Design and implementation of the Globe middleware

Report IR-CS-003

Arno Bakker
Ihor Kuz
Maarten van Steen
Andrew S. Tanenbaum
Patrick Verkaik

Department of Computer Science
Faculty of Sciences
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Email: arno@cs.vu.nl
Telephone: +31-20-4447762
FAX: +31-20-4447653

Abstract

The Globe middleware platform for the development of large-scale Internet applications is designed to provide the flexibility that is required to meet the diverse nonfunctional requirements of such applications. We arrive at a flexible middleware by adopting a new model of distributed objects in which objects are in control of all aspects of their implementation, including nonfunctional aspects such as replication protocol and security. This article explains the design of the Globe middleware and its services and gives a detailed description of its implementation. Copyright © 2003 John Wiley & Sons, Ltd.

Keywords: *distributed programming; wide-area networks; distributed objects*

1 Introduction

The Globe middleware platform is designed to enable the development of large-scale Internet applications [1]. What sets it apart from other efforts in this area, such as CORBA or J2EE, is its focus on flexibility. Our assumption is that different large-scale Internet applications have different requirements with respect to performance, security and fault tolerance. Furthermore, we assume that the key to implementing these requirements efficiently lies in exploiting specific properties of the application in these areas. These two assumptions imply that a middleware platform needs to support many different protocols and policies rather than assume that "one size fits all." A middleware layer should therefore be adaptable to the application that uses it, such that the distributed implementation of the application's functions becomes highly efficient and easier to build.

In Globe, flexibility is achieved by basing the middleware platform on a new model of distributed objects, called *distributed shared objects* from which applications are built. A distributed shared object is in control of all aspects of its implementation, including nonfunctional aspects such as replication and security. A distributed shared object can therefore be said to bring its own middleware to the machines it uses, and thus enables a developer to apply application- or even object-specific solutions in the nonfunctional aspects of the object's implementation. The separation of interface and implementation afforded by the object concept hides these differences from the objects' clients.

This article describes the design and implementation of the Globe middleware. We will do so by describing the implementation of a large-scale application, called the *Globe Distribution Network*. The Globe Distribution Network (GDN) is an application for the efficient, worldwide distribution of freely redistributable software packages, such as the GNU C compiler, the Apache Web server, Linux distributions and a great deal of shareware [2, 3]. The article is structured as follows. We first describe the general architecture of the GDN, introducing the basic structure and components of the Globe middleware. Second, we present the programmer's view on application development with Globe. The third section explains the implementation of the various Globe components in more detail. Next, some practical experience with using the GDN are described. The article concludes with a summary and some lessons learned.

2 Application architecture

One of the key concepts underlying Globe is that shared data is encapsulated by distributed objects. Unlike most other object models, objects in Globe can be physically distributed and replicated across multiple machines [4]. Important is that each object not only encapsulates its state and the implementation of operations on that state, but also that it implements its own distribution strategy. In other words, each object separately implements a strategy that governs how its state is partitioned, replicated, and migrated between hosts. Likewise, each object carries its own implementation for security, persistence, and so on.

The Globe Distribution Network is no exception. Files that are to be shared via the GDN are encapsulated in distributed Globe objects. We refer to these objects as *(distributed) storage objects*. A storage object typically contains a version of a software package (also known as a *revision* [5]), including its variants for different platforms in a number of file formats. Different versions of the same software package are contained in separate storage objects. In this way, we obtain the flexibility to adjust the distribution of a software package to its popularity. In particular, old versions may be stored at only a few places, whereas the most recent version of a package is replicated across many sites, for example, in anticipation of flash crowds, or simply for high availability and performance. In

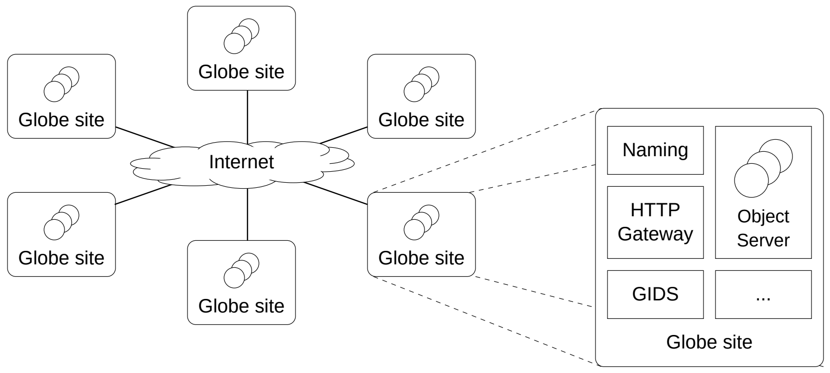

Figure 1: A Globe application runs across a number of Globe sites. Each Globe site consists of an object server hosting the objects comprising the application, and additional servers for, for example, naming of objects and accessing objects via HTTP.

short, GDN allows the distribution of its storage objects to be tuned per object.

The distributed objects used in a Globe application are hosted on a collection of *Globe object servers*. An object server is a user-level process that can host *local representatives* (i.e., replicas) of a large number of objects. A machine running a Globe object server is known as a *Globe site*. In addition to an object server, a Globe site may run additional processes that implement the Globe naming service or allow non-Globe clients to access distributed shared objects, as shown in Figure 1.

A local representative of a distributed shared object is implemented as a local (i.e., nondistributed) object. Such a local object is hosted by an object server similar to the way that local objects in systems such as CORBA [6] and Legion [7] are hosted. A local representative of a GDN storage object logically contains the files that comprise the state of the object. The files are physically stored on disk.

Each storage object in the Globe Distribution Network has an associated human-friendly global name similar to a UNIX pathname. An example of such a name is /com/vendor/gdn/somepkg-v1-0. Names in Globe are globally defined. In other words, all object names together form a worldwide global name space. To access an object in Globe, a client will pass a name for the object to the Globe naming service, which will eventually return the address of an object server hosting a replica of the object. The naming service aims to return the address of an object server that is close to the client. To this end, we have split the naming service into two parts.

The first part supports human-friendly names that are bound to identifiers known as *object handles*. An object handle is a stable and location-independent reference to storage object, comparable to a UUID as used in DCE and now available on many (distributed) computer systems such as Linux. As we explain in detail later, maintaining name-to-object bindings is implemented in Globe by means of DNS.

The second part of the naming service is formed by a separate location service, which is responsible for maintaining the bindings between an object handle and the current addresses of the replicas of a storage object. By separating human-friendly names from addresses via the two-part naming service, we have been able to develop a worldwide scalable service that allows efficient lookups of addresses even if addresses change regularly [8].

The location service returns a contact address describing exactly where and how the requested storage object can be contacted. A contact address usually contains the IP address and port number of an object server along with a server-specific local-representative identifier. Also, the contact address

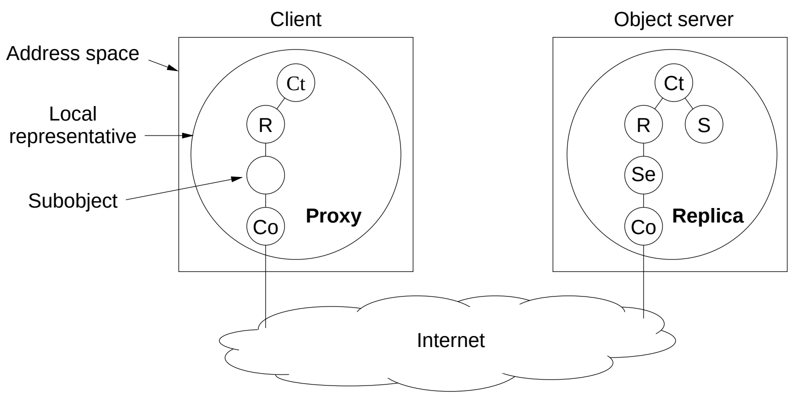

Figure 2: The structure of local representatives. *Ct* stands for control subobject, *R* for replication subobject, *Se* for security subobject, *Co* for communication subobject, and *S* stands for semantics subobject.

contains an implementation identifier, pointing, for example, to a Java jar file containing the code required to setup a connection with the object. Using this information the client constructs its own local representative of the object in its local address space. The client can now transparently access the distributed object by invoking the object's methods on its own local representative. The process of resolving a name with the naming service and constructing a local representative from the returned contact addresses to get access to a distributed object is called *binding*.

