventing illegal distribution is necessary, in particular, to protect the people and organizations who make resources available to the GDN from prosecution. The Internet is now in a phase where governments and stake holders are imposing regulations on what goes on in the network and are actually enforcing those regulations (e.g., prosecution of hackers and music-sharing services). It is therefore necessary that a distributed application adheres to the law, especially if that application depends on volunteers for its infrastructure. The risk is that volunteers may not make their resources available to the application if there is a legal risk attached. As a result, it may not be possible to run the application due to lack of resources.

It may thus not be possible to apply replication as a technique for improving the performance of large-scale distributed applications if they depend on volunteers, as these people and organizations do not want to be held accountable for, or associated with, illegal content or services provided by others. In cases where the infrastructure is provided by commercial companies such as content-distribution networks the risk of prosecution for the infrastructure providers does not exist, as these companies are contracted by the publishers to distribute the content, and can, most likely, claim the publishers are responsible. Whether or not the GDN's measures for keeping illegal content out are legally sufficient to avoid prosecution remains unclear, as legislation regarding the Internet is still evolving.

9.2.3. Step 3: Countering External and Internal Attacks

After having described how the GDN meets its application-level security requirements, the next step in securing the Globe Distribution Network was specifying how the GDN can be protected against external and internal attacks. Unfortunately, the security facilities of the Globe middleware did not crystallize sufficiently in the period this research was conducted, and I therefore cannot provide a clear picture of what this step in the development process of the GDN looks like.

Observation 4 Consequently, only a (simple) access control model for the GDN was defined, giving a high level description of what the legitimate GDN users are allowed to do, and therefore implicitly defining what outsiders should not be allowed to do. In addition, for the initial implementation of the GDN I designed and built a mechanism for disallowing unauthorized updates to a DSO's state and securing replica management (described in Sec. 5.5), based on the Transport Level Security (TLS) protocol. Arguing in favor of Globe is the fact that implementing this mechanism did not require radical changes to the existing architecture of the Globe middleware.

Observation 5 A noteworthy observation is that when an application has large numbers of users, it may need to protect itself against internal abuse as thoroughly as it protects itself against outside abuse. For the GDN, designing measures against internal denial-of-service attacks was relatively straightforward due to the properties of the application. What again played an important role during design was finding counter measures that require little system administration to run, as the GDN as a whole should be easy to maintain (see above).

9.2.4. Step 4: Ensuring Fault Tolerance

Observation 6 Fault tolerance of applications is an area in which the Globe middleware still leaves much to the application developer. An important improvement would be the addition of a method prescribing how to design fault-tolerant applications, which is a complex undertaking as the applications are large and fault tolerance should be an application-wide property. The method should preferably decompose this design problem into smaller problems that can be studied individually. The approach I took for the GDN was to define a model describing the dependencies between GDN components, and study the possibilities for making each of these components fault tolerant, starting with the components that do not depend on components other than the physical hardware (i.e., the leaf components in the dependency graph). This bottom-up approach worked fairly well for the GDN.

9.3. FUTURE WORK

To better assess the claim that Globe constitutes a useful middleware platform for developing large-scale applications, more applications will need to be designed, preferably in two phases. The purpose of designing applications in the first phase is to gain more experience with designing large-scale applications and to identify requirements for Globe's distribution, security, and fault tolerance facilities. After this phase, the experience gained and the requirements identified should be translated into (improved) designs for these facilities, and a method, or at least, guidelines describing how to develop applications using these facilities. The purpose of the second phase of developing applications is then the verification of the method and the facilities.

Creating a development method for Globe in addition to providing the necessary facilities is important if Globe is to be a usable tool for large-scale application development in addition to being a powerful tool. Such a method should address the issues identified above: (1) how to translate an application domain to distributed shared objects in a way that separates functionality from performance and scalability concerns as much as possible, and that results in DSOs with good performance, and (2) how to design fault-tolerant applications. The former is important, because for applications with more complex application domains and domain-to-DSO mappings than the GDN, it may be hard for a developer to determine which are the mapping and interface choices that result in the required performance. A first guideline put forward in this dissertation are two principles for designing interfaces of DSOs: (1) methods of DSOs should be made read-only whenever possible, and (2) a DSO should not store any per-client state when possible (see Sec. 4.3.2).

Some of the resource-management solutions designed for protecting the Globe Distribution Network against denial-of-service attacks are fairly generic (e.g. local resource management); others are specific to the application domain. For example, the GDN Quota Service limiting the number of DSOs a user can create depends on the fact that the rate at which free software is published is rather stable and low. I speculate that the observation that efficient solutions against denial-of-service attacks are found in the application domain is one that can be made in other applications as well, due to the fact that how resources should be allocated is ultimately prescribed by the application. If this speculation holds true, an interesting challenge for the developers of the Globe security facilities is to design a framework that can support application-specific solutions against DoS attacks.

Fault-tolerance facilities for the Globe middleware is an interesting research area. We should further develop our checkpointing mechanism for object servers to allow even more application- and object-specific optimizations, and develop replication protocols that can handle failing and recovering object servers. The latter encompasses an interesting research topic, specifically, efficient failure detection for distributed shared objects. Two research questions in this area are how large-scale, wide-area failure detection can be made efficient, and what the role of application-specific optimizations in this area is.

When designing the security and fault-tolerance facilities, special attention should be paid to the system-administration effort associated with the facilities. In particular, failure handling and recovery can require a considerable amount of work from administrators. An important requirement for the fault tolerance measures offered by Globe is therefore that they enable an application to automatically repair itself, or at least, recover itself to a consistent state. Mechanisms for self-repair will, most likely, require application-specific knowledge as input, or completely application-specific solutions, and is therefore an area in which we may find more evidence in support of the claim that middleware for large-scale distributed applications should be highly flexible. Low maintenance effort and self-repair are currently receiving much attention, for example, in IBM's research into autonomic computing [IBM Corporation, 2002].

