Integrating Distributed Composite Objects into Java Environment

Guray Yilmaz[1] and Nadia Erdogan[2]

[1] Turkish Air Force Academy, Computer Eng. Dept., Yeşilyurt,
34149 İstanbul, Turkey
g.yilmaz@hho.edu.tr

[2] Istanbul Technical University, Electrical-Electronics Faculty,
Computer Engineering Dept., Ayazaga,
80626 İstanbul, Turkey
erdogan@cs.itu.edu.tr

Abstract. This paper introduces a new programming model for distributed systems, distributed composite objects (DCO), to meet efficient implementation, transparency, and performance demands of distributed applications with cooperating users connected through the internet. It allows the representation of an object as a collection of sub-objects and enhances the object distribution concept by implementing replication at the sub-object level and only when demanded. DCOBE, a DCO-based programming environment, conceals implementation details of the DCO model behind its interface and provides basic mechanisms. An important feature of the programming framework is transparency.

1 Introduction

This paper introduces a new programming model for distributed systems, *distributed composite objects (DCO),* to meet efficient implementation, transparency, fault tolerance and performance demands of cooperative applications with users connected through the internet. DCO model incorporates two basic concepts. The first concept is **composition**, by which an object is partitioned into *sub-objects (SO)* that together constitute a single *composite object (CO)***.** The second basic concept is **replication**. Replication extends the object concept to the distributed environment. Sub-objects of a composite object are replicated on different address spaces to ensure availability and quick local access. Decomposition of an object into sub-objects reduces the granularity of replication. To a client, a DCO appears to be a local object. However, the distributed clients of a DCO are, in fact, each associated with local copies of one or more sub-objects and the set of replicated sub-objects distributed over multiple address spaces form a single distributed composite object.

A software layer, ***D****istributed* ***C****omposite* ***O****bject* ***B****ased* ***E****nvironment (DCOBE)* provides a programming framework that is based on the DCO model [5]. DCOBE is a middleware built on Java Virtual Machine and presents functionalities that facilitate

T. Yakhno (Ed.): ADVIS 2004, LNCS 3261, pp. 322–331, 2004.
© Springer-Verlag Berlin Heidelberg 2004

the development of internet wide distributed applications, through a well-defined interface.

An important feature of the programming framework is transparency. Users of DCOs acquire the benefits of a centralized environment as DCOBE takes care of issues such as distribution and replication of object state, management of consistency, and concurrency control. They are automatically programmed separately from the application code, thus enabling developers to concentrate on the semantics of the application they are working on.

The paper is organized into six sections. Section 2 presents the distributed composite object model. The structure of a DCO is explained in Section 3. Section 4 presents developing steps of a DCO. Related work is explained in Section 5. Finally, Section 6 presents our conclusion.

2 Distributed Composite Object Model

The DCO model allows applications to describe and to access shared data in terms of objects whose implementation details are embedded in several sub-objects. Each sub-object is an elementary object, with a centralized representation, or may itself be a composite object. Several sub-objects are grouped together in a container object to form a composite object.

The developer of the composite object distributes the object's state between multiple sub-objects and uses them to implement the features of the composite object. SOs of a composite object are replicated on different address spaces to ensure availability and quick local access. A CO is first created on a single address space with its constituent SOs. When a client application on another address space invokes an operation on a CO which triggers a method of a particular SO, the state of that SO only, rather than that of the whole CO, is copied to the client environment. With this replication scheme, SOs are replicated dynamically on remote address spaces upon method invocation requests. The set of SOs replicated on a certain address space represents the CO on that site. Thus, the state of a CO is physically distributed over several address spaces. Active copies of parts, or whole, of a composite object can reside on multiple address spaces simultaneously. We call this conceptual representation over multiple address spaces a *distributed composite object*.

Fig.1 depicts a DCO that spreads over four address spaces. It is initially created on *Site2* with all of its sub-objects (SO1, SO2, and SO3), and is later replicated on three other sites, with SO1 on *Site1*, SO2 and SO3 on *Site3*, and SO1 and SO3 on *Site4*. The three sites contribute to the representation of the DCO. The set of address spaces on which a DCO resides evolves dynamically as client applications start interactions on the target CO.

