From Remote Objects to Physically Distributed Objects

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

Abstract

Present-day object-oriented middleware provides little support for the distribution, replication and caching of the state of a distributed object. This makes these platforms unsuitable for the development of large-scale distributed applications. We argue that the model of distributed objects on which these middleware platforms are based hinders the addition of comprehensive distribution and replication support to these platforms. We present an alternative view of distributed objects, in which objects are not only in control of the functional aspects of their implementation but also in control of their nonfunctional aspects, in particular, the distribution and replication of their state. We claim that a middleware platform based on this view of distributed objects is better suited for developing the large-scale applications of the future.

1. Introduction

In the not-so-distant future the Internet will grow to a network connecting hundreds of millions of people all over the world, maybe even a billion. To keep this network from a permanent state of congestion, network services and applications will need to make heavy use of replication and caching techniques [4]. Unfortunately, current middleware platforms provide little or no support for these techniques, making them unsuitable for the development of large-scale distributed applications.

In this paper we argue that the present models of distributed objects on which these middleware platforms are based are not fit for dealing with the problems of large-scale systems. We claim that these models will prevent platforms from providing comprehensive support for inherently large systems through distribution, replication and caching. We present an alternative way of looking at distributed objects and argue that it is a better basis for supporting replication and caching of the state of objects.

2. The legacy of RPCs: CORBA and DCOM

Two middleware platforms are currently popular: CORBA [7] and DCOM [1]. A distributed object as defined in CORBA is an object running on a single machine, presented to remote clients as a local object by means of proxies. An Object Request Broker mediates between the clients and the object and, in particular, takes care of the transport of requests and replies from the client to the object over the network. CORBA currently has little support for the replication of objects.

The implication of this remote-object view is that distribution and replication of a distributed object are managed by the Object Request Broker. This implies, in turn, that the choice of the application programmer with respect to these nonfunctional aspects is limited by what the Object Request Broker (or ORB Service) offers. For example, an ORB might offer only active replication protocols and no primary-backup solutions.

The CORBA model does not easily allow the introduction of object-specific replication and distribution protocols. CORBA *Interceptors*, small pieces of software that can be introduced in the invocation (and response) path from client to object, are an improvement, but they are currently not properly worked out [6]. We argue that taking a different view of distributed objects allows object-specific policies for nonfunctional aspects in a more comprehensive way.

The other popular middleware platform is DCOM [1]. DCOM is the combination of Microsoft's COM with remote procedure calls following the DCE standard [8], to allow clients to interact with COM components on other machines.

The choice for DCE RPCs makes DCOM basically a client/single server system. The custom marshalling feature of DCOM allows application programmers to write their own proxies and server-side skeletons. This feature can be used to implement object-specific replication and security protocols. However, implementation is left entirely to the application programmer.

3. A different view of distributed objects

The Globe distributed system [10] is a middleware platform specifically designed for developing large-scale distributed applications. In Globe, processes communicate by invoking methods on a special kind of distributed object, called a *distributed shared object* (DSO). The distributed shared object is the unifying concept in the system. It provides a uniform representation of both information and services and implementation flexibility by decoupling interface and implementation.

3.1. Physically distributed objects

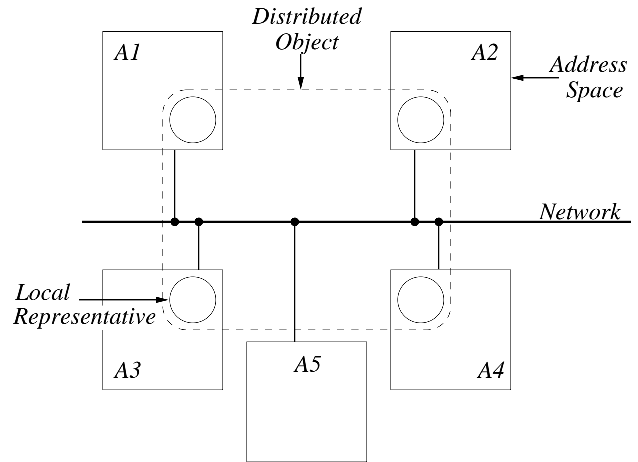
The fundamental idea behind the design of the distributed shared object is that it is, what we call, *physically distributed*. Instead of viewing a distributed object as an entity running on a single machine, possibly with copies on other machines, we view a distributed shared object as a conceptual object, distributed over multiple machines with its *local representatives* (proxies and replicas) cooperating to provide the single (consistent) image. In other words, a distributed shared object is a wrapper encompassing all the object's proxies and replicas, rather than a remotely accessible object implementation. This view is illustrated in Figure 1a.

