PARTITIONED OBJECT MODELS FOR DISTRIBUTED ABSTRACTIONS

Guray YILMAZ
Turkish Air Force Academy
Computer Engineering Department
Yesilyurt, 34807, Istanbul – TURKEY
e-mail : g.yilmaz@hho.edu.tr

Nadia ERDOGAN
Istanbul Technical University
Computer Engineering Department
Ayazaga, 80626, Istanbul – TURKEY
e-mail : erdogan@cs.itu.edu.tr

Abstract - Distributed systems provide sharing of resources and information over a computer network. In this paper we examine two scalable architectures based on the partitioned object model for distributed systems.
Index Terms -partitioned object model, distributed shared objects, fragmented objects

1. INTRODUCTION

The concept of an object offers both transparency and flexibility for distributed systems. The encapsulation of state and its operations into a single object allows a clear separation between the services provided by an object, its implementation. Aspects such as communication protocols, replication strategies, and distribution and migration of state can be completely hidden behind an object's interface. Therefore, adopting an object based model for distributed systems provides solutions to problems of transparency and flexibility. In this paper we examine the **partitioned object model** which differs significantly from distributed object-based models. According to partitioned object model, an object is assumed to be physically distributed across several sites and all distribution aspects of an object, including its location, migration relocation, and replication, are part of its implementation. We will discuss and compare two projects that focus on the partitioned object model. The first one is the **distributed shared objects** in Globe project which is developed at the Vrije University in Netherlands [3,4,5].The second one is **fragmented objects** which is developed by the SOR group at INRIA in France [1,2].

2. FRAGMENTED OBJECTS

The Object Model: The fragmentation of a single logical entity across several locations is useful in many distributed applications. On this point of view, SOR group has proposed the uniform concept of a **fragmented object (FO***)* for designing and building distributed abstractions. As is usual in object-oriented approach, a FO has two aspects an external or abstract and an internal or concrete view. Abstractly, a FO appears, to its external clients, as a single shared object. It is shared by several client objects, localised in different address spaces, possibly on several sites. It is accessed via a programmer defined interface. It can offer distinct, strongly typed, interfaces to different clients. Its components, and in particular their distribution, are not visible. Internally, the FO encapsulates a set of cooperating fragments. Each fragment is an elementary object (i.e. a fragment has a centralised representation). The fragments cooperate using lower-level FOs, such as communication channels. The programmer of a FO, its implementor, controls the location and the communication between the fragments. For the operating system, a FO is a group of elementary objects, with a common inter-address-space communication privilege which is checked by the primitive connective object, implemented by the system: e.g. communication channels or shared memory regions.

Accessing a Fragmented Object: The abstract interface of a FO is provided to some client by a local interface fragment of that FO. A fragment is an ordinary local object. Its public interface may be invoked locally. The client can not distinguish between the interface of the fragment and that of the FO itself. A method of the fragment interface can be entirely implemented by the fragment itself, or it can trigger invocations to other fragments. The internal representation of a FO is fragmented on several address spaces. The implementor considers criteria such as protection, efficiency and availability, to decide on the distribution of data among fragments.

Binding to a Fragmented Object: An object must be instantiated before use, and a client must first *bind* to a FO. To request access to some particular FO, the client invokes a binding procedure, associated with the type of the expected interface. This procedure returns a proxy. Binding takes place in three steps. In the first step, a name look up yields a *provider* object for the named interface. In the second step, the binding request is forwarded, by the distributed object manager, to the particular method of the provider. In the third step, this method may dynamically instantiate a proxy implementation, based (for instances) on the user's identity, on the binding request arguments (e.g. type of access required), on the type of the underlying system or architecture, or on the load of the client's host.

3 DISTRIBUTED SHARED OBJECTS

Object Model: The Globe architecture introduces **distributed shared objects (DSO)** In this model, processes interact and communicate through DSO. Objects provide methods made available through interfaces. Objects are passive. Activity is provided by processes that can share objects and can invoke their methods concurrently. A major distinction with other object-based models is that an object's state can be physically distributed through *local objects*. A local object resides in exactly one address space and communicates with other local objects to form a distributed object. State and operations on that state are completely encapsulated by the object, so 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.

Structure of the Distributed Shared Object: A distributed shared object is a collection of distinct local objects which consist of the object's current value or **state**, the object's **methods** and a collection of **interfaces**. To simplify the implementation of new distributed objects, Globe developers have proposed a composite local object that consists of four (sub)objects [3]: a *communication object* ,a *replication object* a *semantics object* and a *control object* .

*Semantics Object:*This is a local object that implements (part of) the actual semantics of the distributed object. A developer is responsible for constructing a class object for each different kind of semantics object that is part of the distributed object.

Communication Object: This is generally a system-provided local object. It is responsible for handling communication between parts of the distributed object that reside in different address spaces, offering either point-to-point communication, multicast facilities, or both.

*Replication object:*The global state of the distributed object is made up of the state of its various semantics objects. Semantics objects may be replicated for reasons of fault tolerance or performance. The replication object is responsible for keeping these replicas consistent according to some replication strategy.

Control Object: The control object takes care of invocations from client processes, and controls the interaction between the semantics object and the replication object.

Binding a Distributed Shared Object: Before a process can use a distributed object, it has to *bind* to that object. Names are used to allow binding. During binding, code and data needed to support the object are loaded into the process' address spaces. Multiple processes can share an object by binding to it simultaneously. The code that is loaded during binding is contained in a *class object*, which represents a class of object with the same implementation. A class object is a local object which contains the implementation of the methods of such objects. It is used to instantiate a new local object representing the distributed object. This local object is initialized with the distributed objects identifier and the address of the chosen communication end-point. The distributed object can now be accessed through this local object. Depending on the actual implementation of the class object, the local object can act as a proxy forwarding requests to another address space, or it can get a copy of the current state of the distributed object and act as a new replica.

4 DISCUSSION AND CONCLUSION

The two models, details of which are discussed above, differ from other models in the sense that an object's state may be distributed and replicated across multiple address spaces. Distribution and replication schemes are encapsulated by an object; they form part of its implementation. Fragmented objects are mostly language independent. Distribution is achieved manually by allowing interfaces to act as object references that can be freely copied between different address spaces. An important difference with Globe's distributed shared objects, is that fragmented objects make use of relative object references. In contrast, Globe's object handles are absolute and globally unique The most important difference between the two models is that fragmented objects have not been designed for wide-area networks. For example, there are no facilities for incorporating object-specific replication strategies. Yet, in Globe model, partitioning, replication, and migration of an object's state is supported on a per-object basis

REFERENCES

1. M. Makpangou, Y. Gourhant, J.P.LeNarzul, M. Shaphiro, "Structuring Distributed Applications as Fragmented Objects", Technical Report 1304, INRIA, Jan., 1991.
2. M. Makpangou, Y. Gourhant, J.P.LeNarzul, M. Shaphiro, "Fragmented Objects for Distributed Abstractions", IEEE Software, Oct., 1991.
3. M.V.Steen, P.Homburg,A.S.Tanenbaum,"The Architectural Design of Globe: A Wide-Area Distributed Systems", Internal Report IR-422, Mar., 1997.
4. M. V. Steen, F.J. Hauck, G. Ballintijn, A.S. Tanenbaum. "Algorithmic Design of the Globe Wide-Area Location Service" Technical Report IR-440, December 1997.
5. M. V. Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. "Locating Objects in Wide-Area Systems." IEEE Communications Magazine, January 1998, pp. 2-7.