Encapsulating Distribution by Remote Objects

Marten Jansen
Océ-Technologies, Venlo
mja@oce.nl

Eelco Klaver
Graddelt International, Amsterdam
eelco.klaver@graddelt.com

Patrick Verkaik
Maarten van Steen
Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
{*patrick,steen,ast*}*@cs.vu.nl*

Internal report IR-476
August 2000

Abstract

Middleware for modern office environments and many other application areas needs to provide support for a myriad of different, highly-mobile objects. At the same time, it should be able to scale to vast numbers of objects that may possibly be dispersed over a large wide-area network. The combination of flexibility and scalability requires support for object-specific solutions that is hardly addressed by current object-based systems such as CORBA. We have developed a middleware solution that seamlessly integrates traditional remote objects with physically distributed objects that can fully encapsulate their own distribution strategy. We describe how this integration takes place, and how it can be applied to existing systems such as CORBA.

Keywords: distributed objects, mobile objects, middleware, distribution encapsulation

vrije Universiteit

Faculty of Mathematics and Computer Science

1 Introduction

People in many office environments are no longer fixed to a specific location. Instead, they carry their own mobile devices, such as a laptop, connect to the local infrastructure at an arbitrary point, and expect to continue their work where they left off. The traditional notion of an office is thus gradually being replaced by that of a virtual, highly personalized office that an individual connects to as he or she moves from location to location. Such a virtual office is supported by devices such as personal digital assistants, smart cards, notebooks, and so on. The working environment of an employee in such an office will simply appear to travel along as that person moves.

One approach to implementing such virtual offices is to statically configure a number of servers in a network, and allow clients to access those servers from very different locations using a myriad of access devices. This approach is followed by a number of *application service providers*. However, there are several drawbacks to this approach, the most important one being the lack of flexibility. In practice, resources and services in many office systems change rapidly in number and functionality. Also, the underlying infrastructure by which those resources and services can be accessed is continuously changing.

As a simple example of the need for flexibility, consider the implementation of a company's intranet Web site. Until recently, it was common practice to construct such a site by means of a server that has access to the local file system containing the HTML pages comprising the site. In addition, user interaction was supported through CGI-BIN scripts. At present, a shift is taking place through Content Delivery Networks (CDNs) which offer Web hosting services that can replace a company's traditional Web server. At the same time, the traditional notion of Web servers is changing as Web services become accessible on small-scale devices such as PDAs [2].

The components that comprise a user's working environment are thus seen to be highly dynamic in time and space. Basic resources and services such as printers, CPUs, storage, access devices, and so on, change as the user moves. Also, components that make those resources and services available, as well as the applications that are part of the working environment, change in a similar fashion.

Any system that is to support future virtual offices, or similar environments, will thus need to accommodate mobility in the broadest sense of the word. Users should be able to move easily between locations implying that they should equally well be easy to locate when needed. Components that comprise a user's environment should be able to migrate, and be traceable as well. Likewise, finding and binding to the appropriate resources and services to accommodate a user's environment at a specific location should be easy to do. For example, a roaming user who wants to print his e-mail should be able to do so without having to (manually) configure his notebook for the type of printer that happens to be locally available. What we are thus looking for is flexibility.

Middleware can offer the flexibility we are seeking in such environments. However, current solutions, be they based on a middleware approach or not, fail to adequately address scalability. Not only are we faced with problems regarding the sheer amount of data and number of people that need to be supported, problems are also complicated by the fact that in many cases, people and resources may be dispersed across a wide geographical area. Further complications are caused by the highly dynamic nature of data, people, devices, functionality, and media, both in time and space.

This paper describes the integration of two middleware solutions, CORE and Globe, that jointly address the flexibility and scalability issues that are needed in future office environments. CORE is a CORBA-like middleware layer which has been designed to support highly mobile objects, tailored to office environments. Globe too is a middleware system, but instead of concentrating on mobile objects, its designers concentrated on supporting highly replicated and widely distributed shared objects.

The main contribution of this paper is that we show how CORBA-like systems, such as CORE, can be extended in such a way that scalability problems can be more effectively tackled. Part of our approach is to adapt the remote-object model allowing objects to be physically distributed, while at the same time fully encapsulating their own distribution strategy. The adaptation does not affect existing interfaces.

We start by describing CORE, an object-based middleware system whose object model strongly resembles that of CORBA. The objects as used in CORE need to be enhanced to support object-specific distribution, as we explain in Section 3. Details on these enhancements are discussed separately in Section 4, where we concentrate on keeping enhancements transparent to existing applications. We conclude in Section 5 by reflecting on our work and compare it to work by others.

2 The architecture of CORE

CORE is a middleware platform aimed at mobility. To enable objects to be highly mobile, instead of only having portable code, CORE mobile objects are written in Java. There are Java Virtual Machines (JVMs) available for most processing devices. Another key property of CORE is the small footprint. As the target environment contains many devices with limited processing power and resources, CORE should claim minimal resources.

