The Globe Infrastructure Directory Service

Ihor Kuz

Delft University of Technology, Delft, The Netherlands

Maarten van Steen

Vrije Universiteit, Amsterdam, The Netherlands

Henk J. Sips

Delft University of Technology, Delft, The Netherlands

Abstract

To implement adaptive replication strategies for Web documents, we have developed a wide area resource management system. This system allows servers to be managed on a local and global level. On a local level the system manages information about the resources and services provided by the servers, while on a global level the system allows servers to be searched for, added to, and removed from the system. As part of the system, and also in order to implement adaptive replication strategies, we introduce a hierarchical location representation for network elements such as servers, objects, and clients. This location representation allows us to easily and efficiently find and group network elements based on their location in a worldwide network. Our resource management system can be implemented using standard Internet technologies and has a broader range of applications besides making adaptive replication strategies possible for Web documents.

Key words: resource management, distributed systems, Web site replication

---

1 Introduction

Slow document transfers are a familiar problem for all users of the World Wide Web. As such, speeding up document transfers and reducing access times to Web sites have been goals of researchers since the early days of the Web. Many problems can be alleviated by replicating the Web documents that make up a Web site, thus spreading the load over multiple servers and network connections as well as reducing access time by bringing the content closer to the clients. Current techniques for doing so may, however, lead to new problems such as inconsistent replicas. Also, they are often too restrictive as they generally impose the same replication strategy on all Web documents. For example, no distinction is made between pulling or pushing updates; either updates are always pushed or they are always pulled in by replicas from a primary.

We claim that for Web sites to be optimally replicated it is necessary to apply replication strategies to every Web document individually, depending on its needs and characteristics. Furthermore, we also claim that because a Web site's usage changes over time it will be necessary to dynamically adapt these strategies to suit every document's current situation [12].

1.1 Background

To make this flexibility possible we have developed **GlobeDoc** [18], an architecture that encapsulates Web documents in distributed objects. A GlobeDoc object is an instance of a Globe distributed object [17]: a physically distributed object whose state may be partitioned and replicated across multiple servers at the same time. Besides being physically distributed, each GlobeDoc object fully encapsulates its own replication strategy, meaning that there is no systemwide policy imposing how an object's state should be replicated and kept consistent. Clients need not be aware of an object's replication strategy as it is kept hidden behind an object's interface.

A client wishing to use a GlobeDoc object must first bind to the object. Binding results in a **local representative** (**LR**) of the object being placed in the client's address space. An LR implements the object's interface and allows the client to perform local method calls on it. How the method calls are processed by the LR depends on the object's replication strategy. For example, one strategy could dictate that a method call should be performed locally, while another could dictate that it should be forwarded to the object's representatives at other address spaces and that nothing should be done locally. When an LR contains a copy of the object's state we generally call it a replica (of the object).

For each object we distinguish between three different kinds of LR: a permanent LR, an object-initiated LR, and a client-initiated LR (Figure 1). A permanent LR forms the base for a GlobeDoc object, is persistent, and always has permission to perform authoritative changes to the object's state. It is comparable to a Web page's home server. An object-initiated LR is created by an object to, for example, help relieve the load on its existing LRs. It contains a copy of the object state, but cannot perform authoritative changes on that state. An object-initiated LR is comparable to the copy of a Web page on a mirror site. Finally, a client-initiated LR serves to temporarily enhance performance for an object's clients. As such it can be compared to the copy of a Web page stored in a browser or proxy cache. See [8] for more details on how the different kinds of LRs can affect and determine an object's replication strategy.

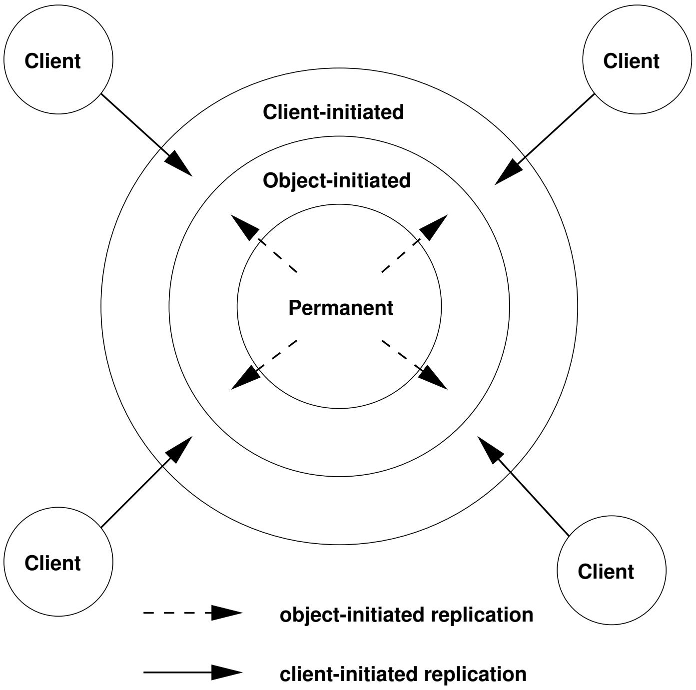


Fig. 1. Different types of replicas

1.2 Replica management

Each replica is hosted on a server called a **Globe Object Server** (**GOS**). Besides managing the lifecycle of an LR and providing services such as persistence and fault tolerance, a GOS also provides an LR with access to its resources (e.g., memory, CPU, disk, network interface). A detailed description of resource management by a GOS is outside the scope of this paper. It is sufficient to note that this management is largely covered by traditional operating systems techniques.

A process (for example, a client or an object replica), possibly running on a remote machine, can request a GOS to host a replica of an object. For this reason, a GOS must provide information about the resources and services it offers, as well as other properties such as its location and security policies. A process may decide to query several GOSs about their current status, and select one based on the information received in response. We refer to the capability of providing this information on a global scale as global resource management.

