DCOBE: Distributed Composite Object Based Environment

Güray YILMAZ[1], Nadia ERDOGAN[2]

[1] Computer Engineering Department, Turkish Air Force Academy
34807, Yesilyurt, Istanbul–TURKEY
Tel: +90 212 6622288
Fax: +90 212 5565118

e-mail: g.yilmaz@hho.edu.tr

[2] Computer Engineering Department
Electrical-Electronics Faculty, Istanbul Technical University
34469, Ayazaga, Istanbul–TURKEY
Tel: +90 2122853592
Fax: +90 212 2853679

e-mail : erdogan@cs.itu.edu.tr

Corresponding Author: Nadia Erdogan
e-mail: erdogan@cs.itu.edu.tr
Postal Address: Computer Engineering Department
Electrical-Electronics Engineering Faculty
Istanbul Technical University
34469, Ayazaga, Istanbul-TURKEY
Telephone: 0 212 285 35 92
Fax: 0 212 285 36 79

Keywords: *distributed composite object, distributed object-based systems, composition, replication, cooperative computations.*

DCOBE: Distributed Composite Object Based Environment

Güray YILMAZ[1], Nadia ERDOGAN[2]

[1] Computer Engineering Department, Turkish Air Force Academy

34807, Yesilyurt, Istanbul–TURKEY

Tel : +90 212 6622288

Fax : +90 212 5565118

email: g.yilmaz@hho.edu.tr

[2] Computer Engineering Department,

Electrical-Electronics Engineering Faculty, Istanbul Technical University

34469, Ayazaga, Istanbul–TURKEY

Tel : +90 212 2853592

Fax: +90 212 2853679

email: 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. DCO model incorporates two basic concepts. The first is composition, by which an object is partitioned into sub-objects that together constitute a single composite object****.*** *The second one is replication, which extends the object concept to the distributed environment. 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 and only when demanded.* ***DCOBE****, a DCO-based programming environment, conceals implementation details of the DCO model behind its interface and provides basic mechanisms for object composition, distribution and replication of object state, consistency management, concurrency control and dynamic deployment of restructured objects. It facilitates the design of distributed applications, reducing significantly the overall time for development by taking care of distributed system issues. DCOBE, being implemented on JVM, provides an environment that works on heterogeneous platforms. A distributed application is developed using the Java language in a centralized manner and then made available on the internet. Objects are dynamically deployed to requesting client nodes. This allows users to deal with various environments that exist in a wide area network and to separate applications from the implementation of shared objects.*

Keywords: *distributed composite object, distributed object-based systems, composition, replication, cooperative computations.*

1. Introduction

The increase in the number of interconnected computers and networks has led the community of software developers to distribute applications in order to support cooperative work. It has, therefore, become important to focus on distributed systems software that provides an infrastructure to enable user interactions and collaborations on common goals and shared data. Such software facilitates implementation because programs are written on top of a high-level execution environment. Programmers are no longer concerned with the complexities of distribution and maintenance of shared data, while maintaining acceptable levels of interactive performance.

This paper introduces a new programming model for distributed systems, *distributed composite objects (DCO)* [1], to meet efficient implementation, transparency, fault tolerance and performance demands of cooperative applications with users connected through the internet. The distributed composite object 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**)* [2] provides a programming framework that is based on the DCO model. DCOBE is a middleware built on Java Virtual Machine and presents functionalities that facilitate 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 eleven sections. Related work is explained in Section 2. Section 3 presents object composition and replication concepts. The distributed composite object model is described in Section 4. Management of sub-objects is explained in Section 5. Section 6 presents DCOBE architecture, while Section 7 explains the creation steps of a distributed composite object. An automatic class generator is presented in Section 8. A sample application is described in Section 9. Section 10 provides an evaluation of DCOBE. Finally, Section 11 presents our conclusion and plans for future work.

2. Related Work

The DCO model has benefited from the large amount of research dedicated to consistency strategies on shared memory systems [3]. DCOBE provides a flexible framework for associating various replication coherence protocols for different sub-objects of a composite object.

The majority of work on distributed object models follows an approach that allows clients to transparently access an object through *remote method invocations*. Related work includes DCOM [4], CORBA [5] and JAVA RMI [6]. In all cases, an object is presented to remote

clients by means of a *proxy* that is installed at the client and offers the same interface as the actual object. Remote method invocation basically uses the same technique as RPC [7] and, when combined with object serialization (JAVA RMI), it forms a powerful technique for transparently invoking remote objects. The main difference with all these models to our work is that they provide remote objects, rather than physically distributed objects and, hence, fail to handle complex distribution issues such as replication and concurrency control.

Smart proxies [8,9], on the other hand, add more functionality into stubs, e.g., adding caching mechanisms to reduce communication overhead and latencies, or forwarding method invocations to some member of a replica group for the purpose of load balancing or fault tolerance. Current smart proxy implementations usually either modify the middleware such that compatibility is no longer maintained, or they use means such as portable interceptors. In the latter case, a signifcant overhead in remote invocation mechanisms is introduced by adding additional levels of indirection on the client side. Smart proxies stay closer at the traditional client-server structure.

An alternative approach is to fully encapsulate distribution in an object, which leads to a model of partitioned objects. The partitioned object model, used by Spring [10], SOS [11], Globe [12] and AspectIX [13] allows combining multiple distributed parts into a single distributed object with a single identity.