2.1 Global architecture

A CORE system consists of a heterogeneous network of machines, which together form a distributed system. Each machine runs a number of services—hardware or software entities that can perform a specific task. To enable communication between services on different machines and to provide some basic functionality, such as lookups of services, a small piece of middleware runs between the operating system and the services. This piece of middleware makes network access and location of services transparent to both the user and the programmer of services.

As most other object-based distributed systems such as CORBA [16] and Java RMI [17], CORE uses a **remote-object model**. In this model, an object is placed in the address space of a single object server. The object server exports the object's interfaces to remote clients, allowing them to invoke the object's methods. The interface implementation at the client is called a **proxy** [15]. A proxy marshals a method invocation into a message that is sent to the object server. An incoming invocation request is then unmarshaled at the server, after which the method is invoked at the object, as shown in Figure 1.

The object is identified by a globally unique **object identifier**, referred to as a **CORE OID**. A CORE OID is a true identifier [23]. In particular, it contains no location information. In addition, each of the services offered by the object by means of its interfaces are identified by a separate service ID. Given a service ID, it is possible to look up the object that provides the identified service. A service ID is somewhat comparable to an interface identifier in CORBA or DCOM [4], except that it is also uniquely tied to an object.

In CORE, no assumptions are made concerning the location of an object. Objects are allowed to move freely between servers on possibly different machines. When an object moves to a different server, the proxy is responsible for reconnecting to that server. Likewise, the proxy is responsible for masking failures as much as possible, although fatal errors such as crash failures or persistent communication failures may raise an exception at the client (see also [21]). When a method is invoked during the migration of the object to another server, the invocation is queued until the object is up and running again. The object itself is not aware of any of its proxies. In our current implementation, strong consistency is guaranteed by forbidding the proxy to cache results of method invocations.

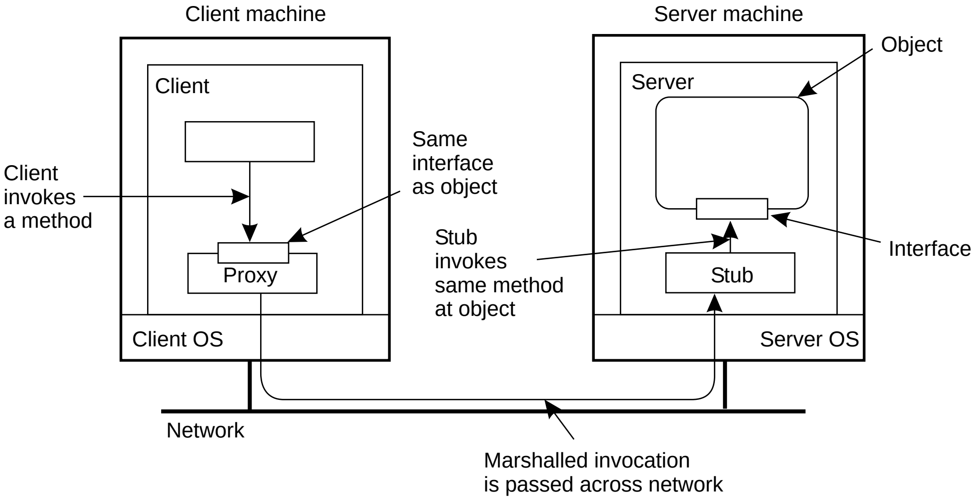

Figure 1: The principle of a remote object.

2.2 CORE platform

A CORE-based distributed system consists of a collection of interconnected hosts, where each host executes a number of basic CORE components. These components collectively form the **CORE Runtime System** (RTS). Each host can run one or more CORE processes. Each process can be accessed through its transport-level address, that is, an *(IP address, port number)*–pair. The CORE RTS at each process partly consists of the Voyager object broker,[1] along with CORE extensions that allow the process to, for example, communicate with other CORE processes. This organization is shown in Figure 2, and is explained next. In the following, the most important CORE components are discussed.

Registry

The **Registry** is used to register (and unregister) objects locally, including the basic CORE components that are part of the platform. The registry also keeps some usage information on each object, such as the number of times it has been invoked, the last time it was invoked, etc. Registration of an object proceeds by providing its interfaces to the registry. Subsequent lookups are done by means of the registry as well, possibly with the help of other CORE components that we discuss below.

Distribution Manager

CORE objects are accessible to remote clients. To that end, a client must **bind** to an object by locally installing a proxy for each object it wants to invoke. Installing a proxy is done automatically by uploading it to the client as we describe below. As we explained, proxies are object specific. In CORE, objects themselves are responsible for creating and handing out proxies to their clients, but for this they can make use of a number of CORE services. The **Distribution Manager** (**DM**) is such a service. This component can be called by an object with the request to generate a proxy object implementing one or more of the object's interfaces. The proxy contains the object's CORE OID, as

[1] See `http://www.objectspace.com` for information on Voyager.

