A Law-Abiding Peer-to-Peer Network for Free-Software Distribution

Arno Bakker, Maarten van Steen, Andrew S. Tanenbaum
Vrije Universiteit Amsterdam
Department of Mathematics & Computer Science
De Boelelaan 1081a, 1081 HV
Amsterdam, The Netherlands
{arno,steen,ast}@cs.vu.nl

Abstract

The Globe Distribution Network (GDN) is an application for worldwide distribution of freely redistributable software packages. The GDN 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 objects containing the software can run on untrustworthy servers. A first version of the GDN has been implemented and has been running since October 2000 across four European sites.

1. 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* [9]. 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 of this application, in particular its security aspects.

The Globe Distribution Network, or GDN for short, is a peer-to-peer network 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 replicating the software near to the downloading clients. 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.

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.

The remainder of the article is structured as follows. We start with a description of the basic architecture of GDN, and explain the basic operation of our application. Section 3 comprises the major part of the paper, in which we present our security measures. The current implementation is briefly discussed in Section 4. We conclude in Section 5.

2. Architecture

The architecture of the Globe Distribution Network is shown in Figure 1. The core of GDN 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. As shown in Figure 1, 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 Access Control Service (GDN ACS)* is a GDN-specific service handling key distribution and validation. These services are discussed next.

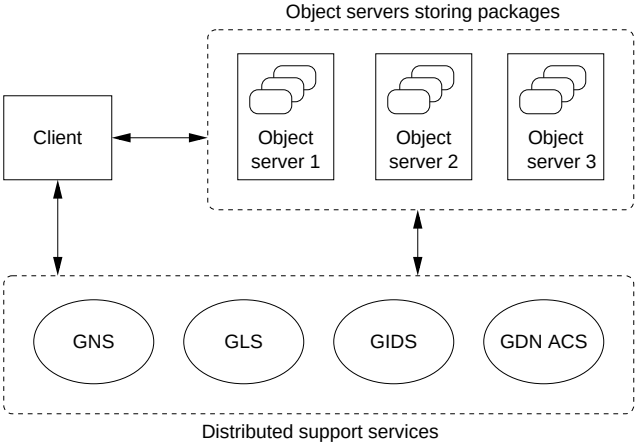

Figure 1. The basic architecture of the Globe Distribution Network.

Clients downloading software from the GDN connect to the most convenient (e.g. geographically or network-topologically 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 location-independent *object handle*. The object handle of a software-package object is its permanent identifier that does not change during the lifetime of the object [10]. This resolution step is carried out by the *Globe Naming Service (GNS)* [1].

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)* [8]. 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 (i.e., flash crowds). 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 [3]. 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.

2.1. Mapping software packages to Globe objects

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 [5]. 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.

Software packages are encapsulated in Globe's *distributed shared objects (DSOs)*. A distinguishing feature of a DSO is its ability to manage the replication and location of its state, as explained in [9]. 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 or Debian's DEB format. 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 GDN, 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. Downloading is also done in large blocks using the getFileContent method.

3. Security

Having explained the basic operation of the GDN we now discuss its security design. We focus on the GDN application; security of the supporting, application-independent services such as the Globe Location Service are not discussed here, see, for example, [2]. For the remainder of this article these services are therefore assumed to be fault tolerant and run by a trusted organization on trusted hosts.

The security design of the GDN 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.

3.1. 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 [6]. 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.

The 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.

3.2. Content liability

We must take action to prevent the illegal distribution of copyrighted works or illicit content via the GDN so that the owners 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 [7]. 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 GDN 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 also 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. Furthermore, 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 GDN. 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.

3.2.1. 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 [6] 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.

The organization owning an object server can decide which producers it wants to give access to its object servers by specifying which AGOs it trusts 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 that organization's object server. Organizations can also block individual producers.

3.2.2. 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. This policy provides protection against malicious GDN producers trying to remove well-known software packages from the GDN.

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. 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. Although blocking the accusing producer himself when the accusation is false may seem like a drastic measure, it is necessary in order to keep the amount of work for access-granting organizations low, as we expect that many false allegations will be made if there is no threshold for an accuser in the form of a possible sanction. Alternative sanctions are a future research topic.

3.2.3. 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 [7].

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 be 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.

3.3. Ensuring the availability of the GDN

The GDN should have high availability; that is, it must be up and running most of the time. Two factors influence availability: failures, and deliberate attacks on the GDN. We concentrate only on attacks.