A model called *Subcontract* is offered in Spring [14]. The Spring Subcontract structures an object around the so-called object representation, a table of method entries and a Subcontract. Spring offers two replication Subcontracts; the replicon and caching Subcontracts. The replicon is more basic; it binds each client to a replica and permits multi invocation on replicas. The caching Subcontract is more elaborate. However, as a general mechanism, it is too limited. Subcontracts do not provide generic support for a variety of consistency protocols

and other requirements inherent for object cashing. For example, it is hard to develop Subcontracts that keep a group of objects shared by several clients consistent.

Our work has been influenced closely by the SOS, AspectIX and Globe projects, which support state distribution through physically distributed shared objects. The SOS system is based on the Fragmented Object (FO) model [15]. 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 required consistency and coherence properties of the fragmented object and provides an internal communication substrate for the FO. In this view of FO, a given address space appears only to contain local references. However, the lowest level of the FO structure, the connective object, encapsulates communication facilities, which are equivalent to remote invocation or message passing mechanisms. Even though the work hides the cooperation between fragments of a FO from the clients, the programmer of the FO 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 of an object, but those details are expected to be implemented by the developer of an object.

AspectIX describes a middleware system that integrates the concept of fragmented objects into a CORBA environment. Their fragmented objects support implicit binding using the ORB's marshalling mechanism by defining customized IOR (Interoperable Object Reference) profiles, while full interoperability with traditional CORBA applications is maintained.

One of the key concepts of the Globe system is its model of *Distributed Shared Objects* (DSOs) [16]. In Globe, processes interact and communicate through DSOs. Each shared object offers one or more interfaces. A Globe object 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 process has to bind to an object, which results in placement of a local object in the client's address space. A local object may implement an interface by forwarding all method 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, allowing object developers to concentrate on the semantics of the object.

SOS, Globe and AspectIX projects provide similar frameworks for partitioning of an object and implementations of replicating local objects. The major difference between our work and SOS and Globe 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 other programming framework that supports replication at the sub-object level. Also, in SOS and Globe, 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. DCOBE, 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. Another essential difference is the binding mechanism used for getting access to an object. Dynamic loading of sub-objects is a feature that is not supported by either of the projects. Such implicit binding is a prerequisite for true object-based programming as object references have to be transparently passed around an application.

3. Object Composition and Replication

The distributed composite object model includes two basic concepts: composition and replication. Composition allows aggregating multiple sub-objects into a single composite object**.** Replication extends the composite object concept to the distributed environment. Sub-objects of a composite object are replicated on different address spaces when they are referenced, rather than having the whole composite object being accessed remotely. This section elaborates on object composition and replication concepts.

3.1 Object Composition

Composition, also referenced as aggregation, is a mechanism for forming an object *whole* using other objects as its *parts*. It reduces complexity by treating many objects as one object. Three basic properties of composition are identified as the following:

- *Configuration* – whether or not the parts bear a particular functional or structural relationship to one another or to the object they constitute.
- *Homeomerous* – whether or not the parts are of the same kind as the whole.
- *Invariance* – whether or not the parts can be separated from the whole.

To clarify the structural connections between an object whole and its parts, [17] describes in detail six different kinds of composition, based on particular combinations of those basic properties. Composite objects are objects that have an externally-distinct structure, and this structure can be addressed via the public interface of the composite object. The objects that comprise a composite object are referred to as *component* objects (sub-objects in this paper). A composite object has a single root object (container object in this paper), and the root references multiple children objects, each through an instance variable. Each child object can in turn reference its own children objects, again through instance variables. The instance that

constitutes a composite object belongs to classes that are also organized in a hierarchy. The composition hierarchy can span an arbitrary number of levels. If a composite object design has component objects that are themselves composite objects, a two-level composition hierarchy is created. This hierarchy could be repeated at several layers of composition.

Representing an object through the composition of several sub-objects can provide benefits to applications by improving manageability and performance. As it allows larger objects to be partitioned into smaller and more manageable units, the object designer gains the ability to apply the "divide and conquer" approach to data management. Furthermore, by limiting the amount of data to be examined or operated on, it provides performance benefits as well.

3.2 Replication

Distributed replication allows for multiple copies of an object to reside in several address spaces. It is, in general, an important approach to increasing availability, achieving fault-tolerance, and improving efficiency of a system. Replication reduces the cost of read operations that do not alter the object state, since it is possible to simultaneously execute such operations locally on multiple nodes. However, operations that modify the state of the object become more expensive because of coherence operations to maintain consistency.

Read-Replication : multiple readers/single writer strategy divides object invocations into two types: read accesses that do not change the state of the object and write accesses that modify the object. Thus, either one of the following two situations are allowed for at any time:

- *multiple nodes* with read-only replicas of the shared object - the object is replicated on two or more nodes and each node has read access to its copy while none of the nodes have write access, or

- *one node* with a read/write replica - no two nodes may be modifying separate copies of an object at the same time and any node that requests read access to an object is not allowed to if a writer to the object already exists.

As stated above, replication improves performance by allowing concurrent access to replicas at multiple nodes. However, if the concurrent accesses are not controlled, they may be executed in an order different from that expected. Memory coherence requires two conditions: a write must eventually be made visible to all nodes and writes to the same location must appear to be seen in the same order by all nodes[18]. Thus, to maintain the coherence of replicated objects, a mechanism that controls or synchronizes the accesses is necessary. A *consistency model* defines a specific kind of coherence provided by the system while a *coherency protocol* is responsible for managing object data so that the required level of consistency is actually provided. Consistency models define the order in which accesses to replicated data are seen by interested parties. A number of different models have been proposed in the literature, such as *sequential consistency*, *causal consistency*, *PRAM consistency*, *weak consistency*, *release consistency*, *and entry consistency* [18]. Consistency models can be divided into two major categories: *strict models* and *relaxed models*. In general, the stronger the consistency level, the higher is the latency its implementation produces [19].