How to design large-scale applications with low-maintenance cost in mind and what support the middleware should offer is an important area of future research, in general. A worthwhile research area related to system administration is evolvability of an application. Evolvability of an application was defined as its ability to adapt to new functional requirements and changes in usage patterns and in its operating environment. At the lower end of the evolvability spectrum is, for example, the ability to support new, more efficient, replication protocols; at the other end is the ability to dynamically update the application while it is running. Scalability of an application can be seen as the ability to adapt to new usage patterns without modification to its design. A large-scale application that is to be durable, that is, persist for a longer period of time should have some form of evolvability.

With respect to the Globe Distribution Network there are three interesting areas for future research. The first area is to further develop and test replication protocols for efficient wide-area distribution of data. In particular, we should try to develop protocols that can take into account network topologies and current network-link conditions. A challenging subclass of these protocols are protocols that can deal with untrusted object servers. A notable observation that may be exploited to achieve better performance is the observation that the flash-crowd phenomenon is generally caused by people, as the name suggests, and not by automated procedures. As a result, the region in which flash crowds occur is most likely the part of the Earth where people are currently active, that is, where is it daytime or evening. The GDN object servers in other parts of the world and their wide-area connections might therefore be underutilized. It would be interesting to investigate whether there is indeed underutilization and whether this dormant capacity could somehow be used to deal with flash crowds, or even for normal network load balancing.

The second area of future research into the Globe Distribution Network is extending the GDN's measures for preventing illegal distribution of content to incorporate per-country differences with respect to what constitutes legal content.

Finally, it is interesting to investigate how one can efficiently distribute other types of content via the GDN. For each type of content we should analyze:

1. How the content can be mapped to distributed shared objects. Content types such as streaming audio or video can present interesting technical challenges.

2. How we can achieve fair allocation of resources among the publishers of that content, some of whom will have bad intentions, without introducing a lot of system administration.

3. How to prevent illegal distribution of copyrighted instances of the new type of content. An important factor is again the amount of work for the administrators. For example, distribution of free digital music via the GDN may not be practically feasible due to the administrative effort involved. The risk of abuse for this type of content is high, and would require content moderation to keep the amount of illegal content low. As content moderation is labour intensive for the GDN's administrators, it may not be feasible to distribute free digital music via the Globe Distribution Network.

SAMENVATTING

Een software distributie netwerk gebaseerd op objecten

Het Internet heeft het leven van miljoenen mensen radicaal veranderd. Onderzoekers bezoeken nog zelden de bibliotheken omdat zoveel publicaties *on line* beschikbaar zijn. Studenten en bedrijven gebruiken gratis software die niet alleen verspreid wordt via het Internet, maar ook ontwikkeld is door programmeurs van over de hele wereld die samenwerken via datzelfde Internet. Steeds meer mensen kopen boeken, CD's, concertkaartjes en kleding via het Net. Sommigen doen zelfs hun bankzaken en aandelentransacties *on line*.

Veel mensen geloven dat dit slechts het begin is. De afgelopen jaren is er veel geld geïnvesteerd in bedrijven die de volgende succes applicatie voor het Internet proberen te ontwikkelen. Het ontwikkelen van applicaties die in staat zijn 24 uur per dag miljoenen mensen verspreid over de hele wereld te bedienen is echter lastig. Dit komt vooral door het ontbreken van een goed ontwikkelplatform dat programmeurs in staat stelt grootschalige, veilige en betrouwbare Internet applicaties te bouwen. Het doel van het Globe project is het ontwerpen en bouwen van zo'n zogeheten middleware platform [Van Steen et al., 1999a].

In dit proefschrift beschrijf ik het ontwerp en de implementatie van een nieuwe Internet applicatie welke ontwikkeld is met behulp van het Globe middleware platform, met als doel de ideeën achter, en het ontwerp van, dit middleware platform te valideren. Deze nieuwe applicatie, genaamd het *Globe Distribution Network* maakt het mogelijk vrij verspreidbare software, zoals de GNU C compiler, de Apache HTTP server en Linux distributies efficiënt wereldwijd te distribueren. Met andere woorden, het Globe Distribution Network (afgekort GDN) stelt Internet gebruikers op een efficiënte manier in staat waar en wanneer dan ook een kopie van de vrije software waarin ze geïnteresseerd op te halen.

Hoofdstuk 1 van het proefschrift legt uit dat de moeilijkheid bij het ontwikkelen van nieuwe grootschalige applicaties niet alleen ligt bij de complexe functionaliteit van de applicatie, maar vooral bij de complexe niet-functionele eisen waaraan voldaan moet worden. Deze eisen met betrekking tot prestaties, veiligheid en fout tolerantie worden geïntroduceerd door de schaal van de applicatie en de eigenschappen van het Internet.

De manier om het ontwikkelen van Internet applicaties minder lastig te maken is daarom de ontwikkelaars de juiste middelen te geven om met deze complexe niet-functionele aspecten om te gaan. Dit is de insteek die gekozen is voor de Globe middleware. De juiste middelen bestaan in de Globe middleware uit: (1) een model van gedistribueerde objecten, genaamd *distributed shared objects*, dat een ontwikkelaar in staat stelt om functionele van niet-functionele aspecten te scheiden, en dat de flexibiliteit biedt om alle mogelijke protocollen, technieken en strategieën toe te passen teneinde aan de niet-functionele eisen te voldoen, en (2) een grote bibliotheek met implementaties van protocollen en technieken. Flexibiliteit met betrekking tot protocollen, technieken en strategieën is van het allerhoogste belang. Om aan de lastige niet-functionele eisen van een applicatie van Internet grootte te kunnen voldoen is het namelijk noodzakelijk dat de oplossingen gebruikt worden die het meest geschikt zijn voor die applicatie. Om aan te tonen dat de Globe middleware inderdaad een goed platform is zijn hiermee een aantal grootschalige applicaties ontwikkeld, waaronder het Globe Distribution Network.

