A Distribution Network for Free Software

Arno Bakker
Maarten van Steen
Andrew S. Tanenbaum

Internal report IR-485
February 2001

Abstract

The Globe Distribution Network (GDN) is an application for the efficient, worldwide distribution of freely redistributable software packages. Distribution is made efficient by encapsulating the software into special distributed objects which efficiently replicate themselves near to the downloading clients. The Globe Distribution Network takes a novel, optimistic approach to stop the illegal distribution of copyrighted and illicit material via the network. Instead of having moderators check the software archives at upload time, illegal content is removed and its uploader's access to the network permanently revoked only when the content is discovered. An important feature of the GDN is that the distributed objects containing the software can run on untrustworthy servers. By exploiting the replication of the software and using fault-tolerant server software, the Globe Distribution Network achieves high availability. A first version of the GDN has been implemented and has been running since October 2000 across four European sites. This article describes the design and implementation of the GDN.

Keywords: *software distribution; wide-area networks; distributed objects; traceable content*

vrije Universiteit

Faculty of Mathematics and Computer Science

INTRODUCTION

Developing a large Internet application is a difficult task due to the complex nonfunctional aspects that have to be taken into account. A developer has to deal with a potentially large number of users, high communication delays, security threats, and machine and network failures. The key to making large-scale application development easier is therefore providing the developer with the means for dealing with these complex aspects. Current middleware platforms, such as CORBA and DCOM, however, do not provide adequate support in this area, as they are mainly aimed at local area networks. We are designing and building a new middleware platform that will provide the developer with the support needed to build worldwide distributed applications more easily. This middleware platform is called *Globe* [1]. To demonstrate the feasibility of our ideas and the design of the Globe middleware we have been building a new large-scale Internet application, called the *Globe Distribution Network*. This article describes the design and implementation of this application.

The Globe Distribution Network, or GDN for short, 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. Efficiency is achieved by efficiently replicating the software near to the users. The GDN does not require servers hosting replicas to be trustworthy. The server capacity required to host the replicas of the software can therefore be donated by untrusted volunteers. To protect these volunteers against legal action the GDN takes special measures to prevent the distribution of illicit content. The GDN has high availability and well-defined semantics when failures can no longer be masked.

We chose the distribution of freely redistributable software (henceforth free 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, resulting in an application that is large both in terms of numbers of users and in the amount of data that needs to be handled. The application also has interesting security aspects. Unauthorized modification of the software being distributed must be impossible and neither should malicious persons be able to use the GDN to illegally distribute copyrighted or illicit material. What makes the security aspects particularly interesting is our intention to let the GDN use spare server capacity provided by anyone who wishes to contribute, implying that the majority of machines used are not to be trusted. This design goal also makes making the application fault tolerant more challenging.

A second reason for choosing software distribution is that the current Internet applications for distributing information 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.

The remainder of the article is structured as follows. We start with a description of the basic architecture of the Globe Distribution Network and provide the necessary background on the Globe middleware platform. This description will explain the basic operation of the application. The next two sections present our security and fault tolerance measures, respectively. We then continue with a description of our current implementation and conclude with future enhancements to the design presented here.

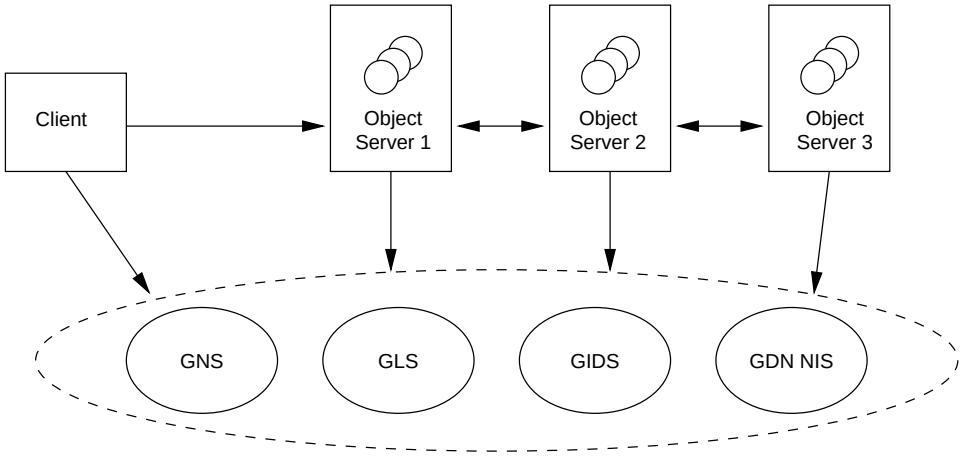

Figure 1: The basic architecture of the Globe Distribution Network (squares represent processes, ovals represent (distributed) services, arrows represent communication). Clients download software packages from a collection of object servers. Being a Globe application, the GDN depends on the three standard Globe middleware services: the *Globe Naming Service (GNS)*, the *Globe Location Service (GLS)* and the *Globe Infrastructure Directory Service (GIDS)*. The *GDN Network Information Service (GDN NIS)* is a GDN-specific service handling key distribution and validation.