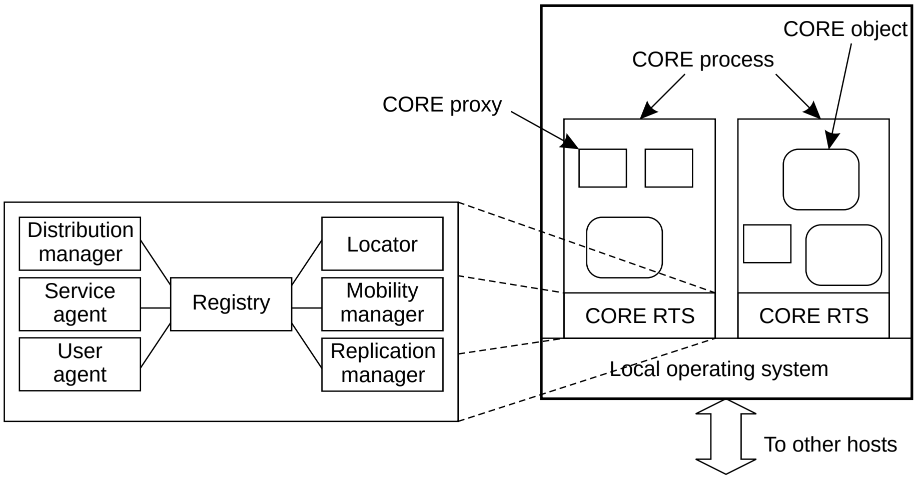

Figure 2: The organization of a CORE-enabled host.

well as a reference to where the object currently resides. A proxy can be serialized and handed out to a client.

Service and User Agent

The **Service Agent** (**SA**) and **User Agent** (**UA**) are two related components. An SA is responsible for registering services at remote platforms, whereas a UA is responsible for looking up services at remote platforms. The implementation of these agents is based on the Service Location Protocol (SLP) described in RFC 2165.

An SA maintains a list of descriptions of currently available services at various CORE platforms. In essence, each description takes the form of an *(attribute, values)*–pair and is normally provided directly by the objects themselves. An SA is responsible for broadcasting removals of service descriptions, and also continually listens to requests and updates coming in over the network.

To look up a service, a UA will broadcast a systemwide lookup request by specifying a predicate formulated in terms of these *(attribute, values)*–pairs. When a lookup request comes in, the SA tests the associated predicate against all descriptions in its list and returns the IDs of each service of which the description matches the predicate. The UA collects replies from SAs (containing service IDs) for some period of time or until it has received the requested number of services.

It is thus seen that the SAs jointly maintain a distributed database of service descriptions. However, broadcasting as is done in CORE has inherent scalability problems. To solve these problems, more sophisticated techniques are necessary. For example, the distributed database could easily be replicated to facilitate local lookups by UAs. This approach is feasible as the set of services is expected to be relatively stable. In other words, the mapping between *(attribute, values)*–pairs and service IDs of the corresponding interfaces will not change often. In addition, it is not necessary that the SAs always share the exact same view of available services. The combination of relative stability and weak consistency allows us to use highly scalable techniques such as deployed in epidemic algorithms [3, 14].

Locator

The **Locator** locates an object with a specified CORE OID on behalf of a client, and takes care of binding the client to the requested object. Binding is accomplished by returning a serialized proxy that implements the interface of the object. Note that this proxy contains the current address of the object. Consequently, by merely deserializing the proxy, the client can immediately invoke methods of the object. We return to the Locator when discussing the Globe location service below.

Mobility Manager

The **Mobility Manager** (**MM**) is responsible for moving objects from one location to another. Instead of implementing a single migration scheme, different schemes are possible by making use of **Object Movers** (**OM**). For example, some objects can be migrated merely by shipping their state to the target machine because the necessary classes are already available. In other cases, migration may need to take place by shipping code and state, or by letting the target first load the appropriate classes. As another example, when the target machine is a notebook, it will probably be necessary to ensure that it has all the necessary classes available to instantiate the object, for the notebook may need to operate in disconnected mode. On the other hand, for a continuously online target, classes can be dynamically downloaded when they are actually needed.

In effect, an OM implements a specific migration policy. An MM is responsible for coordinating the migration between two locations. When an object is to be moved from one host to another, the MM at the source is given the address of the target host, and optionally a preferred OM. This MM then contacts the MM of the target host and negotiates an OM that both machines support, keeping the preferred OM in mind. The negotiated OM is returned to the caller of the MM. The OM is then used to actually move the mobile object. As soon as the object is in a transferable state, the OM marshals the object, transports it, and lets the OM at the target machine rebuild the object.

Replication Manager

The original version of CORE supported only remote objects. However, a **Replication Manager** (**RM**) has later been added and which is responsible for replicating objects to other locations. Keeping the state of replicated objects consistent with each other is, however, the task of the replicated objects themselves, because they know best how to do that most efficiently. When an object is to be replicated to another host, the RM is contacted specifying the target host and port number, and the object to replicate. Subsequently, the RM asks the object to first create a replica of itself and then moves this replica to the specified target machine. Replication is discussed in more detail below.