The coherency protocol is responsible for managing replicated objects so that the conditions to provide consistency are satisfied. The main issue is the synchronization of write accesses to objects in such a way as to insure no application reads old data once a write access has been completed on some replica of the object. There are two approaches: *write-update* and *write invalidate* [20]. Write-update broadcasts the effects of all write accesses to all nodes that have replicas of the object. This approach is usually considered to be expensive since a broadcast is

needed on every write. In the write-invalidate scheme, on the other hand, invalidations are sent and modifications are requested. The basic concept is to send an invalidation message to all nodes that hold a replica before doing an update. Applications ask for updates as they need them.

4. Distributed Composite Object Model

The *distributed composite object* 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 either an elementary object with a centralized representation, or may itself be a composite object that comprises of further sub-objects. Several sub-objects are grouped together in a *container object* to form a composite object, as depicted in Figure-1. 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. The clients see the interface, which the developer has defined for the composite object, rather than the interfaces from the collection of embedded sub-objects. Methods in the interface of the composite object issue calls to sub-objects in order to carry out the functionality expected of it. Therefore, from the client's point of view, a composite object is a single shared object that has only one access point, the public interface it exports. He is not aware of its internal composition and, hence, has no explicit access to the sub-objects that make up their states. 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 whose methods client applications invoke. Thus, dynamic adaptation of the object over time becomes possible, without affecting the applications that use it.

The proposed model relies on replication. Sub-objects of a composite object are replicated on different address spaces to ensure availability and quick local access. A replica encapsulates a local copy of the replicated state in the address space of the client and offers an interface for internal access, to manipulate this state. A composite object is first created on a single address space with its constituent sub-objects, as on *Site2* in Figure-1. When a client application on another address space invokes an operation on the composite object which triggers a method of a particular sub-object, the state of that sub-object *only*, rather than that of the whole composite object, is copied to the client's environment. With this replication scheme, sub-objects are dynamically replicated on remote address spaces upon method invocation requests. The set of sub-objects replicated on a certain address space represents the composite object on that site. Thus, the state of a composite object is physically distributed over several address spaces. Copies of parts of, 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 (DCO)*. Figure-2 shows a DCO that spreads over four address spaces. It is initially created on *Site2* with all its sub-objects (SO1, SO2, and SO3), and is later replicated on three other sites, with only SO1 on *Site1*, SO2 and SO3 on *Site3*, and SO1 and SO3 on *Site4*. These four 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 composite object.

Clients of a DCO are aware neither of its composition, nor of its distribution. As the objects in our model are passive, a client accesses a DCO by invoking methods in the interface provided by the composite 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 DCO simultaneously. When the state of an object is modified, all replicas are kept consistent through consistency management protocols that involve remote interactions.

We need to distinguish the programmer who is the composite object developer, and the user, the client of the composite object. The model we propose allows clients to perform operations on distributed composite objects without needing to know that there exist several sub-objects inside and they are actually distributed physically on several sites. The object developer is responsible of the design of the inner structure of the composite object. The nature of the application will determine the structural connections between the composite object and its parts. The kinds of composition referred to in Section 3.1 provide a practical guide to how a programmer should partition an object into sub-objects. An important key to better performance would be partitioning an object into discrete sub-objects because a chain of method calls between sub-objects will require each target sub-object to be copied on the local host, thus degrading performance.

The following is a summary of the beneficial features of the DCO model:

Transparency: DCO model, with the support of the DCOBE middleware, conceals all implementation details behind its interface, as expected of the object-oriented programming paradigm. Both the developer and clients of a DCO are isolated from issues dealing with:

- distribution of object state,
- replication of object state,
- management of consistency of object state,
- management of concurrency control, and
- the underlying communication technology.

Dynamic adaptation: DCO model builds on the concept of separation of interface and implementation. Clients of objects depend on interfaces, not on implementations. This allows for modifications on the internal composition of a DCO without affecting users that are

presently bound to the object. Thus, object implementations may dynamically evolve to adapt themselves to changes in the environment.

Dynamic deployment: Objects are dynamically deployed to requesting addresses from a machine that holds a valid representation; thus object installation prior to execution is not required.

Reduced response time: Local accesses on sub-object replicas result in reduced method invocation time, especially for read or write access requests issued after the completion of the first write invocation, which inevitably introduces overhead due to getting access permission, concurrency control and consistency maintenance actions.

Conserved bandwidth consumption: Due to its replication policy, communication cost of a DCO is lower when compared to remote method invocation mechanisms. In addition, replication of only target sub-objects, not the entire object, and only when demanded, helps in conservation of bandwidth consumption.

Ease of use: Use of DCO incurs no additional task on the application developer as programming steps to create a DCO are very similar to those needed for a regular object.