ARCHITECTURE

The architecture of the Globe Distribution Network is shown in Figure 1. The core of the Globe Distribution Network is formed by a collection of *Globe object servers*. A Globe object server is a user-level process that stores and manages replicas of a subset of the software packages being distributed through the GDN.

What is special about the Globe Distribution Network is that the replication of software packages is not handled by the object servers directly. Instead, software packages are encapsulated in Globe's *distributed shared objects (DSOs)* which manage the replication and location of their state (the software package) and all other nonfunctional aspects themselves [1].

Clients downloading software from the GDN connect to the most convenient (e.g. geographically nearest) object server that holds a replica of the object containing the desired software package. To find this most convenient replica, clients perform a two-step lookup process. In the first step, the symbolic name of the software-package object is resolved to a binary *object handle*. The object handle of a software-package object is its permanent identifier that does not change during the lifetime of the object. This resolution step is carried out by the *Globe Naming Service (GNS)* [2].

In the second step, the object handle of the package object is mapped to the *contact address* of its nearest replica. This contact address contains, among other items, the IP address of the object server running the replica. We have developed a special service for mapping the location-independent object handles to the actual replica locations, called the *Globe Location Service (GLS)* [3]. The special property of this service is that its lookup costs are proportional to the distance between client and replica. So, if a replica is located near the client, lookup costs are low. Using the information in the contact address the client constructs a proxy for the software-package object and uses this to retrieve the software from the object.

Software-package objects replicate themselves over the object servers following current client demand and the history of the object. This object-controlled automated replica management not only

makes things easier for the publisher of a software package, but also allows faster and effective response to sudden increases in popularity. When the popularity of a certain software package suddenly rises (e.g., there is a new version and everybody wants to download it) the software-package object locates additional object servers and requests them to create a new replica. The additional object servers are located using the *Globe Infrastructure Directory Service (GIDS)* which keeps track of the object servers available worldwide [4]. When popularity drops and it becomes inefficient to maintain a replica at a certain object server the object removes the replica and deregisters it from the Location Service.

Before we go into how software packages are encapsulated in Globe distributed shared objects we first describe how these automatically replicating distributed objects are actually implemented.

Globe's distributed-object model

The fundamental idea underlying the Globe middleware is that a distributed object should have complete control over its (distributed) implementation. In Globe a distributed object manages all its non-functional aspects, such as transport of method invocations, location and replication of its state, and security itself, using only a minimum of supporting services [1]. We call this model of distributed objects the *distributed shared object (DSO) model*. Focusing on the management of the replication of the state, having control over one's implementation means, concretely, that proxies and replicas of a Globe distributed object contain all code for doing replication and (group) communication in an object-specific way.

A replica of a Globe distributed shared object typically consist of 4 modules, or *subobjects* as we call them, as illustrated in Figure 2. The *replication subobject* (labeled R in the figure) contains the implementation of the replication protocol used by this object. The *communication subobject* (Co in the figure) satisfies the replication subobject's communication needs, for example, by offering reliable group communication primitives. The *semantics subobject*, labeled S in the figure, contains the actual implementation of the object's methods and logically holds the state of the object. As illustrated in the figure, the state may actually be stored on local storage, but this fact is transparent to the other subobjects. The *control subobject* (labeled Ct) manages the interaction between the replication protocol and the object implementation and bridges the gap between the application-defined interfaces of the semantics subobject and the standardized interface of the replication subobject. Proxies of distributed shared objects have a similar modular structure. A typical proxy consists of only a communication, a control and a replication subobject.

The replication subobject is the component that monitors client traffic. When subobject detects that a lot of clients are located in a particular geographical region and determines that it is more efficient to create a replica there, it contacts the Globe Infrastructure Directory Service to find object servers to create new replicas on. This process of network-load balancing is described in detail in [5]. The current Globe implementation does not support server load-balancing, but we expect that well-known techniques to do this will be incorporated in the future.

When a new replica or proxy is created, its subobjects are loaded dynamically into an object server or client, respectively, from a trusted *implementation repository*, comparable to automatic class loading in Java. In the case of a proxy and for all but the initial replica, the information detailing which subobjects to load is contained in the contact address which the process obtains from the Location Service in the two-step lookup process described in the previous section.