Clients see the interface, which the developer has defined for the composite object, rather than the interfaces from the collection of embedded sub-objects. Therefore, from the client's point of view, a composite object is a single shared object that has only one access point, a single interface. He is not aware of its internal composition and, hence, has no explicit access to the sub-objects that make up its state. This restriction is an important aspect of our model and allows the object

developer to dynamically adapt composite objects to changing conditions. The developer may add new sub-objects to a composite object to extend its design, remove existing ones or modify the implementation of some, without affecting the interface of the composite object. Thus, dynamic adaptation of the object over time becomes possible, without affecting the applications that use it.

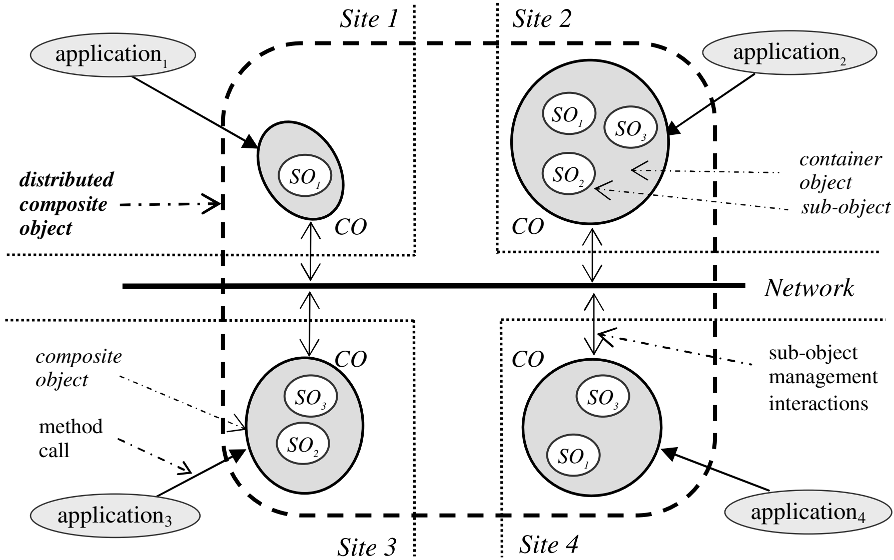

Fig. 1. A distributed composite object that spreads over three sites

Clients of a DCO are aware neither of its composition, nor of its distribution. As the objects in our model are passive objects, a client accesses a DCO by invoking methods in the interface provided by the object. Invocations are ordinary local object invocations as the client has a local implementation of the object in its address space. Multiple clients may access the same object simultaneously. When the state of an object is modified, all replicas are kept consistent through consistency management protocols that involve remote interactions.

3 The Structure of a DCO

We have defined an enhanced object structure to deal with implementation issues and thus provide the object developer and its clients with complete transparency of distribution, replication and consistency management. This structure includes two intermediate objects, namely, *a connective object* and *a control object*, which are inserted between the container object and each target sub-object.

Connective and control objects cooperate to enable client invocations on DCOs. A connective object is responsible for dynamic client to object binding which results in the placement of a valid replica of a sub-object in the caller's address space. A control

object is a wrapper that controls accesses to its associated replica. It implements coherence protocols to ensure consistency of sub-object state. A client object invocation follows a path through these intermediate objects to reach the target sub-object after certain control actions.

An Automatic Class Generator (ACG) that has been developed in the context of this work is used to generate classes for connective and control objects from interface descriptions of sub-objects. Hence, the developer has to focus only on the design of the sub-objects that make up a composite object. The others are generated automatically, according to the coherence protocol specified by the developer.

3.1 Connective Object

The connective object is the target of all local client invocation requests. Structurally, it is an object with the same abstract type and implements the same interface as the sub-object it is associated with. For a client invocation to be possible, it is necessary that the client bind to that object. Each connective object contains a reference that points to the control object of the referenced sub-object. If the reference is bound, it means the control object is already present. The connective object forwards the invocation request to the control object. It is the control object's task to make a valid replica of the sub-object available locally. In case the reference is null, a copy of the control object is fetched and the reference is updated.

3.2 Control Object

The control object is located between the connective object and the local implementation of the sub-object and exports the same interface as the sub-object. It receives both local and remote invocation requests and directs them to the local sub-object. Consistency problems arise as sub-object replicas on different address spaces are modified. The control object is responsible for the management of consistency of object state and concurrency control to ensure mutually exclusive access. It implements certain coherence and access synchronization protocols before actually allowing a method invocation request to execute on the sub-object it is associated with. The system uses *entry consistency* [4] for memory coherency.