As indicated above, Globe objects control their own distribution. In the GDN, storage objects automatically replicate themselves to object servers in areas with many downloaders. Automatic replication not only leads to efficient network usage (most downloaders will download from a nearby server), 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 storage 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 [9]. When popularity drops and it becomes inefficient to maintain a replica at a certain object server the storage object removes the replica and unregisters it from the location service.

2.1 The structure of Globe objects

The local representatives of a Globe object are composed of *subobjects*, modules that take care of a particular aspect of the object's implementation. A replica local representative of an object minimally consists of five subobjects, as illustrated in Figure 2.

Semantics subobject: This subobject is the actual implementation of the distributed object's methods and logically holds the state of the object (which may be on persistent storage physically). It can be developed without having to take many distribution or replication issues into account. Accesses to a semantics subobject are serialized: at most one thread is active in a semantics subobject at a particular point in time. We discuss the rules for writing a semantics subobject in more detail in a subsequent section.

Replication subobject: A Globe object may have semantics subobjects in multiple local representatives (i.e., be replicated) for reasons of fault tolerance or performance. The replication subobject is responsible for keeping the state of these replicas consistent according to the consistency model chosen for this particular distributed object. In addition, the replication subobject may also manage the degree and placement of replicas (i.e., how many replicas the object should have and where they should be located). In other words, the replication subobject is in charge of the object's distribution strategy. To perform this task, the subobject communicates with its peers in other local representatives using an object-specific replication protocol. Different distributed objects will use different (sets of) replication subobjects depending on the protocol chosen. An important aspect of replication subobjects is that there is one standardized interface for all replication subobjects. This also holds for the communication subobject, allowing us to build a large library of such subobjects that are reusable by other applications.

Security subobject: The security subobjects implement the security policy for the Globe object. In particular, a security subobject controls access to the local representative and ensures that the local representative talks only to authorized peers. To perform these tasks it can use different security protocols and keying schemes. The Globe security framework is described in detail in [10], and is currently being built into our Globe implementation.

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

Control subobject: The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. This subobject is needed to bridge the gap between the programmer-defined interfaces of the semantics subobject, and the standardized interface of the replication subobject. For example, the control subobject marshalls and unmarshalls method invocations and replies. Because of its bridging function a control subobject has both a standardized interface used by the replication subobject and programmer-defined interfaces used by client processes (in Globe, as in Java, a (sub)object can have multiple interfaces.) Instead of using a stub compiler to generate control subobjects, we use Java's reflection APIs [11, 12] to generate the application-defined interfaces at run time and connect them to a generic (i.e., application independent) control subobject.

3 Programmer's View

In the previous section we gave an overview of the architecture of a Globe-based application, such as the GDN. In this section we present the programmer's view on application development with Globe. In particular, we explain how a programmer develops a new class of distributed objects (the GDN's "storage object" class), and how he can create and use instances of this new class using Globe's novel *deputy* concept.

3.1 Developing a new object class

The current Globe implementation is based on Java, and developing a new object class is syntactically similar to writing a new class of remotely accessible Java objects via Java's Remote Method Invocation (RMI) [13]. Developing a new class of Globe objects therefore consists of two steps:

1. Defining the interfaces of the new object class.

2. Writing an implementation of these interfaces.

A Globe interface definition should thus adhere to the rules for RMI object interfaces: the interface inherits from `java.rmi.Remote`, each method throws a `java.rmi.RemoteException`, and all arguments are of serializable classes [14]. Additionally, for Globe, the names of all methods in the interface that modify the state of the object should end in `_w`. This information about the read/write behavior of methods is provided as input to the Globe replication protocols which use it to determine their course of action.

The definition of the GDN storage object is shown in Figure 3. To add a file, a client first calls `startFileAddition_w` passing the intended name for the file and its size. Next, the client makes repeated calls to `putFileContent_w` to place the contents of the file in the storage object. Finally, it calls `endFileAddition_w` which makes the file accessible to other clients, and assigns it a unique 64-bit number, its *incarnation ID*. During the upload the file is not accessible to other clients, although it is visible. In particular, a client trying to upload a file with the same name will receive a "name already in use" exception. The storage object interface supports files whose origin is traceable via digital signatures, hence the `traceCert` and `traceSig` parameters. For a detailed explanation of file traceability as a measure for preventing illegal distribution of copyrighted works via the Globe Distribution Network, see [15].

To download a file from a storage object, a client first calls `getIncarnationID` to determine the file's incarnation ID. Incarnation IDs are necessary to safely support interrupted downloads: when a download of a large file is interrupted, either intentionally (user pause) or unintentionally (client crash), the storage object interface allows this download to resume where it left off. To protect a client against a replacement of the file that may have occurred in the mean time, the client must check that the incarnation ID of the file is the same as at the start of the download. The actual contents of the file are downloaded using repeated invocations of the `getFileContent` method. The auxiliary method `getFileSize`, `allFiles` and `deleteFile_w` can be used to retrieve the size of a stored file, obtain the list of files stored in the storage object and to delete a file, respectively.

3.1.1 Discussion

Although defining a Globe object's interfaces in Java syntax is a trivial exercise, *designing* these interfaces most certainly is not. We briefly discuss the different factors that must be taken into account when designing a new interface. A more detailed explanation is found in [3].

The first issue an interface designer must address is what real-world entity the object must represent, or more simplistically, what data should be stored in the object. In case of the Globe Distribution Network, the designer has four apparent options:

1. A storage object contains all files pertaining to a particular software package (e.g. all revisions of the Linux kernel).

2. A storage object contains all files pertaining to a single revision of the software package (i.e., a storage object contains only the files of version 1.0, but not 2.0)

```
public interface Storage extends java.rmi.Remote
{
      public void startFileAddition_w( String name, long size,
                                       byte[] traceCert ) throws ...;
      public void putFileContent_w( String name, long offset,
                                    byte[] data )  throws ...;
      public void endFileAddition_w( String name, byte[] traceSig )
      public long getIncarnationID( String name) throws ...;
      public byte[] getFileContent( String name, long inc, long offset,
                                    long maxBlockSize ) throws ...;
      public getFileTraceInfoResults getFileTraceInfo( String name,
                                                       long inc ) throws ...;
      public long getFileSize( String name, long inc ) throws ...;
      public String[] allFiles() throws ...;
      public void deleteFile_w( String name ) throws ...;
}
```

Figure 3: Interface of a storage object. For legibility we have left out the declaration of the possible exceptions that may be thrown, which is the same for all methods: the required `java.rmi.RemoteException` and a `StorageException` which covers all GDN-specific exceptions. The `getFileTraceInfoResults` is a serializable class that encapsulates a DER-encoded public key certificate and digital signature.

3. A storage object contains a single *variant* [5] of a particular revision of the software package. Examples of variants are: the source code, binaries for Linux on the x86 Intel platform, or binaries for MacOS X 1.0.2 on PowerPCs.

4. A storage object contains a single file belonging to a particular software package. For example, the tar-ball of the source code, or an RPM for RedHat Linux on Intel.

These four options represent four different granularities for the storage object (from coarse grained to fine grained). A fine granularity, such as Option 4, is desirable because it means we can replicate only the popular file without having to copy along other files that may not be so popular. However, as there is a certain overhead associated with a Globe object, having many small objects is too expensive. Hence, we used Option 2, revision-sized storage objects in our implementation of the GDN.

It is obvious that determining what an object should represent is an important step, as the choice affects the object's interface. In the GDN, for example, choosing Option 4 would result in an interface that reflects that only a single file is stored in the storage object, as opposed to the multiple-file interface used now.

The second factor to be taken into account when designing an interface is scalability of the interface. A scalable interface has two properties: (1) it generates few updates to the state of the object and (2) it does not require the object to maintain (session) information about its clients (i.e., the object is stateless in the sense of a stateless file server). To understand the notion of scalable interfaces, consider the alternative interface shown in Figure 4.