4.1 Application Domain of DCO Model

Over the past decade, an evident shift from individuals to groups engaged in collaborative work has been observed in the design and implementation of computer systems. Collaborative information management, sharing and exchange on the WWW, collaborative design work of engineering teams, and collaborative authoring and editing are a few of such computer systems which involve work that may be distributed either physically, carried out at different places, or temporally, carried out at different times. In parallel with the trend, there is a critical need for tools supporting collaboration among distributed users with similar interests,

or who are part of the same group with some common purpose. The research field Computer Supported Collaborative Work (CSCW) focuses on techniques and tools to provide individuals working on related tasks with support to make distributed work more effective. Collaborative tools themselves need to be distributed and dynamic, and support the discovery and dissemination of information. The object composition and replication characteristics of the DCO model, together with the DCOBE framework, make it particularly suitable for the design and implementation of such computer systems, where participants require to access and share information without having to rely on any centralized repositories. Collaborative authoring and editing, distributed CAD packages, distributed information retrieval systems, and distributed news facility are a few examples that would benefit the features of the proposed model.

5. Management of Sub-objects

A DCO is structured as a collection of composite objects, each with a set of replicated sub-objects, which communicate with each other on issues for replication and consistency of state. 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. As illustrated in Figure-3, 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. In Figure-3, sub-objects have "*sub_*", control objects have "*control_*" and connective objects have *"connective_"* prefixes.

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.

5.1 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, that is, a local object implementing the object interface be installed in the client's address space. Each connective object contains a reference that points to the control object of the referenced sub-object. It also contains a unique *object-id* associated with the sub-object which allows it to be located and copied on the local host. 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, through a call to the DCOBE middleware system a copy of the control object is fetched and the reference is updated.

In order for a sub-object to be copied to a site for the first time, its class definitions should be made available. They may be loaded prior to the start of the application or the action may be postponed to execution time, to the point in time when they are actually needed. DCOBE allows class definitions to be loaded at runtime, by means of Java’s dynamic class loading facility through the system calls described in detail in Section 6.3.

The replication policy is independent of the frequency with which a sub-object gets referenced. Therefore, those sub-objects which are called very infrequently are copied on the local host as well. Once copied, DCOBE makes no attempt to remove a sub-object and completely relies on the Java garbage collector to clear objects which are no longer referenced. It would be interesting to keep statistics of the frequency with which a sub-object gets referenced to help the object developer to refine his object design, which would consequently lead to better system performance. Such an enhancement to the system may be considered as future work.

5.2 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. Due to its composite and distributed nature, the state of a DCO is the union of the states of its sub-objects. 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 [21] before actually allowing a method invocation request to execute on the sub-object it is associated with. The system implements *entry consistency* [22] to maintain the coherence of shared objects. In an entry consistent environment, object invocations that modify the object (write accesses) require synchronization. Two synchronization operations are defined to differentiate between entry and exit to critical regions which enclose write accesses to shared objects: an **acquire** operation tells the system that a critical region is about to be entered and a **release** operation indicates that a critical region has just been exited. Under this model, each replicated object is associated with a

synchronization variable, actually a lock, that must be explictly acquired and released to enforce concurrent accesses to happen sequentially.

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* [3]. 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. This increases latency on subsequent accesses, but decreases bandwidth consumption if the object is not accessed again or is invalidated several times before the next access. DCOBE implements both coherence protocols. The developer chooses the one which suits the requirements of his application the best and the control object is generated accordingly by the ACG.

Each sub-object in the system has a single owner, a unique node, which is either the creator of the object or holds the only writable copy of the object. The owner node also maintains a list of nodes, namely the *valid-list*, which holds valid replicas of the sub-object.

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 a standard 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, where the corresponding method of the sub-object replica is invoked. 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 symbol *(R/W)* that follows the method name in the interface declaration of a sub-object. For an *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 (a local implementation is not created before it receives its first call or the current replica may have been invalidated) 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. In the implementation of the write-update protocol, however, no invalidation takes place but the valid list is returned to the caller along with an access permission to enable further update operations on those replicas on remote addresses through remote invocation requests directed to nodes on the valid list.

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 the call which it had received from the connective object. If the implementation of the method of a sub-object further includes a method invocation on another sub-object, it is forwarded to the connective object of that sub-object and the same method invocation steps are repeated.

Step 3. Complete invocation: This step completes the method invocation after issuing update messages, which involve method call requests to 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 meanwhile 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. The system does not deal with situations that would result in starvation due to programmer errors, such as a non returning method call.

6. Distributed Composite Object Based Environment: DCOBE

The main objective of DCOBE is to present a convenient environment for the realization of distributed computations that utilize the DCO model. It provides a DCO-based programming environment, as depicted in Figure-4, which hosts various numbers of applications dispersed on several nodes, interacting and collaborating on common goals and shared data. DCOBE conceals implementation details of the DCO model behind its interface, allowing users to concentrate merely on application logic rather than on issues dealing with activity on a distributed environment. It provides the basic mechanisms for object composition, distribution and replication of object state, consistency management, concurrency control and dynamic deployment of restructured objects. It is a middleware architecture that is implemented on a

network of heterogeneous computers, each capable of executing Java Virtual Machine (JVM). Its place in the software hierarchy is between a Java application and the JVM.

DCOBE architecture consists of two main components that handle the core functionalities of the middleware: a system-wide coordinator *(DCOBE Coordinator–DC)* and a server component *(DCOBE Server-DS)* on each node that participates a DCO-based application in the distributed environment. Java RMI [23] is used as the communication mechanism for the interaction between the coordinator and the servers on different nodes. Figure-5 illustrates the architecture of DCOBE and the inner composition of its components.