3 Distribution of CORE objects

The original version of CORE does not support replication, and has some serious limitations with respect to its scalability. The first problem is that the state of a remote object always resides at a single location. Consequently, in a geographically-dispersed environment, access times to such objects may be considerable. If an object is not simultaneously shared by multiple clients, access times can be improved by migrating the state to the object's current client. However, this solution does not work when several clients require access to the object at the same time. Another problem is that if objects are allowed to migrate a lot, we need a scalable mechanism to track and locate them. The original naming service in CORE offers only a centralized solution which scales poorly to large networks.

There are basically only three solutions to tackle scalability problems: distribution (i.e., partitioning), replication, and caching [12]. In the case of replication or caching, objects are copied to several machines. In principle, service requests are simply forwarded to the nearest copy. The main drawback of replication is that whenever a copy is updated, that copy becomes inconsistent with the others. Consequently, we need to update the other copies as well. In a worst-case scenario, propagation of updates may require global synchronization between the replicas. Unfortunately, global synchronization has itself inherent scalability problems. The only approach to alleviate such problems is to provide weaker consistency guarantees.

An important observation, however, is that the ideal replication policy is dependent on the usage of the object being replicated. For example, as reported in [13], applying a single replication strategy to all documents from the same Web site will never lead to the same performance as selecting a policy for each document separately. In other words, applying caching and replication as a means to enhance the scalability of a system, is most effective if each object is allowed to have its own replication strategy. Originally, CORE, like many distributed systems based on remote objects such as CORBA and Java RMI, lacked this support.

We have improved the scalability of CORE in two ways. First, we have replaced part of the naming service by a scalable location service that allows us to track frequently migrating objects in an efficient way. Second, the remote-object model has been partly replaced by physically distributed objects that encapsulate their own strategy regarding how their state is distributed, replicated, or migrated. These two extensions to CORE are described next.

3.1 A scalable location service

Our first extension to CORE was to partly replace the naming service with a scalable location service as developed in the Globe project [18]. The CORE Locator was rebuilt to act as a front end to this location service. The basic idea is that each distributed object in CORE is assigned a lifelong object identifier. It is crucial that this identifier never changes, and, in particular, that it is location independent. In contrast, a (location-dependent) **contact address** is used to refer to the current location of an object.

As we explained, a contact address in CORE is implemented as a serialized proxy, similar to the use of proxies in Java RMI [22]. When a client wants to bind to an object it looks up the object's current location by providing the object identifier to the location service. The location service returns a contact address in the form of a serialized proxy. This proxy is subsequently deserialized by the client and automatically initialized. The proxy already has the current location of the object as part of its state, so that after deserialization, the client can immediately invoke the object's methods as made available through the proxy.

Note that when an object moves, the proxy as stored in the location service is updated with the object's new location. As we explain next, updating means removing the old proxy and inserting a new one. However, for those processes that are already bound to an object, and thus already have a proxy in their address space, it is the proxy's responsibility to keep track of the current location of the object. Currently, this has been implemented by raising an exception when a client attempts to access the object again, enforcing the client to look up the object's current address in the location service.

The location service organizes the underlying network into a hierarchical collection of domains. For example, a lowest-level domain may represent the entire network of a science park, whereas the next higher-level domain represents the city where that science park is located. Each domain is represented in the location service by a directory node. The concept of a domain in the location service is thus somewhat similar to the concept of a domain in DNS.

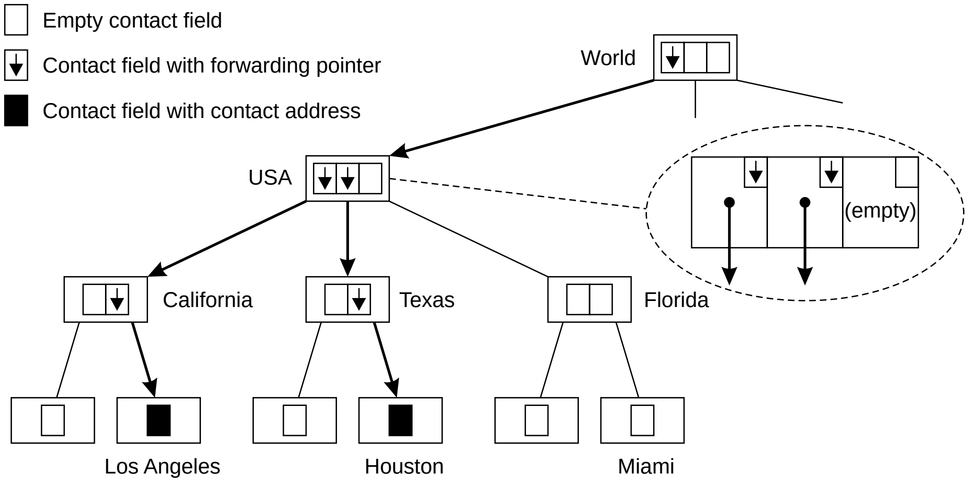

Figure 3: The organization of the Globe location service.