Global resource management is needed to support adaptive replication strategies. Besides static replication strategies, we have been looking at strategies that can change their characteristics depending on the needs of an object. Such adaptive strategies are capable of creating new LRs, destroying existing LRs, and modifying the replication and consistency strategies used by these LRs (e.g., dynamically changing the distribution of an object) when the access patterns of an object change. When creating a new LR the initiator, be it an object or a client, must be able to find a server to host that LR. This means that the initiator must search through all the available GOSs and find one that offers the required resources and services.

Because that collection of GOSs may be spread worldwide, global resource management is potentially faced with a scalability problem. Fortunately, searching for a specific GOS is often done within a restricted region. For example, a replica may need to be created in a region from where many requests are currently coming. A major problem that we must solve is identifying such regions. Current (network) location representation schemes, such as IP addresses and DNS names, do not allow us to easily determine the proximity of a group of processes. Thus, for example, given a list of IP addresses, it is not easy to determine the geographic or network topological proximity of the nodes running the corresponding processes. For this reason, we decided to develop a new model for representing the location of resources. This model is an integral part of our overall resource management system.

In this paper, we present the **Globe Infrastructure Directory Service** (**GIDS**), a system for managing resources and services made available by a worldwide distributed collection of Globe object servers. The GIDS allows servers to register resources, services, and other properties, while providing clients an efficient search facility for locating the servers they need.

GIDS is highly scalable with respect to geographical distribution and number of processes. In other words, it can efficiently handle a collection of processes regardless of whether these processes are all on the same local network, or spread out over the entire Internet. Likewise, its performance stays within reasonable bounds even if the number of processes is extremely large. The main contribution of this paper is that we describe a general architecture for location-based worldwide resource management. As we also demonstrate, this architecture can be implemented efficiently using existing Internet technologies.

The rest of this paper is organized as follows. Section 2 describes the architecture of this system and Section 3 discusses the details of implementing this architecture using existing Internet technologies. Section 4 reviews related work and Section 6 presents our conclusions and discusses future work.

2 Globe Infrastructure Directory Service

In GIDS, we make a distinction between local resource management and global resource management. Local resource management deals with managing resources and services within a base region. A **base region** represents a small geographical or network-topological area containing a group of servers (Globe object servers as well as other servers). It is the smallest unit of proximity known to the GIDS. In other words, if two processes are in the same base region, the GIDS will consider them as being at the same location.

Global resource management deals with grouping the information from base regions into larger units, and making that information available to clients. In particular, global resource management deals with globally tracking resources and services, and providing efficient search facilities for clients, regardless of a client's location.

2.1 Base regions and local resource management

2.1.1 Region Service Directory

Every base region has a **Region Service Directory** (**RSD**). An RSD is the central access point for a base region, storing, managing, and making available data about all the servers in that region. A server that wishes to make its information available through an RSD must register itself with that RSD by providing details of its resources, services, and additional properties. A server may register at an RSD only if it was previously authorized by the base region's administrator.

In the RSD, servers are identified by a set of attributes that describe their resources, services and properties. General properties include information such as the type of server, its address, its communication protocol, the platform it runs on, etc. Servers also have server-specific properties. For example, a GOS has properties that describe whether it offers persistence, how much disk space it makes available to each object, how much memory is available, the available local network bandwidth, authentication and authorization information, etc.

Figure 2 lists the operations that an RSD provides for accessing and modifying server information.

local operations
`add_server(server, properties)`
`delete_server(server)`
`modify(server, properties)`
`search(properties)`

remote operation
`remote_search(properties, location)`

Fig. 2. RSD operations

The `add_server`, `delete_server`, and `modify` operations allow management of an RSD and the server information that it stores. The `search` operation initiates a search in the RSD's base region for servers with the given properties. In contrast, the `remote_search` operation is used to search for servers *outside* the RSD's base region. We return to these operations later.

In other words, an RSD can be accessed both from within and from outside its base region. That is, it can be queried by clients registered in the same base region, and clients from other base regions.

2.1.2 Naming and locating base regions

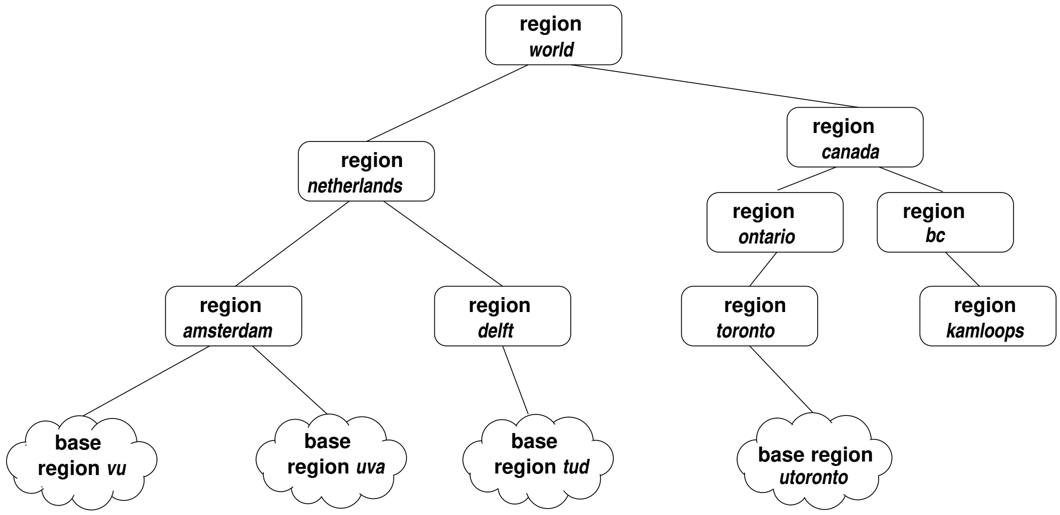
Base regions are grouped into nonoverlapping regions, which, in turn, are grouped into larger regions. This organization leads to a hierarchicy of regions, similar to the organization of domains in DNS. An important difference with DNS domains, however, is that regions *always* represent a notion of proximity. For example, the base regions within a city can be grouped into a separate region representing that city, which, in turn, can be part of a region representing a province or country.

