THE GLOBE DISTRIBUTED SYSTEM

Philip Homburg
Maarten van Steen
Andrew S. Tanenbaum

1. INTRODUCTION

In my first lecture, I gave a general introduction to distributed systems. In my second lecture I discussed Amoeba, a distributed system intended for use on a local area network. In this lecture I will discuss another system I have helped design, GLOBE. In contrast to Amoeba, GLOBE is intended as the ''operating system'' for a worldwide distributed system (Homburg et al., 1996; Van Steen et al., 1995). As a consequence, a number of quite different issues arise. As a simple example, resources are located in Amoeba by broadcasting a request. In a worldwide system, such an approach will not work, so a completely different design is needed.

In the 1960s and 1970s, the computing universe was dominated by mainframes and minicomputers that ran batch and timesharing operating systems. Typical examples of these systems were OS/360 and UNIX. These system were primarily concerned with the efficient and secure sharing of the resources of a single machine among many competing users.

In the 1980s, personal computers became popular. These machines had different kinds of operating system, such as MS-DOS and Windows, and were primarily concerned with providing a good interactive environment for a single user. Resource allocation was not a major issue.

As we move closer to the year 2000, it is becoming clearer that a new environment is appearing, one dominated by millions of machines interacting over wide area networks such as the current Internet. While all these machines have individual local operating systems, the worldwide system as a whole raises many of the same issues that are found in local operating systems, including interprocess communication, naming, storage, replication, and management of files and other kinds of data, resource management, deadlock prevention, fault tolerance, and so on. Only instead of being performed on a single machine, these issues arise in the large.

In effect, we need an ''operating system'' for this worldwide system. This operating system will of necessity be different from existing operating systems in that it will run in user mode on top of existing operating systems (as do the servers in many modern microkernel-based operating systems). Nevertheless, it will have to perform traditional operating system functions on a huge scale.

At the Vrije Universiteit, we are developing such an operating system, GLOBE, for the worldwide computing environment of the future. In this paper we will describe its architecture.

2. THE STATE-OF-THE-ART IN WIDE-AREA SYSTEMS

Current wide-area systems tend to lack a common model. For example, the World Wide Web, email, news, and FTP all provide ways of communicating, but they are all based on different conceptual models. For example, each one has its own naming scheme (URLs, DNS names, newsgroup names, and host+path names). Mail and news are ''push-driven'' (the sender actively transmits the data), whereas FTP and WWW are ''pull-driven'' (the receiver goes and gets the data from a remote site). They also differ in terms of replication strategy, encoding, containers, and authentication, although there is no real reason for all these differences to exist.

In contrast, there are other systems that have a more unified model. For example, in FORTRAN 4, all external data sets are modeled as tapes. In UNIX, everything is a file. In the WWW, everything is a document. In many distributed shared memory systems, everything is a page, and so on. What we need for worldwide computing is a single unifying model, like these.

3. THE GLOBE PARADIGM

GLOBE is a layer on top of existing operating systems that effectively functions as a kind of worldwide operating system that presents a uniform abstraction to applications and users. GLOBE is based completely on objects in the sense that all the important items in the system are modeled as objects. Typical objects include machines, web pages, mailboxes, news articles, etc. Associated with

each object is a set of methods that authorized users can invoke without regard to the relative location of the user and the object.

GLOBE objects are unusual in that they are distributed shared objects What this means is that an object can be mapped into the address spaces of many different machines at the same time, even though these machines may be spread all over the world. Any process holding the object can invoke its methods. How the objects are internally organized is up to each object's designer.

Several models are possible. In the simplest case, one instance of an object holds all the state, and the others are just proxies that forward all invocation requests to the primary object to get work done. A second model has the state being fully replicated on all sites, so methods that only read the data can be done locally. Methods that change the data require some kind of (object-specific) internal synchronization protocol, for example two-phase commit. Yet another model is to partition the data over the object copies. Many other models are possible. Since the data replication is hidden inside each object, different objects can, and generally will, use different strategies, depending on what is appropriate.

Communication in GLOBE is quite different from that in other systems. Most systems, especially wide-area ones, are based on messaging. Typically two processes that want to communicate first established a (TCP or ATM) connection, then pump bits through the connections. In other systems a datagram-style of communication is used (e.g., UDP packets), but here, too, the basis is still message passing.

GLOBE works differently. Here processes do not send messages to communicate. Instead they communicate using the GLOBE distributed shared object mechanism. Both (or all) interested parties first bind to some common distributed shared object, then perform operations on it. For example, to send email, a process might bind to the recipients mailbox and then invoke an INSERT ITEM method. To read mail, the receiver would invoke a GET ITEM method. While the object would have to use physical communication internally to move the bits, the users would just deal with objects, and not with sockets, TCP connections, and other message-oriented primitives. We believe this model allows us to unify many currently distinct services. For example, GET ITEM could be used to read mail, read a news article, read a web page, or read a remote FTPable file in exactly the same way: in all cases, the user just invokes a method on a local object.