Hoofdstuk 2 identificeert de functionele en niet-functionele eisen voor een wereldwijd distributie-netwerk voor vrije software, zoals het GDN. Het feit dat veel mensen vrije software gebruiken en ook produceren en dat deze interesse wereldwijd is vereist dat de applicatie efficiënt gebruik maakt van het onderliggende netwerk en 24 uur per dag beschikbaar is. De laatstgenoemde eis impliceert dat de applicatie fouttolerant moet zijn en bestand moet zijn tegen aanvallen die pogen de applicatie buiten werking te stellen. Het domein en de aard van de applicatie maken het GDN ook een aantrekkelijk doelwit voor andere soorten aanvallen. Het is zeer waarschijnlijk dat men zal proberen het GDN te gebruiken als een handig distributie-kanaal voor de illegale verspreiding van auteursrechtelijk beschermde werken, zoals commerciële software of digitale muziek. Verder bestaat er het risico dat kwaadwillende personen zullen proberen de vrije software die verspreid wordt te wijzigen om zo toegang te kunnen krijgen tot de computers waarop deze software gedraaid wordt. Een distributie-netwerk als het GDN dient maatregelen tegen te nemen tegen dit onrechtmatig gebruik.

Na de presentatie van de noodzakelijke achtergrondinformatie over de Globe middleware in hoofdstuk 3 gaan we in de hoofdstukken 4, 5, en 6 in op hoe het Globe Distribution Network aan de geïdentificeerde eisen gaat voldoen. Hoofdstuk 4 beschrijft hoe de distributie van software efficiënt gemaakt wordt, dat wil zeggen, hoe het GDN ervoor zorgt dat bij het ophalen van de software door cliënten er weinig gebruikt gemaakt wordt van gedeelde lange-afstandsverbindingen waarop de bandbreedte vaak beperkt is. Om dit doel te bereiken kapselt het GDN de vrije software in Globe *distributed shared objects* (DSO's) welke een replicatie-protocol gebruiken die het object dicht bij zijn cliënten repliceert. Om precies te zijn wordt elke revisie van een software pakket in zijn eigen *Revision DSO* geplaatst. Om grote collecties van software, zoals Linux distributies, te publiceren wordt gebruikt gemaakt van *DistributionArchive DSO's* welke een specifieke variant van een revisie van de gehele software collectie in één bepaald bestandsformaat bevatten.

Het replicatie-protocol van Revision en DistributionArchive DSO's houdt bij hoe vaak en waarvandaan de software in het object opgehaald wordt. Iedere keer dat een cliënt software ophaalt wordt er geregistreerd in welk *autonomous system* de cliënt zich bevindt. De DSO's gebruiken deze informatie om periodiek hun replicatie-scenario te herzien, dat wil zeggen, het aantal replica's van het DSO en hun plaats in het Internet. Als het huidige replicatie-scenario van het DSO niet efficiënt is, d.w.z., als het ophalen van de software uit het object teveel bandbreedte gebruikt op lange-afstandsverbindingen, dan worden er extra replica's gecreëerd in de *autonomous systems* van waaruit de software vaak opgehaald wordt en worden onderbenutte replica's verwijderd. Een DSO past ook automatisch zijn replicatie-scenario aan als er een plotselinge scherpe toename is in het aantal keren dat de software opgehaald wordt. Dit is een veel voorkomend fenomeen bij het uitkomen van een nieuwe versie van een populair software pakket (het zogenaamde *flash crowd* fenomeen).

Hoe het Globe Distribution Network aan de gestelde veiligheidseisen voldoet wordt besproken in hoofdstuk 5. Het GDN heeft een nieuwe aanpak voor het probleem van de illegale verspreiding van beschermde of verboden werken. In plaats van de inhoud van bestanden te controleren vóór deze op het netwerk toegelaten worden (wat hoe dan ook nagenoeg onmogelijk is), maakt het GDN bestanden slechts traceerbaar naar degene die ze gepubliceerd heeft. Wanneer er illegale informatie aangetroffen wordt in een bestand wordt de uitgever van het bestand permanent de toegang tot het distributie-netwerk ontzegd en het bestand verwijderd. Het voordeel van deze aanpak is dat de hoeveelheid werk voor de beheerders van het distributie-netwerk klein is als er weinig illegale publicaties zijn, een belangrijke eigenschap voor een distributie-netwerk dat geen inkomsten genereert. De architectuur van het GDN ondersteunt ook andere manieren om illegale distributie tegen te gaan.

Het vaststellen van de authenticiteit en integriteit van de software die via het GDN verspreid wordt, wordt gezien als een taak die door de eindgebruiker in samenwerking met de auteur van de software uitgevoerd dient te worden, zonder tussenkomst van het GDN. Het GDN beschermt zichzelf tegen aanvallen van buitenaf door het handhaven van een simpel toegangsbeheersingsmodel (*access control model*). Hoofdstuk 5 identificeert ook mogelijke aanvallen tegen de beschikbaarheid van het GDN door ingewijden (software-uitgevers en beheerders van computers gebruikt door het GDN) en hoe deze aanvallen afgeslagen kunnen worden. Dit hoofdstuk beschrijft verder een eerste implementatie van de veiligheidsmaatregelen ter voorkoming van illegale verspreiding en verstoringen van buitenaf.

Hoofdstuk 6 beschrijft hoe het GDN in hoge mate beschikbaar blijft, betrouwbaar gemaakt wordt en beheersbaar blijft ondanks het falen van computers, netwerken en GDN onderdelen. Een grote mate van beschikbaarheid en betrouwbaarheid worden bereikt door een combinatie van fouttolerante servers, het benutten van het feit dat de *distributed shared objects* die de software bevatten reeds gerepliceerd zijn t.b.v. prestatieverbetering, en het expliciet introduceren van extra replica's t.b.v. beschikbaarheid. Beheersbaarheid wordt voornamelijk bereikt door het geven van sterke garanties met betrekking tot het resultaat van een operatie in het GDN wanneer er fouten optreden. In het bijzonder garandeert het GDN dat operaties *atomic with respect to exceptions* zijn, dat wil zeggen, dat een operatie of slaagt of faalt, waarbij in het laatste geval de toestand van de applicatie niet veranderd wordt.