The modularization of the code and standardization of subobject interfaces, combined with dynamic loading not only allows us to easily select different replication protocols for different distributed objects, but also allows the introduction of new replication protocols without changing the object

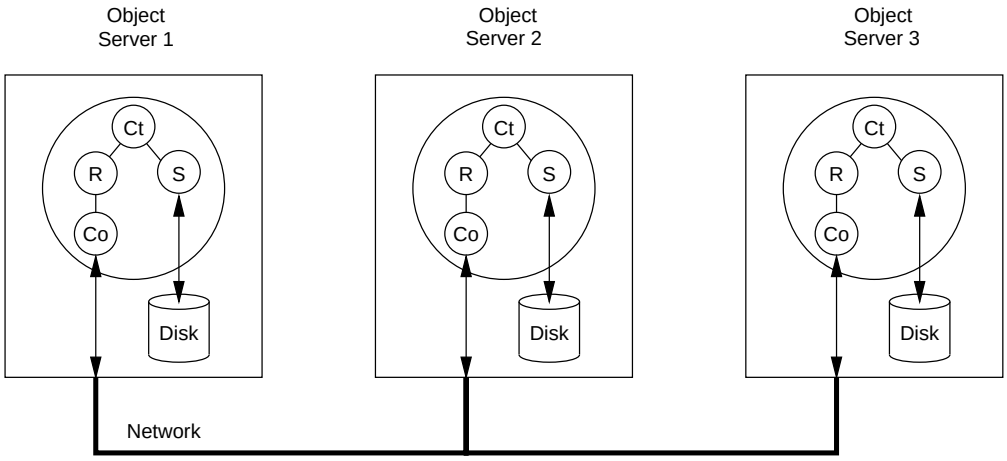

Figure 2: Three object servers each hosting a replica of a Globe distributed shared object. Each replica consists of four so-called *subobjects*, labeled Ct, R, S, and Co.

servers. In our current design of the Globe Distribution Network we use a single replication protocol that is compatible with our security requirements. We come back to this aspect in the section on security.

Mapping software packages to Globe objects

The actual mapping from software package to DSOs involves a number of issues we have not yet discussed. To explain the details of the mapping we first introduce some terminology.

A *software package* in our approach is an application, a library, or any piece of software that is published as a separately named entity. We assume that a software package may continuously evolve as bugs are fixed, new functionality is added, or when it is adapted to changing library APIs. This evolution results in a tree of *revisions*, that is, versions that are meant to replace other, earlier versions. Each revision of a package can have a number of *variants*, that is, versions somehow derived from a revision that are not meant to replace it, but instead coexist with that revision [6]. 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.

The mapping of a software package to distributed shared objects is as follows. We encapsulate each revision of a software package along with all its variants in a single DSO. We refer to such a distributed shared object as a *revision object*. A variant may be distributed in multiple file formats, either a generic file archive format (ZIP, GZIP-ed TAR) or a specialized format for packaging software, such as Red Hat's RPM [7] or Debian's DEB format [8]. In other words, a revision object is basically a collection of archive files, containing the different variants of a particular revision of a software package.

Consider the following example to illustrate our mapping scheme. The GIMP application manipulates images in various image formats. In the Globe Distribution Network the package would be published as a set of revision objects, one for each published revision. The revision object encapsulating revision 1.1.29, for example, would consist of the source code in `tar.gz` format and binaries for Linux on i386 and Alpha processors in `.deb` and `.rpm` package formats.

We chose this mapping because it allows us to apply different replication strategies to different

revisions. Popular new revisions can be replicated on many hosts, while older revisions of a software package need to be replicated on just a few archive sites. This approach allows for efficient use of the available resources. We may switch to an alternative mapping where each individual variant is encapsulated in a separate DSO in the future, if our initial mapping turns out to be too coarse-grained.

Uploading into or downloading an archive file from a revision object is done by invoking the methods of the distributed shared object. To upload a file a user first calls the `startFileAddition` method passing the name and size of the file to be uploaded as arguments. The content of the file is uploaded in large blocks using a series of invocations of the `putFileContent` method. When the upload is finished the user calls `endFileAddition` which finalizes the upload and makes the file accessible for download. The archive files are written to persistent storage by the semantics subobject. Downloading is also done in large blocks using the `getFileContent` method.

Having explained the basic operation of the Globe Distribution Network we now discuss its security design and how it is made fault tolerant. We focus on the GDN application; security and fault tolerance of the supporting, application-independent services such as the Globe Location Service are not discussed here, see, for example, [9]. For the remainder of this article these services are therefore assumed to be fault tolerant and run by a trusted organization on trusted hosts.

SECURITY

The security design of the Globe Distribution Network addresses three issues:

1. How to guarantee the authenticity and integrity of the software being distributed.
2. How to prevent the illegal distribution of copyrighted works or illicit material via the GDN.
3. How to guarantee the availability of the GDN given our design goal to allow object servers to run on untrusted machines. This design goal enables anyone with a permanent Internet connection to run an object server and participate in the GDN. Measures must be taken to prevent attackers from disrupting the GDN by running maliciously modified object servers.

