A Distributed-Object Infrastructure for Corporate Websites*

Ihor Kuz
Delft University of Technology
Delft, The Netherlands
I.T.Kuz@its.tudelft.nl

Patrick Verkaik
Vrije Universiteit
Amsterdam, The Netherlands
patrick@cs.vu.nl

Maarten van Steen
Vrije Universiteit
Amsterdam, The Netherlands
steen@cs.vu.nl

Henk J. Sips
Delft University of Technology
Delft, The Netherlands
H.J.Sips@its.tudelft.nl

Abstract

A corporate website is the virtual representation of a corporation or organization on the Internet. Corporate websites face numerous problems due to their large size and complexity, and the nonscalability of the underlying Web infrastructure. Current solutions to these problems generally rely on traditional scaling techniques such as caching and replication. These are usually too restrictive, however, taking a one-size-fits-all approach and applying the same solution to every *document. We propose Globe as a foundation upon which to build scalable corporate websites, and introduce GlobeDoc, a website model based on Globe distributed shared objects. This paper describes GlobeDoc, highlighting the design and technical details of the infrastructure.*

1. Introduction

A corporate website is the virtual representation of a corporation or organization on the Internet. It is typically a large website that contains a wide variety of information about or related to that corporation. This information can range from publicly available marketing and PR information, through announcements, news, technical and support information, to internal information with access restricted to employees only. Because of its varied character the information is usually maintained by a diverse group of people. Some of the website's contents may be provided by the marketing department, other parts of the site may be designed and maintained by individual product groups, and yet other parts of the site may be maintained by specific regional departments.

Clients of corporate websites vary just as widely. There is often no single client profile with regards to location, access times, access frequency, etc. Clients will access the site from a wide range of locations, at all times of the day, and with differing access patterns. A corporate website will often have to deal with a heavy load, though not every part of the site will be equally burdened. For example, the pages describing their products may be very popular, while a page describing the marketing department's trip to the zoo will be much less popular.

More so than other, smaller, websites, corporate websites face numerous problems due to their large size, complexity, and the nonscalability of the underlying Web infrastructure. These problems manifest themselves in the form of suboptimal access times to the website, broken links to and within the website and the presence of wrong or inconsistent information on the site. Access problems are caused either by the overloading of servers and their network connections from too many requests, or by structural problems such as the server or network being down. Problems with broken links are usually caused by internal reorganization of the site, or by unavailable mirrors, while inconsistent information is caused by improperly updated mirrors or caches.

Current solutions to these problems generally rely on traditional scaling techniques such as caching and replication and include (proxy) caching, mirroring and clustering. The basic principle behind all of these techniques is that replicating (parts of) the site on multiple servers reduces the load on any single server and possibly improves access times by moving the contents closer to the user. Often, however, the problems are only partially solved. For example, clustering solves the problem of overloaded servers, but not that of saturated network connections. At the same time new

*This work was sponsored by a grant from the NLnet Foundation. It is also part of the JERA project funded by the Dutch HPCN foundation.

problems, such as inconsistent documents, are introduced. What's more, these solutions are often ad-hoc, leading to a myriad of different, incompatible, and often unmanageable solutions. There is, for example, no standard way of creating consistent Web site mirrors and Web site administrators often have to create their own solutions, quickly leading to a situation where many different incompatible and suboptimal mirroring approaches are being used.

Many proposed solutions are also too restrictive: they generally take a one-size-fits-all approach, applying the same solution to *every* resource. For example, most caching solutions have one caching algorithm that is applied to every cached Web resource. We claim that, in order for the Web to scale, it will be necessary to apply distribution solutions to individual Web resources depending on their needs and characteristics. Thus, while replication-based content delivery network solutions such as Akamai's FreeFlow [1] and Digital Island's Footprint [7] provide complete replication services and take care of issues such as automatic redirection of requests and document consistency, we feel that their approach of assigning a global replication strategy to all documents is too coarse.

In addition, the naming scheme used in the Web aggravates many of the scalability problems because it is not location transparent. Each URL contains a Web server address, which means that when resolving the URL and retrieving the resource, only the referenced server can be contacted. Solutions utilizing clustering or mirroring of Web sites have to deal with this problem and often come up with schemes that rewrite Web pages, use dynamic DNS tables, or modify IP routing tables to allow the address in the URL to refer to more than one actual server. This problem with naming in the Web has been widely recognized by the Web community and work continues on a location-transparent naming structure (URNs) [17].

Based on these observations, we claim that a good solution to the problems encountered by corporate websites must have the following characteristics. The solution must be scalable, that is, it should offer an infrastructure that is able to handle a growing number of users, resources, and requests per resource worldwide. Resource names and references must be location transparent, and remain valid if the resource is moved or distributed over multiple locations. It must also be flexible and extendible, so that new resources and new solutions can easily be added (without having to resort to solutions outside the system). Furthermore, the solution should not degrade overall system performance, and last, but not least it must be compatible with existing WWW clients and websites.

We propose Globe as a foundation upon which to build corporate websites. Globe is a wide-area distributed system based on the concept of distributed objects that fully encapsulate their own distribution policies - including replication, migration, and partitioning. A detailed description of the Globe model can be found in [19]. We believe that Globe has the necessary characteristics to provide a good infrastructure for very large websites.

By providing a framework that lets scaling techniques be applied on a per-object basis, Globe allows scalable components and applications to be created. Also, because Globe allows distribution strategies to be tailored per object it is possible to provide optimal solutions by applying strategies based on the object's (expected) usage and characteristics.