Om de haalbaarheid van het ontwerp van het Globe Distribution Network aan te tonen is er een eerste implementatie van de applicatie gebouwd. Hoofdstuk 7 beschrijft een aantal kleine experimenten die uitgevoerd zijn met deze implementatie. Deze experimenten zijn gericht op het meten van de prestaties van het GDN in vergelijking met software-distributie via het HyperText Transfer Protocol (HTTP) [Fielding et al., 1999]. Uit deze experimenten bleek dat de prestaties van een GDN server slechts 10% slechter waren dan die van een Apache HTTP server [The Apache Software Foundation, 2002], of minder, bij grote aantallen cliënten. Uit een experiment over het Internet waarbij software opgehaald werd m.b.v. het GDN en HTTP bleek dat software ophalen via het GDN vooral langzamer is door de aanzienlijke tijd die nodig is om de symbolische naam van een *distributed shared object* te vertalen naar een contactadres van de dichtstbijzijnde replica en het creëren van een *proxy* van het DSO dat verbinding legt met deze replica. Deze acties kostten aanzienlijk veel tijd, danwel van nature, danwel omdat de eerste implementatie van de applicatie en de middleware nog niet compleet en volledig geoptimaliseerd zijn. De voordelen van het GDN over een puur op HTTP gebaseerde oplossing zijn dat het GDN automatisch de dichtsbijzijnde replica lokaliseert, zorgt voor automatische omschakeling naar een andere replica bij fouten en dat het sterke consistentie van replicas garandeert.

Hoofdstuk 8 bediscussiëert de ontwerpen van andere system welke gebruikt worden of kunnen worden voor wereldwijde verspreiding van software. We kijken met name naar oudere systemen zoals HTTP, FTP en AFS [Howard et al., 1988], moderne *content distribution networks* zoals Akamai en moderne bestanduitwisselingssystemen zoals CFS [Dabek et al., 2001b] en OceanStore [Rhea et al., 2001]. De oudere systemen zijn minder geschikt voor grootschalige software-distributie, uit technisch danwel uit gebruiksvriendelijkheids oogpunt. *Content distribution networks* zijn veel beter geschikt aangezien deze de problemen van de oudere systemen opgelost hebben. De nieuwste bestanduitwisselingssystemen lijken goed toegerust voor de distributie van vrije software, alhoewel sommigen mogelijk enige efficiëntie inruilen voor een gedecentraliseerde architectuur.

Hoofdstuk 9 bevat een samenvatting van de dissertatie, beschrijft mijn observaties over het ontwerp en de implementatie van het Globe Distribution Network en identificeert een aantal onderwerpen voor toekomstig onderzoek. Het onderzoek naar het GDN leidt tot acht belangrijke observaties:

1. Bij het ontwerpen van de interfaces van *distributed shared objects* kan men niet negeren dat objecten over het algemeen gerepliceerd zullen worden en moet men ook rekening houden met schaalbaarheid en de prestaties zoals ervaren door de gebruiker.

2. De beschikbaarheid van diensten als de Globe Location Service en de Globe Infrastructure Directory Service vereenvoudigen het ontwikkelen van replicatie-protocollen voor *distributed shared objects* aanzienlijk.

3. Wil men een grootschalige gedistribueerde applicatie laten draaien en beheren door vrijwilligers dan dient van de meeste vrijwilligers deze systeembeheertaak weinig werk te vereisen. Tevens dienen er voor deze vrijwilligers geen wettelijke risico's verbonden te zijn aan hun medewerking aan zo'n applicatie.

4. *Distributed shared objects* kunnen eenvoudig beveiligd worden door middel van een Transport Layer Security [Dierks and Allen, 1999] bibliotheek.

5. Een grootschalige gedistribueerde applicatie met veel gebruikers moet zich mogelijk net zo goed beschermen tegen zijn eigen gebruikers als het zich beschermt tegen misbruik van buitenaf.

6. Het modelleren van een applicatie in termen van diensten en afhankelijkheden tussen diensten is waardevol bij het ontwerpen van maatregelen om een op Globe gebaseerde applicatie fouttolerant te maken.

BIBLIOGRAPHY

4C Entity LLC (2000). Content Protection System Architecture: A Comprehensive Framework for Content Protection. http://www.4centity.com/data/tech/cpsa/cpsa081.pdf. revision 0.81.

Adve, S. V. and Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial . *IEEE Computer*, 29(12):66–76.

Akamai Technologies, Inc. (2000). Annual Report to the Securities and Exchange Commission (FORM 10-K). http://www.akamai.com/en/html/investors/10k_2000.html.

Akamai Technologies, Inc. (2002). EdgeSuite. http://www.akamai.com/.

Allock, W., Bester, J., Bresnahan, J., Chervenak, A., Liming, L., and Tuecke, S. (2001). Internet Draft: GridFTP: Protocol Extensions to FTP for the Grid. http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf. (work in progress).

Anonymizer.com (2001). Anonymous Web Surfing. http://www.anonymizer.com/.

Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., and Spiteri, M. (2000). Generic Support for Distributed Applications. *IEEE Computer*, 33(3):2–10.

Bailey, E. (1998). *Maximum RPM – Taking the Red Hat Package Manager to the Limit*. Red Hat Press, Durham, NC, USA, 1.1 edition.