This different view of what a distributed object is gives us flexibility with respect to replication, caching and distribution of the object's state. In this view, a distributed shared object encapsulates its own replication and distribution strategy. The local representatives of an object take care of the replication and distribution of the DSO's state and all necessary communication. Only minimal (protocol independent) support is required from the run-time system.

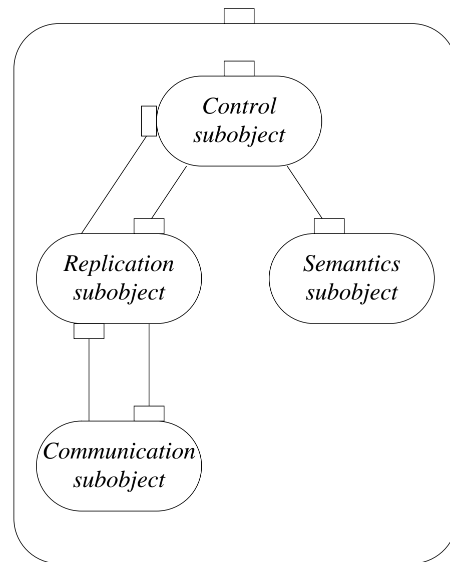
The way the state of the object is replicated can now be governed completely by object- and application-specific requirements with respect to consistency and nonfunctional aspects, such as security, and is under no restriction from the supporting middleware platform. However, we do not leave everything to the application programmer. The structure of local representatives, described below, separates replication and communication code. This means that a programmer can write his or her own replication protocol based on existing communication protocols. Furthermore, we provide the application programmer with implementations of frequently used replication protocols.

3.2. Implementation of the Globe object model

A local representative resides in a single address space and communicates with local representatives in other address spaces. Each local representative is composed of sev-



(a)



(b)

Figure 1. (a) A distributed shared object (DSO) distributed over four address spaces (A1-A4). In each address space the DSO is represented by a local representative. Address space A5 does not currently contribute to the distributed shared object. (b) A local representative is composed of a number of subobjects. The exact composition depends on the role the local representative plays in the distributed shared object.

eral subobjects as shown in Figure 1b. A typical composition consists of the following four subobjects.

Semantics subobject: This is a local object that implements (part of) the actual semantics of the distributed object. As such, it encapsulates the functionality of the distributed object. The semantics object consists of user-defined primitive objects written in programming languages such as Java, C or C++. These primitive objects can be developed independent of any distribution or replication issues.

Communication subobject: This is generally a system-provided subobject (i.e., taken from a library). It is responsible for handling communication between parts of the distributed object that reside in different address spaces, usually on different machines. Depending on what is needed from the other components, a communication subobject may offer primitives for point–to–point communication, multicast facilities, or both.

Replication subobject: The global state of the distributed object is made up of the state of semantics subobjects in its local representatives. A DSO may have semantics subobjects in multiple local representatives for reasons of fault tolerance or performance. In particular, the replication subobject is responsible for keeping the state of these replicas consistent according to some (per-object) coherence strategy. Different distributed objects may have different replication subobjects, using different replication algorithms. An important observation is that the replication subobject has standard interfaces.

Control subobject: The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. This subobject is needed to bridge the gap between the user-defined interfaces of the semantics subobject, and the standard interfaces of the replication subobject.

A key role, of course, is reserved for the replication subobject. Replication (and communication) subobjects are unaware of the methods and state of the semantics subobject. Instead, both the replication subobject and the communication subobject operate only on opaque invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication and communication subobjects. This approach is comparable to techniques applied in reflective object-oriented programming [2].

3.3. Binding to a distributed shared object

To access a distributed shared object, a client first needs to install a local representative of the object in its address space. The process of installing a local representative in an address space is called *binding*. An important property of Globe is that each DSO is identified by a worldwide unique, location-independent object identifier (OID). During binding, this OID is mapped to one or more contact addresses, describing where (network address, port number) and how (which replication and communication protocol) the distributed shared object can be contacted. This information is then used by the local run-time system to create a new local representative in the client's address space and integrate the new representative into the DSO.