Flexibility and extendibility are provided by Globe's interface-based object design and modular object structure. An interface-based design means that Globe-object clients call methods through interfaces that are independent of actual method implementations. Method implementations can, therefore, change (or be replaced) without modification of clients that use them. Internally, Globe objects are built up modularly out of subobjects. This means that specific object parts can be replaced without affecting any of the other parts. It is therefore possible for an object's distribution strategy, for example, to be replaced without having to go through the trouble of reimplementing the whole object.

Globe also has a scalable naming service that provides location transparency. In Globe, object names are separate from, and independent of, their location: an object may change its location, or even be replicated, yet keep the same name. This transparency is achieved by splitting the naming and locating of objects into two separate services. A name service is used to resolve symbolic user-defined names to fully location-independent and globally unique persistent object identifiers called object handles. Object handles are, in turn, resolved by a location service to object contact addresses that describe where and how an object can be contacted. The name and location services will be described in more detail later.

The goal of this paper is to describe a Globe-based infrastructure for corporate websites called GlobeDoc. (Note, however, that we do not propose to replace the whole Web with GlobeDoc. We present GlobeDoc as a solution that can coexist with and be integrated into the existing Web.) We will focus on the design and technical details of the infrastructure rather than on motivation of our (design) choices as these are already covered elsewhere. Contributions made by this paper include solutions to how large websites (and other distributed applications) can be organized and built using distributed objects in a way that solves many of the current problems. Recognizing that the Web's strength is that everything can be accessed through standard browsers, we also show how Globe-based websites can be fully integrated into the current Web structure.

The rest of the paper is structured as follows: Section 2 will present the model and system architecture of the

GlobeDoc corporate website infrastructure followed by a detailed description of all the system components in Section 3. Section 4 will delve deeper into the design of the corporate website, describing the objects used and issues that must be dealt with. Section 5 will examine related work and Section 6 will conclude with a summary of the project status and directions for future work.

2. The GlobeDoc corporate website model

2.1. Assumptions and definitions

The following assumptions about (corporate) websites and their environment are made. A corporate website is accessible from the Internet or from an internal intranet and access to the site will be through regular Web browsers. As the users of corporate websites may reside anywhere in the world, the site will be accessed from a variety of geographic regions. The majority of such a website's contents will be based on static data and contain regular Web content (e.g. static HTML pages, images, etc.); only a small percentage will be dynamically generated or contain streaming content such as information about the company's stocks, or a speech by the company's president. The website will be heavily used (by either internal or external users), however this usage will not be evenly balanced (i.e., some documents will be very popular while others will rarely be accessed).

To facilitate further discussion of corporate websites and distributed-object based websites, we present definitions of some key concepts. We define a **website** as a collection of related Web documents and applications. For example, the website of a corporation contains a collection of documents that are in some way related to the corporation. Note that a website, in our view, may be physically distributed accross multiple locations. A **Web document** is defined as a collection of related Web resources. A **Web resource** is simply anything that can currently be accessed over the Web, such as, HTML pages, images, video clips, audio clips, applets, etc. The relation between the resources contained in a Web document is stronger than that between the documents contained in a website. For example, a Web document may contain the HTML pages that make up a news story plus the icons and other multimedia elements that are referenced in the HTML pages. Note that Web documents are static, that is, they do not contain dynamic content such as dynamically generated, or interactive pages. **Web applications** are used to provide such dynamic content. In this paper we concentrate only on (static) Web documents.

2.2. Distributed-object based website

In our model all Web documents are encapsulated in distributed objects called **GlobeDoc** [20] objects (or simply GlobeDocs). These objects provide a standard interface that allows the resources making up the document (i.e., its elements) to be retrieved. To access a website, a client must look up the GlobeDoc objects that it is interested in and connect to them. Once connected, the client calls appropriate methods to retrieve the object contents and present them to its user.

GlobeDoc is based on Globe and as such every GlobeDoc object is an instance of a Globe distributed object. Globe distributed objects are physically distributed, meaning that they are literally spread out over multiple address spaces; we call them **distributed shared objects** (DSOs). Each DSO consists of a number of local objects, called **local representatives** (LRs), one in each address space covered by the object (see Figure 1). Local objects are completely contained in one address space and can be implemented in any supported (not necessarily object-oriented or object-based) language.

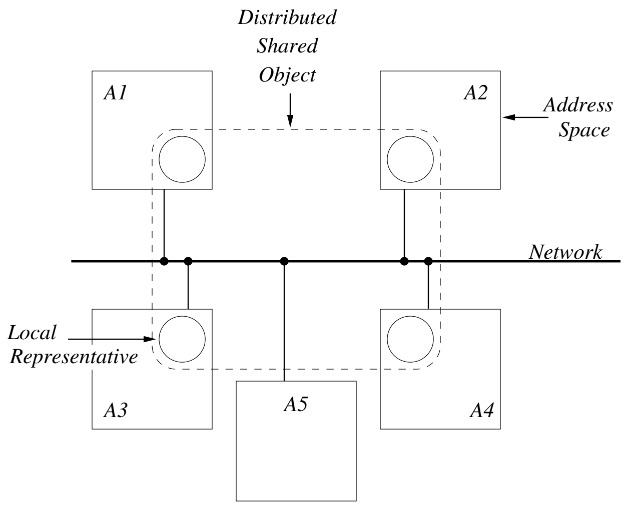

Figure 1. A distributed shared object

The benefit of a DSO is that its state can be copied or partitioned over any of the LRs. In some distributed shared objects the LRs might contain replicas of the state, in others the full state might be contained in only one of the LRs, and in still others each LR might contain only a part of the whole state. Globe DSOs allow this distribution of state to be determined by the object implementation itself. Because the state distribution is encapsulated within the object, the replication or partitioning is transparent, that is, neither clients, nor other system components need to be aware of an object's **distribution policy**. An object's distribution policy can therefore be set to one that suits the object's needs (i.e., the way that it is used), and need not depend on some global system policy.