We discuss the issues of authenticity and integrity, content liability, and availability in turn.

Authenticity and integrity of the software

People downloading software from the Globe 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 a malicious Trojan horse?

In our design, 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 gives only very limited authenticity guarantees, by providing the verified name of the person who uploaded the software (which is recorded for traceability reasons, as we explain later). Stronger 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 [10]. 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*. Vital to this authentication scheme is 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.

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 key ring supplied by the downloading user. If the key ring does not contain the required public key, the user is prompted to supply it. People can, of course, choose to do end-to-end signature verification themselves (using, for example, PGP [11]) if they do not trust the GDN client software to do this faithfully.

Most important reason for not having the GDN provide strong authenticity guarantees is that we expect GDN users not to trust any statements the GDN makes about the authenticity of the software they download. We expect GDN 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 object 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 object 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.

Content liability

We must take action to prevent the illegal distribution of copyrighted works or illicit content via the GDN so that the owner's of object servers 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 [12]. The same risk of liability exists for pornography and other illicit materials.

Avoiding the problem of liability by ensuring that no illegal content is uploaded into the Globe Distribution Network is practically impossible. The only solution is to manually check each piece of content before it is allowed onto the network. Manual checks are error prone and may be defeated by cleverly encoding illicit content into inconspicuous content. We can, therefore, try only to limit distribution of illegal content.

We believe that manual checks at upload time are an unsuitable mechanism for limiting the amount of illegal distribution. Manual checks at upload time, or *content moderation* as we refer to it, has several disadvantages. Unpacking software archives and checking them for illicit content is tedious work. In addition, if there is little abuse, we expect the people doing the moderation to perceive the work as superfluous. Moreover, content moderation introduces a delay between the initial submission for publication and the actual publication in the distribution network. We expect that software maintainers wanting to publish via the GDN will find this delay irritating.

Given the disadvantages of content moderation we chose a different, novel solution to limit the illegal distribution of content in the Globe Distribution Network. All content that is published through the GDN is made traceable to the person who published it. If it is discovered that a person published

inappropriate content through the GDN all content published by that person is immediately removed and he or she is banned from the GDN. Intuitively, the GDN is 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.

Implementing content traceability

Content traceability is implemented in the GDN as follows. If someone wants to start publishing the software he maintains via 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 means of identification. In addition, the organization checks with the other AGOs to see if this person has not been banned from the GDN. If the candidate is clean, the Access-Granting Organization creates a certificate [10] linking the identity of the candidate to a candidate-supplied public key and digitally signs this certificate. This certificate allows the candidate to upload content into the GDN. We call a person who is allowed to upload content a *GDN producer*. We call the key pair of which the public key on this certificate is one part the *trace key pair*. The trace key pair may be the same key pair as used for the end-to-end signature but this is not required.

A producer signs all content that he uploads into the GDN using the trace key pair. We call this the *trace signature* to distinguish it from the end-to-end signature. Concretely, the `startFileAddition` method invoked at the beginning of an upload has two additional arguments: a digital signature created with the trace private key, and the certificate containing the trace public key signed by the Access-Granting Organization. The trace signature is created automatically by the GDN upload tool. When the upload is finished and the producer calls `endFileAddition` the object verifies the trace signature. When the signature is false (either because the producer has been banned from the GDN, the certificate did not contain the right public key, or the file did not match the digital signature) the object removes the uploaded file from its state. This procedure guarantees that all content in the GDN is traceable to a particular producer.

Object-server owners can decide which producers they want to give access to their object servers by specifying which AGOs they trust to do a proper identity and black-list check. Only producers that have certificates signed by those AGOs will be allowed to place content on the object server. Object-server owners can also block individual producers.

Revoking access to the GDN

To ban a producer from the GDN when illicit content traceable to him is found, the following procedure is executed. When a downloading user or object-server owner 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 violator access of the presence of illicit content. The Access-Granting Organization in addition receives a copy of the signed illicit content and verifies that this content is indeed inappropriate and signed by the violator. If this is the case, the violator is then placed on the central black list shared by all AGOs and is thus banned from the GDN.

The actions taken by the object-server owners depend on their 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 system-wide policy to guarantee certain system-wide properties with respect to illegal distribution. We currently require object servers to follow a system-wide policy where all content published by a violator is deleted, but only after verification of the evidence.

The reason accusation is delegated to a producer is to keep the number of accusations an AGO has to process low. More specifically, the accusing producer will be banned himself if the accusation he makes proves false. The system-wide policy provides protection against malicious GDN producers trying to remove well-known software packages from the GDN.