A directory node has a contact record for every (registered) object in its domain. The contact record is divided into a number of contact fields, one for each child node. A directory node stores either a forwarding pointer, or an object address in the contact field. A forwarding pointer indicates that an address can be found at the child node. Contact records at leaf nodes are different, they contain only one contact field storing the object's address in that leaf domain.

As an example, consider Figure 3, which shows a configuration for a single object that has been replicated across Los Angeles and Houston. The root node has a forwarding pointer to the USA node, which in turn points both to California and Texas. Now, consider a client in Miami who wants to contact the object. In that case, a lookup request will travel from the Miami node upwards to the first node where the object is known. In our example, this is the USA node. From there on, the request is forwarded either to Texas or California, and eventually reaches a leaf node where an address is stored. The important issue is that scalability is obtained by exploiting locality: a request issued within the USA will never leave the USA domain if a replica is present in that domain.

The location service has been implemented in Java and integrated with CORE. Further details concerning its algorithms, including the partitioning of higher-level nodes, can be found in [20, 1].

3.2 Physically distributed objects

Efficiently locating mobile objects in a geographically dispersed network is not enough for scalability. We also need support for caching and replicating objects. The ideal replication strategy for all objects is that updates are (1) immediately propagated to all copies, and (2) each copy sees all updates in the same order. Unfortunately, such a strong degree of consistency is impossible to implement efficiently in large-scale systems as it requires global synchronization. As we discussed, the solution to scalability is to weaken consistency requirements so that alternative and more efficient replication strategies can be followed. Such an approach requires that we take the usage and update patterns of individual objects into account. Consequently, it makes sense to let an object fully encapsulate its own replication strategy. This is the approach followed in Globe [19], and which we adopted for CORE.

One of the key concepts of the Globe system is its model of **distributed shared objects**. Like other object-based models each object offers one or more interfaces, each consisting of a set of methods. Objects are passive; activity comes from processes. Multiple processes may access the same

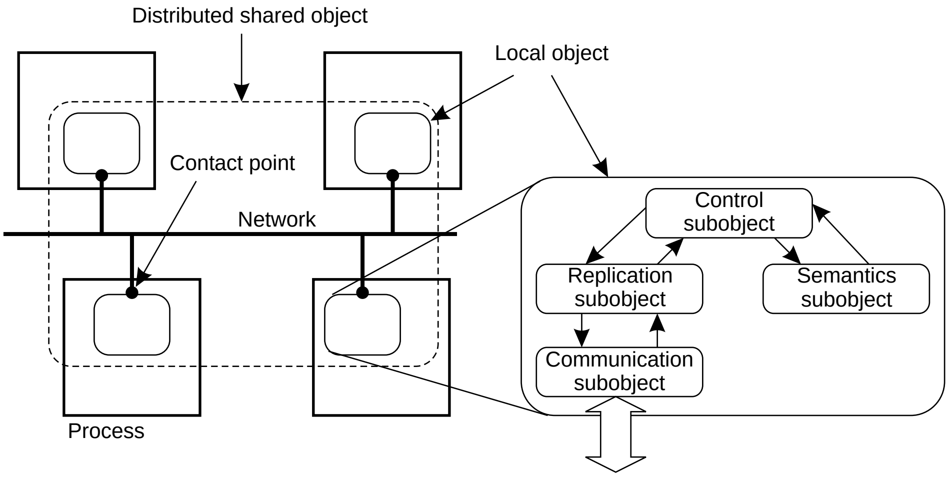

Figure 4: A Globe distributed shared object.

object simultaneously. A major distinction with other object-based models is that objects are physically distributed, meaning that copies of an object's state can and do reside on multiple machines at the same time. However, processes are not aware of this: state and operations on that state are completely encapsulated by the object. This means that all implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of the object but are hidden behind its interface. This model is illustrated in Figure 4.

A distributed shared object has the possibility to distribute (part of) its state among several machines. It is the responsibility of the object to decide what distribution or replication policy is taken and to keep its state consistent. The policy can vary from replicating the whole state on different machines (replication) to dividing its state in distinct parts and locating each part on different machines (partitioning). Also, how, when, and where updates are propagated is entirely left to the object to decide. As a consequence, different objects can and do implement different replication policies.

Each distributed object is spread across multiple processes by installing a **local object** at each process. Each local object is composed of several subobjects, and is itself again fully self-contained as also shown in Figure 4. A minimal 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 or C++. These primitive objects can be developed independent of any distribution or scalability issues.

Communication subobject. This is generally a system-provided subobject. It is responsible for handling communication between parts of the distributed object that reside in different processes. Depending on what is needed by 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 its various semantics subobjects in different processes. Semantics subobjects may be replicated for reasons of fault tolerance or performance. The replication subobject is responsible for keeping these replicas consistent according to some (per-object) coherence strategy. Different distributed objects may have different replication subobjects, using different replication algorithms.

The replication subobject has a standard interface. However, implementations of that interface will generally differ between replication subobjects. In a sense, this subobject behaves as a meta-level object comparable to techniques applied in reflective object-oriented programming [8].