The `read` method in this interface lacks both scalability properties. As this method lacks a parameter for passing the offset from which to read, the object implementing this interface is forced to maintain the offset internally. Adding an `lseek` method to this interface does not solve this prob-

```
public interface UNIXStorage extends java.rmi.Remote
{
      public int open( String pathname, int flags, mode_t mode );
      public int read( int fd, byte[] buf, int count );
      public int write( int fd, byte[] buf, int count );
      public int close( int fd );
      ...
}
```

Figure 4: Alternative interface of a storage object, based on the familiar UNIX I/O interface. For simplicity we left out the declaration of the possible exceptions that may be thrown. We also did not mark the methods as read or write.

lem, as the object still has to remember the set offset between calls. The implication of choosing this method signature is therefore that the object has to update its state on each `read` operation (in particular to update the offset associated with the specified file descriptor). This, in turn, implies that the replication protocol has to synchronize all replicas after each call to `read` (assuming strong replica consistency), turning what should be an inexpensive operation (reading) into an expensive one. This interface also requires the object to maintain per client state (about open file descriptors), which can have three negative effects:

1. It may prevent the object from scaling to large numbers of clients (e.g. millions).
2. This per-client information can be maintained only in the object's state, and thus violates the first property of scalable interfaces (i.e., generate few state updates).
3. If an object has many clients that fail permanently or misbehave, a garbage collection mechanism is needed to purge their stale entries, complicating the object's implementation.

3.2 Writing an object implementation

Given a properly designed interface the next step is to write an implementation of the interface in the shape of a semantics subobject. Semantics subobjects are written as regular Java classes but with a number of Globe-specific rules [16]:

- A semantics subobject must be a finite state machine, that is, the current state plus the arguments of a method invocation completely determine the next state of the object, and the state does not change between method invocations. This requirement is imposed to enable *active replication* [17] in Globe objects. As a result, one cannot generally call other Globe objects from inside a semantics subobject.
- Long-term blocking (i.e., waiting on a condition variable) is not allowed, to limit the number of blocked threads and reduce resource usage [18]. *Continuations* should be used instead [19].
- The semantics subobject should be able to marshall and unmarshall its state. For historic reasons, a programmer cannot, at present, fully exploit Java's serialization facilities, but has to implement a Globe-specific `SemanticsSubobject` interface. This interface consists of two methods, one for marshalling and one for unmarshalling the state of object in blocks.

- The implementation must obviously obey the specified write behavior, that is, a method that is not marked as modifying the state (i.e., its name does not end in `_w` (see above) should not modify the semantics subobject's state.
- The implementation has to take into account and can exploit the fact that all accesses to the semantics subobject are serialized by the middleware (i.e., there are no concurrent accesses to the object).
- A semantics subobject, if so desired, can swap state out to persistent storage, but only using the I/O interface offered by the Globe runtime system to achieve operating-system independence and enable garbage collection of persistent data.

3.3 Using an object class

In Globe, a programmer is not finished when the semantics subobject is written. To turn the new object class into something that can be instantiated in a distributed system he also has to configure the nonfunctional parts of the object's implementation. In other words, the programmer has to select the protocols and policies for replicating the state of the object and securing the object. The result of this selection process is what we call a *working object class*.

The concept of a working object class is where Globe shows its flexibility with respect to non-functional aspects. Because different objects may require different non-functional implementations, there may be several working object classes based on a single object class. To define a working object class, programmers can also write new application-specific subobjects (e.g. a new replication protocol), if a suitable one is not available from the Globe library. It is important to note that developing a new object class and using it to create a new working object class are independent activities. In other words, one programmer can reuse another programmer's object class to define a new working object class.

In this section we show how a programmer can use an object class, that is, create and use instances of the class, which includes defining a working object class. We do so by explaining how a programmer writes client programs for the distributed application. In general the programmer does not have to concern himself with any server programs as the Globe object server provides a generic platform for hosting object replicas.

A novel feature of the Globe middleware that facilitates client development is the concept of a *deputy*. A deputy is an object reference (in the CORBA sense) that can be used not only to invoke methods on an instance of a Globe object, but also to create working object classes and create and manage object instances (e.g., add or remove replicas). A deputy is an application of the *facade pattern* [20] which simplifies the access to the Globe middleware. Each object class requires its own deputy (a Java class) which is automatically generated from the object implementation using Java's reflection interfaces [11].

To best illustrate this functionality we will show how a deputy is used in the context of the Globe Distribution Network to create and access a storage object. The code to create a new storage object is shown in Figure 5. Step 1 is to create a `StorageDeputy`, an instance of the deputy class automatically derived from the storage object's implementation. In Step 2, the programmer defines an ad-hoc working object class. In this case, he specifies only which replication protocol the object should use. The protocol chosen is a master/slave protocol that is distributed with the Globe middleware (identified by the constant `REPL_MASTER_SLAVE_AUTO`). This particular replication protocol does *automatic replication*; that is, the protocol decides when and where to create or delete replicas autonomously and automatically, based on access patterns. The protocol therefore needs to be initialized with a policy

```
// Step 1
StorageDeputy s = new StorageDeputy();

// Step 2
/*
 * The object should use a master/slave replication protocol that
 * automatically creates and deletes replicas based on demand, and
 * this behavior should be governed by the default replication policy.
 */
ARSimpleReplicationPolicy policy =
                          new ARSimpleReplicationPolicy( false );
String policyString = policy.getEncodedFullReplicationPolicy();
s.setReplicationProtocol( s.REPL_MASTER_SLAVE_AUTO, policyString );

// Step 3
ObjectServerAddress osa =
                    new ObjectServerAddress( "gos1.vendor.com", 23000 );

// Step 4
s.create( "/com/vendor/gdn/somepkg-v1-0", osa );
```

Figure 5: Creating a storage object using a deputy.

that specifies the rules governing when it should add or delete replicas. In this case, it is made to follow the default policy as defined by the `ARSimpleReplicationPolicy` class. In the future, the programmer will also be able to specify in this step which security and fault tolerance mechanisms and policies the object should use.

Configuring the nonfunctional parts of the object's implementation is in fact done entirely in terms of the subobjects to use in the object's local representatives and initialization strings for those subobjects. In particular, the `setReplicationProtocol` is just a utility method that makes it easier to select a replication subobject from the Globe library. To more easily create Globe objects of a particular working object class, a programmer can create a subclass of the object's deputy class that sets aspects such as replication subobject and policy in the subclass' constructor.

Step 3 in creating an instance of a storage object is to select the Globe object server that will be asked to host the first replica of the new storage object. Creating the first replica of a storage object is equivalent to creating the object. In Step 4, the programmer tells the deputy to actually create the storage object and to register it in the Globe naming service under the name /com/vendor/gdn/somepkg-v1-0 (the GNS was discussed above).

The storage object is now ready to be used. The code for uploading a file into the object is shown in Figure 6. The invocation of `bind` causes the deputy to bind to the storage object and install a local representative in the local address space. The deputy has the same interfaces as the storage object it provides access to, so GDN the client can directly invoke the object's upload methods on the deputy, which forwards them to the local representative. Invoking those methods without binding first will throw a "not bound" exception. A GDN download client has similar code. The client will typically invoke the deputy's `bind` method with the storage object's human-friendly object name as parameter, after which it can download the stored file using the methods described above.

```
s.bind();
s.startFileAddition_w( "somepkg-1.0.i386.rpm", fsize, null );
while( totalwritten < fsize )
{
  block = ... // = read block from file
  s.putFileContent_w( "somepkg-1.0.i386.rpm", totalwritten, block );
  totalwritten += block.length;
}
s.endFileAddition_w( "somepkg-1.0.i386.rpm", null );
s.unbind();
```

Figure 6: The code a client uses to upload a file into a storage object using a deputy.

In the above example, the storage object uses an automatic replication protocol. If we chose manual control over replication the programmer can control replica creation and deletion himself using the deputy's `addReplica` and `deleteReplica` methods. Deputies used to create objects contain some management information, such as the address of the object server that hosts the first replica, and the object's name and object handle. To preserve this information between program runs, (groups of) deputies can be saved and loaded from disk.