Bakker, A., Amade, E., Ballintijn, G., Kuz, I., Verkaik, P., Van der Wijk, I., Van Steen, M., and Tanenbaum, A. (2000). The Globe Distribution Network. In *Proceedings 2000 USENIX Annual Technical Conference (FREENIX track)*, pages 141–152, San Diego, CA, USA.

Bakker, A., Kuz, I., Van Steen, M., Tanenbaum, A., and Verkaik, P. (2002). Global Distribution of Free Software (and other things). In *Proceedings System Administration and Networking Conference (SANE 2002)*, Maastricht, The Netherlands.

Bakker, A., Van Steen, M., and Tanenbaum, A. (1998). Replicated Invocations in Wide-Area Systems. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal.

Bakker, A., Van Steen, M., and Tanenbaum, A. (1999). From Remote Objects to Physically Distributed Objects. In *Proceedings of the 7th IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS'99)*, pages 47–52, Cape Town, South Africa. IEEE Computer Society.

Bakker, A., Van Steen, M., and Tanenbaum, A. (2001a). A Distribution Network for Free Software. Technical Report IR-485, Division of Mathematics and Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam.

Bakker, A., Van Steen, M., and Tanenbaum, A. (2001b). A Law-Abiding Peer-to-Peer Network for Free-Software Distribution. In *Proceedings IEEE Symposium on Network Computing and Applications (NCA'01)*, Cambridge, MA. IEEE Computer Society.

Bal, H., Kaashoek, M., Tanenbaum, A., and Jansen, J. (1992). Replication Techniques for Speeding up Parallel Applications on Distributed Systems. *Concurrency Practice & Experience*, 4(5):337–355.

Ballintijn, G., Van Steen, M., and Tanenbaum, A. (1999). Simple Crash Recovery in a Wide-Area Location Service. In *Proceedings 12th International Conference on Parallel and Distributed Computing Systems*, pages 87–93, Fort Lauderdale, FL, USA.

Ballintijn, G., Van Steen, M., and Tanenbaum, A. (2000). Scalable Naming in Global Middleware. In *Proceedings 13th International Conference on Parallel and Distributed Computing Systems (PDCS-2000)*, pages 624–631, Las Vegas, NV, USA.

Ballintijn, G., Van Steen, M., and Tanenbaum, A. (2001). Scalable User-Friendly Resource Names. *IEEE Internet Computing*, 5(5):20–27.

Bates, T., Gerich, E., Joncheray, L., Jouanigot, J.-M., Karrenberg, D., Terpstra, M., and Yu, J. (1995). RFC 1786: Representation of IP Routing Policies in a Routing Registry.

Bea Systems, Expersoft Corporation, GDM Fokus, Hewlett-Packard, Inprise, IBM, IONA Technologies, Objective Interface Systems, Object Oriented Concepts, Sun Microsystems, Adiron LLC, and Humboldt-Universität zu Berlin (1999). Portable Interceptors: Joint Revised Submission. OMG Document orbos/99-12-02, Object Management Group, Framingham, MA, USA.

Birman, K. P. and Van Renesse, R., editors (1994). *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E. (1993). Network Objects. In *Proceedings 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 217–230, Asheville, NC, USA.

Burk, D. (2001). Copyrightable Functions and Patentable Speech. *Communications of the ACM*, 44(2):69–75.

CAIDA (2001). NetGeo - The Internet Geographic Database. http://www.caida.org/tools/utilities/netgeo/.

Carzaniga, A., Rosenblum, D., and Wolf, A. (2000). Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *Proceedings 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, Portland, OR, USA.

Cederqvist et al. (2001). Version Management with CVS. http://www.cvshome.org/docs/index.html.

Chappell, D. (1996). *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, USA.

Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. (1994). RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185.

Clarke, I., Hong, T., Miller, S., Sandberg, O., and Wiley, B. (2002). Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. (2001). Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Federrath, H., editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany.

Conradi, R. and Westfechtel, B. (1998). Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282.

Cornelli, F., Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. (2002). Choosing Reputable Servents in a P2P Network. In *Proceedings 11th International World Wide Web Conference*, Honolulu, HI, USA.

Cristian, F. (1991). Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78.

Czajkowski, G. and von Eicken, T. (1998). JRes: A Resource Accounting Interface for Java. In *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 21–35, Vancouver, Canada.

Dabek, F., Brunskill, E., Kaashoek, M., Karger, D., Morris, R., Stoica, I., and Balakrishnan, H. (2001a). Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany. IEEE Computer Society.

Dabek, F., Kaashoek, M., Karger, D., Morris, R., and Stoica, I. (2001b). Wide-area Cooperative Storage With CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada.

Dailey Paulson, L. (2001). Hacker Lauches Cyberattack from Security Site. *IEEE Computer*, 34(4):23.

Dasgupta, P., LeBlanc, R., Ahamad, M., and Ramachandran, U. (1991). The Clouds Distributed Operating System. *IEEE Computer*, 24(11):34–44.

Davis, R. (2001). The Digital Dilemma. *Communications of the ACM*, 44(2):77–83.

Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C.-G., and Wei, L. (1996). The PIM Architecture for Wide-Area Multicast Routing. *IEEE/ACM Transactions on Networking*, 4(2):153–162.

Dierks, T. and Allen, C. (1999). RFC 2246: The TLS Protocol Version 1.0.

Druschel, P. and Rowstron, A. (2001). PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany. IEEE Computer Society.

Dwork, C. and Noar, M. (1992). Pricing via Processing or Combating Junk Mail. In Brickell, E., editor, *Advances in Cryptology — Crypto'92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer-Verlag.

Eddon, G. and Eddon, H. (1998). *Inside Distributed COM*. Microsoft Press, Redmond, WA, USA.

Edwards, J. (2000). Building the Optical-Networking Infrastructure. *IEEE Computer*, 33(3):20–23.