In this sense, each (base) region has an associated location. A location is represented by a name that reflects the hierarchical organization of regions. For example, if the campus of the Vrije Universiteit forms a base region, a possible name for this region could be *vu.amsterdam.netherlands.world*, with each component name representing a region. In this example, the top-level region is named *world*.

In our system, a base region is the smallest grain of location representation, meaning that every server inside a base region has the same location as the base region itself. A *full* location name always refers to a single base region. A *partial* name (which is always a suffix of a full name), on the other hand, can refer only to a group of (base) regions, and never to a base region.

It is not necessary that the region hierarchy forms a balanced tree. As such, some branches in the hierarchy may be deep and broad, while others may be shallow and narrow. For example, in a region hierarchy that reflects geographic location, branches representing areas with a high density of servers will tend to be deep and broad, while those representing areas with few servers will be shallow and narrow.

Figure 3 shows an example of a region hierarchy and some associated full and partial names. Note that the partial name *kamloops.bc.canada.world* refers to an area with no base regions. In this case the set of base regions belonging to *kamloops.bc.canada.world* is empty. This feature (that a branch of the region hierarchy does not always end in a base region) is necessary to support the extensibility and flexibility of the hierarchy. It allows the hierarchy to be built up dynamically by adding regions and base regions as they are needed.



region names: vu.amsterdam.netherlands.world refers to base region vu
tud.delft.netherlands.world refers to base region tud
toronto.ontario.canada.world refers to region toronto
kamloops.bc.canada.world refers to region kamloops
canada.world refers to region canada

Fig. 3. Base regions and a simple hierarchy

2.1.3 Local management in a base region

Like Globe object servers, every client also belongs to a base region and has an associated location. Unlike servers, however, clients do not register with an RSD; they simply contact their local RSD to determine their location. Besides GOS details, an RSD also contains a wide variety of other information about its base region. For example, an RSD can also provide the address of a local or nearby DNS server, or otherwise that of a proxy to such a server.

Similarly, information on Web (proxy) servers, local file servers, and so on, can be readily provided by an RSD. In this sense, an RSD is similar to domain controllers as used in the Windows 2000 Active Directory. There are also important differences with Active Directory that are presented later.

2.1.4 Security in a base region

There are a number of tasks that can be performed with respect to an RSD that must be protected against abuse. The RSD's security model deals with authenticating clients and confirming that they are authorized to add, remove, or modify entries (i.e., information about services and their properties). The authentication can be performed automatically by the RSD. However, granting authorization is arranged out-of-band (e.g., as part of a service contract). Note that the authorization needed for modifying an RSD is separate from the authorization that a client might need to actually use a service in the base region (which is determined by the service itself).

2.2 Global resource management

The previous section described management of servers and resources on a local scale (i.e., within a base region). This section deals with the management of servers and resources on a global level involving multiple regions.

2.2.1 Searching for servers

The main functionality of GIDS is that it allows one to search for servers with given properties. In doing so, it distinguishes between two kinds of searches: local searches and remote searches. A **local search**, initiated at a specific base region, is limited to the servers in that base region, while a **remote search** allows a GIDS client to search for servers in base regions other than the one where the search was initiated.

As described above, a local search is invoked through an RSD's `search` operation. A remote search is invoked through a similar `remote_search` operation. The difference between the two operations is that the `remote_search` operation requires an additional location parameter.

For a local search, the RSD simply searches through its internal database of server properties trying to find servers that match the given property requirements. The client may limit the number of results it wants returned, ranging from one, to all the possible matches.

Remote searching is somewhat more complicated, and proceeds in two steps. The first step involves resolving the given location name into a set of RSD addresses that represent all the base regions referred to by that name. The second step involves performing a local search on each of these RSDs for servers that match the given property requirements. A remote search returns a set of servers within the specified location that fulfill all the specific client requirements.

Because locations are represented as names, the first step, location-name resolution, is done using a name service. This name service resolves a location name to the set of base regions (actually, the RSDs belonging to those base regions) that fall under that location. As an example, Figure 3 shows a number of base regions and a simple (geographic) location hierarchy. Resolving the location name *vu.amsterdam.netherlands.world* in this hierarchy would result in the RSD of base region *vu* being returned. Resolving the name *netherlands.world*, on the other hand, would result in the RSDs of base regions *vu*, *uva* and *tud* being returned because those three base regions lie in region *netherlands*. Finally, resolving the location *kamloops.bc.canada.world* would result in an empty set being returned because there are no base regions in region *kamloops*.

Such a name service can be implemented by associating a **Region Name Server** (**RNS**) with every region. An RNS stores the following information about its associated region. For each subregion, it maintains a mapping of that subregion's location name to the address of the server representing that subregion. Such a server is either an RNS in case the subregion is not a base region, and otherwise its RSD. To resolve a location name, name resolution starts at the root RNS, and moves down the hierarchy following the components of the location name, completely analogous to DNS name resolution. A partial name is always resolved to an RNS; a full name is resolved to an RSD.

When name resolution ends in an RNS, the naming subtree rooted at that RNS is traversed by resolving each pathname to a leaf node. This traversal results in a set of RSDs, which is subsequently returned to the client that initiated the original search operation. When name resolution ends in an RSD, that RSD is returned. Attempting to resolve a nonexisting name (i.e., one that refers to a nonexisting region) returns an error.