We make the assumption that an OID never changes, but that an object's contact addresses may change regularly. For this reason, we cannot make use of traditional naming services such as DNS, to look up a contact address. Although we have developed an efficient *location service* for tracking distributed shared objects [9], binding to an object will always require looking up a contact address, making it a relatively expensive operation.

We stress that the performance of binding would not be a problem if contact addresses would never change. This is the reason why look-up operations in the World Wide Web perform reasonably well. However, even the Web demonstrates that contact addresses (i.e., URLs), *do* change. In Globe, we do not make simplifying assumptions about the mobility of objects, as we believe that such assumptions have no place in the next generation of distributed systems.

4. Granularity of distributed shared objects

A distributed application in which caching and replication play an important role is the World Wide Web. This makes it an important area of research for us.

Consider, for example, a moderate-size Web site consisting of hundreds of Web pages, along with the images, animations, etc. that go with these pages. Like most Web sites, some Web pages will be popular while others are hardly ever accessed. Likewise, this site will contain pages that are hardly ever changed, and pages that are changed every day. There are several ways that such a Web site can be modeled in Globe.

One approach is to take each page, along with its images and such, and turn it into a Globe distributed shared object. This offers a fine-grained approach to distribution, as we are able to associate a separate distribution strategy with each page. For example, pages that enjoy large popularity could be replicated using a copy-invalidate scheme, reducing the average download time because their contents are closer to the viewer. Less popular pages could possibly do without replication. Although in theory this approach works fine, it would also mean that a client (i.e., a Globe-enabled Web browser) would have to go through the entire binding process for each page, since each page would be a distributed shared object. Given the current cost of contact address look ups, we deem this approach as yet to be too expensive.

An alternative is to model the entire site as a single Globe

distributed shared object. However, it is clear that this approach can work only for small sites consisting of a few Web pages. The main problem is that having a single distribution strategy for *all* pages of the site is not a good idea. For example, if 90% of all accesses involve only a few pages, then it does not make sense to use a master-slave replication strategy for all pages. On the other hand, the advantage of the approach is that only a single binding step is needed to access the site and all its pages. Getting access to the Web site object is thus cheap.

The solution, of course, lies in the middle: a large site consisting of hundreds of pages should be split into smaller units, where each unit has its own distribution strategy. In principle, each unit could be represented by a distributed shared object. The problem with this approach is that we are grouping pages into objects based on distribution strategy. As a consequence, the objects we create may not be meaningful at the functional level, because the only thing that relates the pages in those objects, is that they should be distributed in the same way.

It is possible that the objects we create correspond to logical relationships (e.g., a group of pages that should be replicated the same way might turn out to be the group of pages describing a certain product line). However, in general, this will not be the case and therefore this approach violates the basic principles of good (object-oriented) software engineering, notably separation of functional and nonfunctional aspects.

The fundamental problem to be solved here is that we want to differentiate between data elements according to nonfunctional requirements (i.e., apply different distribution strategies to different data elements) at low cost, without creating relationships between those elements at the functional level (i.e., group logically unrelated data elements into the same object). To illustrate this point, consider the example of a company's Web site where the most popular pages are the product and support pages. For performance reasons it would make sense to put the product and support pages in a single DSO with an optimized replication strategy. However, logically products and support are different entities and should be kept separate.

We describe how this problem can be solved using the distributed shared object concept in the next section. The approach in which logically related pages are grouped into distributed shared objects, that is, where the DSOs are logical units, is currently being investigated for feasibility by our project team [11].

5. Containers and clusters

Our solution is to use a clustering mechanism. Clustering has traditionally been applied to object-oriented databases to physically group related objects close together so that they may be efficiently retrieved [3]. In our case, we use clusters to group elements according to a common distribution strategy. An important aspect of our solution is that elements are no longer only pieces of data, but are turned into objects encapsulating their own implementation. This both facilitates the implementation of our clustering mechanism and adds expressiveness to the system. In our Web example, an element would be a Web page with all its HTML text, images, etc., modeled as an object containing a set of files with methods to add, retrieve, update and remove these files.