Eliassen, F., Andersen, A. Blair, G., Costa, F., Coulson, G., Goebel, V., Hansen, Ø., Kristensen, T., Plagemann, T., Rafaelsen, H., Saikosi, K., and Yu, W. (1999). Next Generation Middleware: Requirements, Architecture, and Prototypes. In *Proceedings of the 7th IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS'99)*, pages 60–65, Cape Town, South Africa.

Exodus (2002). 2Deliver Web Service. http://www.exodus.net/solutions/2deliver/.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.

Foster, I. and Kesselman, C. (1997). Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128.

Foster, I. and Kesselman, C. (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA.

Free Software Foundation, Inc. (1991). GNU General Public License Version 2. http://www.fsf.org/licenses/gpl.txt.

Fu, K., Kaashoek, M., and Mazières, D. (2000). Fast and Secure Distributed Read-Only File System. In *Proceedings 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, USA.

Gadde, S., Chase, J., and Rabinovich, M. (1998). A Taste of Crispy Squid. In *Proceedings Workshop on Internet Server Performance (WISP98)*, Madison, WI, USA.

Goa, J., Chen, C., Toyoshima, Y., and Leung, D. (1999). Engineering on the Internet for Global Software Production. *IEEE Computer*, 32(5):38–47.

Goldschlag, D., Reed, M., and Syverson, P. (1999). Onion Routing for Anonymous and Private Internet Connections. *Communications of the ACM*, 42(2):39–41.

Gong, L. (1999). *Inside Java 2 Platform Security (Architecture, API Design, and Implementation)*. The Java Series. Addison-Wesley, Reading, MA, USA.

Grid Forum (2001). Grid Forum Account Management Working Group. http://www.gridforum.org/groups/wg.html#AM.

Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M. (1999). Wide-Area Computing: Resource Sharing on a Large Scale. *IEEE Computer*, 32(5):29–37.

Hänle, C. and Tanenbaum, A. (2000). A Security Architecture for Distributed Shared Objects. In *Proceedings 6th Annual ASCI Conference*, pages 350–357, Lommel, Belgium. Advanced School for Computing and Imaging, Delft, The Netherlands.

Hayton, R., Herbert, A., and Donaldson, D. (1998). Flexinet: A Flexible, Component-Oriented Middleware System. In *Proceedings 8th ACM SIGOPS European Workshop*, pages 17–24, Sintra, Portugal.

Homburg, P. (2001). *The Architecture of a Worldwide Distributed System*. PhD thesis, Faculty of Sciences, Division Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

Housley, R., Ford, W., Polk, W., and Solo, D. (1999). RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile.

Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. (1988). Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81.

Ibiblio (2001). Linux Archives: How to Submit. http://www.ibiblio.org/pub/linux/howtosubmit.html.

IBM Corporation (2002). Autonomic Computing. http://www.research.ibm.com/autonomic/.

IDC (2000). Linux Is Red Hot in the Server Market. http://www.idc.com/Hardware/press/PR/ES/ES041000pr.stm. Press Release.

IDC (2001). Microsoft Strengthens Its Grip, Narrowing the Window of Opportunity for Other Operating Environments. http://www.idc.com/software/press/PR/SW022801pr.stm. Press Release.

Internet Software Consortium (2001). Internet Domain Survey. http://www.isc.org/ds/WWW-200101/index.html.

ISO (1995). Open Distributed Processing - Reference Model - Part 3: Architecture. International Standard / ITU-T Recommendation 10746-3 / X.903, ISO/IEC.

Jalote, P. (1989). Resilient Objects in Broadcast Networks. *IEEE Transactions on Software Engineering*, 15(1):68–72.

Johnson, K., Carr, J., Day, M., and Kaashoek, M. (2001). The Measured Performance of Content Distribution Networks. *Computer Communications*, 24(2):202–206.

Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133.

Jung, J., Krishnamurthy, B., and Rabinovich, M. (2002). Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings 11th International World Wide Web Conference (WWW2002)*, Honolulu, HI, USA. ACM Press.

Katz, R. H. (1990). Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408.

Ketema, J. (2000). Towards a Fault-Tolerant Globe. Master's thesis, Faculty of Sciences, Vrije Universiteit Amsterdam, The Netherlands.

Kiczales, G., Rivières, J., and Bobrow, D. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

Krishnamurthy, B., Wills, C., and Zhang, Y. (2001). On the Use and Performance of Content Distribution Networks. In *Proceedings ACM SIGCOMM Internet Measurement Workshop (IMW'2001)*, San Francisco, CA, USA.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings 9th International Conference on Architectural Support for Programming Languages and Operatings Systems (ASPLOS'2000)*, Cambridge, MA, USA. ACM Press.

Kuz, I., Van Steen, M., and Sips, H. (2002). The Globe Infrastructure Directory Service. *Computer Communications*, 25(9):835–845. Elsevier Science, Amsterdam, The Netherlands.

Kuz, I., Van Steen, M., and Tanenbaum, A. (2001). The Globe Infrastructure Directory Service. Technical Report IR-484, Faculty of Sciences, Division Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691.

Leighton, F. and Lewin, D. (2000). Global Hosting System. United States Patent 6,108,703.

Leiwo, J., Hänle, C., Homburg, P., Gamage, C., and Tanenbaum, A. (1999). A Security Design for a Wide-area Distributed System. In *Proceedings 2nd International Conference on Information Security and Cryptology (ICISC'99)*, pages 236–256, Seoul, South Korea. (Lecture Notes in Computer Science #1787).

Leiwo, J., Hänle, C., Homburg, P., and Tanenbaum, A. (2000). Disallowing Unauthorized State Updates in Distributed Shared Objects. In Qing, S. and Eloff, J., editors, *Proceedings IFIP TC-11 16th Annual Working Conference on Information Security (SEC2000)*, Beijing, PRC. Kluwer Academic Publishers.

Lethin, R. (2001). Reputation. In Oram, A., editor, *Peer-to-Peer: Harnessing the Power of Disruptive Techn ologies*, chapter 17, pages 341–353. O'Reilly and Associates, Sebastopol, CA, USA.