In addition, this model isolates programmers and users from issues such as the location and replication of data. In a communication-oriented system, users have to know how the data are replicated and where the copies are. In our model, the user of an object does not have to know how many copies there are or where they are. The object itself manages the replication transparently. To see the implications of this design, think about accessing a copy of a replicated web page or FTP file. In both cases the user must make a conscious choice about which copy to access, since different copies have different URLs or DNS names, respectively. In GLOBE, each object has a single unique name that users deal with. Managing the replicated data is done inside the object, not outside, so users are presented with a higher level of abstraction than is currently the case. This higher level of abstraction makes it much easier to design new worldwide applications.

4. OBJECTS IN GLOBE

The smallest unit in GLOBE is the local object. It resides in a single address space and can only be invoked there. Objects in GLOBE exist at runtime and occupy space in memory, that is, they are not merely compile-time abstractions that the compiler optimizes away.

Each object belongs to a *class*. A class defines the methods that can be invoked on objects of its class and contains (at runtime), the actual code for doing the work. The distinction between a class object and a regular object is that class objects are primarily there to generate new objects of the appropriate class. Thus, every class object has a method GENERATE NEW OBJECT, which results in a new object being created and a pointer to it returned to the caller.

GLOBE objects can contain state information. A web object would normally contain one or more web pages that it made available to users. A bank account object would contain information about some bank account, and so on. Once an object is created using GENERATE NEW OBJECT, it can load state into itself from some source and generally evolve on its own independent of other objects.

GLOBE objects are passive; they do not contain any active elements. In the GLOBE model, each process has an address space into which some objects are mapped, but the active elements are threads. Threads invoke methods on objects. Threads, processes, and address spaces are managed by the local operating system and fall outside the scope of GLOBE itself.

Every GLOBE object has one or more *interfaces*. An interface is a list of (method, state) pointer pairs. The first one points to the code to be executed and the second one points to the state. Interfaces provide threads the necessary access to objects. Each object exports certain methods, and these are contained in its interface table. The only methods that can be executed are those present in the interface table.

Objects can have multiple interfaces. For example, the primary interface may be the one most users use to get work done, but there may be a secondary one as well for the system administrator to configure and manage the object.

Individual objects can be collected into *composite objects*. A composite object contains multiple objects inside of itself, but exports one of more interfaces to the outside world. The methods in the exported interfaces are a subset of the interfaces of the internal objects. In this way, small objects can be aggregated into large ones, and so on, recursively.

5. DISTRIBUTED OBJECTS

Objects in GLOBE can be distributed. In fact, the whole basis of GLOBE is distributed shared objects. A distributed shared object can be present in the address space of multiple processes on distant machines at the same time. The processes then communicate by invoking methods on their distributed shared objects. Changes to the state made by one process are observed by other processes. There is no a priori distinction between clients and servers.

Before a process can invoke an operation of a distributed shared object, it has to *bind* to that object (at runtime). This means that an interface and its implementation are loaded into the process' address space. Multiple processes share an object by being bound to it simultaneously.

To simplify the implementation of new distributed shared objects we propose an implementation that consists of four local subobjects: a communication object for handling low-level communication with other instances of the object, a replication object dealing with state distribution, a semantics object providing the actual semantics of the distributed shared object, and a control object handling local concurrency control.

Communication objects provide a simple interface to send data to and receive data from other address spaces. A communication object can support connection oriented or connectionless communication and point-to-point or multicast data transfers. A replication object is responsible for keeping the state of the distributed shared object consistent. It sends marshaled copies of (parts of) the object's state, and marshaled arguments and return values of operations to other address spaces.

Communication and replication objects are generally written by system programmers and are available from libraries or class repositories. Application programmers implement the semantics objects needed for an application. A control object is generated based on the interface of the corresponding semantics object. We expect that the system comes with standard semantics objects like files and directories. Furthermore, some distributed shared objects may require replication or communication objects that are optimized to work with a specific semantics object.

A distributed shared object may replicate its state for reasons of fault-tolerance, availability, or performance. It may also partition its state, but that will not be discussed further. For each address space bound to a distributed shared object, the object decides whether to store a (full) copy of its state in that address space or not. When a process binds to an object, the object may decide to create a replica in that address space and control the consistency through update or invalidate messages. This choice optimizes the performance of read operations, which can then be executed locally. Special (server) processes can be run at strategic sites. These processes can bind to an object to create a replica. These replicas are needed to optimize read access to an object (caches) and to provide persistence and fault tolerance.