2.3. System architecture

2.3.1. Binding and services. To communicate with a DSO (e.g., a GlobeDoc), a client must **bind** to the object. This causes a new LR to be created in the client's address space, effectively connecting that address space to the rest of the DSO. Once an LR is created in a client's address space, the client can communicate with the whole DSO by calling (local) methods on the LR. The binding process is illustrated in Figure 2. It can be divided into two main phases: finding an object and installing the appropriate LR.

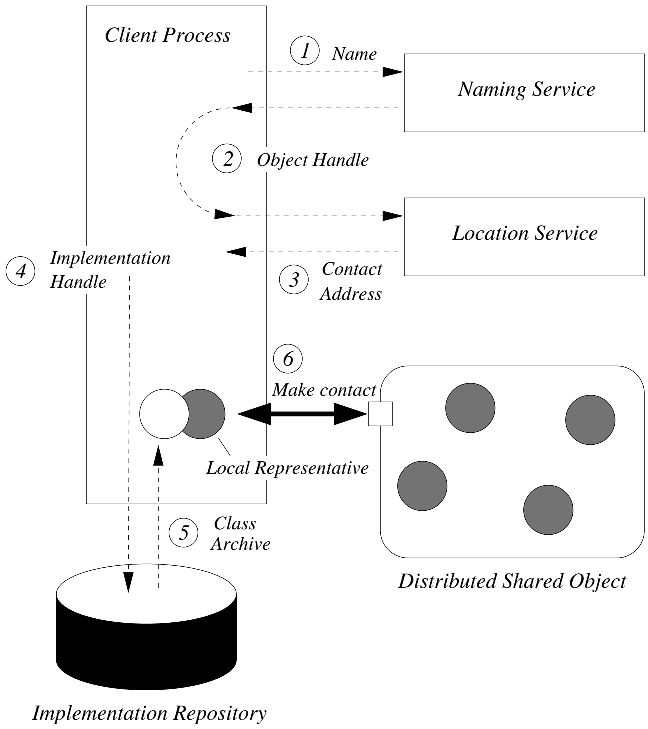

Figure 2. The binding process

In the first phase, a binding client starts by passing a name of the DSO to the **naming service**. The Globe naming service is responsible for mapping a name to a globally unique, location-independent **object handle**. The naming service returns an object handle, which is then passed on to the **location service**. The location service maintains a mapping of each object handle to a set of **contact addresses**, which represent the contact points of a DSO (analogous to service access points in computer networks). Although normally more than one of these addresses may be returned to the binding client, we assume, for simplicity, that only one address is returned.

In the second phase, the contact address is used to find and install an appropriate LR in the client's address space. The first step of the second phase involves extracting an **implementation handle** (which identifies an implementation) from the contact address and passing it to an implementation repository. The implementation repository finds a corresponding implementation and returns it in the form of a **class archive**. A class loader subsequently extracts the implementation code from the class archive, loads it into memory, creates the actual LR and initializes it. Once the LR is initialized, the client will be able to communicate with other parts of the DSO. We say that the client is now **bound** to the DSO. The LR in the client's address space is said to be **connected** to the rest of the DSO.

Splitting the binding process into these different steps makes the whole system more flexible. As mentioned above, naming and location are split into separate services so that object names and object locations can be kept separate. By separating naming from location, we avoid the need to change names (as is the case with current URLs) when an object changes its location or is replicated. The implementation repository is kept separate from the location service for a similar reason: an object's location and its implementations remain independent of each other. Because performance is important, it is conceivable that contact addresses will be stored and reused by clients to avoid having to resolve names and object handles. Although our contact addresses are comparable to the (location-dependent) object references in CORBA and Java RMI, a Java RMI object reference, for example, is actually a complete serializable proxy that is handed out between different processes. By separating implementations from contact addresses, it becomes possible for us to return client-specific implementations. Thus, for example, a client that prefers to use only certified LR implementations may use the same contact address as one who also accepts non-certified implementations.

2.3.2. Structural support. Implementing a website as a collection of Globe DSOs requires structural support for the DSOs. This support includes providing address spaces for LRs, providing access to the services used during binding (i.e., name service, location service, etc.), and providing a means to access objects from client Web browsers. Figure 3 shows an infrastructure that provides such support. The following description of requesting a Web page from an object will highlight the most important components in the figure. A detailed description of each component is given in the following section.