There are two approaches in the synchronization of write accesses to objects so that no client reads old data once a write access has been completed on some replica: *write-update* and *write-invalidate* [2]. Write-update broadcasts the effects of all write accesses to all address spaces that hold replicas of the target object. In the write-invalidate scheme, on the other hand, an invalidation message is sent to all address spaces that hold a replica before doing an update. Upon receipt of an invalidation message, objects are simply marked invalid, but not immediately retrieved. Clients ask for updates as they need them. DCO model implements both coherence protocols. The developer chooses the one which suits the requirements of his application the best and the control object’s class is generated accordingly by the ACG.

The interface of a control object is divided into two parts. The first part is identical to that of the sub-object and its methods are called by the connective object to access the sub-object replica. The second part is an upcall interface that is used to implement

the coherence and access synchronization protocols. Control objects on different sites communicate through this interface to keep the object state consistent.

The control object implements a method invocation request in three main steps. They are briefly explained below, omitting specific details.

Step 1. Get access permission: This step involves a set of actions, possibly including communication with remote control objects, to obtain access permission to the sub-object. It is blocking in nature, and once activity is allowed, it proceeds to step 2. The control object recognizes the type of the operation the method invocation involves, either a write *(W)* operation that modifies the state of the object or a read *(R)* operation that does not, and proceeds with this information. The object developer specifies the access type of each method with an appropriate keyword *(R/W)* that follows the method signature in the interface declaration of a sub-object. For a *R-type* of invocation request, the actions are similar for both types of coherence protocols. They result in the placement of a valid sub-object copy in the local address space if one is not already present and return a permission to proceed, if currently there is no active writer to the object and the list of pending requests is empty. The client is added to the valid list of the target sub-object. If those conditions do not hold, the client is suspended temporarily and the request is queued in a waiting list.

A *W-type* of invocation request is queued for both coherence protocols, if a writer is already active or the pending list of requests is not empty. Otherwise, for the write-invalidate protocol, all reader clients in the valid list are sent an invalidation message and the valid list is purged. The operation returns a valid copy of the target sub-object on the caller's address space, if not already present, along with its ownership granting write access permission to the invoker.

Step 2. Invoke method: This is the step when the method invocation on the local sub-object takes place. After receiving permission to access the target sub-object, the control object issues a call which received from the connective object.

Step 3. Complete invocation: This step completes the method invocation after issuing update requests for remote replicas on the valid list to meet the requirements of write-update protocol. After the call returns, the control object activates invocation requests that have blocked on the object. The classical multiple-reader/single-writer scheme is implemented, with waiting readers given priority over waiting writers after a write access completes and a waiting writer given priority over waiting readers after the last read access completes.

4 Developing a New DCO

In this section, we will demonstrate with an example how a DCO is created and how it is accessed from a remote Java application. No language extensions or system support classes are required during coding. The developer generates code as he does for a conventional centralized application. As an example, we assume a DCO named `Employee`, whose state and implementation is distributed between three sub-objects: `Person`, `Account`, and `Job`. Fig.2 shows the class definition a developer would prepare for the container object `Employee`.

```
public class Employee {
   //Definitions of the sub-objects and other variables
    Connective_Account bankAccount;
    Connective_Person  person;
    Connective_Job     job;
    float              amount;
    .....         .......
    public Employee() {
          bankAccount = new Connective_Account();
          person      = new Connective_Person();
          job         = new Connective_Job();
          amount      = 0;
          ...................
    }
    public void deposit_BankAccount(float amount) {
          bankAccount.deposit(amount);
    }
    public void withdraw_BankAccount(float amount) {
          bankAccount.withdraw(amount);
    }
    public float balance_BankAccount() {
          amount = bankAccount.balance();
          return amount;
    }
    ..........................
}
```

Fig. 2. Container class definition for DCO Employee

```
public class Sub_Account {
     float total = 0;
     public void deposite(float amount) {
            total = total + amount;
     }
     public void witdraw(float amount) {
            total = total - amount;
     }
     public float balance() {
            return total;
     }
}
```

Fig. 3. Code for sub-object class `Sub_Account`

In this example, for clarity, we have only included methods that utilize the sub-object class `Account` and their contents are extremely simplified as to include only a single method invocation on the target sub-object. Actually, there is no restriction on the semantics of the methods of a DCO. Next, the class definitions and interface descriptions for each sub-object are prepared. Class definitions are typical Java definitions, except for the prefix 'Sub_' that precedes the name of the class. Fig.3 shows the code for sub-object `Sub_Account`.