The difference between a deputy and a general proxy is that a deputy also allows the user to manage the distributed object it represents. It makes all middleware functions applicable to an object such as naming, replica management and access control accessible via a single interface and reference. It also hides the details of the run-time system and service interfaces from the user, resulting in a simpler management interface and allowing the RTS and service interfaces to change without always affecting client code.

4 Globe internal organization

Recall that a Globe application, such as the Globe Distribution Network, runs on a number of so-called Globe sites. Internally, each Globe site consists of a number of servers that jointly comprise the implementation of Globe for that site. We distinguish five different types of servers:

- an object server for hosting the local representatives of distributed (storage) objects
- a name server to implement the Globe naming service
- a location server for implementing the location service
- a directory server to provide sitewide information, and finally
- a Globe HTTP server to allow non-Globe clients to access Globe objects

The organization of these servers and how they relate to the Globe middleware services are discussed next.

4.1 Object server

A crucial server is the *Globe object server* (GOS). An object server is responsible for hosting replicas of Globe objects. As a application-independent hosting platform it provides several facilities to these

```
public interface GOSPersistentSemanticsSubobject
{
    public void setPerstID( long pid )
        throws GOSPersistentSubobjectException;

    public byte[] getIncoreState()
        throws GOSPersistentSubobjectException;

    public void setIncoreState( byte[] state, boolean recoverMode )
        throws GOSPersistentSubobjectException;

    public void prepareDestruction()
        throws GOSPersistentSubobjectException;
}
```

Figure 7: Interface to be implemented by a semantics subobject to become persistent.

replicas, which we discuss in turn. Whether or not replicas make use of these facilities depends on their working object class, that is, programmers can decide which of the object server's facilities a replica will use.

4.1.1 Persistent-object support

The Globe object server offers persistent-object support. A persistent object in Globe is defined as an object that can continue to exist even it is not currently hosted by an object server. In contrast, a transient (distributed) object can exist only if there is at least one server that is hosting the object. To implement a persistent object, an object server needs to ensure that the state of the replica local representative it is hosting is written to persistent storage when the server is shut down. Likewise, that state should be able to be read from storage by another server at startup time. The GDN's storage objects are implemented as persistent objects.

The programming interface for making objects persistent is simple. To use the persistence facilities, the programmer of the semantics subobject just has to implement the `GOSPersistentSemanticsSubobject` interface and instruct the creating deputy to create a persistent object. The interface has four methods shown in Figure 7.

When a local representative is first created, the persistence manager of the object server calls `setPerstID`, which assigns a *persistence ID* to a local representative. This method is actually needed by the object server's fault-tolerance facilities, which are integrated with persistence support. We discuss this method below.

When an object server is asked to do a graceful shutdown, it informs the local representatives that belong to persistent objects of the pending event. Upon this notification, a local representative starts marshalling its state, which includes the states of its subobjects. To obtain the (marshalled) state of the semantics subobject, the local representative calls `getIncoreState` from the `GOSPersistentSemanticsSubobject` interface. This method should return those parts of the state of the semantics subobject that are not already stable on disk. Recall that a semantics subobject can store some or all of its state on disk (to reduce its memory footprint) via a special I/O interface offered by the Globe runtime system. After gathering the in-core state of all its subobjects, the local representative returns the combined state to the object server, which writes it to disk, and destroys the local representative. When all persistent local representatives are thus properly passivated, the object server writes its own administration to disk and quits.

When the object server restarts it looks for local representatives to reactivate. To reactivate a local representative (LR) it creates a new instance based on information about the working type of the local representative in the server's administration, and provides the instance with the marshalled state of its predecessor. The local representative unmarshalls the state and reactivates its subobjects in turn. To reactivate the semantics subobject the LR calls its `setIncoreState` method, passing it the marshalled state as returned by `getIncoreState`. `setIncoreState` has an argument that indicates whether the server is recovering from a controlled reboot or from crash. In the latter case, the semantics subobject may want to check that the parts of its state it kept on disk are still intact. This argument is again a result of the integration of the persistence and fault tolerance facilities. The local representative may also contacts its peers to see if there have been updates to the object during the period the object server was down, if so required by its consistency model.

When a persistent local representative is explicitly removed from the object server, the LR calls `prepareDestruction` on the semantics subobject. The semantics subobject should now delete the parts of its state that it kept on disk. Note that passivation of a local representative is different from removing an LR. Although in both cases the current LR instance is destroyed, in the passivation case, the LR will be recreated when the object server starts again. For this reason, it is important that the semantics subobject never deletes any of its persistent state in its `finalize` method.

To facilitate persistence, our object servers allow contact points (i.e., the TCP ports where a replica can be contacted) to become persistent as well. Persistent contact points remain valid while (the local representative of) an object is not running; when the object is running again, its contact points can be used as before. The main advantage of persistent contact points is that the contact addresses that were registered in the Globe location service do not have to be updated after a server reboot or (pause) crash. These *persistent contact points* are part of the object server's multiplexed communication support which we discuss below.

4.1.2 Fault tolerance

Globe object servers are made fault tolerant by enabling them to quickly recover after a crash with most of their state intact. To this end, object servers currently support a simple checkpointing mechanism. Periodically, the object server creates a checkpoint by halting the processing of incoming requests, waiting until current requests have been processed, and then saving the in-core states of the local representatives and the server's administration to disk. Once this data is stable on disk, the previous checkpoint is deleted in an atomic disk operation.

After a crash, the new object server reads the last complete checkpoint back from disk, recreates the replicas, and passes them their marshalled state. Each replica then reinitializes itself and synchronizes with its peers in an application- or even object-specific manner. In the Globe Distribution Network a storage object replica will recover from a server crash by contacting the other local representative to see if its state is still current, and thus ensure strong consistency. If this is the case, the storage object checks the integrity of the free software it stored on disk using any available digital signatures on the files it stores (files in the GDN are generally digitally signed by their uploaders to enable content traceability, see [15]).

We have yet to explain the `setPerstID` method from the `GOSPersistentSemanticsSubobject` interface that assigns a persistence ID to a semantics subobject. This ID is necessary to allow garbage collection of persistent data. When a semantics subobject writes data to persistent storage via the runtime's special I/O interface it must pass its persistence ID as one of the parameters, such that all data on persistent storage becomes traceable to the local representatives that allocated it. If after a crash of the object server, a local representative cannot be recovered, the server can trace the persistent data that belonged to this LR and remove it.

Because we save only the in-core parts of the local representative's state in a checkpoint, our approach is also known as *user-directed checkpointing* [21]. We decided to implement just periodic, user-directed checkpointing for performance reasons. The alternative is to checkpoint the state at each update operation, but this was felt too expensive for a general solution as it requires a synchronous disk operation. Further research is needed to see whether we can improve our fault-tolerance mechanisms. In particular, we intend to explore object-specific fault tolerance via a choice of local and global mechanisms (i.e., allow an object to choose from a set of object server facilities and various forms of replication). Some initial work on this matter is described in [22].

4.1.3 Multiplexed communication support

Until the release of the Java Development Kit version 1.4, Java only supported nonblocking TCP sockets which imposed a limit on the number of local representatives our object server was able to support. In practice, our server could support a number of local representatives that was roughly equal to the number of threads per process afforded by the underlying operating system (about 1000 for Linux). To alleviate this problem we implemented a communication layer that multiplexes multiple connections to the same destination over a single TCP connection. This multiplexing layer sets up just a few TCP connections to another object server, and thus radically reduces the number of actual TCP connections (and therefore the number of threads used) when two object servers have many connections between their local representatives. At present it just uses a single TCP connection. The multiplexing layer is a general, required facility rather than an optional facility, such as persistence or fault-tolerance, to ensure an object server can scale to (ten)thousands of hosted replicas.

4.2 Naming service

As mentioned above, each Globe object can have a human-friendly name that is resolved to an object handle. To this end, the Globe name space consists of a strict hierarchical name space that is represented as an edge-labeled rooted tree with each node representing either a directory or an object. This name space is implemented using DNS technology. For example, an object name such as /com/vendor/gdn/somepkg-v1-0 is internally rewritten to the DNS name somepkg-v1-0.gdn.vendor.com. and passed to a local DNS server to be resolved.