In our approach (path A), a browser sends a request for a Web page (as a URL) to a proxy server that filters GlobeDoc-specific names from regular URLs. GlobeDoc-specific names are forwarded to a **GlobeDoc gateway**, and regular URLs are forwarded in the normal way (an alternative, path B, is that requests for *embedded URIs* are sent di-

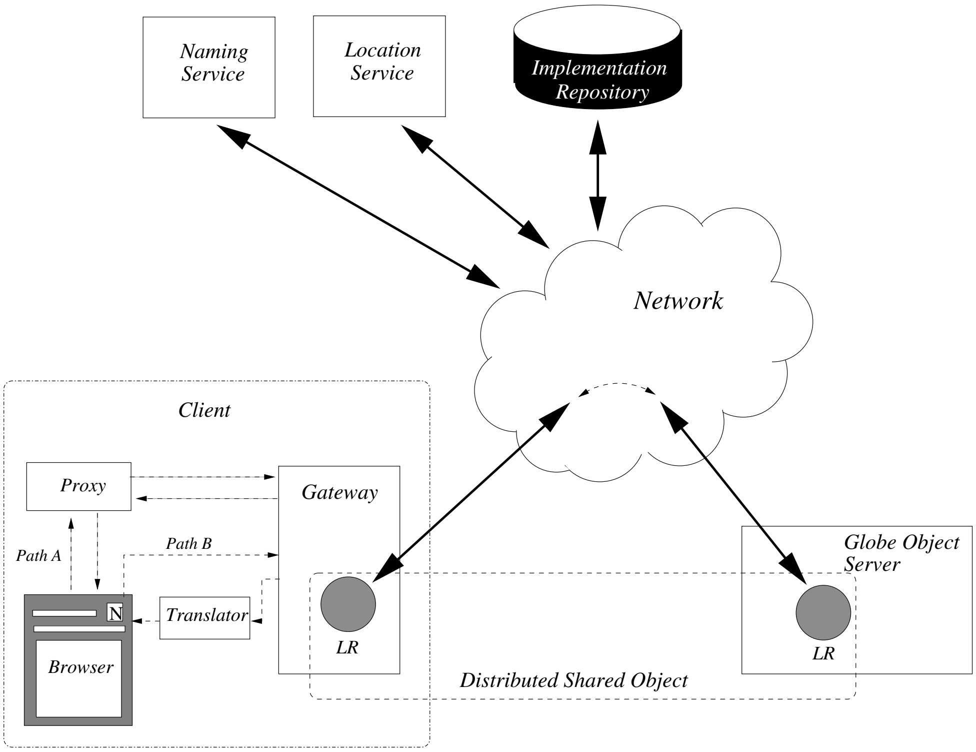

Figure 3. The Globe website infrastructure

rectly to a gateway and results are returned through a **translator**, this will be described in more detail later). The gateway is a special instance of a **Globe object server** and provides address spaces and service access to LRs. It binds to the referred object causing a new LR to be created in the gateway's address space. This newly created LR connects to another LR (or replica) hosted by a remote Globe object server (with functionality similar to the GlobeDoc gateway), and becomes part of the DSO. Once it is bound to the DSO, the gateway calls a method on the LR requesting the Web page. This causes the LR (depending on the replication strategy) to request the page from the remote LR or look it up in its local state, and return it to the gateway. The gateway passes the page on to the proxy or translator where it is packaged in a proper HTTP reply and sent to the Web browser. Note that when requesting Web pages from objects that have already been bound to, the whole binding step can be skipped and the page can be immediately requested from the LR.

3. System components

We now describe each of the components shown in Figure 3 in more detail.

3.1. Naming service

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

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

The current name space implementation is largely based on DNS [14] name servers. In this implementation it is assumed that the root as well as (hierarchically) higher-level nodes in the name space correspond to regular DNS domains. In theory, leaf nodes, which represent actual DSOs, and lower-level interior nodes also correspond to DNS domains, but these are implemented in a Globe-specific way. Such *Globe domains*, (i.e., Globe-specific as opposed to regular DNS domains) are implemented by **Globe domain servers**. A Globe domain server consists of two parts: a name server and a naming authority. The *name server* is the main part and implements the subtree rooted at the node represented by the Globe domain. This subtree corresponds to a DNS zone. Currently our name servers are implemented using BIND8 [2]. The *naming authority* is a server colocated on the same machine as the name server and is the only entity allowed to invoke update operations at the name server.

To adhere to DNS naming syntax, we transform a name such as globe://nl/vu/cs/object/foo into foo.object.cs.vu.nl. When resolving it, the DSO name (e.g., foo.object.cs.vu.nl) is passed to a DNS resolver as though it were a regular host name. The resolution eventually reaches a Globe name server (e.g., the server for object.cs.vu.nl), where the remainder of the name is resolved to the appropriate object handle. Details on the name service implementation can be found in [4].

3.2. Location service

An object handle is resolved to one or more contact addresses by the location service. As mentioned, an object handle is a location-independent and universally unique object identifier that can be used as a worldwide object reference. A contact address, on the other hand, describes a contact point, which is an address where a DSO can be contacted. It contains information about where and how the object can be reached. This information is stored in the form of an implementation handle, which identifies the implementation of the LR needed to contact the object, and data used to initialize the LR, which includes the actual network address of the contact point. While a DSO has only one object handle that does not change throughout its life, contact addresses can be added, removed or updated as necessary.

The location service stores every DSO's contact addresses and maintains a mapping of every object handle to a set of contact addresses. Because of this, it must be capable of storing and supporting frequent updates of large numbers of contact addresses. It must also be able to efficiently resolve object handles to contact addresses. To ensure scalability, it is essential that the location service exploits locality.