6. NAMING

Our architecture includes a naming system that maps a human readable pathname to a set of contact addresses for a distributed shared object. The contact addresses provide independent ways to bind to a distributed shared object. Typically, a replicated object provides multiple contact addresses for availability and improved performance. The set of contact addresses can change over time, and is maintained by the object itself. A replicated object typically adds contact addresses when new replicas are installed, and a mobile object creates a new contact address when it moves to a different location. At the same time, users can create and delete pathnames that refer to the object.

In our architecture, object naming is split into two services. The first part, the *name service*, maps a pathname to an *object handle*. This object handle is passed to the second part, the *location service*, which maps it to the set of addresses. This scheme allows a user to create multiple names that resolve to the same object handle, and allows the object to update the set of contact addresses that the object handle refers to. Logically, the mapping from object handle to contact addresses is stored in one place in the location service. The fact that the location service can replicate location information is hidden from the object.

An object handle consists of an object identifier and some additional information for the location service. An object identifier is worldwide unique and is never reused after the object is destroyed. Object identifiers are used during binding as an end-to-end check for the location service. The additional location information which is part of an object handle is allocated by the location service when the object registers its initial set of contact addresses.

The name service can be implemented as a worldwide hierarchical collection of directories. The feasibility of this approach has been shown by DNS. With worldwide location services there is less experience. A straightforward approach is to divide objects into different categories and use different algorithms for each category. For example, many objects will be located on a single site. These objects may have a few replicas, but the replicas are not far apart. We expect that the majority of the objects fall in this category. Traditional solutions will suffice here. A second category are highly replicated objects. For example objects that encapsulate Usenet news articles or news groups. Since replicas of such an object are stored at many sites, we expect those sites to cooperate in maintaining location information for the object. We also have to support mobile objects, that move from one place to another and require frequent updates of their location information.

The location information of these three categories of objects can be stored in three independent, worldwide location servers. A few bits in the object handle can be used to specify the location service used. A disadvantage of this approach is that an object can not move to a different category without changing its object handle.

We have designed a location service that is able to dynamically adapt to the migration patterns of an object. A location service provides four operations:

- Register a new object
- Unregister a deleted object
- Change the set of contact addresses for an object
- Lookup the set of contact addresses for an object

The goal of the location service is to implement all four of these operations efficiently. This means optimizing the placement of the information needed to map an object handle to a set contact addresses such that updates to the set of contact addresses are cheap (i.e. require a small amount of (local) communication) and that lookups are also cheap. Since updates to the set of contact addresses are initiated by the object itself, it makes sense to store the mapping in a part of the location service that is near the object. Note that a replicated object is not located at a single site. Furthermore, objects that migrate require that the location of the mapping follows the object.

Our design of the location service uses a worldwide, distributed search tree. To build this search tree, we divide the world into a hierarchical set of domains. At the bottom we have one domain per site. A collection of sites form a region. An object is registered at each site where it has a contact address. Furthermore, each object is registered in all regions that contain a site where the object is registered, recursively up to the top of the tree.

Initially, registrations at the site level contain the actual contact addresses and registrations at region or world level contain pointers to the next lower level. Registering at multiple levels allow searches with expanding rings: a search starts at the local site, followed by the local region, then the next higher level region, etc., and eventually followed by the root. Searching with expanding rings provides the desired locality. If the object has a contact address in the site or region of the requesting process, then contact addresses will be found with only local communication.

To be able to actually implement regional and world nodes, we have to partition them. Each site stores a part of the root node and parts of the regional nodes. The partioning can be done by computing a hash value from the object handle.

A key element that makes the search process work efficiently is the fact that contact records are stored at the lowest point in the tree that covers all the locations the object normally moves to. For example, if an object tends to wander around Amsterdam, but not outside Amsterdam, its contact record will be stored at the Amsterdam level of the tree. If, however, the object tends to wander around all of Holland, its contact record will be stored at the Dutch national server level. Finally, if the object constantly wanders around Europe, its contact record will be stored at the European level. As a result of this design, even though an object may move a lot, its contact record is stable. This fact means that once a remote user has looked up the contact record for an object, it can cache a pointer to where it found the record, since the record is likely to be stable, even though the object itself may be highly mobile within a certain range.

Homburg, P.C., Van Steen, M.R., and Tanenbaum, A.S.: "An Architecture for a Wide Area Distributed System," *Proc. Seventh ACM SIGOPS European Workshop*, 1996.

Van Steen, M., Homburg, P., Van Doorn, L., Tanenbaum, A.S., and De Jonge, W.: "Towards Object-based Wide Area Distributed Systems," *Proc. Fourth Int'l Workshop on Object Orientation in Oper. Systems*, IEEE, pp. 224-227, 1995.