Interface descriptions list the methods the sub-object implements for internal use. At this point, the developer is required to identify, for each method, the type of operation its invocation involves using an appropriate symbol: W (short for Write) for one that modifies the state of the object and R (short for Read) for one that does not. This is the only difference between an RMI and DCO interface description. Fig.4 shows the interface description for sub-object `Sub_Account`.

```
public interface Sub_Account {
    public void deposit_W(float amount);
    public void withdraw_W(float amount);
    public float balance_R();
}
```

Fig. 4. Interface description for class `Sub_Account`

The next step involves the generation of class definitions for connective and control objects of each sub-object. The Automatic Class Generator creates them automatically using the information extracted from interface descriptions of the sub-objects. Fig.5 and Fig.6 show the class definitions for the connective object and the control object generated respectively from interface `Account`.

```
public class Connective_Account {
     int obj_id;
     Control_Account control_object;

     public Connective_Account() {
           control_object = new Control_Account ();
           obj_id = control_object.get_id();
     }

     public void deposite(float amount) { ....... }

     public void witdraw(float amount)  { ....... }

     public float balance() {
           if (control_object == null) (1)
               control_object = (Control_Account)
               dcobe_server.get_control_object(obj_id);(2)
           return control_object.balance();(3)
     }
  }
```

Fig. 5. Class definition for the connective object; `Connective_Account`

The connective object of `Sub_Account` is named as `Account` and implements the same interface as `Sub_Account` because a method invocation on a sub-object is actually directed to its connective object first. It contains a pointer to the control object. Whenever one of its methods is activated, it first checks the binding of the

reference to the control object ((1) in Fig. 5). If the reference has not yet been bound, a call is issued, which returns copies of both the control object and the sub-object (2). The method invocation is then forwarded to the control object (3).

```
public class Control_Account {
      Sub_Account  subObject;
      int    obj_id, server_id;

      public Control_Account() {
             subObject = new Sub_Account();
             server_id = dcobe_server.get_serverId();
             obj_id = dcobe_server.
                          register_object(this, subObject);(1)
      }

      public void deposite(float amount) { ....... }

      public void witdraw(float amount)  { ....... }

      public float balance() {
             access_right(R);  (2)
             float account = subObject.balance();(3)
             access_end(server_id, R);
             return account;
      }
}
```

Fig. 6. Class definition for the control object; `Control_Account`

The control object registers the sub-object and, in return, receives a unique identifier, obj_id ((1) in Fig.6), which is used by the connective and control objects on successive accesses to the sub-object. Control object implements coherence protocols to ensure consistency of the sub-object's state. After getting access permission through a lock (2), the method is invoked on the sub-object (3). The control object also includes internal methods (upcalls not presented in Fig.6) that may be invoked by DCOBE_Server in order to check the status of the lock on the sub-object and block a lock request from a remote node until the lock is explicitly released.

After completing the class definitions for a composite object class, these class files are made available to other nodes by an HTTP-server so that they may be dynamically loaded from remote addresses. The following piece of code first instantiates a composite object in an application program. Immediately afterwards, connective objects, control objects and the sub-objects are automatically created on that node (1). Second, the newly created composite object is registered with a name (2), and third, in order to make class definitions dynamically loadable, class base information is also registered (3).

```
(1) employee = new Employee();
(2) dcobe_server.register(employee, "John");
(3) dcobe_server.register_class("Employee",
                                          "http://Class_Base/");
```

For a DCO to be accessible from a remote node, a user has to bind to the object through a lookup operation, that is, a registered composite object needs to installed in its address space. With this process, connective objects are also installed on the requesting node automatically. However, a local method invocation on the object becomes possible only after the control object and a replica of a sub-object is loaded.