Discussion

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, "provider[s] of online services", such as Internet Service Providers can request legal protection from copyright infringements by their users. Under this protective measure, copyright holders cannot seek compensation from the service providers for these infringements. To receive this legal protection ISPs are required only to remove the copyrighted content once they have been notified by the copyright holders [12].

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 immediately and its uploader 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 ousted from the GDN.

What this scheme currently does not fully take into account are the differences between countries of what content may be legally distributed. 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 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 as anything that is not freely redistributable software or part thereof.

Ensuring the availability of the Globe Distribution Network

The GDN 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 next section.

Recall that our design goal is to make anyone with a permanent Internet connection a candidate for running an object server for the GDN. This design goal creates a vulnerability as people may attempt to undermine the availability of the GDN from the inside by running a modified and maliciously acting object server. We, as GDN designers, do not have and can never have complete control over object-server machines and thus cannot prevent this malicious behavior. We, therefore, take measures which to make sure these denial of service attacks have little effect.

We divide our discussion on countermeasures into two parts. We first discuss measures that protect against malicious object servers trying to affect the operation of other object servers. After that, we discuss the measures that protect a downloading user against a misbehaving object server. We do not consider denial-of-service attacks by network flooding.

Protecting object servers

Object servers can maliciously affect other object servers by sending fake replication-protocol messages. In particular, they can send fake state-update messages (i.e., method invocations, a new version of the state, and state invalidates) and sabotage collective decisions, for example, by reporting failure in a two-phase commit protocol or faking replies from other object servers during such decisions.

Our first measure for protecting good object servers is to have all revision objects use a safe replication protocol. In this replication protocol each object has a small number of so-called *core replicas*. These replicas run on machines trusted by the owner of the object (a GDN producer) and have access to the producer's trace private key (or a derivative thereof [13]). In addition to these core replicas, an object can have a number of replicas hosted by untrusted machines. Untrusted and core replicas accept only state-update messages that are signed with the producer's trace private key. Given that only the producer and the core replicas have access to this key, no untrusted replica (i.e., malicious object server) can modify the state of any other replicas.

The second measure is to limit the number of collective decisions, as we illustrate with the following two examples. When a state-modifying method, for example, `deleteFile` is invoked, only the core replicas dictate whether or not this update operation succeeds. As a consequence, when all core replicas have successfully executed a state-modifying method, but an untrusted replica cannot perform the update for whatever reason, the operation on the object is never rolled back. Instead, the untrusted replica is no longer considered a part of the object and is left to destroy itself.

Another example of limiting collective decisions relates to replica placement. We make each untrusted replica decide for itself if there is a need for a new replica and where it should be placed. In some cases better decisions could be made by taking the load and geographic location of more replicas into consideration, but that requires replicas not to sabotage this collective decision.

Not all cooperation between untrusted object servers can be avoided, however, so there can still be some interference from malicious object servers. For example, when an object server detects an influx of traffic from a particular region it will ask an object server in that region to create a new replica. The latter (malicious) object server could grant the request, but kill the new replica just after, thus hindering the former object server. We provide some limited means to deal with these types of situations. Object-server owners are allowed to specify which object servers they want to cooperate with and can block others (e.g. by blocking certain IP-address ranges). These rules are used in selecting a candidate object server (using the Globe Infrastructure Directory Service) and to evaluate "create replica" requests the object server receives itself.

Protecting downloading users

It is important to realize that an object server can be only obnoxious to a downloading user since any malicious modifications to the software are detected by the end-to-end authenticity and integrity checks discussed in the previous section.

One source of interference with normal operation is fake contact addresses in the Globe Location Service (GLS). Object servers need to register the replicas they host in the GLS such that downloading clients can find them. Object servers should, however, not be able to insert fake addresses. We implement this requirement as follows. Object servers are not allowed to register a contact address for a certain Globe object, unless they can present the *GLS-access ticket* for this object to the GLS. A GLS-access ticket is basically the object handle of the object signed by the GDN producer that created the object and is given to each object server in the "create replica" request. So to register a fake address an object server must first have been asked to create a replica of the object by one of the existing replicas, limiting the possibility of malicious object servers inserting fake addresses considerably.

Even if object servers have been asked to create a replica of a certain object they can still hinder a downloading user by putting different content in that replica. This content could even be traceable (i.e., a malicious object server could serve us the content of a totally different object) which means that users will not notice the problem until they do the end-to-end authenticity check. This problem makes the end-to-end authenticity check absolutely vital to the secure downloading of software from the GDN. Again, by allowing 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) we give users a way to also protect themselves against this type of misbehavior.

FAULT TOLERANCE

The Globe Distribution Network should be able to handle failures of hosts and networks. More concretely, we want the GDN to:

1. be highly available; that is, it must be up and running most of the time.