6.1 DCOBE Coordinator – DC

DC is the component that initializes the DCO based execution environment and coordinates the interaction of DCOBE Servers. It runs as a separate process, which is explicitly started at a predefined network address such that it may be accessible by all servers that participate in the environment. It has a remotely accessible interface that allows distributed DS's to request services from it. When a DS is started, DC supervises a handshake protocol ensuring that each DS is initialized knowing the address of every other DS participating.

Being unique makes DC very critical as it plays a major role in the system. In order to protect the system against failures, we have added a secondary DC process as a backup unit to the DCOBE architecture, component BDC in Figure-4. It is initiated on another node and monitors the primary DC continuously, duplicating its information base. If the primary fails, the backup recognizes the situation and takes over, and introduces itself as the current primary DC by notifying all servers.

6.2 DCOBE Server – DS

Its main goal is provide execution support for DCO objects. A DS is actually instantiated within the context of each application and provides facilities that implement the distributed

composite object model. As a server is integrated in each client application, the application can directly perform method calls as both are in the same address space. Servers on remote address spaces cooperate to process application requests and to ensure consistency of the replicated DCO state.

6.3 DCOBE Services

DCOBE may provide service to several groups of users working on different applications. Location transparency is an important issue in a distributed environment. DCOBE allows applications to associate symbolic names to objects that are independent of location. A name service resolves user-defined names to *obj_id*'s which are globally unique and location independent. A location service relates each *obj_id* with the physical address where it can be contacted. Core functionalities of DCOBE are briefly described below:

Start a new application: Starting a new application requires the creation of a DCOBE server object at the requesting node. DC registers the new server under a unique *server_id* in a *server-access* table and broadcasts the '*server_id, node_address*' information pair to all current servers. The content of the *server-access* table is replicated at each node, providing the information for servers to communicate directly with one another.

```
DCOBE_Server dcobeServer = new DCOBE_Server();
```

Create and register a new DCO: A new DCO creation includes allocation of memory for the composite object on the creator's node and its registration within the system. A connective and a control object are created for each sub-object. DC associates the DCO with a unique *obj_id* that is used as the object reference on further access requests. The creation process completes with the registration of the previously created DCO under a user-defined name. This facility enables applications to associate symbolic names to objects, which are later used as reference in subsequent look-up operations from remote applications. An attempt to

register under a name that already exists, results in an error return, which alerts the programmer to repeat the operation with another symbolic name.

```
DCO_Class my_dco = new DCO_Class);

register(my_dco,"dco_name");
```

Execute a method call on a DCO: A replica of a target sub-object may or may not be present on a node at the time of method invocation request. DCOBE handles each case differently.

Case 1: This is the case where the requesting application is the creator of the DCO or has already received a local replica on its address space. The first step involves inquiring the existence of the control object that belongs to the sub-object, the target of the method call. This is necessary because replication of a DCO involves copying of the connective object only. The control objects and sub-objects are not copied until a particular method call requires their presence on the node. This mode of operation decreases bandwidth consumption, as certain sub-objects may never receive method calls on particular nodes, making their presence on those nodes unnecessary. If the control object of the target sub-object does not exist, a copy of the control object and the sub-object are retrieved from the server that holds a valid replica. The control object then proceeds with the method invocation process as described in Section 5.2

Case 2: This is the case where an application issues a method call for the first time, on a DCO which has been created by a remote application. As stated before, a method call may proceed only if a container object for the target is present on the current address space. This requires the availability of DCO class definitions in order to be able to create a container object of the specified type. Class definitions may be statically loaded prior to the start of the application, or they may be dynamically loaded during execution time, at the point where they are needed, using Java's *dynamic class loading* [24] facility. In DCOBE, users can implement their

applications using either one of these approaches. However, the first one imposes additional effort on the users and conflicts with distribution transparency. For the alternative approach, the creator of a DCO registers its class name and the URL location where the class definitions may be loaded with a DCOBE call to enable dynamic class loading (1).

```
register_class("DCO_Class",http://guray.hho.edu.tr/dco_classes/); (1)
```

The remote application loads class definitions with a DCOBE call that requires the name of the class as a parameter (2).

```
load_class("DCO_Class");                                    (2)
```

Next, to bind to the DCO, it issues a look-up request (3) for the remotely created DCO through its user-defined name. This call returns replicas of the container and the connective objects of the DCO to the new address space. From this point on, the method invocation process proceeds as described above in Case 1.

```
lookup("dco_name");                                         (3)
```

Modify the structure of a DCO: The structure of a DCO may be dynamically altered by addition of new sub-objects, or modification or complete removal of existing ones. The application which has created the DCO halts its execution temporarily and restarts with the new class definitions of the DCO. Meanwhile, distributed clients of the DCO continue their execution independently, possible using the older version of the object. The developer application reregisters the restructured DCO under its previous symbolic name with which it is known throughout the distributed system (4).

```
re_register(my_dco,"dco_name");                             (4)
```

This call notifies the DC to update its records associated with the DCO and to return a user list of the object. The local DS multicasts control messages to each DS on the list, which in turn, inform client applications of the object modification. An application is expected to

invoke the following method calls (5, 6) to reload the new class definitions and have the container and connective objects of each sub-object to be copied to its address space. However, an application may choose to act differently, depending on its use of the DCO.

```
load_class("DCO_Class");                    (5)

lookup("dco_name");                         (6)
```

Terminate an application: An application informs the system of its termination with the following DCOBE call (7);

```
leave_system();                             (7)
```

The coordinator transfers the ownership of any object the terminating application holds to another that is on the user list of the associated object. After removing representative data from several tables, the registration of the server is cleared.