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

A key role is, of course, reserved for the replication subobject. An important observation is that communication and replication subobjects are unaware of the methods and state of the semantics subobject. Instead, both the communication subobject and the replication subobject operate only on invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication subobjects and communication subobjects.

As in CORE, each distributed shared object has a globally unique, location-independent object identifier. To distinguish such identifiers from those used in CORE, we refer to them as **Globe OIDs**. Again, a client can invoke the methods of a distributed shared object by first binding to the object. Binding requires that a contact address is looked up in the location service. In contrast to CORE, a contact address for a Globe distributed shared object consists of a transport-level address, along with a **protocol identifier**, specifying exactly what subobjects the client should implement in order to communicate with the object. In the current implementation, a protocol identifier takes the form of a URL referring to an implementation of the required subobjects that the client should load into its address space.

The effect of binding to a distributed shared object is that a local object is placed in the client's address space. This local object has the same internal organization as before, in that it consists of the four subobjects described above. In this sense, there is no strict distinction between the client and other processes bound to the same distributed shared object. In principle, the client may register a contact address for the object in the location service, thus allowing other processes to bind to the object as well, using the freshly bound process as an intermediate. However, a security policy may possibly prohibit such registration.

4 Jackets: Integrating CORE and Globe

To address the scalability problems inherent to the CORE object model, we have integrated CORE and Globe into a single system. An important objective was to make the integration transparent to CORE applications. In other words, the existing CORE object model should essentially remain the same. Likewise, we did not want to affect the Globe object model. This approach has led to an integrated object model shown in Figure 5. We refer to objects in this model as **CORE distributed shared objects** (**CORE DSO**s).

Conceptually, the integration is relatively simple. Each local object of a Globe distributed shared object (referred to as a Globe DSO) is fully encapsulated in a special CORE proxy, called a **jacket**. A jacket offers all the standard interfaces that are normally provided by CORE proxies. In this sense, the CORE DSO cannot be distinguished from other CORE objects. This encapsulation also allows a

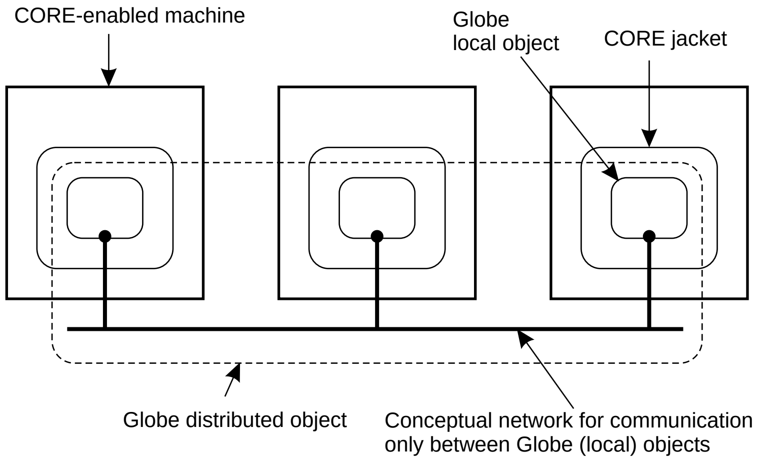

Figure 5: The integration of remote and distributed shared objects into a single model.

Globe local object to treat a jacket as just a client. In other words, the local object can remain ignorant of how a process is actually bound to it.

Consequently, CORE applications and Globe DSOs are, in principle, fully independent of one another. However, there are a number of subtleties that need to be dealt with to maintain this independence, which we discuss next.

4.1 Creating a CORE DSO

Our first concern is the creation of a CORE DSO. In Globe, a distributed shared object is created by having a process first create a local object with all its subobjects. In practice, this means that the process loads a file containing the complete implementation of a specific local object after which it instantiates the object.

Together with instantiating a local object, a Globe object identifier (Globe OID) is generated for it, effectively establishing a Globe DSO, albeit one with only a single local object. The Globe OID is used to register a Globe contact address in the location service. This contact address allows another process to contact the process that just created the object, and bind to it.

In essence, the same procedure is followed when creating a CORE DSO. First, a CORE process creates a jacket. Because a CORE process can, in principle, support any objects written in Java, it is straightforward to embed a Globe local object that has been written in Java into a CORE process. After instantiating a jacket, initialization of the jacket continues by creating a Globe local object as just described, along with a Globe OID. Only the latter is stored in the serializable state of the jacket. The Globe local object that has just been created does not form part of this state, as shown in Figure 6. We return to this issue below.

To make the Globe local object known to other objects, the CORE process first creates a contact point, such as a socket. It then registers the local object at the location service by handing it a *(Globe OID, Globe address)*–pair, with the address containing all the necessary information to enable binding. Note that such a pair is useless to CORE processes, which expect a serialized CORE proxy. Therefore, to facilitate such processes, the associated jacket is serialized and inserted into the location service under its own CORE OID. Of course, jackets for local objects that belong to the same Globe DSO will have the same CORE OID.