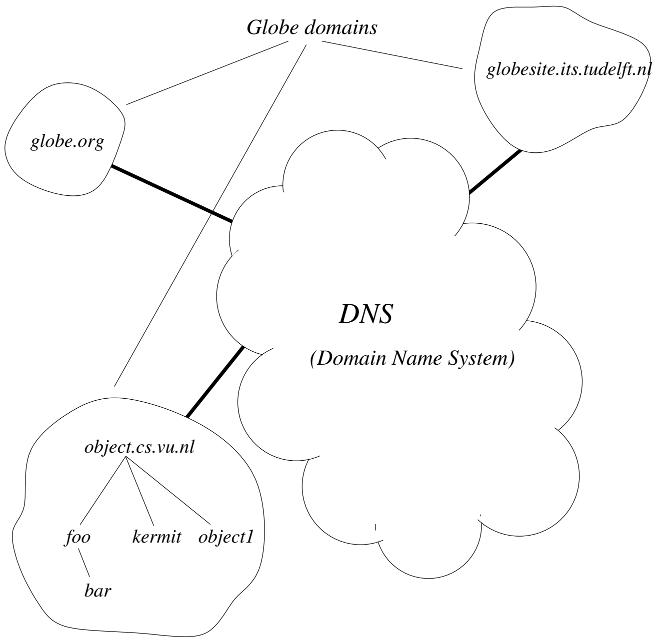

Figure 4. DNS based Globe name space

The location service is implemented as a worldwide distributed search tree in which all requests for updates and look-ups are initiated at leaf nodes. If a leaf node cannot handle a request, the request is forwarded to its parent. In this way, we exploit locality and achieve scalability. To prevent higher-level nodes from being swamped with requests, we partition these nodes by dividing the set of object handles using a hashing technique. It is beyond the scope of this paper to explain in detail the implementation of the location service. Further information can be found in [18].

3.3. Implementation repository

The implementation repository is a service that stores LR implementations and makes them available to binding clients. These implementations are stored and transferred as class archives, which are files that contain all the implementation code needed by an LR. Storing the entire implementation of an LR in a single class archive makes its transportation and management easier compared to having multiple files. In our implementation, a class archive is a Java jar file and contains the Java class files that form an LR implementation.

When an LR implementation is registered at the implementation repository it is assigned an implementation handle. The implementation handle is placed in a contact address and subsequently used by a binding client to retrieve (copies of) the implementation. An implementation handle is an opaque identifier that is generated by the implementation repository. Currently, we support only *file* URLs as implementation handles, that is, a handle simply contains the path name of a locally available class archive file. Other schemes, such as those based on *ftp* or *http* URLs, may be preferred for a wide-area system such as the Web. We plan to support such URLs as well.

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

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

3.4. Globe object server (including the gateway)

The GlobeDoc gateway and Globe object server both provide **address spaces** and **runtime services** to LRs. The difference between the two is that the gateway's main goal is to provide clients with access to GlobeDoc LRs and their methods, while the Globe object server provides an environment for non-client Globe LRs. The gateway is usually placed either very close to a client (e.g., on the same machine or the same local network) or is actually part of the client process (e.g. built into a browser). It provides facilities that allow clients to bind to GlobeDocs and call methods on the resulting LRs.

When the gateway is a separate process, it must provide an external interface through which clients can bind to a GlobeDoc and call its methods. This can take the form of a dedicated RPC-style interface, or a server that accepts custom HTTP requests from clients. When the gateway is integrated with the client, the client can perform method calls directly on the LRs as both are in the same address space. The client will also have direct access to the Globe runtime system and can use its services and resources to bind to DSOs.

A Globe object server always runs as a separate process. It has a remotely accessible interface that allows LRs, other Globe object servers, or administrators to request services from it. These services include binding to an existing DSO, unbinding from a DSO, creating a DSO and destroying a DSO. A binding request causes the Globe object server to bind to the given DSO, resulting in an LR of that DSO being created in the Globe object server's address space. Likewise, a Globe object server can be requested to unbind from a DSO, resulting in all LRs of that DSO being removed from the server's address space.

In the remainder of this section, we concentrate on the Globe object server. The GlobeDoc gateway has very similar semantics, except that it can support only client LRs. In practice, this means that a GlobeDoc gateway cannot offer a contact point for a DSO. The most important functions of both (the gateway and Globe object server) are, however, that they provide access to services such as the naming and location service, facilities for binding to a DSO, and local services to LRs contained in its address space. These issues are described next.

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

3.4.2. Support for binding. The Globe object server also provides the facilities needed for binding. These are encapsulated in a *binding object*, a local object that is part of the runtime system. Binding in Globe consists of at least three steps: (1) name resolution, (2) object handle resolution, and (3) loading and initialization of a LR. Normally, binding starts at the first step. It is, however, possible to begin binding at any other step, as long as the information needed by that step is present. For example, to start binding at the second step, a client would need to have an object handle to pass to the location service. A Globe object server might store an object handle as previously returned in step 1 to avoid a name look-up when it is requested to bind to that same object again later.

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

However, it is not always wise to immediately fulfill a request to unbind from a DSO. Consider, for example, a GlobeDoc gateway that has just bound to a DSO to retrieve information for a client. In the same style as HTTP, the gateway could decide to immediately unbind from the DSO as soon as it has passed the information to the browser. However, it may be much more efficient to stay bound to the DSO, anticipating more requests for that object. In effect, a server or gateway can decide to *cache* a binding for later use. In our current implementation, which supports only passive Web documents, the effects of caching bindings turns out to be comparable to that of traditional Web caches.

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

Although LRs are not active objects, they do require thread management facilities. For example, a thread is started whenever a message comes in from another LR. The thread facilities are provided by the runtime system. The runtime system also offers access to low-level resources such as communication points (e.g., sockets) and persistent storage (such as files on disk). These resources are all offered through standard platform-independent interfaces.

3.5. Local representative

As mentioned earlier, a local representative is a local object that is wholly contained in one address space. A local representative implements the interfaces exported by its DSO. Each LR may implement these interfaces in a different way, depending on its role in the distribution strategy of the DSO. For example, in a DSO with only one copy of the state, there will be a "primary" LR that contains that state. Other LRs in that DSO will implement the DSO's interfaces by simply forwarding requests to the primary. However, when the state has been replicated across multiple machines, an LR may hold a local copy of that state. In that case, when a client invokes a write method, that method may have to be propagated to all other LRs, as in active replication.