7. Creation and Access of a Distributed Composite Object

In this section, we will demonstrate with an example how a DCO is created and how it is accessed from a remote Java application. As DCOBE middleware supports the DCO model directly, 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 class named `Employee`, whose state and implementation is distributed between three sub-objects: `person`, `account`, and `profession`. Figure-6 shows the class definition a developer would prepare for the container object `employee`, identifying its composite structure and its user interface. As explained before, the user of the object has no direct contact with its sub-objects. These objects are accessed over the methods exported by the container class interface. In this example, for clarity, we have included methods whose 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 by the object developer. Class definitions are typical Java definitions, except for the prefix '`Sub_`' that precedes the name of the class. Figure-7 shows the code for sub-object `Sub_Account`. Interface descriptions, on the other hand, 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 sub-object class interface description. Figure-8 shows the interface description for sub-object `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. Figure-9 and Figure-10 show the class definitions for the connective and the control objects generated respectively from interface `Account`.

The connective object of the sub-object is named as `Sub_Account` and implements the same interface as the sub-object because a method invocation on a sub-object is actually directed to its connective object first, as shown in Figure-3. The connective object is responsible for dynamic binding. It contains a pointer to the control object of the sub-object. Whenever one of its methods is activated, it first checks the binding of the reference to the control object (line denoted by (1) in Figure-9). If the reference has not yet been bound, a DCOBE call is issued, which returns copies of both the control object and the sub-object (line denoted by (2) in Figure-9). The method invocation is then forwarded to the control object (line denoted by (3) in Figure-9), which also implements the same interface as the sub-object.

When a composite object is instantiated, it creates instances of its sub-objects. However, as the flow of control through the constructor methods indicates, first a connective object is created, which in turn creates a control object, which finally creates the sub-object itself. The control object registers the sub-object and, in return, receives a unique identifier, `obj_id` (line denoted by (1) in Figure-10), 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 (line denoted by (2) in Figure-10), the method is invoked on the sub-object (line denoted by (3) in Figure-10). The control object also includes internal methods (upcalls not presented in Figure-10) 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 firstly instantiates a composite object in an application program. Immediately afterwards, connective objects, control objects and the sub-objects are automatically created on that node (8). Second, the newly created composite object is registered with a user-defined name (9), and third, in order to make class definitions dynamically loadable, class base information is also registered to the DCOBE Coordinator (10).

```
employee = new Employee();                                          (8)

dcobeServer.register(employee, "John");                             (9)

dcobeServer.register_class("Employee","http://Class_Base/");(10)
```

For a distributed composite object 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 using the dynamic class loading facility of Java (11) and binds to one of its instances, "`John`" (12).

```
Employee = dcobeServer.load_class("Employee");          (11)

employee = dcobeServer.lookup("John");                  (12)
```

Now, the remote user is ready to invoke a method on the distributed composite object (13). Only a replica of the sub-object a`ccount` will be loaded to the user's address space. In addition, according to the coherence protocol specified, all other replicas of `account` will either be invalidated or updated after this method completes execution.

```
employee.deposit_Account(1000);                         (13)
```

8. Automatic Class Generator

As manual coding of connective and control objects by the DCO developer is both inconvenient and error-prone, DCOBE supports automatic generation of these classes via an Automatic Class Generator (ACG). ACG generates class definitions for connective and control objects from the interface descriptions of sub-objects. The DCO designer prepares the class and interface descriptions for sub-objects and the container object that comprises of the sub-objects as he would for a centralized application. The connective and control objects belonging to each sub-object are then generated automatically using the ACG, which has an interface to accept two parameters: firstly the name of the file which contains the interface description of the sub-object and, secondly, a description of the coherence protocol desired to be applied to the sub-object; *update* to denote the write-update protocol and *invalidate* to

denote the write-invalidate protocol. As different applications have different consistency requirements, DCOBE allows different coherence protocols to be applied to different sub-objects of a given DCO. Consequently, class definitions of the control object of each sub-object are generated in agreement with the specified protocol. The following is a sample command which will generate the class files "*Connective_Account.java*" and "*Control_Account.java*" for the sub-object named *Account* from its interface descriptions from the file named "*Account_itf.java*" to conform to the mode operation required to implement the write-update protocol.

```
java Class_Generator Account_itf.java update
```

ACG consists of four main modules, namely, a lexical analyser (scanner), a syntax analyser (parser), a code generator, and an error reporter, as depicted in Figure-11. First, the scanner separates the characters in the interface description into tokens allowed by the Java language: keywords (public, interface, void, int, etc), identifiers to denote method names and parameters, and special characters such as the parenthesis or the colon. Next, the parser groups tokens together into syntactic structures and checks if a given sequence of tokens conforms to the grammar rules for interface description given in Figure-12. The parser is a recursive descent parser which is easily implemented for the rather small grammar. A call to the code generator is issued at the point when the parser recognizes in its input a sequence that can be reduced to the definition of a method according to Rule 4. of the grammar in Figure-12. The code generator produces code for connective and control objects by appending a standard sequence of code that is executed to implement each of the coherence and access synchronization protocols to sub-object method definitions to create the class files for connective and control objects.

Both scanner and parser modules report errors they discover to the error reporter module, which, in turn, issues an appropriate diagnostic message. For example, the scanner may be unable to proceed because the next token is misspelled, or the parser may be unable to recognize a structure because of a syntactic error, such as missing parenthesis.