In our new model, elements are held in a *container*. Each element has a worldwide unique identifier. An element can be held in only one container. Within the container, elements are grouped into clusters, where each cluster supports a single distribution strategy. As an initial approach, we do not allow an element to be moved from one cluster to another without changing its (object) identifier. As before, a client is first required to bind to an element, resulting in the installation of a special local representative supporting the element's methods in the client's address space, analogous to binding in our original object model.

Binding to an element object is not as expensive as binding to a distributed shared object, however. To bind to an element we only need to look up a contact address of the cluster it belongs to in the worldwide location service. Consequently, when a client binds to an element in a cluster C, it need only look up a contact address if it is not yet bound to another element in C. In this way, we avoid many expensive look-up operations, while allowing multiple distribution strategies within a large distributed shared object (i.e., a container).

The underlying assumption is, of course, that a cluster contains more than one element. If each cluster contains exactly one element (i.e., each element has its own replication strategy), this approach is as expensive, in terms of number of look-up operations on the worldwide location service, as turning each element object into a distributed shared object. In all other cases there will be less load on the location service.

However, we are not required to use the worldwide location service for finding a cluster's contact addresses. The fundamental idea of Globe is that a distributed object has complete control over all aspects of its implementation, over its functional aspects and, in particular, over its nonfunctional aspects. This idea enables us employ object-specific solutions for each part of a distributed shared object's implementation. This applies to finding the contact addresses of clusters as follows.

Clusters can be viewed as part of the implementation of a container DSO. As a consequence, the problem of finding the contact addresses of a cluster that is part of a certain container object is an aspect of the implementation of that

container DSO. We should therefore be free to use object-specific solutions for this problem if this results in a more efficient implementation. In other words, it should be possible to put the container DSO in charge of keeping track of where its clusters can be contacted, if the container DSO can do this more efficiently than the (external) location service.

For example, if the contact addresses of a container's clusters hardly ever change (i.e., each cluster is always replicated on the same set of machines), these addresses could just as well be replicated at each of the container's local representatives, avoiding expensive look-up operations all together. This idea and how it can be actually implemented in Globe is currently under investigation. At present, we choose to use the worldwide location service for discovering a cluster's contact addresses.

An important property of our clustering mechanism is that clusters are visible only to the people and services managing the nonfunctional aspects of the container object and its elements. A developer can model his distributed application in terms of container objects containing element objects. The partitioning of elements into clusters does not play a part at this stage, and can be dealt with at the appropriate time.

The concept of clusters is new to the Globe distributed system. Until now, a distributed shared object was not only the unit of functionality but also the unit by which nonfunctional aspects could be changed. The clustering mechanism now allows large distributed shared objects to replicate parts of their state according to different strategies. Or, taking a different viewpoint, this clustering mechanism introduces a form of composite distributed objects where the component objects (i.e., the element objects) are more lightweight than regular distributed shared objects, because they are replicated in clusters, but are almost equally powerful.

6. Implementing clusters

We describe the implementation of our clustering mechanism in Globe by looking at the internal structure of a local representative for a cluster of elements. We also discuss how elements and clusters are added to and removed from a container DSO.

A local representative for a cluster consisting of two elements is shown in Figure 2. It can be divided into two parts: the cluster-specific part and the element-specific part. The element-specific part consists of a set of control subobjects, one for each element, supporting the methods of the elements and possibly semantics subobjects. Semantics subobjects are loaded only when the local representative is acting as a replica for the cluster of elements. Whether or not a local representative should act as a replica is determined by the replication strategy of the cluster. If it should

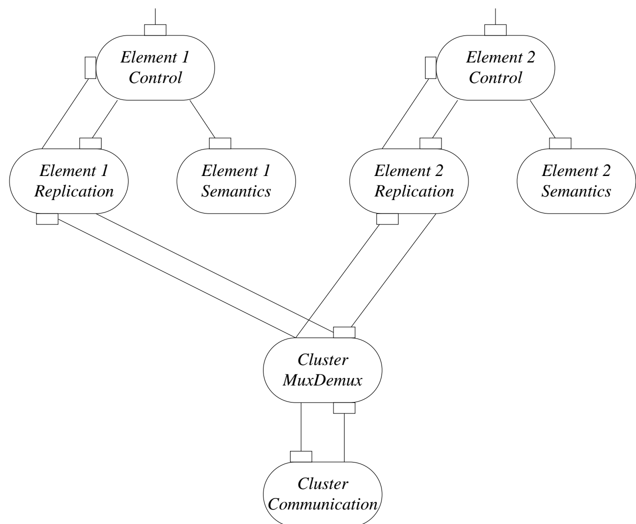


Figure 2. A local representative for a cluster of element objects, Element 1 and Element 2. There is no single replication subobject for the cluster, instead each element has its own replication subobject. The replication subobjects access the communication subobject for the cluster through a new subobject, called the multiplexer/demultiplexer subobject (abbreviated MuxDemux), which multiplexes and demultiplexes the communication streams to the replication subobjects.

act as a replica there are semantics subobjects for all elements, as is the case in Figure 2. Otherwise, there are no semantics subobjects in the local representative.