Once the location name is resolved to a set of RSDs, these RSDs are searched for appropriate servers using their local search operations. Clients may limit the number of results they wish to receive from the remote search. In this case, the search stops when that number of servers is found, otherwise it continues until all RSDs have been searched. The RSDs are searched in a random order to prevent any one RSD's servers from being chosen more often than others. This introduces simple load balancing and prevents servers from being favored or overloaded because their RSD is always found first during the location hierarchy traversal.

2.2.2 Scalability

Although having a root RNS through which all requests are made may pose a problem to scalability, we apply techniques (from DNS) such as caching of mappings at every RNS, which prevent every request from going through the root. Similarly, as in DNS, we may replicate the root RNS over multiple physical servers to prevent overloading of any single server.

2.2.3 Global region management

The region hierarchy is dynamic, meaning that there is no predefined structure determining the number and locations of all (base) regions. Rather, the hierarchy is built up as regions and base regions are added and removed. This addition and removal of regions is controlled by a global management policy. The management policy determines which clients are authorized to add or remove regions, and where and when such operations can take place. For example, it determines whether it is possible to add regions at any RNS or only at the leaves of the hierarchy, whether it is possible to remove or add whole subtrees at once, etc. A detailed discussion of management policies is, however, beyond the scope of this paper. In the GIDS the management policies can be implemented either internally (e.g., by the RNSs themselves) or externally (e.g., as an external management service). They can also be either automated (e.g., a region can be added by performing an operation on an existing RNS) or manual (e.g., an administrator must edit a configuration file and restart the parent RNS for a region to be added). As we discuss below, in our current implementation we adopt the policy implemented in DNS.

3 Implementation

In this section we describe an implementation of GIDS based on widely available LDAP [19] and DNS [10] technology.

3.1 LDAP

LDAP (the Lightweight Directory Access Protocol) is a lightweight protocol for accessing X.500 directory services. The protocol is based on the IP protocol stack and as a result has become popular on the Internet. Despite being a protocol for accessing X.500 directory services, LDAP's popularity has led to implementations of stand-alone (that is, independent of X.500) LDAP directory servers.

The RSD, as described in the previous section, is implemented using a stand-alone LDAP server provided by the openldap project [1] . We use LDAP because it is a standardized directory protocol that provides all the functionality required for the RSD. By basing the RSD on a standard protocol, all existing LDAP clients can be used to interface with our RSDs. Similarly many libraries for building LDAP applications exist, greatly facilitating the building of custom GIDS clients.

LDAP defines both a network protocol for accessing information in a directory and a data model that defines the form and character of that information. The data model is based on *entries* that contain typed *attributes* and their associated values. An LDAP *schema* defines the classes of available entries and the attributes contained in them. As part of our implementation we have created a schema for GIDS. This schema contains class definitions for the various servers and services (e.g., GOS, location service, naming service, etc.) that an RSD keeps track of. Figure 4 shows a number of these classes and some of their associated attributes. The GLocation and GNaming entries represent standard Globe services and contain information about the network address of their nearest access points. The GOS entry is more complex and contains information about a GOS itself, such as its network address and its operating environment, as well as information about the resources and services. The GIDS schema is extensible, meaning that new attributes can be added to entry definitions without invalidating data stored in the RSD based on older schemas.

class	**attributes**
`GNaming`	`address`
`GLocation`	`address`
`GOS`	`address, os, memory, diskspace, persistence`

Fig. 4. Part of the GIDS LDAP schema

The local RSD operations, as presented in Section 2, map directly onto equivalent LDAP operations. Thus, `add_server` maps onto the LDAP `Add` operation and `delete_server` maps onto LDAP's `Delete` operation. Similarly, `modify` maps onto `Modify` and `search` maps onto `Search`. Unfortunately, there is no direct mapping for the RSD's remote operation `remote_search`. To solve this problem we overload the LDAP `Search` operation and map both the local `search` and remote `remote_search` operations onto it. This means that the

[1] see http://www.openldap.org

RSD interface, as implemented, does not strictly follow the syntax of the interface as presented in Section 2. We felt that this is acceptable given the benefits that using the LDAP protocol brings.

The LDAP `Search` operation uses a *search filter*, which defines the conditions that must be fulfilled for the search to match a given entry. The filter specifies attributes, attribute values, and match criteria for the search. Composite filters can be created by combining other filters using *and*, *or*, and *not* operations. LDAP also defines a string format for a human-readable filter representation [7]. This representation uses a prefix notation for the three logic operations (*and*, *or*, and *not*), and an infix notation for the attribute/value matching criteria. Figure 5 shows an example string representation of a filter where the `language` attribute's value must equal `Java` and the `memory` attribute's value must be greater or equal to `32`.

```
(&(language=Java)(memory>=32))
```

Fig. 5. Example of an LDAP search filter

In order to distinguish between a local and remote search operation we require that the filter for the remote operation always contains a *location* attribute. The value to match for this attribute corresponds to the `location` parameter from the `remote_search` operation as specified in Section 2. Figure 6 shows an example of the search filter for a local and a remote search. The only difference between the two is the presence of a location attribute in the remote search filter.

```
local:
      (&(language=Java)(memory>=32))
remote:
      (&(language=Java)(memory>=32)(location=amsterdam.nl))
```

Fig. 6. Examples of a local and remote LDAP search filter

Because both the local and remote search operations are represented as regular LDAP `Search` operations, an RSD must distinguish between the two before processing search requests any further. This means that the RSD must first scan the search filter to see if a location attribute is present. As this is not a standard function of the LDAP server used, we have implemented the RSD as combination of a preprocessor server and an LDAP server as shown in Figure 7. The preprocessor implements the same operations as the LDAP server and receives all incoming requests, separating remote operations from local ones. Remote operations are performed by the preprocessor itself, while local operations are passed on to the LDAP server.