9. An Application: Collaborative Book Writing

This section describes a typical application that can benefit the distributed composite object model; a real-time collaborative writing system that allows two or more physically dispersed people working together to produce a document, a book in our case. Collaborative book writing on the internet is not a new approach. There are several academic and industrial studies on collaborative writing such as IRIS [25], NetEdit [26], and Duplex [27]. Our goal is to show how the DCO model facilitates the design of the distributed application, reducing significantly the overall time for development by taking care of distributed system issues such as distribution, replication, consistency, concurrency and communication.

The application aims to develop a web-based collaborative environment that allows users to view and edit shared text at the same time. Collaborative writing involves periods of synchronous activity where the group works together the same time, and periods of asynchronous activity, where group members work at different times. The authors need to be able to read and update (write) any displayed document content. They also require immediate feedback on their actions. The system has the following characteristics:

- Low response time: Response to local user actions is quick (as quick as a single-user editor) and the latency for remote user action is short (determined mainly by external communication latency)

- Distributed environments: Cooperating authors reside on different machines connected by different communication networks.
- Unconstrained collaboration: Multiple users may concurrently and freely edit any part of the text at any time.

There are two aspects that should be regarded separately: the storage of the text objects and the author interface for manipulating text content.

9.1 Storage of Text Objects

In this application, the text of a book is a persistent document. It is represented by a single distributed composite object, namely `book`, which is shared among authors. It consists of a collection of related sub-objects, each representing a specific part of the book text: its *contents*, *preface, index, references*, and *chapters* denoted by sections and sub-sections. A chapter object is still a composite object that may include several *section* sub-objects, and similarly, these section sub-objects are composite objects that may further be divided into *sub-section* sub-objects. Figure-13 illustrates the composite structure of the `book` object at an instant of time when it is accessed by three sites.

The replicated architecture of the composite object model plays an important role in achieving good responsiveness and unconstrained collaboration. The shared sub-objects are replicated at the local storage of each participating author, as shown in Figure-13, so updates on different sub-objects are first performed at the local address space immediately, possibly simultaneously, and then propagated to remote sites, according to the consistency protocol. Multiple authors are allowed to access any part of the shared text. An invocation of a read type of method in the user interface results in the loading of the target sub-object into the local memory if it is not already present or if the present copy has been invalidated meanwhile. Any number of updates may be performed on the local replica, and the new content is submitted

with a write type of method invocation. This action may be performed whenever desired, after a word has been changed, or after the author has been working several hours on the content. The write-invalidate coherence protocol resolves conflicts that arise on simultaneous requests to update operations on identical parts of the book by different authors. While authors carry out these operations, they will not be aware of the composite and distributed structure of the object they are working on.

An interesting characteristic of the composite object `book` is that, initially at the start of the application, as no part of the book has been written yet, the set of sub-objects that will represent various parts of the book text is empty. As work progresses, the object *book* evolves, sub-object by sub-object, as new parts of the book text are added to the composite object. A sub-object named `parts` keeps track of the sub-objects thus added to the composite object in a table, where it stores the name of the part, a string, with a reference to the connective object associated with it. `parts` offers an interface with methods to add, remove, update, and search an entry in the table for internal use. It is created by the constructor method when an instance of `book` is created, and is automatically replicated on each address space where that instance is loaded.

9.2 Author Interface

Authors access object `book` via a user-friendly graphical interface, which contains methods that fall into three groups:

- **Content based methods:** They involve the reading, writing, or editing of the content of a part of the book text. Presently, a simple class `Editor` implements the functionality required, providing a text-based editing environment with filing facilities. As further work, we plan to integrate standard editing tools into the framework to provide authors with an enhanced working environment.

- **Attribute based methods:** They involve adding, modifying, or retrieving the attributes of a part, such as its author, last modification date, other authors' opinions, etc.
- **Document based methods:** They involve listing, adding, removing, renaming, or searching of parts or index entries. The *contents* sub-object is updated automatically if the invocation of a method requires such a modification.

An instance of the graphical user interface is reproduced in Figure 14.

10. Performance Evaluation

We evaluate DCOBE according to two main criteria: a discussion of DCOBE overheads compared to those for another distributed object model, namely, that employed by Java RMI and a study of behaviors of DCOBE with respect to different parameters, such as varying numbers of DCOs and sub-objects, varying sizes of sub-objects, and different consistency requirements.

To assess DCOBE performance, we conducted simple tests to measure the cost of basic operations on a LAN environment of ten INTEL architecture machines (Pentium IV 1.5 GHz with 512 MB RAM) connected through a 100 Mbit Ethernet and running Windows XP operating system. J2SDK 1.4.1 was used in the testing environment. Each time shown is a duration measured in milliseconds by calls to `System.currentTimeMillis()`. Tests were repeated ten times and the arithmetic average of the measurements are reported.

10.1 Comparison with Java RMI

In order to establish a baseline for Java-based communication to which DCOBE implementation can be compared, we carried out a similar test with Java RMI, using the

standard *java.rmi* package. The same object sample used for DCOBE test, a composite object with two sub-objects, was initiated on a node and remote method invocations were issued from a different node. Method argument data types were of small size and therefore their marshalling/unmarshalling had a little effect on overall invocation times. The test was repeated ten times and the arithmetic average of the measurements are reported. Table-1 summarizes the results we have obtained.