2. exhibit failure semantics by which an operation on the GDN is a transaction, and either gives a complete and correct response or otherwise reverts back to the state before the operation and reports an error.

Having the GDN meet the second requirement makes it easier to use. If someone tries to upload a new version of a software package and the upload fails, he does not have to worry about some object servers keeping a partial copy.

To make the Globe Distribution Network highly available we make use of the fact that the software packages are replicated to make their distribution efficient. Because object servers do not have to be trusted we expect there will be no shortage of object servers. We can therefore safely assume that all software packages will have more than one replica. This assumption allows us to transparently fail-over to another replica in most failure situations and thus provide uninterrupted service.

In our safe replication protocol some replicas have a special status, however, as we saw in the previous section. In particular, our safe replication protocol depends on the presence of at least one trusted core replica that assists in the creation of other replicas on other, untrusted machines, and which controls updates. To ensure availability of this central component we require a revision object to have multiple core replicas. We do not expect a shortage of trusted object-server capacity since these

servers have to be trusted only by the maintainer of a particular software package. Given that different maintainers will trust different machines, we expect a natural distribution of trust and therefore core replicas over the collection of available object servers.

We benefit not only from the replication of the software packages, but also from the fact that software packages are stored persistently on local storage by the replica's semantics subobject (see above). By also making the internal state of the replica and administration of an object server persistent we can create a highly available object server. When an object server suffers from a nonhalting crash, it can restore the replicas of the revision objects it was hosting and thus quickly resume service, increasing availability.

Making operations on the Globe Distribution Network transactional, in particular uploads, is more complex. We discuss the two most important operations on the GDN, downloads and uploads, in turn.

Making downloads transactional is easy. In most cases a download will be successful given that the download tool can fail-over to other replicas. 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 only involves deleting the incomplete file from the downloading user's disk. This can be done by the download tool, immediately or when it is restarted.

As a convenience to the GDN user we take special measures to allow a user to continue a download after a crash of his download tool or temporary unavailability of the revision object, as follows. We assume that after a crash the user will restart the tool and have it continue the download. The download procedure is now as follows. The GDN download tool starts a download by retrieving the desired file's trace signature from the revision object and storing it on disk. If the tool crashes during this 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 this signature, the tool checks if any of the already downloaded blocks of the file were damaged during the crash and, if so, downloads these damaged blocks again.

To enable this behavior, our trace signatures are special signatures that can be used to check 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 [10] of the individual blocks and a cryptographic digest of the whole file encrypted with the producer's trace private key (required for traceability). The download tool can now detect any damage on the blocks by first decrypting the trace signature using the producer's trace public key and then recalculating the digest of each downloaded block and comparing it to the block's digest in the trace signature.

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.

Creating transactional uploads is very complex. Source of the complexity is the fact that uploads into the GDN consist of multiple method invocations (we upload files in blocks, as described above). The Globe middleware currently lacks support for transactionally executing a series of method invocations, therefore we have to resort to an *ad hoc* solution. We choose a solution where the uploading user is responsible for the rollback, making uploads a nontransactional operation. However, this solution strives to make the upload succeed whenever possible. We consider developing a proper rollback mechanism outside the scope of this research and given we expect the number of uploads into an

object to be low, we consider the current solution sufficient for now.

Our solution is as follows. We distinguish three types of failures that can happen during an upload: 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, when a replica, either core or untrusted, crashes during the upload it is pronounced dead and no longer considered part of the object even if the object server recovers. When all core replicas fail, this is detected by the remaining untrusted replicas which destroy themselves thereby destroying the object.

To handle the second type (noncrash-replica failures) we take two measures. The first measure is to reserve the required disk space at the start of the upload. As explained above, 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.

The second measure to deal with noncrash-replica failures is to adopt the following update strategy, implemented as part of our safe replication protocol. All write-method invocations are first sent to the core replicas. The core replicas forward the invocation to all untrusted replicas and also carry out the method themselves. Every untrusted replica that fails to execute the method destroys itself. As for the core replicas, each one reports the result (failure or success) of the method execution to its 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.

This procedure results in a successful upload if at least one of the core replicas succeeds in carrying out all the method invocations. The only case where these measures are not sufficient is when all core replicas fail to execute the method. In this case, the core replicas instruct the remaining untrusted replicas to destroy themselves. The core replicas will not destruct themselves, instead they report a failure of the method invocation to the upload tool. It now is the responsibility of the uploading user to rollback the upload by deleting the partially uploaded file from the revision object.

This is the same rollback an uploading user will have to perform to recover from the third type of failure we distinguish: a crash of the upload tool. We take no extra measures to handle this type of failure.