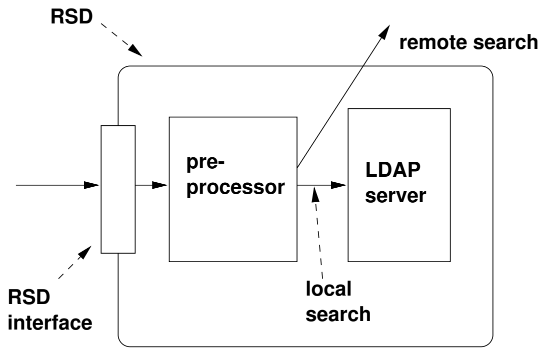


Fig. 7. The RSD with preprocessor and LDAP server

3.2 DNS

Our implementation of the region hierarchy is based on DNS. In this implementation RNSs are implemented as standard DNS name servers, and are arranged in a hierarchy similar to that used in DNS. While domain names in DNS are similar to region names in GIDS, it is important to note that DNS domains do not always represent a notion of proximity. This means that we cannot simply map the region hierarchy onto the existing DNS domain hierarchy. For example, addresses in the *.com* domain do not represent any kind of geographic or network topological proximity. However, where the existing DNS domains do represent proximity (e.g., the top-level country domains), that part of the hierarchy can be used directly.

The DNS servers at the leaves of the hierarchy contain mappings from base region names to corresponding RSD addresses. These mappings are stored in DNS SRV records. An SRV record [5] is a *service* record and is used to specify the location of named network services. This enhances DNS by identifying the specific services provided at a particular domain name, allowing different services to share a logical domain name but be hosted on different physical servers. A particular service is identified by prepending a service and protocol name to a domain name. Thus, an RSD server at the location *vu.amsterdam.nl* would be identified as *gids.tcp.vu.amsterdam.nl* where *gids* identifies the GIDS service and *tcp* the TCP/IP protocol. Storing RSD mappings in SRV records allows us to leverage a large part of the existing DNS infrastructure without disturbing current users of the service.

When an RSD performs a remote search, it acts as a DNS resolver and looks up the given location name. It first tries to look up the location name as though the name represented a base region, by prepending *gids.tcp* to it. If this works, the RSD receives the remote RSD's address and can contact it directly to perform a local search on it. If it fails, however, the RSD performs a zone transfer on the domain server represented by the location name, acquiring a list of domain names managed by that name server. The RSD then repeats the above process on these domain names until it finds all the underlying base regions and their corresponding RSD addresses.

3.3 Example 1: Creating remote replicas

To illustrate how the GIDS is used, we present examples of three practical applications of the system. The first is that of a process that uses the GIDS to find a remote server on which to host a replica. Assuming that this process has already decided (possibly after evaluating an object's past client requests) that it needs to create a new replica at location *amsterdam.nl*, its first step is to compile a list of property requirements for the remote server. It may require a server that, for example, can load GlobeDoc objects written in Java, has at least 32 megabytes of memory, and does not require any authentication. This list of requirements is formulated as an LDAP search filter, and a search query is built based on that filter. Because this will be a remote search request the search filter is extended with a location attribute as shown in Figure 8. When the search filter is ready, the process contacts its local RSD (whose address is already known) and performs an LDAP search query on it (step 1 in Figure 9).

```
(&(objectclass=GOS)(language=Java)(bandwidth>=32)
(authentication=none)(location=amsterdam.nl))
```

Fig. 8. Remote search filter

The preprocessor part of the RSD receives the request and scans the search filter for a location attribute, which it finds. It then extracts the location name and performs a DNS lookup (as previously described) for all the RSD addresses at that location (step 2). This returns a list of remote RSD addresses, in this case those in the base regions *vu.amsterdam.nl* and *uva.amsterdam.nl* (step 3). The local RSD then contacts each of these remote RSDs and performs a search on them (using the original filter with the location attribute removed) (step 4). The results, consisting of the addresses and properties of suitable GOSs, are returned to the process (step 5), where one GOS is chosen (in this case a GOS at *uva.amsterdam.nl*) and a replica created on it (step 6).

3.4 Example 2: Creating shared local replicas

Often, when many clients from the same location access the same Web document, it is preferable to create a local shared copy of that document than to have every client keep its own copy. This sharing of a local replica is the idea behind proxy caches in the Web [1]. Generally, however, using a proxy cache requires clients to explicitly name the proxy that they want to use. In GlobeDoc we allow the creation of shared local replicas, but make the use of