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.

3.3.1. 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 the authority to update the state of the object. Core replicas accept only state-update messages that are signed by the owner's trace private key.

In addition to these core replicas, an object can have a number of replicas hosted by untrusted machines. Untrusted replicas accept only state updates that are either signed by the trace private key or that are cryptographically verified to originate from a core replica. Untrusted replicas can verify the origin of a state update as follows. Each object server has its own public/private key pair and a certificate containing its public key signed with the owner's trace private key. Any state-update messages the object server sends out contain the certificate and are signed with the object server's private key, allowing an untrusted replica to verify that this update comes from a core replica with delegated authority to make updates.

Given that only the producer has access to the trace private key and the core replicas' authorization is tied to their own key pair, no untrusted replica (i.e., malicious object server) can modify the state of any other replicas. An alternative to delegation via certificates is to use *proxy signatures* [4]. As indicated above, the machines running the core replicas have to be trusted by the GDN producer who owns the object. In particular, these machines should be trusted not to send out fake updates and act as trustworthy sources for the state of the object. We expect that finding machines will not be hard, as they require only limited trust (digital signatures prevent malicious modification of the software itself), and we expect well-known object servers will appear (as has happened with FTP) that can be used, in particular, by producers who do not have easy access to trustworthy servers themselves (through befriended users, for example).

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, see Section 2) and to evaluate "create replica" requests the object server receives itself.

3.3.2. 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 the user 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. 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.

4. Current implementation

A first version of GDN 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 two sites in the United States and one in Israel soon. All code is written in the Java programming language.

The current implementation supports the basic functionality for uploads and downloads and replication. The design of our security framework is currently being implemented. The current implementation consists of the code for the revisions objects that encapsulate the software, an object management and upload tool, and a HTTP-to-Globe gateway used for downloading software via a standard Web browser.

The current implementation has proven quite stable and is being used to distribute the GDN software itself, in addition to a number of operating systems (MINIX, Amoeba, RedHat Linux and the Linux kernel). Furthermore, it is currently used to replicate a number of personal Web pages. A live demo can be visited at `http://enter.globeworld.org/nl/vu/cs/globe/proj/clubglobe`. As our priorities lie with the implementation of the security features, we have only recently started analyzing and optimizing the performance of our implementation, the results of which are not yet available. The analysis is done in preparation of a large-scale experiment to distribute the content of a well-known free-software site through the Globe Distribution Network.

5. 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 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.

Our experiences with the current implementation of the GDN in a test spanning four European sites are promising. We are currently working on a complete implementation of the described security 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 (see `http://www.cs.vu.nl/globe`).

Acknowledgments

We 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] G. Ballintijn and M. van Steen. "Scalable Naming in Global Middleware." In *Proc. 13th Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 624–631, Las Vegas, Aug. 2000. ISCA.

[2] G. Ballintijn, M. van Steen, and A. S. Tanenbaum. "Simple Crash Recovery in a Wide-Area Location Service." In *Proc. 12th Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 87–93, Fort Lauderdale, FL, Aug. 1999. ISCA.

[3] I. Kuz, M. van Steen, and H. J. Sips. "The Globe Infrastructure Directory Service." Technical Report IR-484, Vrije Universiteit, Department of Mathematics and Computer Science, Jan. 2001.

[4] M. Mambo, K. Usuda, and E. Okamoto. "Proxy Signatures for Delegating Signing Operation." In *Proc. Third Conf. Computer and Communications Security*, pp. 48–57, New Delhi, India, Mar. 1996. ACM.

[5] G. Pierre, I. Kuz, M. van Steen, and A. Tanenbaum. "Differentiated Strategies for Replicating Web Documents." *Comp. Comm.*, 24(2):232–240, Feb. 2001.

[6] B. Schneier. *Applied Cryptography*. John Wiley, New York, 2nd edition, 1996.

[7] *Digital Millenium Copyright Act*. United States Plublic Law No. 105-304, Oct. 1998.

[8] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems." *IEEE Commun. Mag.*, 36(1):104–109, Jan. 1998.

[9] M. van Steen, P. Homburg, and A. Tanenbaum. "Globe: A Wide-Area Distributed System." *IEEE Concurrency*, 7(1):70–78, Jan. 1999.

[10] R. Wieringa and W. de Jonge. "Object Identifiers, Keys, and Surrogates–Object Identifiers Revisited." *Theory and Practice of Object Systems*, 1(2):101–114, 1995.

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