The aim in Globe is to support object developers by separating functionality from distribution. In principle, an object developer should be able to concentrate only on designing and implementing the object's basic functionality as specified in that object's interfaces. Separate from this activity, a developer should concentrate on how that functionality is to be distributed and replicated across a network. We refer to the latter as designing and implementing a distribution strategy. It is this separation of concerns that gives Globe much of its flexibility.

Separation is achieved by constructing LRs in a modular way. An LR is built up of (at least) four subobjects, each responsible for a different part of the functionality, as shown in Figure 5. The *communication* and *replication* subobjects work together to implement the distribution strategy of a DSO. The replication subobject takes care of replication and consistency issues, while the communication object is responsible for exchanging messages with other LRs. The *semantics* subobject implements the actual functionality of the DSO. A DSO's state is generally stored in the semantics subobject of its LRs. Finally, the *control* subobject takes care of invocations from client processes and controls interaction between the semantics and replication subobjects. Details can be found in [19].

We return to precise definitions of interfaces below.

3.6. Browser and translator

Ideally, users should be able to use regular Web browsers to access GlobeDoc Web documents. Unfortunately, current browsers are incapable of resolving GlobeDoc URIs as they do not understand `globe:` schemes. A way around this problem is to use GlobeDoc-aware proxies. These are Web proxies that filter out GlobeDoc requests and send them to a (local) GlobeDoc gateway. The gateway binds to the appropriate objects and performs methods on it on behalf of the user. Any results from the methods are returned to the

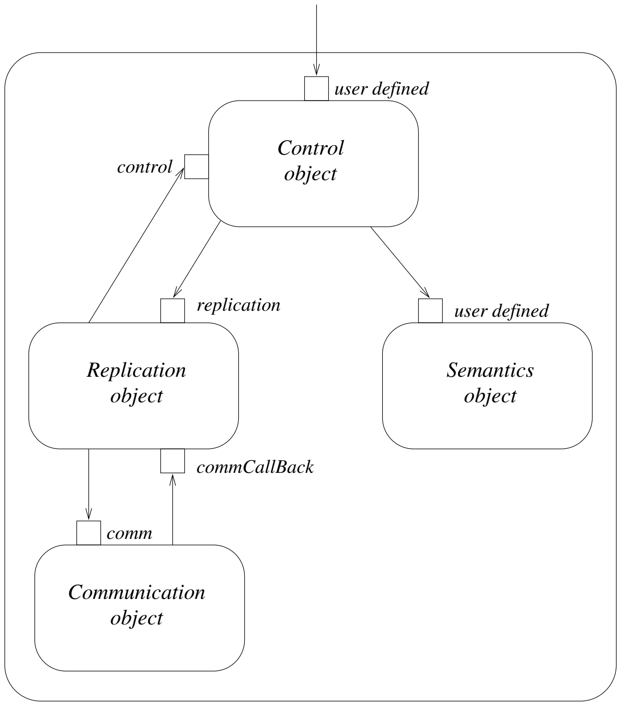

Figure 5. Local Representative

user's browser through the proxy. Non-Globe requests are passed to appropriate servers, as in regular proxies.

A disadvantage of the proxy approach is that *all* requests from the browser (including non-GlobeDoc requests) must be forwarded through the proxy. As a result, the proxy must be able to handle all the various kinds of schemes supported in URLs, or forward them to a proxy that can. An approach that avoids this problem uses a GlobeDoc translator. This component translates GlobeDoc URIs to what we call **embedded URIs**. An embedded URI is a regular HTTP URL that contains an object name and a gateway address, such as http://globedoc.cs.vu.nl/nl/vu/cs/foo/object. When an embedded URI link is clicked, an HTTP request for the embedded object name is sent to the gateway. The gateway binds to the object and calls methods on it as usual, except that results are passed to the translator. At the translator, each link consisting of a GlobeDoc URI, is rewritten to contain an equivalent embedded URI. The modified result is then passed on to the browser. In this way, access to non-GlobeDoc Web resources is not affected by the added ability to access GlobeDoc resources.

We have recently built a GlobeDoc-aware Web browser. This is a browser that can natively resolve GlobeDoc URIs and bind to the corresponding GlobeDoc objects, that is, it has the gateway functionality built into it. Rather than build a GlobeDoc-aware browser from scratch, we are investigating the use of browser plug-ins to add GlobeDoc functionality to existing browsers. Such plug-ins are loaded and used when URIs with appropriate scheme identifiers are accessed. We have currently modified Mozilla (the open-source version of Netscape's browser) to support protocol plug-ins. Microsoft's Internet Explorer already supports this extensibility, while Mozilla is officially adding it as well.

4. GlobeDoc objects

As mentioned earlier, in our model, a website consists of related Web documents, each encapsulated in a GlobeDoc object. A GlobeDoc encapsulates an entire Web document and contains a collection of logically related **elements** including Web pages and other resources such as icons, images, sounds, etc. Elements in a GlobeDoc may contain internal as well as external hyperlinks. An internal link refers to an element in the same GlobeDoc, whereas an external link refers to an element in another GlobeDoc. Every GlobeDoc assigns one element to be the root, which provides access to other elements through internal links, and is comparable to the index.html file. Because we do not say anything about the contents of an element, every element has a set of properties associated with it. At the least, these properties include a MIME type that describes an element's contents.