The following piece of code loads the class `Employee` dynamically (4), and binds to one of its instances, `John`, (5). Now, the remote user is ready to invoke a method on the distributed composite object (6). Only a replica of the sub-object `bankAccount` will be loaded to the user's address space. In addition, according to the coherence protocol, all other replicas of `bankAccount` will either be invalidated or updated after this method completes.

```
(4) Employee = dcobe_server.load_class("Employee");
(5) employee = dcobe_server.lookup("John");
(6) employee.deposit_BankAccount(1000);
```

5 Related Work

Our work has been influenced closely by the SOS [1] and Globe [6] projects, which support state distribution through physically distributed shared objects. The SOS system is based on the Fragmented Object (FO) model [3]. The FO model is a structure for distributed objects that naturally extends the proxy principle. FO is a set of *fragment objects* local to an address space, connected together by a *connective object*. Fragments export the FO interface, while connective objects implement the interactions between fragments. A connective object embodies the consistency and coherence properties of the distributed object and provides an internal communication substrate for the FO. Even though the work hides the cooperation between fragments of a FO from the clients, the programmer is responsible to control the details of the cooperation. He has to decide if a fragment locally implements the service or is just a stub to a remote server fragment. FO hides data replication and consistency management from the user, but those details are exposed to the implementer.

One of the key concepts of the Globe system is its model of *Distributed Shared Objects* (DSOs) [6]. A DSO is physically distributed, meaning that its state might be partitioned and replicated across multiple machines at the same time. However, all implementation aspects are part of the object and hidden behind its interface. For an object invocation to be possible, a local object is bound in the client's address space. A local object may implement an interface by forwarding all invocations, as in RPC client stubs, or through operations on a replica of the object state. Local objects are partitioned into sub-objects, which implement distribution issues such as replication and communication, so developers concentrate on the semantics of the object.

The major difference between our work and above projects is that, they both do not support the composite object model and caching is restricted to the state of the entire object. However, the DCO model allows the representation of an object as a collection of sub-objects and enhances the object distribution concept by implementing replication at the sub-object level, providing a finer granularity. To the best of our knowledge, there is no programming framework that supports replication

at the sub-object level. Also, in both projects, deciding where and when to create a replica is left to the application. Even though Globe provides a general mechanism for associating replication strategies with objects, at present, a developer has to write his own implementation of a replication sub-object. DCO, in contrast, replicates sub-objects at all sites they are used and management of consistency of state and concurrency control is transparent to both object developers and users. Dynamic loading of sub-objects is also a feature that is not supported by either of the projects. Another important difference is that, DCO developer may add new sub-objects to a composite object to extend its design, remove existing ones or modify the implementation of some sub-objects without affecting its users.

6 Conclusion

In this study, we proposed a new object model, distributed composite object, for internet-wide collaborative computing. This distributed composite object model allows users to describe applications in terms of a single composite object whose implementation details are embedded and encapsulated in different types of sub-objects. In this model, a distributed object is not an entity running on a single machine, possibly with full copies on other machines, it is partitioned on several sub-objects and these sub-objects are replicated across multiple sites at the same time.

Applications are developed using Java language as centralized manner and made available on the internet. Applications are dynamically deployed on client nodes and the distributed objects are transparently shared among applications. This allows users to deal with the diverse environments that exist in a wide area network and to separate applications from the implementation of the objects. Application programmers may really concentrate on the application specific logic. This approach makes distribution and replication almost transparent to the application programmers.

References

1. Shapiro, M., Gourhant, Y., Herbert, S., Mosseri, L., Ruffin, M. and Valot, C.: SOS: An Object-Oriented Operating System-Assessment and Perspectives. Computing Systems, Dec. 2(4) (1989) 287-338.
2. Mosberger, D.: Memory consistency models. Operating Systems Review, Jan. (1993) 18-26.
3. Makpangou, M., Gourhant, Y., LeNarzul J.P. and Shaphiro, M.: Fragmented Objects for Distributed Abstractions. in: T.L. Casavant and M. Singhal (eds.), Readings in Distributed Computing Systems, IEEE Computer Society Press (1994) 170-186.
4. J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. ACM Transactions on Computer Systems, Aug. 13(3) (1995) 205–243.
5. Yılmaz G., Distributed Composite Object Model for Distributed Object-Based Systems. PhD Thesis, Istanbul Technical University Institute of Science and Technology, Istanbul, Turkey (2002).
6. Bakker, A., Kuz, I., Steen, M.V., Tanenbaum, A.S. and Verkaik. P.: Design and Implementation of the Globe Middleware. Technical Report IR-CS-003, June (2003).