A node in the Globe name space is implemented as a DNS TXT record. In this way, we can use existing implementations of DNS to implement our name space. It is important to note that we do not require DNS to be replaced. Instead, we are making use of the existing DNS servers and organization, and expand the current DNS name space by adding leaf domains.

A directory is represented by a DNS domain name and has a special TXT record associated to it containing the keyword GLOBEDIR and the zone name in which that directory is supposedly contained. For example, the DNS entry

```
$ORIGIN  vendor.com.
gdn       IN TXT  "GLOBEDIR ns.vendor.com."
```

identifies the directory with Globe name /com/vendor/gdn/, which is contained in the DNS that is maintained by the DNS server at ns.vendor.com. Directory entries can contain subdirectories or object handles, each of which are again implemented as TXT records. For example, the following specifications identify two entries /org/globeworld/gdn/somepkg-v1-0 and /org/globeworld/gdn/somepkg-v2-0 storing object handles, as well as an entry /com/vendor/gdn/beta indicating a subdirectory for beta releases:

```
$ORIGIN  gdn.vendor.com.
somepkg-v1-0  IN TXT  "GLOBEOBJHANDLE ACARZJgItYlPUkSIB590cUN42gfon4"
somepkg-v2-0  IN TXT  "GLOBEOBJHANDLE AAAAIAACX5FN84nRS+JIImw9SqceKs"
beta          IN TXT  "GLOBEDIR ns.vendor.com."
```

A Globe name server consists of two processes normally colocated on the same machine. One process is a DNS name server running BIND [23], which is actually responsible for storing and maintaining the name space. The other process is a naming authority responsible for securely updating the local DNS database on behalf of clients. Updates are carried out by BIND, but it accepts update requests only from its naming authority.

Organizing the Globe name server as a colocated pair of processes is primarily done for simplicity. First, letting BIND accept updates from only one process avoids expensive locking schemes to handle nonatomic update operations. For example, removing a directory requires a read operation to first check whether the directory is empty, and a subsequent write operation to actually remove it. Second, if a naming authority is responsible for only a single domain, then updates for only that domain are affected if the naming authority is disrupted. Finally, colocating the naming authority and the DNS name server has the advantage that the former can be identified using a normal DNS lookup request. The IP address returned by the DNS name server is the same one as its associated naming authority.

4.3 Location service

The Globe Location Service (GLS) is designed to provide a worldwide scalable solution to locating mobile and replicated objects. For each object, GLS maintains a mapping between an object's object handle and the contact addresses where the object's replicas can be found. Only those replicas that can be used for binding a client to the object need to be registered with GLS.

The location service is organized as a worldwide distributed search tree, based on a hierarchical partitioning of the underlying network into domains. The top-level domain covers the entire network, whereas the lowest-level domains typically correspond to a moderately-sized network such as a university campus or the office network of a corporation's branch in a certain city. In the current setup, we have a relatively small five-level tree, as shown in Figure 8.

In this setup, each node is represented by a separate logical server. Each server maintains location information on the storage objects that reside in its associated domain. This location information is either a contact address or a pointer to a lower-level domain where the object resides. Location information is stored in a *contact record*. For example, consider a object that has been replicated to the VU1 and the Erlangen domain. The server at Erlangen will store the object's contact address, whereas the server for Germany will store a pointer to the Erlangen server, and so on, leading to the situation shown in Figure 9.

To look up a contact address, a client contacts its nearest leaf node. If that node does not contain any location information on the requested storage object, the request is forwarded upwards until a node is reached that does. In the worst case, a lookup request travels to the root. The root and intermediate nodes store only forwarding pointers to child nodes. Therefore, the lookup request subsequently travels a path of forwarding pointers down to one of the leaf nodes.

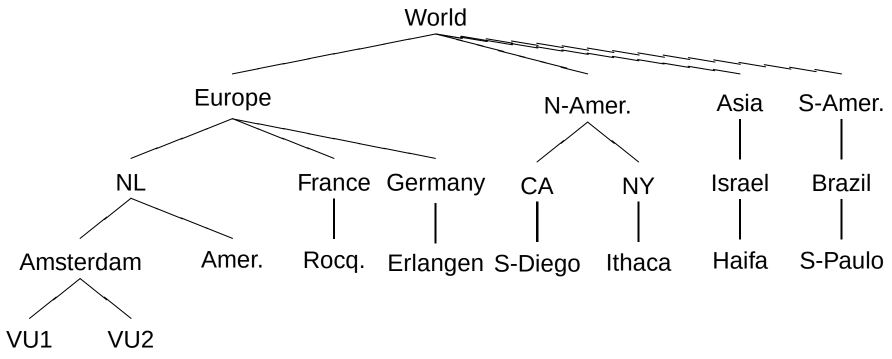

Figure 8: The current organization of the Globe location service.

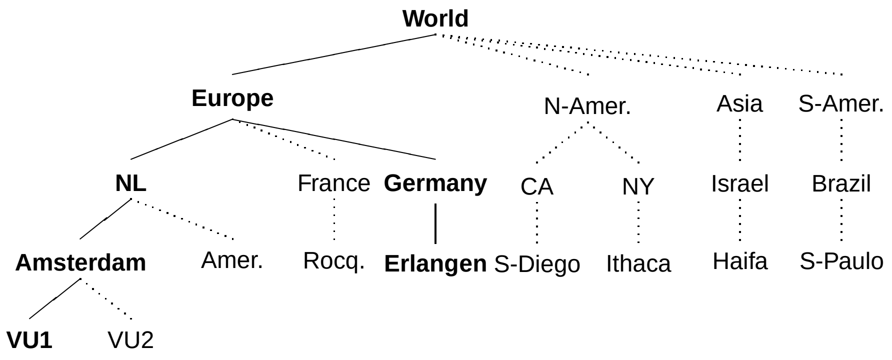

Figure 9: Nodes containing a forwarding pointer or address for a storage object replicated at VU1 and Erlangen.

Returning to our example tree of Figure 9, suppose a client in Rocquencourt (France) issues a lookup request. That request is first processed at node Rocquencourt, from where it is forwarded first to France and then to Europe. The latter stores two forwarding pointers: one to node NL and one to Germany. In such cases, an arbitrary choice is made in the current setup. Assuming that the request is forwarded to Germany, it eventually reaches node Erlangen where a contact address is found, which is then returned to the client.

An important observation is that the lookup request travels only between servers in the smallest domain in which both the client and the requested object reside. In a similar fashion, we exploit locality for update operations. Details on these operations, as well as various important refinements and optimizations are described in [24, 25].

Implementing GLS as a single-rooted tree obviously introduces a scalability problem for higher-level nodes. The solution to this problem is to apply partitioning techniques. In other words, we implement a logical node using multiple servers. It is beyond the scope of this paper to describe these techniques in any detail. However, it is worth noting that it is possible to distribute contact records in such a way that hosts running servers for GLS are equally loaded while maintaining the desirable locality properties of having a search tree. Details on such partitioning schemes, along with a description of simulation experiments are described in [26].

4.4 Infrastructure service

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

The *Globe Infrastructure Directory Service (GIDS*[1]*)* keeps track of all object servers that are currently available worldwide. The GIDS allows applications to discover suitable object servers based on a specification of desired properties, related to technical capabilities (amount of memory, available bandwidth, operating system and hardware platform), security attributes (which person or organization operates this object server), and the location of the object server. Once a matching server has been found the application and server enter a negotiation phase to determine whether or not they want and can cooperate, and negotiate the exact details of that cooperation.

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

The current implementation of the GIDS uses the Light-weight Directory Access Protocol (LDAP) and standard LDAP servers [27]. The GIDS divides the world into a set of base regions (generally subdivisions of the leaf domains identified for the Globe location service). Per base region there is an LDAP server, called the *Regional Service Directory (RSD)* that keeps track of the available object servers and their properties. The base regions are organized into a hierarchy, currently based on their geographical location, which allows clients (i.e., objects looking to create a new replica somewhere)

[1]GIDS is Dutch for "guide."

in other base regions to locate the appropriate Regional Service Directories.