4.1. The GlobeDoc semantics subobject

A GlobeDoc allows elements to be added and removed, as well as the contents and properties of existing elements to be modified. Its functionality is implemented by a semantics subobject having a set of predefined interfaces as shown in Figure 6. Clients use methods from these interfaces to access and modify the elements contained in a GlobeDoc.

The *document* interface contains methods that act on the document as a whole. It allows elements to be added and removed, as well as element names to be retrieved. An element is always referenced by its name, which is a character string. The *content* interface is used to retrieve and set an element's contents. The contents are contained in a byte array. An element's properties can be set and retrieved through the *property* interface. Properties are represented as strings of (attribute,value) pairs.

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

```
interface document {
        void            addElement(name, elementType, contents);
        void            deleteElement(name);
        name            getRoot();
        name[]          allElements();
}

interface content {
        contents        getContent(name);
        void            putContent(name, contents);
        void            putAllContent(name[], contents[]);
}

interface property {
        properties      getProperties(name);
        void            setProperties(name, properties);
}
```

Figure 6. The GlobeDoc interfaces

4.2. Naming

A GlobeDoc, like other Globe DSOs, is referenced by a location-independent object name. GlobeDoc element names are, on the other hand, valid only in the context of a GlobeDoc. To refer to a GlobeDoc element, therefore, both the GlobeDoc and element names are required. For convenience, we allow a GlobeDoc URI to contain both a GlobeDoc home and an element name. The URI globe://nl/vu/cs/object/gdo:/element.html, for example, refers to an element named /element.html in a GlobeDoc named /nl/vu/cs/object/gdo. A GlobeDoc URI with an empty element name implicitly refers to the root element. For integration in the current Web, GlobeDoc URIs can be embedded in URLs, for example as http://globe.cs.vu.nl/nl/vu/cs/object/gdo:/element.html.

4.3. GlobeDoc replication

Our claim for Globe's scalability, and thus the reason for basing corporate websites on Globe, rests on its flexible approach to distribution strategies. To experiment with this flexibility we have implemented a number of simple replication strategies, and are looking into other strategies that are optimal for large (corporate) websites. The simplest distribution strategy is client/server interaction. In this strategy there is one LR that acts as a server and contains all of the object state (i.e., all the elements). The rest of the LRs are stateless proxies that forward all requests to and receive all replies from the server. We have also implemented active replication where all the LRs contain full replicas of the state. Read operations are served locally by an LR, while write operations are forwarded to and executed on all LRs. Another strategy that we have implemented is a simple master/slave variation. Here all LRs have a replica of the state, but only one master LR is allowed to perform updates. When the master performs an update it sends a message to all the other LRs informing them of the update. Examples of more complex strategies are those that combine replication and caching — some LRs acting as consistent full replicas, and others acting as less consistent pull caches.

These are all examples of what we call *static* strategies, strategies that do not adapt to an object's circumstances. We are currently experimenting with *adaptive* distribution strategies. These are strategies that monitor an object, detect when its configuration becomes suboptimal, and cause it to change to a new strategy. Adaptive distribution strategies are especially useful in the face of *flash crowds*. These are extreme (but temporary) increases in requests caused by unexpected interest in some documents. Because a flash crowd is unexpected the mass of requests coming in usually overwhelms the server, causing it to fail. By determining that it is experiencing a flash crowd, a GlobeDoc with an adaptive distribution strategy can start making widely distributed replicas, preventing any single replica from becoming overloaded, and thus keeping the document available. When the flash crowd subsides, the GlobeDoc can recall the replicas that are no longer being used.

To gain insight into the various replication strategies, we have performed experiments based on simulations (using Web server logs) of static and adaptive distribution strategies under normal and flash crowd conditions. The results show us that not only do different documents require different strategies (i.e., a one-size-fits-all approach to replication strategies is not appropriate) but that adaptive distribution strategies can (and do) correctly respond to and prevent problems from flash crowds. Details about these experiments and our results can be found in [15].

Two important aspects of adaptive distribution strategies are determining when and how to change the strategy (e.g., discovering flash crowds), and actually making the changes. While the former involves monitoring an object's network usage and performance and applying heuristics to determine appropriate changes, the latter involves actually creating and destroying replicas (LRs). In order to create an LR, a Globe object server that can host the LR must be found. To make finding Globe object servers possible we are developing a **Globe virtual network**. This is a service that keeps track of all available Globe object servers and contains information about every server's location, available resources (e.g., disk, memory, network access, persistence, fault tolerance, etc.), and authorization requirements (i.e., who is allowed to use the server).

5. Related work

With the increasing importance of the Web, improving the performance of the Web in general and large corporate websites in particular has become a high priority topic. As such, many commercial and research projects have been proposed to solve this problem, with the two most popular approaches being clustering and content delivery networks (CDN).

Clustering offers distribution of requests among a fixed set of mirror servers. The greatest challenge in clustering is to transparently redirect requests to appropriate servers. Approaches to this transparent redirection range from redirection of requests at the DNS level [11] to redirection in the routers themselves [6] [5]. While most of these approaches solve problems such as load balancing, none offer dynamic or adaptive replication.

CDNs improve on clustering and tackle the dynamic replication problem. Solutions such as Footprint [7], Free Flow [1] and RaDaR [16] provide networks of servers that mirror customer's sites on demand. They implement dynamic strategies that make replication decisions based on a site's traffic patterns and client locations. This is similar to what we propose in our own model, however, we go one step further and offer an architecture where the actual replication algorithm can be (dynamically) determined per document. We show in [15] that this is essential for achieving optimal performance.