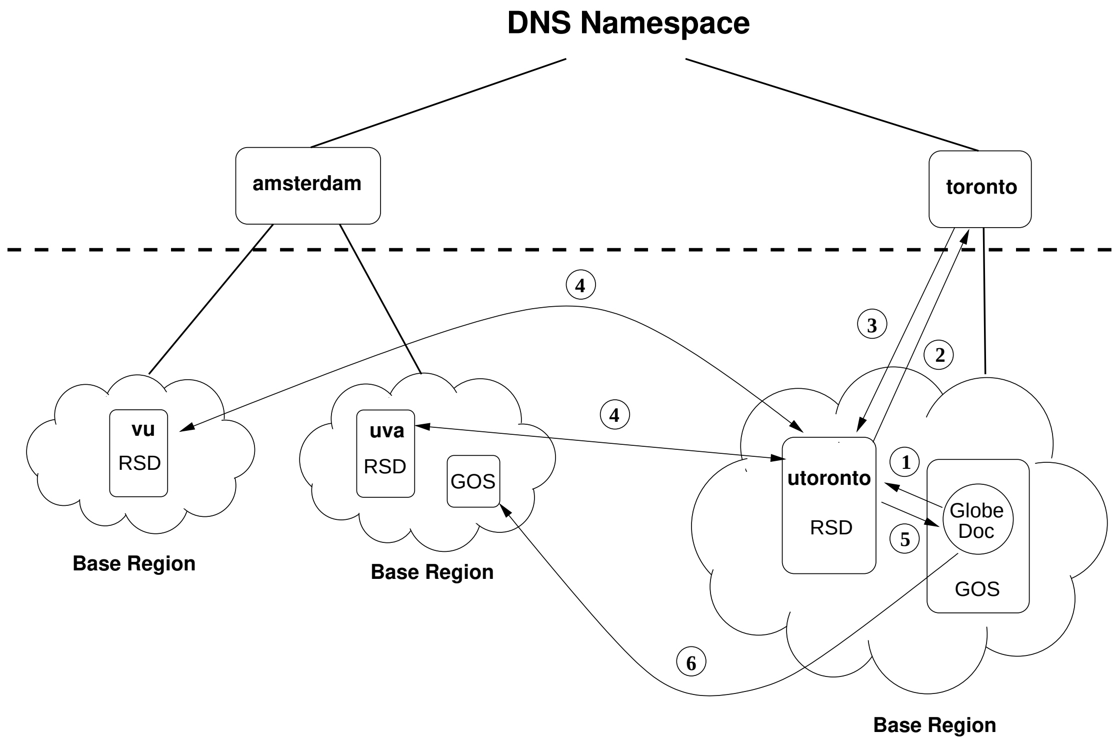


Fig. 9. Example of object searching for remote GOSs

them transparent to the client. That is, a client does not need to explicitly specify that it is going to use a shared replica; moreover, a client will generally not know whether it is using a shared replica or not.

To achieve this transparency, we introduce a **Bind Through Server** (**BTS**). Recall that in GlobeDoc, whenever a client wishes to invoke a distributed object it must first bind to that object. This binding is a three-step process (for a more detailed description of binding than the following see [17]). In the first step the client looks up an object's location in the Globe location service [16]. The result of such a lookup is the contact address of the object's nearest (or only) replica. Like a reference to a remote object in Java, this contact address specifies exactly what that the client needs to install an LR into its address space. In the second step the client contacts a class repository, loads in the necessary code for the specified LR, and instantiates it to create a LR. In the third step the LR actually connects to the rest of the object (e.g., a transport-level address contained in the contact address). The essence of the BTS is that it performs the first step of binding for the client, possibly contacting or creating a local replica as a side-effect. The client simply passes the BTS an object name or reference and the BTS returns a contact address. The BTS determines if there is a local shared replica available to use, or if it should create one. As a consequence, whether the returned contact address is for an existing local replica, a newly created local replica, or a remote replica, is transparent to the client.

When binding using a BTS, a client forwards its bind request transparently to the BTS (step 1 in Figure 10). Note that the request may include details of, for example, required consistency guarantees. The BTS first checks whether a local GOS, which meets the additional binding constraints required by the client, is already bound to the object. If so, the client is returned a contact address for the object at that GOS (step 5) and can continue binding.

However, if no appropriate locally bound replica is found, the BTS attempts to create a local replica. It first tries to find an appropriate object server by performing a *local* search at the local RSD (step 2). If a server is found, it is requested to bind to the object (step 3). The server will then load an LR in its address space and connect to the rest of the object. After completing the binding, the GOS returns a local contact address (step 4) that can be used by the client to complete its own binding (step 5).

If no appropriate local GOS is found, the BTS simply forwards the bind request to the Globe location service. The contact address it receives is returned to the client (step 5), allowing the client to proceed with binding as usual.

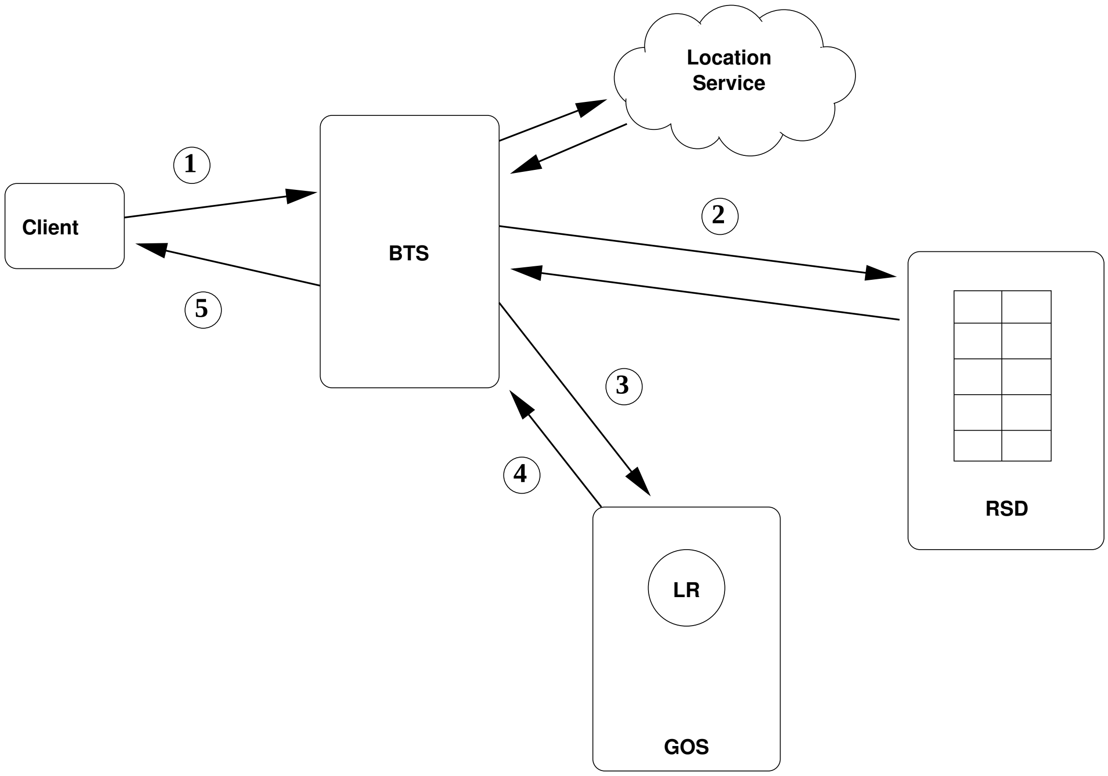


Fig. 10. Creating a shared local replica

The GIDS may also be used to supply local configuration information to GlobeDoc clients and GOSs. They use this information to initialize the runtime system (RTS), which is responsible for finding and binding to other GlobeDoc objects.