A Regional Service Directory also contains virtually all configuration information of the Globe sites in its region. An RSD can be located using its DNS name, which corresponds to the DNS name of the site the server is part of. In this sense, GIDS is similar to Active Directory [28], an important difference being that GIDS servers constitute only leaf nodes in the name space. Intermediate nodes are always DNS servers. This difference makes GIDS more efficient. Details of GIDS are described in [9].

4.5 Services for integration in the Web

Many users on the Internet prefer to use their existing client-side software to access services such as offered by the Globe Distribution Network. We did not find it reasonable to force users to install Globe-enabled clients in order to be able to access GDN. Instead, we have chosen to provide a means that allows the use of standard browsers to download the files contained in storage objects. However, at this moment, uploading files requires using one of our GDN clients. In the following, we briefly describe how GDN and other Globe applications are integrated into the Web.

4.5.1 Naming issues

A minimal requirement for integration in the Web is that we adopt a naming convention that is accepted by Web browsers. In practice, this means that a user should be able to use a browser-recognizable Uniform Resource Identifier (URI). Unfortunately, not all browsers are capable of recognizing the same URIs. In some cases, such as Netscape, the types of URIs that are supported is fixed, Other browsers, like Mozilla and Internet Explorer provide more flexibility.

We have decided to take a two-step approach. First, Globe pathnames are extended to *Globe URNs* by attaching the prefix "globe:/" to each pathname, leading to names such as globe://com/vendor/gdn/somepkg-v1-0. Underlying this naming convention is that extensible browsers should be able to easily support new scheme identifiers such as "globe."

However, most browsers recognize only pre-configured scheme identifiers. We therefore take a second step by embedding Globe names into *embedded Globe URNs*, which are plain HTTP URLs. A name such as http://enter.globeworld.org is added as a prefix to a Globe name, resulting in, for example, the embedded Globe URN http://enter.globeworld.org/com/vendor/gdn/somepkg-v1-0. This URL refers to an object known under the Globe name /com/vendor/gdn/somepkg-v1-0. We allow a user to specify any prefix he likes, but every prefix requires the support from specially configured Globe HTTP servers. As we explain next, these servers are responsible for handling Globe URNs.

4.5.2 Globe HTTP server

To support downloading of files from GDN through Web browsers, we make use of a service that translates HTTP requests into read operations on the appropriate storage object and returns the results of an operation as an HTTP reply. This service, referred to as the *Globe HTTP server*, consists of two parts, each currently implemented as a separate process as shown in Figure 10.

One part is called the *Globe gateway* and acts as a Globe client that can bind to any storage object. As explained above, when a process binds to a storage object a local representative of that object is loaded into the process's address space such that it can invoke the object's methods. The Globe gateway accepts HTTP requests that contain a Globe URN referring to the requested object by its name as explained above. Note that the gateway uses HTTP only as its communication protocol; the

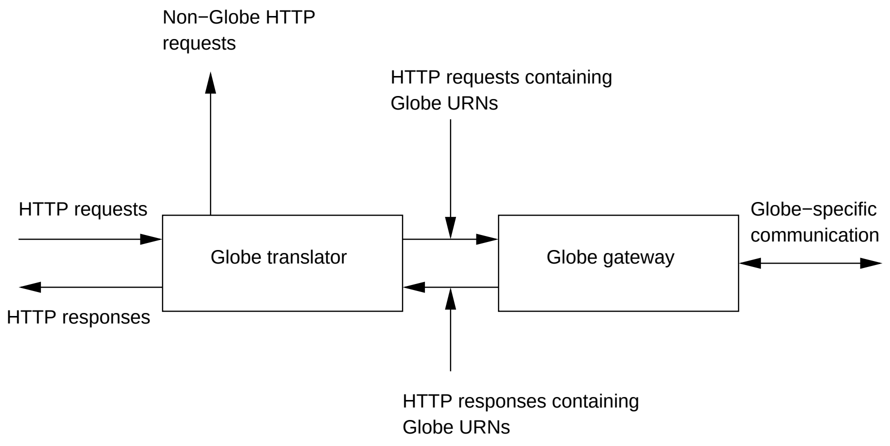
Non−Globe HTTP requests
HTTP requests containing Globe URNs
HTTP requests
Globe translator
Globe gateway
Globe−specific communication
HTTP responses
HTTP responses containing Globe URNs

Figure 10: The Globe HTTP server.

names it accepts are always expressed as Globe URNs (i.e., with URI scheme identifier globe). The gateway then binds to the named storage object, and invokes the object's methods for downloading a file from the object (see above). The Globe gateway is setup as a generic framework for bridging between HTTP and different types of Globe objects. For example, it is also used by another Globe application called *GlobeDoc*, a scalable replacement for the (hypertext) Web [29].

The other part of a Globe HTTP server is the *Globe translator*. This part acts as a regular HTTP proxy server that accepts incoming HTTP requests containing URLs having http as their scheme identifier. Each translator is configured to recognize a specific prefix as explained above. A URL containing this prefix is treated as an embedded Globe URN and is translated into a Globe URN. This URN is then passed to the Globe gateway. In case of the GDN, the translator is otherwise passive. In case of GlobeDoc (our Web replacement) the translator also accepts HTML pages that are returned by the gateway. Any Globe URNs contained in such a page are translated into embedded URNs, after which the modified page is passed on to the client.

For optimal performance, a client should use a Globe HTTP server that is nearby. A Globe HTTP Server itself always contacts the nearest replica of an object using the Globe Location Service. Finding a nearby HTTP server is, however, a nontrivial problem that applies not just to Globe but to replicated Web sites in general (cf. Akamai [30]). To solve this problem, we apply a redirection mechanism that is based on the NetGeo service from CAIDA [31]. NetGeo maintains a mapping between IP addresses and geographical location. When a Web browser accesses the GDN, it does so by sending an HTTP request to a set of servers known collectively as enter.globeworld.org (via round-robin DNS). Using the client's location as provided by NetGeo and a map of available Globe HTTP servers, these servers compute the Globe HTTP server that is geographically closest to the requesting client and return an HTTP response requesting the client to retrieve the requested content at the selected Globe HTTP server. After it has been redirected, the client continues to use that server with further redirections. We use HTTP cookies storing the address of the nearest server to speed up redirection on subsequent requests to `enter.globeworld.org`.

Table 1: Results of Experiment 1.

Server	Kilobytes per second	Downloads per second
Apache	9942.59	19.65
GDN	9486.75	15.66
Difference	-4.81%	-25.47%

5 Practical experience with Globe

We have conducted a number of experiments to test the performance of our prototype implementation of Globe and the Globe Distribution Network. In the first experiment we compare the performance of a single Globe object server to that of the Apache HTTP server [32]. In the second experiment we provide evidence that that download time in the GDN can indeed be improved by using a nearby server. We discuss each of these experiments in turn.

5.1 Experiment 1: object server performance

The first experiment is aimed at measuring the average throughput per client for a large number of clients simultaneously downloading a heterogeneous set of files. Both servers were loaded with the 50 most popular files on the SourceForge free-software site in October 2001. In the GDN case, each file was placed in a separate storage object. The size of these files ranged from 21 KB to 15 MB (average 1.5 MB). At the client side 50 clients were started and each client continuously downloaded the same file from the server. After 30 minutes all clients were killed and the total number of successful downloads was counted.

We used the following hardware and software. The client machines were dual-processor Pentium III at 1 Gigahertz with 1 Gigabyte of memory, running RedHat 7.2. The server machine was a dual-processor Pentium III at 933 MHz with 2 Gigabyte of memory and 10,000 RPM Ultra-160 SCSI disks, running RedHat 7.1. All machines ran custom configured kernels, and were connected via a full-duplex 100 Mb/s Ethernet. For the tests with Apache, we used Apache version 1.3.19, and the `wget` HTTP client version 1.6. The Apache server was configured following advice from RedHat [33]. For the tests with GDN, we used version 1.0 of the GDN implementation[2] which is written in Java. To execute the Java code we used the IBM Developer Kit for Linux, Java 2 Technology Edition, version 1.3-2001-06-26, which includes a high-performance just-in-time (JIT) compiler [34].