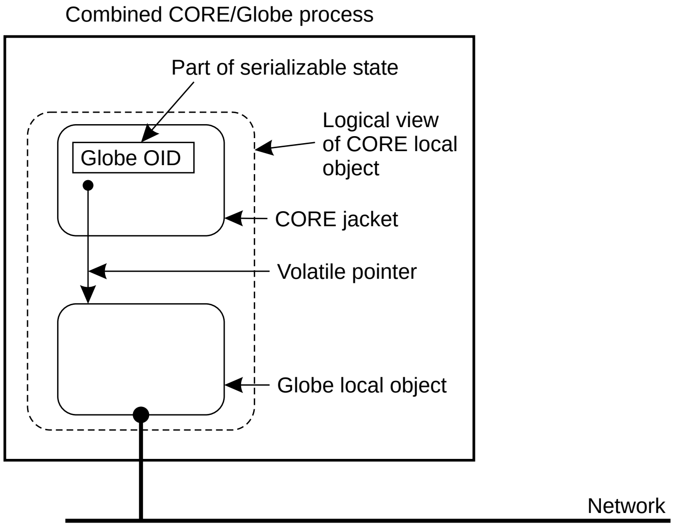
Combined CORE/Globe process
Part of serializable state
Globe OID
Logical view of CORE local object
CORE jacket
Volatile pointer
Globe local object
Network

Figure 6: The relation between a jacket and its associated Globe local object.

Binding to a CORE DSO is now straightforward. A process looks up a CORE contact address in the location service by handing it a CORE OID. The location service returns a serialized jacket, containing a Globe OID in its serialized state. After deserializing and instantiating the jacket, a second lookup takes place, but now by means of the Globe OID found in the jacket. This lookup returns a contact address for a Globe local object. The process subsequently creates a local object that connects to the associated Globe DSO through the returned contact address.

4.2 Migration and replication

Objects in Globe and CORE are highly independent in the sense that mutual adaptations are actually not necessary to allow integration. However, migration and replication as provided by CORE require some subtle changes.

As we explained, migration in CORE takes place by means of the mobility managers. In essence, a CORE object is moved by serializing it, transferring the serialized object to the target server, and deserializing it again. A different approach needs to be followed for a CORE DSO. Migration is done in two phases. The first phase consists of migrating the jacket, whereas the second phase consists of migrating the associated Globe local object.

Migrating the jacket is straightforward. Like any other CORE object, the jacket is serialized and moved from source to destination with the help of the respective mobility managers and object movers as described above. Note that the volatile pointer shown in Figure 6 is *not* serialized. After this phase, there is still a Globe local object at the source, but none at the target. To Globe, unless special measures are taken, this situation appears as if the local object at the source is no longer used. In principle, it could therefore be cleaned up. However, the local object might contain state that must be safeguarded first.

To avoid removing the local object, a binding to the local object is maintained at the source as long as necessary, as we explain next. In line with the approach that objects should fully encapsulate their own distribution strategy, an object-specific scenario is followed for completing the migration process. Let us consider two example scenarios.

In the first scenario, the jacket at the destination creates a Globe local object that connects to the local object at the source. A state transfer between these two Globe local objects subsequently takes place. Then, the contact address of the source is removed from the location service, whereas a contact address at the destination is inserted. At that point, all state of the Globe distributed shared object has been preserved, so that we can safely remove the Globe local object at the source. Removal includes informing all other local objects of the Globe DSO to disconnect from the local object at the source, if necessary, and possibly reconnect to the local object at the destination.

A completely different scenario is the following. After having migrated the jacket to the destination, the local object at the source removes its contact address from the location service. This removal prevents other processes from binding to it. The local object subsequently instructs processes to disconnect from it, possibly after having shipped its state to other local objects such that the state of the DSO as a whole is preserved. Meanwhile, the jacket at the destination simply starts binding to the Globe DSO referred to by the Globe OID that is part of its state. Binding at the destination can take place simultaneously with the unbinding of the local object at the source.

Variants of these two scenarios can easily be thought of. However, it is up to the distributed shared object to decide which strategy is the best one to follow. It is precisely this separation between mechanism and policy that we feel is currently lacking in many object-based distributed systems.

Replication is currently dealt with in a similar fashion. A CORE replication manager may be instructed to create a new replica, for which it creates a jacket as described above. The newly created jacket simply binds to the associated Globe DSO and, following an object-specific scenario, maintains local state consistent with other local copies. Deciding on where to create a new replica is currently left to applications. However, it should also be possible for an object to decide that more or fewer replicas are needed. In that case, an object will itself contact a replication manager to install or remove a replica. We are currently developing mechanisms to support such object-initiated replication (see also [7]).

5 Discussion and related work

Distributed objects in virtually all existing systems are actually implementations of a remote-object model in which the object resides at a single server. Remote clients are offered transparent access to an object by means of proxies. We argue that the remote-object model misses two important properties.

First, it can be argued whether remote objects are actually distributed. In many cases, remote objects are nothing but traditional objects contained in a single object server, but which can be transparently accessed by remote clients by means of proxies. Further distribution transparency is supported by allowing the object to migrate between servers, but hiding all location-awareness inside the proxies.