Furthermore, there is a replication subobject for each element in the cluster. The replication protocol (implementing the replication strategy) is, of course, the same for all elements in the cluster, suggesting that there should be a single replication subobject. However, we choose to give each element a separate replication subobject. The most important reason for this implementation decision is that we want to maintain some independence between the elements with respect to replication. For example, in a copy-invalidate replication protocol we want to be able to invalidate state on a per-element basis, not per cluster. Furthermore, we want to reuse replication subobjects written for distributed shared objects as much as possible. Rewriting a replication subobject to handle a cluster of objects requires considerable modifications, which can be avoided by letting each element object have its own replication subobject.

The cluster-specific part of the local representative consists of a communication subobject and a new subobject, called the *multiplexer/demultiplexer* subobject. The

communication subobject is a regular communication subobject, providing the communication facilities required by the replication protocol of the cluster. The multiplexer/demultiplexer subobject is introduced to deal with the fact that we have multiple replication subobjects instead of one.

The mechanisms that Globe uses to construct and install local representatives of distributed shared objects in a client's address space can also be used to construct these cluster representatives. Only minor additions to the runtime system are necessary. We do, of course, need to record some additional information, such as the mapping from element identifier to cluster identifier. We use the container DSO for this. A container DSO is a regular distributed shared object that has a set of special methods, in addition to its application-defined methods. This set of special methods is used to retrieve and maintain the additional information regarding elements and clusters (the mapping from cluster identifier to contact addresses is, of course, maintained by the location service). The implementation of this set of methods is provided by the system in the form of a semantics subobject and is automatically combined with the implementation of the container's application-specific methods.

Each cluster has one or more associated *element managers*. Element managers are used to add element objects to the cluster, that is, create new element objects that follow the cluster's replication strategy. Their most important task is to add control and semantics subobjects to all the cluster's local representatives that should function as replicas according the cluster's replication strategy, and to initialize these replicas with the element object's initial state. The exact implementation of element managers is outside the scope of this paper, but basically they are special element objects that are aware of the replication strategy of the cluster they are associated with. Creating a new cluster consists primarily of creating element managers and registering the new cluster and its associated element managers with the container DSO.

Removing elements from a container means destroying the element, because in our model elements cannot exist without a container. Destroying an element is similar to destroying a distributed shared object, and requires no additional mechanisms. Once all elements that follow the cluster's replication strategy have been logically removed for a container, and none are expected to be added, a cluster can be destroyed. This basically consists of deregistering the cluster and its element managers at the container and destroying the element managers.

7. Conclusions

Current middleware platforms are based on the remote-object view of distributed objects. Replication and caching support, necessarily for developing large-scale distributed applications, can and is being added to these platforms [5]. However, we argue that adopting a different view of distributed objects, notably viewing them as physically distributed entities, results in more comprehensive support. The distributed shared object model as implemented in the Globe system makes replication and caching support not only more comprehensive, but also results in a more flexible middleware platform. In particular, we can add new forms of replication support through distributed containers, without changing the architecture of Globe.

References

[1] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[2] G. Kiczales, J. Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[3] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA., 1990.

[4] B. C. Neuman. Scale in Distributed Systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[5] Object Management Group. Fault tolerant CORBA Using Entity Redundancy: Request For Proposal. OMG Document orbos/98-04-01, Object Management Group, Framingham, MA, Apr. 1998.

[6] Object Management Group. Portable Interceptors RFP: Request For Proposal. OMG Document orbos/98-09-11, Object Management Group, Framingham, MA, Sept. 1998.

[7] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.2. OMG Document formal/98-07-01, Object Management Group, Framingham, MA, Feb. 1998.

[8] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Sebastopol, CA, 1992.

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

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

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