For the measurements with 1–50 concurrent clients each node ran a single GDN or HTTP client. For the measurements with 60–100 concurrent clients, each node ran 2 client programs, one on each CPU. In this experiment, we focused on network throughput and, as a result, the download time measured at the client (from which we calculate the throughput) does not include the time used accessing the Globe naming service or Globe location service. The GDN client connects directly to the replica based on a stored contact address. Both the HTTP client and the GDN client discard the downloaded data by writing it to `/dev/null`, so there is no disk I/O at the client side. GDN clients download the file in blocks of 1 Megabyte.

The results of Experiment 1 are shown in Table 1. The experiment was repeated three times, and the numbers shown are the average of the three runs. In terms of the number of files served, Apache

[2]Globe is available from `http://www.cs.vu.nl/globe`.

outperforms the GDN by more than 25%. In terms of number of bytes served, however, Apache is just 4.81% better than the GDN object server. The higher number of files for Apache was due to the fact that the GDN created a new proxy of the object for each download and is generally slower at downloading small files due to overhead, as further study revealed.

5.2 Experiment 2: download time and proximity

Two important properties of the Globe Distribution Network are that:

1. it allows clients to download a copy of the software from a nearby server
2. it *binds*, that is, locates and connects to this server using *proportional communication*. Proportional communication means that the client does not send any wide-area binding messages if the replica is on the local network.

The second property is mainly due to the Globe location service which has lookup costs proportional to the distance between lookup requester and nearest replica (see above).

These properties can improve both download time and scalability as they avoid communicating over shared wide-area networks where bandwidth is assumed to be scarce and latency high due to geographical distance and possibly congestion. To evidence that download time in the GDN can indeed be improved by using a nearby server and that the time for locating a replica is smaller when the replica is closer by, we ran the following experiment.

We used a machine located in San Diego, CA to download a one Megabyte file from two locations: Ithaca, NY (approximately. 4000 kilometers away) and Amsterdam, The Netherlands (approximately. 9000 kilometers away), and measured the download times. The download time is subdivided into three components. The first component, *binding time*, entails the time spent in the first part of the download where the client establishes a connection to the replica, by contacting the Globe middleware services and creating a proxy for the object in its local address space. The second component, *transmission time*, is the total time spent on the network during the download, and the third component, *processing time* is the time used by the client's proxy and the replica to process the client's download request.

The two download times and their components are shown in Figure 11. The numbers used are the average of 99 downloads from both locations. In this experiment, the download to San Diego from the closer server in Ithaca is indeed faster than the download from Amsterdam (by 6.7 seconds). The histograms also shows that in this experiment binding time is decreased considerably when binding to a replica which is nearer. The percentage of total download time spent binding only slightly increases (from 8 to 10 % of the total download time) when the replica in Ithaca is used. This percentage would have been considerably higher if binding time did not decrease with client–replica distance. A more detailed breakdown of the download times and a comparison to downloads via HTTP can be found in [3].

5.3 Showcase

The Globe Distribution Network as it stands today is the result of combining scientific efforts and development work. Development has largely taken place in the form of an externally supported project called SIRS ("Scalable Internet Resource Service"). One of the original goals of SIRS/GDN was to come to a solution for offloading popular FTP sites. The research and development efforts put into GDN amount to approximately 12 person-years of work, excluding the work put into the Globe location service (which is at least another eight person-years).

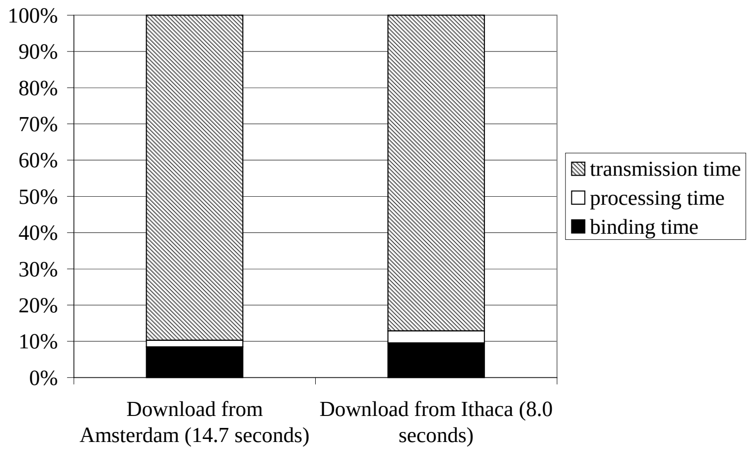

Figure 11: Histogram of the download times from Amsterdam and Ithaca, broken down in three components.

The GDN is now running as a permanent Globe show case on multiple sites across the Internet. There are various "local" sites hosted in The Netherlands; the main international sites are (in alphabetical order):

- Amerongen, The Netherlands (NLnet Foundation)
- Amsterdam, The Netherlands (Vrije Universiteit)
- Ithaca, New York (Cornell University)
- Haifa, Israel (Technion)
- Rocquencourt, France (INRIA)
- San Diego, California, USA (CAIDA)
- Sao Paulo, Brazil (University of Sao Paulo)

GDN is hosting software for Linux (2.5 kernels and the RedHat 7.2 updates), Amoeba, and Minix. We are also hosting various Web sites. The entry point for the showcase is `http://enter.globeworld.org/nl/vu/cs/globe/proj/clubglobe`.

6 Summary and some lessons learned

The Globe middleware platform for the development of large-scale Internet applications is designed to provide the flexibility that is required to meet the diverse nonfunctional requirements of such applications. We arrive at a flexible middleware by adopting a new model of distributed objects in which objects are in control of all aspects of their implementation, including nonfunctional aspects such as replication protocol and security. This model allows developers to choose the mechanisms and policies for meeting the application's nonfunctional requirements that are best suited for the application or a particular object in the application.

The Globe middleware offers a number of services to support its distributed objects. The Globe objects making up an application are hosted by a collection of generic object servers. These object servers are specifically designed to host large numbers of objects and provide facilities to the objects that allows them to persist despite server crashes or controlled reboots. The Globe naming and location services provide scalable location-independent naming for billions of Globe objects. The Globe infrastructure directory service enables the discovery of (underutilized) object servers to support Globe objects with dynamic replication protocols. The extensible Globe HTTP server finally supports access to Globe objects via legacy clients, in particular, Web browsers.

Building Globe-based application is simplified by the concept of a deputy, a facade that hides away the details of the Globe runtime system and the interaction with the various Globe services. This simplification does not, however, come at the expense of flexibility. A deputy affords a developer fine-grained control over the distributed implementation of Globe objects, if so desired. A number of experiments conducted with Globe show that it can be efficiently implemented in Java and provide evidence that Globe's ability to do flexible replication and its scalable services can indeed improve an application's performance.

6.1 Lessons learned

Research and development of Globe and GDN continues and is partly based on our experiences with our development efforts so far.

An important lesson we learned from developing and using the Globe middleware is the importance of ease-of-use. It is not sufficient for a middleware to just implement all required functionality for building applications, it should also be easy to use. Implementing applications should be easy as possible, and both the middleware and the applications developed should be easy to deploy and manage. Our experience is that Globe is easy to use for application developers, but less so for people who want to setup an instance of Globe on their site. We will discuss the usability issues for these two classes of users in turn.

We made a good choice by adopting much of the syntax of Java Remote Method Invocations. Many programmers are familiar with RMI and we were able to use the facilities offered by the Java Virtual Machine (i.e., Java Reflection) to provide almost the same ease-of-use and transparency. An important feature of the Globe middleware is, however, the ability for a developer to turn off transparency and take control of the distributed implementation of the object, for example, to configure a replication protocol and policy optimal for the object (class). Finding an interface to the system that enabled fine-grained control when desired, but that preserved transparency in other cases, was a challenge. We think we have found a good solution in our concept of a deputy to accompany each object class.