Li, K. and Hudak, P. (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

Liang, S. and Bracha, G. (1998). Dynamic Class Loading in the Java Virtual Machine. In *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 26–44, Vancouver, Canada.

Likins, A. (2002). System Tuning Info for Linux Servers. http://people.redhat.com/alikins/system_tuning.html.

Loshin, P., editor (2000). *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. Morgan Kaufmann, San Francisco, CA, USA.

Macedonia, M. (2000). Distributed File Sharing: Barbarians at the Gates? *IEEE Computer*, 33(8):99–101.

Makpangou, M., Gourhant, Y., Le Narzul, J.-P., and Shapiro, M. (1994). Fragmented Objects for Distributed Abstractions. In *Readings in Distributed Computing Systems*. IEEE Computer Society.

Malda, R. (1999). Slashdot Moderation. http://slashdot.org/moderation.shtml.

Mao, Z., Cranor, C., Douglis, F., Rabinovich, M., Spatscheck, O., and Wang, J. (2002). A Precise and Efficient Evaluation of the Proximity Between Web Clients and Their Local DNS Servers. In *Proceedings 2002 USENIX Annual Technical Conference*, Monterey, CA, USA.

Martin, D., Smith, R., Brittain, M., Fetch, I., and Wu, H. (2001). The Privacy Practices of Web-Browser Extensions. *Communications of the ACM*, 44(2):45–50.

McCarty, B. (1999). *Learning Debian GNU/Linux*. O'Reilly and Associates, Sebastopol, CA, USA.

Microsoft Corporation (2001a). Software Management System. http://www.microsoft.com/smsmgmt/.

Microsoft Corporation (2001b). Visual SourceSafe. http://msdn.microsoft.com/ssafe/.

Mitchell, J., Gibbons, J., Hamilton, G., Kessler, P., Khalidi, Y., Kougiouris, P., Madany, P., Nelson, M., Powell, M., and Radia, S. (1994). An Overview of the Spring System. In *Proceedings of the IEEE International Computer Conference (COMPCON'94)*, pages 122–131, San Francisco, CA, USA.

Mockapetris, P. (1987). RFC 1034: Domain Names – Concepts and Facilities.

Moore, D., Periakaruppan, R., Donohoe, J., and Claffy, K. (2000). Where in the World is netgeo.caida.org? In *Proceedings 10th Annual Internet Society Conference*, Yokohama, Japan.

Neuman, B. C. (1994). Scale in Distributed Systems. In Casavant, T. and Singhal, M., editors, *Readings in Distributed Computing Systems*. IEEE Computer Society.

Nielsen, J. (1995). *Multimedia and Hypertext: The Internet and Beyond*. AP Professional, Boston, MA, USA.

Object Management Group (2001). The Common Object Request Broker: Architecture and Specification. Revision 2.4.2. OMG Document formal/2001-02-01, Object Management Group, Framingham, MA, USA.

Oram, A., editor (2001). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly and Associates, Sebastopol, CA, USA.

Partridge, C. (1994). *Gigabit Networking*. Addison-Wesley, Reading, MA, USA.

Penfield Jackson, T. (1998). Findings of Facts. http://www.microsoft.com/presspass/trial/c-fof/fof.asp. United States District Court for the District of Columbia – Civil Action No. 98-1232/98-1233: Antitrust case against Microsoft Corporation.

Pierre, G., Kuz, I., Van Steen, M., and Tanenbaum, A. (2000). Differentiated Strategies for Replicating Web Documents. *Computer Communications*, 24(2):232–240. Elsevier Science, Amsterdam, The Netherlands.

Pierre, G. and Van Steen, M. (2001). Globule: a Platform for Self-Replicating Web Documents. In *Proceedings 6th International Conference on Protocols for Multimedia Systems*, Enschede, The Netherlands.

Pierre, G., Van Steen, M., and Tanenbaum, A. (2001). Self-Replicating Web Documents. Technical Report IR-486, Faculty of Sciences, Division Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

Pitoura, E. and Samaras, G. (2001). Locating Objects in Mobile Computing. *IEEE Transactions on Kowledge and Data Engineering*, 13(4).

Plank, J. (1997). A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software Practice and Experience*, 27(9):995–1012.

Plank, J., Beck, M., Kingsley, G., and Li, K. (1995). Libckpt: Transparent Checkpointing under Unix. In *Proceedings USENIX Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, USA.

Rabinovich, M. and Aggarwal, A. (1999). RaDaR: A Scalable Architecture for a Global Web Hosting Service. In Mendelzon, A., editor, *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada. Elsevier Science, Amsterdam, The Netherlands.

Rabinovich, M., Chase, J., and Gadde, S. (1998). . Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network. In *Proceedings 3rd International WWW Caching Workshop*, Manchester, UK.

Rabinovich, M., Rabinovich, I., Rajaraman, R., and Aggarwal, A. (1999). A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. In *Proceedings International Conference on Distributed Computing Systems (ICDCS'99)*, Austin, TX, USA. IEEE Computer Society.

Rabinovich, M. and Spatscheck, O. (2001). *Web Caching and Replication*. Addison Wesley Professional, Reading, MA, USA.

Raymond, E. (2000). The Cathedral and the Bazaar Giving the original Cathedral and Bazaar paper at Linux Kongress, May 1997. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.

Reiter, M. and Rubin, A. (1999). Anonymous Web Transactions with Crowds. *Communications of the ACM*, 42(2):32–38.

Réseaux IP Européens (2001). RIPE Whois Database. http://www.ripe.net/whois.

Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. (2001). Maintenance-Free Global Data Storage. *IEEE Internet Computing*, 5(5):40–49.

Riggs, R., Waldo, J., Wollrath, A., and Bharat, K. (1996). Pickling State in the Java^{tm} System. In *Proceedings 2nd USENIX Conference on Object-Oriented Technologies (COOTS'96)*, Toronto, Ontario, Canada.