This is a rather straightforward example as the runtime system simply contacts its local RSD (whose address it received as a parameter or from a configuration file) and requests the addresses of the services it requires. These addresses are then stored by the run-time system for later use. Alternatively, to ensure that it always has the corect address, an RTS can periodically perform lookups on the local RSD or perform them every time it needs to access one of the services.

4 Related work

Different approaches have been proposed and implemented to keep track of distributed resources. One of the simplest approaches is manually distributing files containing resource information to interested parties. As resource information changes, new copies of the files must be redistributed to those parties. This approach works well for systems were few copies must be made and resource information rarely changes. However, when the number of interested parties increases and becomes more dispersed, or the resource information becomes more dynamic, such a manual approach becomes infeasible. Systems such as NIS [15] attempt to overcome this problem by automatically distributing these files, however, their scalability is generally limited to local networks. DNS provides a highly scalable service for mapping domain names to Internet addresses, however, it is not designed to contain general directory information nor to provide the flexibility required for a general resource directory (i.e. it does not support attribute-based naming [2]).

X.500 [13] is a standard that defines a wide-area, extensible directory service. Although X.500 provides a suitable directory service model, it is complex and requires the ISO protocol stack. LDAP (the Lightweight Directory Access Protocol) is a lightweight version of the X.500 directory service. It does not require the ISO protocol stack, but defines a protocol based on the IP protocol suite. Data encoding is also simplified and a standard API for directory access is defined. Because it can be used over IP, LDAP is becoming the directory service of choice for the World Wide Web. LDAP provides the capability to create distributed hierarchies of the directory data, however, this capability does not scale to the extent needed for GIDS. The reason for this is that LDAP places a directory server at every node in a hierarchy, which makes it a more heavyweight solution than our approach of using DNS for finding location-bound directory servers. GIDS does not benefit from the extra directory servers in the hierarchy because it does not store any information about the hierarchy. As such, the extra servers provide unnecessary overhead.

Because of its nature as a directory service and its implementation using LDAP, GIDS bears a strong resemblance to both Microsoft's Active Directory [9] and the Globus project's Metacomputing Directory Service (MDS) [3]. It is important to stress that although all three (GIDS, Active Directory, MDS) are based on the same technologies, and therefore have many resemblances, each is tailored to best suit its own particular environment, and neither renders the other obsolete.

Active Directory, in addition to being based on LDAP, also makes use of DNS to organize its directory servers into larger groups or domains. The main difference between GIDS and Active Directory is between the role of domains in Active Directory and base regions in GIDS. In Active Directory organizations define hierarchical domains where each node in the hierarchy is represented by an LDAP directory server. Given an operation on a domain, DNS is used to find the address of the directory server on which the operation is to be performed. Whereas in Active Directory directory servers can represent internal nodes of the hierarchy, in GIDS the directory servers are limited to the leaves of the hierarchy. The reason for this difference is that, being primarily a system for storing administrative information, Active Directory stores information about the domains and subdomains themselves, as well as information about the servers in those domains. In GIDS, however, we are interested only in storing information about actual servers and services and these are present only in the leaves. Thus, in Active Directory the domain hierarchy is part of the directory structure, while in GIDS the location hierarchy is simply a mechanism for finding directory servers that is much more efficient to implement. Active Directory also provides a domain-wide index called the Global Catalog, which provides a quick way to find information in a domain. This index is a centralized summary of the information found in the various directory servers in a domain. Such a centralized index would not be feasible in GIDS because GIDS's location hierarchy is meant to cover a much larger (both logical and physical) area and contains a greater number of directory servers.

Unlike Active Directory, which is primarily an administrative system, Globus MDS is similar to GIDS in that both are resource directories; they provide information about available resources in distributed systems. MDS is a resource directory service for metacomputing environments. As such, one of its properties is that the data maintained by it are highly dynamic, which leads to the requirement that data are made available in a timely fashion. Similarly, the data returned by the system may come from multiple sources and might even be generated on the fly (as opposed to simply being stored in a database on the LDAP server). Because the GIDS is used in an environment where the server data are less dynamic and timeliness is a less important issue, we do not share these requirements with MDS. Similarly, the data stored in the MDS are of a much finer grain than that needed for the GIDS. MDS also relies on the distribution capabilities of the LDAP protocol to create distributed hierarchies of the resource data. As mentioned above, we feel that the distribution provided by LDAP does not scale to the extent needed by GIDS.

Another system that is similar to GIDS is Jini [20], an environment that allows Java programs to provide and use remote services. The Jini infrastructure provides mechanisms for devices, services, and users to join a network, detach from a network, and find each other in that network without the need for any manual administration. A main component of this infrastructure is the lookup service. This is a service that, like the RSD in GIDS, stores and publishes information about the services available on a network. Unlike the RSD, the lookup server in Jini is not based on LDAP, but on associative memory technology similar to JavaSpaces [4]. Although an associative memory approach is better suited to distribution (for example, through the use of hashing [14]), Jini is targeted for smaller-scale networks (at the workgroup level) and as such, wide-area scalability is not a goal in the Jini architecture. Jini, being an extension of the Java architecture, is also heavily dependent on Java and Java Remote Method Invocation (RMI).

5 Evaluation

One of the requirements for GIDS was that it must be easy to search through it given a set of resource and service requirements. GIDS is meant to be used by objects looking to find servers that can host their replicas. Because these objects know where to place their replicas, location plays an important role in the search process. As such, GIDS is designed as a location-aware resource management system. In contrast, most (general-purpose) resource management systems, such as MDS, are location unaware. That is, given characteristics of a service, they attempt to find any matching services, regardless of their location. In fact, such systems are often specifically used to find the locations of desired services. GIDS is not a general-purpose resource management system.