Second, remote objects do not encapsulate *all* implementation aspects. In particular, many distribution policies are implemented by special or configurable object servers [6, 9]. We argue that distribution policies should be part of an object's implementation, similar to why an object encapsulates its state and operations.

There are only a few distributed-object models that follow this approach, notably the fragmented objects as developed by the SOR group at INRIA [10]. At best, support is given for adapting the client proxy to specific objects, as used in Spring's subcontract model [5] or as implemented in Java RMI [22]. To compensate for the lack of flexibility in the supported object model, CORBA provides interceptors as a mechanism to provide object-specific policies [11]. However, interceptors essentially allow only breaking into an existing ORB and modifying its invocation policy. Much more is needed to actually provide object-specific support for distribution.

Our research demonstrates that it is possible to support object-specific policies while retaining the client's view of a remote-object model. In fact, we have shown that integration of a remote-object model and that of physically distributed shared objects can be done in such a way that neither model needs to be affected. Integration takes place by means of a relatively simple type of object, called a jacket. Although we have concentrated on the integrated implementation of CORE and Globe, this same approach can be followed for existing systems such as CORBA.

Acknowledgments

The original concept and the basis for the CORE architecture as described in this paper, was part of the Océ-Technologies research efforts, initiated by Teus Hagen who is now at NLnet Foundation. We also gratefully acknowledge the implementation work done by Ivo van der Wijk on the integration of CORE and Globe, and that of Sebastiaan de Jongh with respect to application development.

References

[1] G. Ballintijn, M. van Steen, and A. S. Tanenbaum. "Exploiting Location Awareness for Scalable Location-Independent Object IDs." In *Proc. Fifth ASCI Annual Conf.*, pp. 321–328, Heijen, The Netherlands, June 1999. ASCI.

[2] G. Borriello and R. Want. "Embedded Computation meets the World Wide Web." *Commun. ACM*, 43(5):59–66, May 2000.

[3] A. Demers et al. "Epidemic Algorithms for Replicated Data Management." In *Proc. Sixth Symp. on Principles of Distributed Computing*, pp. 1–12, Vancouver, Aug. 1987. ACM.

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

[5] G. Hamilton, M. Powell, and J. Mitchell. "Subcontract: A Flexible Base for Distributed Programming." In *Proc. 14th Symp. Operating System Principles*, Asheville, N.C., Dec. 1993. ACM.

[6] B. Jørgensen, E. Truyen, F. Matthijs, and W. Joosen. "Customization of Object Request Brokers by Application Specific Policies." In J. Sventek and G. Coulson, (eds.), *Proc. Middleware 2000*, volume 1795 of *Lect. Notes Comput. Sc.*, pp. 144–163. Springer-Verlag, Berlin, 2000.

[7] A. Kermarrec, I. Kuz, M. van Steen, and A. Tanenbaum. "A Framework for Consistent, Replicated Web Objects." In *Proc. 18th Int'l Conf. on Distributed Computing Systems*, pp. 276–284, Amsterdam, The Netherlands, May 1998. IEEE.

[8] G. Kiczales. "Towards a New Model of Abstraction in the Engineering of Software." In *Proc. Int'l Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tokyo, Nov. 1992.

[9] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. "Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB." In J. Sventek and G. Coulson, (eds.), *Proc. Middleware 2000*, volume 1795 of *Lect. Notes Comput. Sc.*, pp. 121–143, Berlin, 2000. IFIP/ACM, Springer-Verlag.

[10] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. "Fragmented Objects for Distributed Abstractions." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 170–186. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[11] P. Narasimham, L. Moser, and P. Mellior-Smith. "Using Interceptors to Enhance CORBA." *Computer*, 32(7):62–68, July 1999.

[12] B. Neuman. "Scale in Distributed Systems." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[13] G. Pierre, I. Kuz, M. van Steen, and A. Tanenbaum. "Differentiated Strategies for Replicating Web Documents." In *Proc. Fifth Web Caching Workshop*, Lisbon, May 2000.

[14] M. Rabinovich, N. Gehani, and A. Kononov. "Scalable Update Propagation in Epidemic Replicated Databases." In *Proc. Int'l Conf. on Extended Database Technology*, pp. 207–222, 1997.

[15] M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle." In *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, pp. 198–204, Cambridge, MA, May 1986. IEEE.

[16] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley, New York, 2nd edition, 2000.

[17] R. Steflik and P. Sridharan. *Advanced Java Networking*. Prentice Hall, Upper Saddle River, N.J., 2nd edition, 2000.

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

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

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

[21] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. "A Note on Distributed Computing." Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, CA, Nov. 1994.

[22] J. Waldo. "Remote Procedure Calls and Java Remote Method Invocation." *IEEE Concurrency*, 6(3):5–7, July 1998.

[23] R. Wieringa and W. de Jonge. "Object Identifiers, Keys, and Surrogates–Object Identifiers Revisited." *Theory and Practice of Object Systems*, 1(2):101–114, 1995.