Apart from problems that directly influence the correct functioning of the Globe Distribution Network there are a number of problems caused by failures that merely affect the performance of the GDN. In particular, failures of object servers can cause the Globe Location Service and Globe Infrastructure Directory Service to be out of sync with the actual situation. These problems are performance problems, because, for example, a downloading client fails over to another replica if the first one does respond quickly and an object server will ask for another candidate object server if it does not get a reply to its “create replica” request.

When considering the Location Service, inconsistencies are avoided by having object servers that suffered nonhalting crashes bring the Service in sync again when they reboot by deregistering the contact addresses of replicas that could not be recovered. Halting crashes are dealt with by having object servers periodically register contact addresses again. Periodical reregistration is also used to fix inconsistencies in the Infrastructure Directory Service.

CURRENT IMPLEMENTATION

A first version of the Globe Distribution Network has been up and running since October 2000. It currently spans four European sites: the Vrije Universiteit and the NLNet Foundation in the Netherlands,

INRIA Rocquencourt in France and the University of Erlangen in Germany. We expect to include a site in the United States soon. All code is written in the Java programming language. Since a large part of the functionality of the Globe Distribution Network comes from the Globe middleware we start with the implementation status of the middleware and discuss the status of the GDN application thereafter.

The Globe middleware services

The Globe Naming Service as found in the current Globe middleware is a prototype version based on the *Domain Name System (DNS)* [14] and works as follows [15]. Globe object names have a one-to-one mapping to valid DNS names. These DNS names point to a TXT DNS Resource Record that contains the encoded object handle for the DSO. To map a Globe object name, say `/nl/vu/cs/globe/somePackage`, to a Globe object handle, the object name is first translated to a DNS name, in this case `somePackage.globe.cs.vu.nl`. This DNS name is then resolved using the normal DNS name resolution mechanism and returns a TXT record from which the object handle is extracted.

In the current implementation of the Globe Distribution Network software maintainers therefore have to setup their own DNS leaf domain to register Globe object names or contact another maintainer willing to make registrations for them. We have setup a specific DNS domain called `software.gns-dns.globeworld.org` where we register names on request. This domain is subdivided into categories to allow object names such as `/org/globeworld/gns-dns/software/os/Linux/kernel/v2-2-18`. We do not check if the person making the request is actually the maintainer of the package, because that is very hard to establish as we discussed in the section on authenticity of the distributed software. Therefore, downloading users must retrieve the object-name-to-package binding from a trusted source, for example, the package's well-known Web site. We intend to replace the DNS-based prototype with an implementation based on distributed shared objects [2] in the future.

The basic functionality of the Globe Location Service, that is, mapping location-independent object handles to contact addresses in a scalable way has been completely implemented. This implementation is described in full detail in [16].

The Globe Infrastructure Directory Service (GIDS) has been implemented but not yet integrated into Globe. It uses the Light-weight Directory Access Protocol (LDAP) and standard LDAP servers [17]. The GIDS divides the world into a set of base regions. Per base region there is an LDAP server, called the *Regional Service Directory* that keeps track of the available object servers and their properties (such as amount of memory and disk space available, required authentication method, etc.). 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 [4].

The Globe object server

The Globe object server is the Globe middleware's application-independent platform for running replicas of distributed shared objects. It 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 and processes the object-management commands the server receives over the network, such as "create replica" and "destroy replica." 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 such as our revision objects use this interface to store and retrieve 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.

A Globe object server provides facilities for replicas to survive the graceful shutdown and restart of a server. When an object server is shutdown by its owner 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 would change 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.

The Globe Distribution Network

The current implementation of the Globe Distribution Network supports the basic functionality for uploads and downloads and replication. The security and fault tolerance measures described have not yet been incorporated. The implementation consists the code for the revisions objects that encapsulate the software, an upload tool and a HTTP-to-Globe gateway used for downloading software via a standard Web browser.

The code of the revision object consists of a semantics subobject and a replication subobject. The GDN semantics subobject implements the block-based interface for uploading and downloading files into a revision object, described earlier. It interfaces with the persistence manager of its host object server to read and write the archive files that make up the revision object's state to disk. The replication subobject implements a master/slave replication protocol that can handle distributed shared objects with very large state. It does not yet support any of the security and fault tolerance features discussed above. The upload tool can be used to upload files into a revision object and for manually managing the replication of a revision object. We do not currently have support for automatic replication.

The HTTP-to-Globe gateway enables the downloading of software packages via a standard Web browser. This gateway works as follows. We use special URLs that have encoded in them the name of a revision object and the name of the desired archive file (e.g. `source.tar.gz`) in that object. The HTTP-to-Globe gateway extracts the object name from the URL and resolves it to a contact address using the two-step lookup process involving the Globe Naming Service and the Globe Location Service. Using this contact address the gateway constructs a proxy of the desired revision object. It then repeatedly invokes the method `getFileContent` on the revision object to retrieve the contents of the desired archive file. The contents of the file are passed to the requesting browser which saves them to disk.