In practice this means that GIDS works well when searching for services whose (approximate) location is known. It does not, however, work well when performing global searches for services whose location is not known. Such searches require a search of the whole region (DNS) hierarchy, which is a highly inefficient operation. Note that attribute-based searches in large areas are inherently expensive.

GIDS is designed as a system for managing both global and local resources. Using GIDS for local resource management is equivalent to using it for global resource management, except that the location involved is implicit.

In the Globe environment, GIDS is also used by services as a means for finding configuration information. This makes the job of managing a local Globe environment easier as there are considerably less configuration files to be maintained. However, using GIDS in this way, we found that the limitation of storing data only in base regions prevented us from easily sharing information between base regions. For example, currently, when object servers in multiple base regions share an external service, copies of that service's address details must be stored separately in the RSDs of each of these base regions. When this information changes, modifications must be made to each RSD individually. Ideally this information would be stored at a higher level in the region hierarchy, where it could be shared by multiple base regions. Storing information in the hierarchy in this way resembles the approach taken in directory services (such as Active Directory) used for administration.

One thing the GIDS lacks is an RSD discovery protocol. Such a protocol would allow any GIDS-enabled client to find its nearest RSD and in doing so automatically determine its base region. RSD discovery would allow local services to startup, find a local RSD, and fully configure themselves without any interference from an administrator. Various local-area discovery protocols [11] [6] exist that we can readily use for this purpose.

6 Conclusion

In this paper we have described the architecture of the Globe Infrastructure Directory Service. GIDS is a system for collecting and organizing information about resources and services as provided by a globally distributed pool of servers. It is designed as a collection of directory servers that are grouped into regions based on a location hierarchy. The hierarchy defines a location representation (shared by both clients and servers) and reflects a given network or organizational topology. We have implemented this system using LDAP and DNS.

With the design of the GIDS we have been careful to make a strict separation between policy and mechanism. Where possible, the architecture defines mechanisms for implementing various policies, but does not choose any one policy over another. As such, we have not presented a definite list of properties that a GOS may have, nor have we defined a concrete location naming scheme (i.e., whether the location hierarchy should be geographic, organizational, or some mixture of both). In our implementation of the GIDS we have, however, defined some of these policies. For example, in the LDAP schema we define exactly which GOS properties are stored in the directory.

The system has been designed in such a way that it is independent of Globe and GlobeDoc. Because GOSs are not particular to GlobeDoc, GIDS can be used by any Globe application that requires its functionality. Also, by replacing GOSs with non-Globe servers that offer similar functionality (i.e., provide information about their resources and properties), GIDS can be employed in the current Web without having to switch to GlobeDoc. For example, GIDS can be used as the basis for an open content delivery network based on current Internet technology. In this case, GIDS would function as a directory of all Web servers willing and available to host mirrors of other websites in a particular region.

Future work for GIDS will include looking at appropriate policies with regards to location hierarchy, authorization, and authentication. We will also look at the possibilities for GOSs to provide (and possibly negotiate) guarantees or contracts with regards to quality of service provided (e.g., guaranteed uptime). Also interesting is the possibility of blocking or filtering unwanted content from being served by a given GOS, and for providing object and data integrity guarantees.

References

[1] G. Barish and K. Obraczka. "World Wide Web Caching: Trends and Techniques." *IEEE Commun. Mag.*, 38(5):178–184, May 2000.

[2] M. Bowman, L. Peterson, and A. Yeatts. "Univers: An Attribute-Based Name Server." *Software – Practice & Experience*, 20(4):403–424, 1990.

[3] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. "A Directory Service for Configuring High-Performance Distributed Computations." In *Proc. Sixth IEEE Symp. on High-Performance Distributed computing*, pp. 125–136, Portland, Oregon, Aug. 1997.

[4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice.* Addison-Wesley, Reading, MA., 1999.

[5] A. Gulbrandsen and P. Vixie. *A DNS RR for specifying the location of services (DNS SRV)*, Feb. 2000. RFC 2728.

[6] E. Guttman, C. Perkins, J. Veizades, and M. Day. *Service Location Protocol, Version 2*, June 1999. RFC 2165.

[7] T. Howes. *The String Representation of LDAP Search Filters*, Dec. 1997. RFC 2254.

[8] A. M. Kermarrec, I. Kuz, M. van Steen, and A. S. Tanenbaum. "A Framework for Consistent, Replicated Web Objects." In *Proc. 18th Int'l Conf. on Distributed Computing Systems*, Amsterdam, the Netherlands, May 1998.

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

2000.

[10] P. Mockapetris. *Domain Names–Concepts and Facilities*, Nov. 1987. RFC 1034.

[11] C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*, Nov. 1993. RFC 1546.

[12] G. Pierre, I. Kuz, M. van Steen, and A. Tanenbaum. “Differentiated Strategies for Replicating Web Documents.” *Computer Communications*, 24(2):232–240, Feb. 2001.

[13] S. Radicati. *X.500 Directory Services: Technology and Deployment.* International Thomson Computer Press, London, 1994.

[14] A. Rowstron. “Run-time Systems for Coordination.” In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, pp. 78–96. Springer Verlag, Berlin, Aug. 2001.

[15] H. Stern. *Managing NFS and NIS.* O'Reilly, Sebastopol, CA, 1991.

[16] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbau m. “Algorithmic Design of the Globe Wide-Area Location Service.” *The Computer Journal*, 41(5):297–310, 1998.

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

[18] 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.

[19] M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*, Dec. 1997. RFC 2251.

[20] J. Waldo. *The Jini Specifications.* Prentice Hall, Upper Saddle River, N.J., 2nd edition, 2000.