Current distributed object systems such as CORBA, DCOM and Java RMI provide remote objects, rather than physically distributed objects. This means that the actual object state is always kept at a central server and clients use simple proxies or stubs to access it. Although it is possible to modify or augment these systems to provide some form of replication [8] [12], these modifications always prescribe a single global replication strategy to all objects. The reason for this is that modifications are made to the actual middleware layer and services, as opposed to the Globe approach where the replication strategy is part of the object. As mentioned, we feel that support for per-object replication is essential in a wide-area distributed system.

Approaches combining distributed objects and the Web range from using CORBA or DCOM objects for distributed back-end processing in servers [3] and combinations of Java front-ends communicating with distributed object back ends [10] to complete distributed-object based models of the Web [9]. The main difference between these and our approach is the flexibility with regards to distribution strategies that Globe provides.

One model that does provide physically distributed objects is that based on fragmented objects [13]. Although fragmented objects have been designed to encapsulate their own distribution policy, they have not been designed with worldwide scalability in mind and have not been applied to the Web.

6. Conclusion and future work

In this paper we have presented GlobeDoc, a scalable architecture for corporate websites based on Globe distributed objects. Globe allows scaling techniques, such as replication, to be applied on a per-object basis which we believe is an essential property for a scalable wide-area distributed system. We have described the GlobeDoc website model and given details of the system components including GlobeDoc objects, which are Globe DSO representations of Web documents.

We have recently built a small GlobeDoc based website (publicly accessible from the Globe home page: http://www.cs.vu.nl/globe) that implements all the main components described in this paper (translator, gateway, Globe object server, GlobeDoc objects, etc.). The service contains GlobeDocs that encapsulate the Web documents on the Globe website and is being used as a first test of the architecture. The general configuration is similar to that in Figure 3. Each GlobeDoc object is started in its own Globe object server. However, unlike the Globe object servers described in this paper, these servers are very simple and do not yet offer support for persistence and fault tolerance nor the possibility of remote management. We have also implemented a Globe-aware version of Mozilla that recognizes Globe URIs and forwards the requests directly to the gateway (bypassing the translator). All GlobeDoc components, except for our adaptation of Mozilla, are implemented in Java.

Besides this simple GlobeDoc implementation, we are working on the implementation of a full-blown Globe object server, adaptive distribution strategies, and the Globe virtual network. A complete Globe object server will provide support for persistence and fault tolerance. At present we have designed and implemented persistence support and are researching requirements for fault-tolerant DSOs. With regards to adaptive distribution strategies we are currently looking at appropriate heuristics for deciding when and were to create new replicas. We are also investigating what replication algorithms are most appropriate for Web documents and when a document should change algorithms. The Globe virtual network (described in Section 4) is currently in the design phase. We have determined the general architecture and are working towards an implementation. Furthermore, we are investigating how security can be incorporated into the Globe framework so that security policies can be attached to individual Globe objects in a similar way as done with distribution now. All of these components will be combined to implement a fully distributed website implementation that can be used to perform experiments with and validate new distribution strategies.

References

[1] Akamai Technologies, Inc. *Free Flow*. http://www.akamai.com.

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

[3] Apple. *WebObjects*. http://www.apple.com/webobjects/.

[4] G. Ballintijn, P. Verkaik, E. Amade, M. van Steen, and A. S. Tanenbaum. A scalable implementation for human-friendly URIs. Technical Report IR-466, Vrije Universiteit Amsterdam, the Netherlands, Nov. 1999.

[5] Cisco Systems. *Distributed Director*. http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/.

[6] O. P. Damani, P.-Y. Chung, Y. Huang, C. M. R. Kintala, and Y. M. Wang. One-IP: Techniques for hosting a service on a cluster of machines. *Comp. Netw. ISDN Syst.*, 29:1019–1027, 1997.

[7] Digital Island, Inc. *Footprint*. http://www.digisle.net/services/cd/footprint.shtml.

[8] P. Felber. *The CORBA Object Group Service. A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1998.

[9] D. B. Ingham, M. C. Little, C. J. Caughey, and S. K. Shrivastava. W3Objects: Bringing object-oriented technology to the Web. In *Fourth Int'l WWW Conf.*, Boston, Mass., Dec. 1995.

[10] Iona Technologies. *OrbixWeb*. http://www.iona.com/products/orbixweb/.

[11] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. In *First Int'l WWW Conf.*, April 1994.

[12] J. Maassen, T. Kielmann, , and H. E. Bal. Efficient replicated method invocation in java,. In *ACM 2000 Java Grande Conference*, San Francisco, CA, June 2000.

[13] M. Makpangou, Y. Gourhant, J. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press, July 1994.

[14] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Nov. 1987.

[15] G. Pierre, I. Kuz, M. van Steen, and A. S. Tanenbaum. Differentiated strategies for replicating Web documents. In *Fifth Int'l Web Caching and Content Delivery Workshop*, May 2000.

[16] M. Rabinovich and A. Aggarwal. RaDaR: A scalable architecture for a global Web hosting service. In *Eighth Int'l WWW Conf.*, May 1999.

[17] K. Sollins and L. Masinter. Functional requirements for uniform resource names. RFC 1737, Dec. 1994.

[18] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic design of the Globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[19] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, Jan. 1999.

[20] M. van Steen, A. S. Tanenbaum, I. Kuz, and H. J. Sips. A scalable middleware solution for advanced wide-area Web services. *Distributed Systems Engineering*, 6(1):34–42, 1999.