A middleware platform for large-scale Internet applications should not only make applications easy to implement, it should also make deploying and maintaining them less complex. We feel that these management issues should have received more attention in Globe. For example, setting up a collection of Globe sites and deploying a new application to those sites is still laborious, despite a number of tools that were written. The large effort required is mainly due to the amount of work in configuring an individual Globe site. A Globe site requires more than 20 external libraries or tools, most of which have to be installed by the site administrator himself, as they are not typically installed on most systems. Installing all those packages is time-consuming and tedious. Furthermore, after installing the packages the site administrator has to configure the various components of the Globe middleware, which involves setting a lot of parameters. Finally, to finish the configuration the administrator must register his site with our group, which is a manual process that takes some time.

Having a system that is difficult to install made it harder for us to attract outside users and create an infrastructure of Globe sites for testing our middleware in a real (i.e., non-simulated) wide-area environment. An important lesson is therefore that if we are to successfully evaluate systems such as Globe, it is imperative to support ease of deployment. A minimal requirement is that users can install, configure and upgrade their local system without too much hassle, and that the distributed system as a whole should be designed to cope with a regularly changing set of participants. We are currently looking at deployment and management problems in the broader context of self-managing peer-to-peer networks.

Another lesson learned is that Java is a good implementation vehicle for our middleware. Garbage collection made our implementation more robust and the exception facilities allowed for easier debugging. The availability of, in particular, a number of security libraries and a LDAP [27] library allowed us to expand the functionality of our prototype with little effort. More importantly, the performance of Globe did not suffer from the use of Java. Various experiments, including those reported in this paper, confirm that Java itself is rarely on the critical path, and instead Globe performance is largely dictated by network and disk I/O [35, 36].

The development of the current version of Globe and the Globe Distribution Network could not have been accomplished without the financial support from the NLnet Foundation. In addition, the discussions with Teus Hagen, Wytze van der Raay, and Frances Brazier concerning the deployment of GDN have significantly contributed to making GDN publicly available as soon as possible. Wytze van der Raay deserves special credit for being our first and highly constructive beta tester. We also gratefully acknowledge the contributions from Egon Amade and Ivo van der Wijk.

References

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

[2] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. Van der Wijk, M. van Steen, and A. Tanenbaum. "The Globe Distribution Network." In *Proc. 2000 USENIX Annual Technical Conference (FREENIX track)*, pp. 141–152, San Diego, CA, USA, June 2000.

[3] A. Bakker. *An Object-Based Software Distribution Network*. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands, Dec. 2002.

[4] A. Bakker, M. van Steen, and A. Tanenbaum. "From Remote Objects to Physically Distributed Objects." In *Proc. 7th IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS'99)*, pp. 47–52, Cape Town, South Africa, Dec. 1999. IEEE Computer Society.

[5] R. Conradi and B. Westfechtel. "Version Models for Software Configuration Management." *ACM Computing Surveys*, 30(2):232–282, June 1998.

[6] Object Management Group. "The Common Object Request Broker Architecture: Core Specification. Revision 3.0.2." OMG Document Technical Report formal/2002-12-02, Object Management Group, Framingham, MA, USA, Dec. 2002.

[7] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. "Wide-Area Computing: Resource Sharing on a Large Scale." *IEEE Computer*, 32(5):29–37, May 1999.

[8] G. Ballintijn, M. van Steen, and A. Tanenbaum. "Scalable Human-Friendly Resource Names." *IEEE Internet Comput.*, 5(5):20–27, Sept. 2001.

[9] I. Kuz, M. van Steen, and H. Sips. "The Globe Infrastructure Directory Service." *Computer Communications*, 25(9):835–845, June 2002. Elsevier Science, Amsterdam, The Netherlands.

[10] B. Popescu, M. van Steen, and A. Tanenbaum. "A Security Architecture for Object-Based Distributed Systems." In *Proc. 18th Annual Computer Security Applications Conference*, pp. 161–171, Las Vegas, NV, USA, Dec. 2002.

[11] Sun Microsystems, Inc., Palo Alto, CA, USA. *Java^{tm} Core Reflection: API and Specification*, Jan. 1997.

[12] Sun Microsystems, Inc. "Reflection." `http://java.sun.com/j2se/1.4/docs/guide/reflection/`. [14 April 2003].

[13] A. Wollrath, R. Riggs, and J. Waldo. "A Distributed Object Model for the Java System." In *Proc. 2nd USENIX Conference on Object-Oriented Technologies (COOTS'96)*, Toronto, Ontario, Canada, June 1996.

[14] Sun Microsystems, Inc., Palo Alto, CA, USA. *Java^{tm} Object Serialization Specification*, 1.4.4 edition, Aug. 2001.

[15] A. Bakker, M. van Steen, and A. Tanenbaum. "A Law-Abiding Peer-to-Peer Network for Free-Software Distribution." In *Proc. IEEE Symposium on Network Computing and Applications (NCA'01)*, pp. 60–67, Cambridge, MA, Oct. 2001. IEEE Computer Society.

[16] P. Homburg. *The Architecture of a Worldwide Distributed System*. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands, Mar. 2001.

[17] F. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach." *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[18] K. Langendoen, R. Bhoedjang, and H. Bal. "Models for Asynchronous Message Handling." *IEEE Concurrency*, 5(2):28–37, Apr. 1997.

[19] R. Draves, B. Bershad, R. Rashid, and R. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems." In *Proc. 13th ACM Symposium on Operating System Principles*, pp. 122–136, Asilomar, CA, 1991.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1994.

[21] J. Plank, M. Beck, G. Kingsley, and K. Li. "Libckpt: Transparent Checkpointing under Unix." In *Proc. USENIX Winter 1995 Technical Conference*, pp. 213–223, New Orleans, LA, USA, Jan. 1995.

[22] J. Ketema. "Fault-Tolerant Master-Slave Replication and Recovery in Globe." Master's thesis, Vrije Universiteit Amsterdam, Mar. 2001.

[23] P. Albitz and C. Liu. *DNS and BIND*. O'Reilly & Associates, Sebastopol, CA, 3rd edition, 1998.

[24] M. van Steen, F. Hauck, G. Ballintijn, and A. Tanenbaum. "Algorithmic Design of the Globe Wide-Area Location Service." *Comp. J.*, 41(5):297–310, 1998.

[25] A. Baggio, G. Ballintijn, M. van Steen, and A. Tanenbaum. "Efficient Tracking of Mobile Objects in Globe." *Comp. J.*, 44(5):340–353, 2001.

[26] M. van Steen and G. Ballintijn. "Achieving Scalability in Hierarchical Location Services." In *Proc. 26th International Computer Software and Applications Conference (COMPSAC'02)*, pp. 899–905, Oxford, UK, Aug. 2002.

[27] P. Loshin, (ed.). *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. Morgan Kaufmann, San Francisco, CA, USA, Apr. 2000.

[28] A. Lowe-Norris. *Windows 2000 Active Directory*. O'Reilly & Associates, Sebastopol, CA, 2000.

[29] M. van Steen, A. Tanenbaum, I. Kuz, and H. Sips. "A Scalable Middleware Solution for Advanced Wide-Area Web Services." *Distributed Systems Engineering*, 6(1):34–42, Mar. 1999.

[30] F. Leighton and D. Lewin. "Global Hosting System." United States Patent 6,108,703, Aug. 2000.

[31] CAIDA. "NetGeo - The Internet GeographicDatabase." `http://www.caida.org/tools/utilities/netgeo/`, May 2001.

[32] The Apache Software Foundation. "The Apache HTTP Server." `http://www.apache.org/`. [15 April 2003].

[33] A. Likins. “System Tuning Info for Linux Servers.” `http://people.redhat.com/alikins/system_tuning.html`. [15 April 2003].

[34] IBM. “IBM Developer Kit for Linux, Java 2 Technology Edition, version 1.3-2001-06-26.” `http://www-106.ibm.com/developerworks/java/jdk/linux130/`. [26 June 2001].

[35] I. Kuz. *An Approach to Wide-Area Scalable Web Servers*. Ph.D. thesis, Delft University of Technology, Delft, The Netherlands, 2003. (to appear).

[36] G. Ballintijn. *Locating Objects in a Wide-area System*. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2003. (to appear).