Robinson, S. (1999). CD Software Is Said to Monitor Users' Listening Habits. *The New York Times on the Web*. http://www.nytimes.com/library/tech/99/11/biztech/articles/01real.html.

Rodriquez, P. and Sibal, S. (2000). SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution. In *Proceedings 9th International World Wide Web Conference*, Amsterdam, The Netherlands.

Rogerson, D. (1997). *Inside COM*. Microsoft Press, Redmond, WA, USA.

Rowstron, A. and Druschel, P. (2001a). Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany.

Rowstron, A. and Druschel, P. (2001b). Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada.

Rumbaugh, J., Jacbobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Reading, MA, USA.

Saltzer, J., Reed, D., and Clark, D. (1984). End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288.

Samuelson, P. (1997). The Never Ending Struggle For Balance. *Communications of the ACM*, 40(5):17–21.

Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47.

Satyanarayanan, M. (1990). Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21.

Schneider, F. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys*, 22(4):299–319.

Schneier, B. (1996). *Applied Cryptography*. John Wiley & Sons, New York, NY, USA, 2nd edition.

Simons, B. (2000). From the President: To DVD or Not To DVD. *Communications of the ACM*, 43(5):31–32.

Software in the Public Interest, Inc. (2001). Debian Linux. http://www.debian.org/.

Sollins, K. and Masinter, L. (1994). RFC 1737: Functional Requirements for Uniform Resource Names.

Source Gear Corporation (2001). Source*OffSite*. http://http://www.sourceoffsite.com/.

Speedera Networks, Inc. (2002). Speedera Content Delivery. http://www.speedera.com/.

Stefik, M. (1997). Trusted Systems. *Scientific American*, 276(3):78–81.

Stelling, P., Foster, I., Kesselman, C., Lee, C., and von Laszewski, G. (1998). A Fault Detection Service for Wide Area Distributed Computations. In *Proceedings 7th IEEE Symposium on High Performance Distributed Computing*, pages 268–278, Chicago, IL, USA.

Stevens, W. R. (1998). *UNIX Network Programming*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition.

Stix, G. (2001). The Triumph of the Light. *Scientific American*, 284(1):80–86.

Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. (2001). Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, CA, USA.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. (1994). Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, USA.

The Apache Software Foundation (2002). The Apache HTTP Server. http://www.apache.org/.

The Globus Project (2000). GridFTP–Universal Data Transfer for the Grid. http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf.

The Institute of Electrical and Electronics Engineers, Inc. (2000). IEEE Standard 802.3, 2000 Edition. New York, NY, USA.

Tichy, W. F. (1992). Programming-in-the-large: Past, Present, and Future. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 362–367.

Tridgell, A. (2000). *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Canberra, Australia.

United States Government (1998). Digital Millennium Copyright Act. United States Public Law No. 105-304.

U.S. Department of Commerce (2001). Commercial Encryption Export Controls. http://www.bxa.doc.gov/Encryption/Default.htm.

Vahdat, A., Anderson, T., Dahlin, M., Culler, D., Belani, E., Eastham, P., and Yoshikawa, C. (1998). WebOS: Operating System Services For Wide Area Applications. In *Proceedings 7th IEEE Symposium on High Performance Distributed Computing*, Chicago, IL, USA.

Van Renesse, R., Birman, K., and Maffeis, S. (1996). Horus, a Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83.

Van Steen, M. and Ballintijn, G. (2002). Achieving Scalability in Hierarchical Location Services. In *Proceedings 26th International Computer Software and Applications Conference (COMPSAC'02)*, Oxford, UK.

Van Steen, M., Hauck, F., and Tanenbaum, A. (1998a). Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 104–109.

Van Steen, M., Homburg, P., and Tanenbaum, A. (1999a). Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78.

Van Steen, M., Tanenbaum, A., Kuz, I., and Sips, H. (1999b). A Scalable Middleware Solution for Advanced Wide-Area Web Services. *Distributed Systems Engineering*, 6(1):34–42.

Van Steen, M., Van der Zijden, S., and Sips, H. (1998b). Software Engineering for Scalable Distributed Applications. In *Proceedings 22nd International Computer Software and Applications Conference (CompSac)*, Vienna. IEEE Computer Society.

Waldman, M., Rubin, A., and Cranor, L. (2000). Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, USA.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1994). A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Mountain View, CA.

Wang, N., Parameswaran, K., Schmidt, D., and Othman, O. (2001). The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *Proceedings 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, TX, USA.

Wegner, P. (1987). Dimensions of Object-Based Language Design. In *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 168–181, Orlando, FL, USA.

Welsh, M. and Culler, D. (1999). Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538. (Special Issue on ACM 1999 Java Grande Conference).

Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. In *Proceedings 2nd USENIX Conference on Object-Oriented Technologies (COOTS'96)*, Toronto, Ontario, Canada.

Wolski, R., Plank, J., Brevik, J., and Bryan, T. (2000). Analyzing Market-based Resource Allocation Strategies for the Computational Grid. Technical Report UT-CS-00-453, University of Tennessee, Knoxville, TN, USA.

Wolski, R., Spring, N., and Hayes, J. (1999). The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5):757–768.

World Intellectual Property Organization (1996). WIPO Copyright Treaty. In *WIPO Diplomatic Conference on Certain Copyright and Neighbouring Rights Questions*, Geneva, Switzerland. http://www.wipo.int/treaties/ip/copyright/index.html.

Ximian, Inc. (2002). Red Carpet. http://www.ximian.com/products/ximian_red_carpet/.

Zacks, M. (2001). French Court Claims Jurisdiction over Yahoo! *IEEE Internet Computing*, 5(2):10–11.

Zayas, E. (1991). *AFS-3 Programmer's Reference: Architectural Overview*. Transarc Corporation, Pittsburgh, PA, USA, 1.0 edition.

List of Citations

INDEX