We have started implementing the security and fault tolerance measures described in this article and expect to finish their implementation by the end of 2001.

CONCLUSIONS

The Globe Distribution Network (GDN) is an application for the efficient distribution of freely redistributable software packages. It has been developed as a test application for a new middleware platform called Globe which is designed to facilitate the development of large-scale Internet applications. Distribution of the free software is made efficient by encapsulating the software into Globe distributed shared objects and having efficiently replicating the objects near to the clients downloading the software. Replication of the software is automated because distributed shared objects manage their replication themselves based on past and present client demand. An important feature of the Globe Distribution Network is that it can use untrusted servers to replicate the software objects on.

Instead of doing content moderation at upload time to prevent the illegal distribution of copyrighted material or other illicit content, the Globe Distribution Network takes a novel approach where publishers are given direct access to the network. In this optimistic approach all content uploaded into the network is made traceable to its publisher (by means of digital signatures) allowing illicit material to be removed from the GDN immediately after it is found and the publisher of this material to be banned from the GDN. The Globe Distribution Network exploits the replication of the software to achieve high availability and has well-defined failure semantics when failures can no longer be masked.

Our experiences with the current implementation of the GDN in a test spanning four European sites are promising. We are currently working on a more complete implementation of the described design and intend to expand our experiments to involve more sites, in particular in the United States. In the mid-term future we plan to add support for facilitating the management of many different versions of a software package and downloading groups of related packages. The source code for both the Globe Distribution Network and the Globe middleware platform are freely available under the BSD software license [18].

Acknowledgments

We would like to thank Chandana Gamage, our staff programmers, Patrick Verkaik and Egon Amade and our sponsor, the NLNet Foundation for their support in the development of Globe and the Globe Distribution Network.

References

[1] van Steen M, Homburg P, Tanenbaum AS. Globe: a wide-area distributed system. *IEEE Concurrency* 1999; **7**(1):70–78.

[2] Ballintijn G, van Steen M, Tanenbaum AS. Scalable naming in global middleware. In *Proceedings of the 13th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-2000)*, Chaudhry G, Sha E (eds.). ISCA: Cary, NC; 624–631.

[3] van Steen M. Hauck FJ, Tanenbaum AS. Locating objects in wide-area systems. *IEEE Communications Magazine* January 1998; 104–109.

[4] Kuz I, van Steen M, Sips HJ. *The Globe Infrastructure Directory Service*. Technical Report IR-484, Division of Mathematics and Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam, 2001.

[5] Pierre G, Kuz I, van Steen M, Tanenbaum AS. Differentiated strategies for replicating Web documents. *Computer Communications* 2000;**24**(2): 232-240, Elsevier Science: Amsterdam.

[6] Conradi R, Westfechtel B. Version models for software configuration management. *ACM Computing Surveys* 1998; **30**(2): 232–282.

[7] Bailey E. *Maximum RPM – Taking the Red Hat Package Manager to the Limit*. Red Hat Press: Durham, NC, 1998.

[8] Jackson I, Dienes K, Morris D, Schwarz C. Debian Packaging Manual. http://www.debian.org/doc/packaging-manuals/packaging.html/ [2 January 2001]

[9] Ballintijn G, van Steen M, Tanenbaum AS. Simple crash recovery in a wide-area location service. In *Proceedings of the 12th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-1999)*, Olariu S, Wu J (eds.). ISCA: Cary, NC; 87–93.

[10] Schneier B. *Applied Cryptography* (2nd edn). John Wiley & Sons: New York, 1995.

[11] Zimmermann P. *The Official PGP User's Guide*. MIT Press: Cambridge, MA, 1995.

[12] *Digital Millennium Copyright Act*, United States Public Law No. 105-304, 28 October 1998.

[13] Mambo M, Usuda K, Okamoto E. Proxy signatures for delegating signing operation. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*. ACM: New York, NY, 1996; 48–57.

[14] Albitz P, Liu C. *DNS and BIND* (3rd edn). O'Reilly and Associates: Sebastopol, CA, 1998.

[15] Bakker A *et al*. The Globe Distribution Network. In *Proceedings of the 2000 USENIX Annual Technical Conference (FREENIX track)*. USENIX Assocation: Berkeley, CA, 2000; 141–152.

[16] Ballintijn G, Verkaik P, Crawl D, van Steen M. *The Globe Location Server*. Technical Report (in preparation), Division of Mathematics and Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam, 2001.

[17] Loshin P (ed.). *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*, Morgan Kaufmann Publishers: San Francisco, 2000.

[18] GDN Download Page. http://www.cs.vu.nl/globe/ [2 January 2001]