Creation and Registration: DCO creation time is approximately four times the time it takes for RMI object creation. This is due to the number of objects involved. As the sample DCO comprises of a container object and two sub-objects, together with the connective and control objects of these sub-objects, creation time actually includes the time it takes to create seven objects. However, DCO registration is much faster than RMI registration operation. For DCOBE, this is the time spent to register a DCO with a user defined name on a DS and DC. On the other hand, a RMI object is bound on a HTTP server registry with a user defined name.

Lookup and Binding: For RMI, it is the time required for the lookup operation of the RMI object's stub from the remote server. For DCOBE, it is the time taken to load class definitions of a DCO, followed by binding process which returns replicas of DCO container and connective objects to the address space of its remote client (three objects in this case). The DCOBE operation is twice slower than RMI operation because, against a RMI stub, three objects are loaded from the remote site.

Initial and Successive Method Invocation: This is the time it takes for a method invocation to complete. In case of RMI, all method invocations on the same object, regardless of being the initial, require the same amount of time to return (0.42 ms). However, with DCOBE, the time required for the initial and successive calls differ greatly. The initial invocation request involves the loading of the control object and the target sub-object to the address space of the remote client and the processing of access permission requests in order to satisfy consistency

and concurrency requirements. Consequently, the completion time of an initial method invocation on a DCOBE object is more than ten times than that of a RMI object. However, on successive invocations, as the request is handled locally on the local sub-object, completion time improves significantly.

Invalidate replicas: It is the time taken to send invalidation messages to replicas of a sub-object.

Update replicas: It is the time taken to send remote invocation requests on replicas of a sub-object.

The results indicate that, for an initial method invocation request, the time required to bind to an object and to complete the call for DCOBE is almost twice the time taken by Java RMI (7.51 ms for Java RMI and 13.01 ms for DCOBE). At first glance, this may appear as a serious disadvantage. However, after the initialization phase, as DCOBE method invocations are carried out locally, they complete in significantly shorter times while RMI invocation requests are forwarded to a remote address. It is apparent that network latency is the most influential factor in the overall invocation time for Java RMI, and considerable speedups are expected with DCOBE as its replication policy reduces the effect of network latency considerably.

10.2 DCOBE Performance Results

In this section, we describe the results of tests we have carried out to compare the performance of DCOBE under different situations.

Effect of increasing number of DCOs: This test compares system performance under increasing number of DCOs. Figure-15 illustrates the time needed for object creation and registration as the number of DCOs is increased. DCO creation time includes the time spent

to install the container object and the creation of each sub-object together with its connective and control objects. Registration records the object under a system wide recognized unique name. The test shows that the system can tolerate the existence of very large numbers of DCOs, the limit being physical capacity.

Effect of increasing number of SOs: This test compares system performance when the number of sub-objects a composite object comprises of is varied. Figure-16 illustrates the time needed for creation and lookup operations for DCOs with increasing number of sub-objects. As expected, creation time increases with the number of sub-objects as the total number of objects to be created (three objects per sub-object) grows. However, we recognize that the time needed to complete a lookup operation is not closely effected by the number of sub-objects owned by a DCO.

Effect of increasing sizes of SOs: This test compares system performance on implementation of a coherence protocol when object size is varied. Figure-17 illustrates the time needed to update, to invalidate, and to retrieve a valid copy after invalidation, for objects of various sizes. An update involves a method call request message to be sent to remote replicas on the object's valid list. That is, a message including method parameters as well is sent and the method is applied to all copies on remote nodes. With a constant number of replicas to be updated, we observe a slight variation in update time in relation to object size, interpreting it as the increased workload of the method for larger sized objects. The invalidation scheme, on the other hand, requires an invalidation message only to be sent to remote replicas before doing an update. With the number of remote replicas on the valid list of the object kept constant, the time needed for invalidation does not change, as it is not dependent on object size but rather on the length of the valid list. This is clearly observed in Figure-17, where measurement results for invalidation follow a constant value. However, with this scheme, applications ask for updates as they need them. Figure-17 depicts the variation in time

required to retrieve a valid copy after invalidation, in relation to object size. As the operation entails the transfer of the entire sub-object, retrieval time increases significantly as the size of the object is enlarged. As a result, we may conclude that the update scheme that DCOBE implements is more efficient, especially for large objects.

11. Conclusion

This paper presents a new object model, *distributed composite objects*, for distributed computing along with the design of a middleware architecture, DCOBE, which provides basic mechanisms for the development, deployment and management of distributed computations. The proposed model, with the support of DCOBE, allows for collaborative design and control of distributed applications. Users describe shared data in terms of composite objects whose implementation details are embedded and encapsulated in different types of sub-objects. Sub-objects of a composite object are replicated across multiple nodes when required. The main strength of the proposed model is that it provides the appropriate level of distribution visibility to the implementer of a distributed service, while hiding the distribution of sub-objects from its clients.

DCOBE, being implemented on JVM, provides an environment that works on heterogeneous platforms. A distributed application is developed using the Java language in a centralized manner and then made available on the internet. Objects are dynamically deployed to requesting client nodes. This allows users to deal with various environments that exist in a wide area network and to separate applications from the implementation of shared objects. The key benefits of the proposed object model are distribution transparency, ease of application development, conserved bandwidth consumption, and dynamic adaptation and deployment of shared objects. Future work will concentrate on extension of the DCO model

with a transparent data persistence service to incorporate fault-tolerance. We also plan to enhance the system by introducing access control policies for shared objects.

Acknowledgments: We would like to thank the referees for their helpful feedback.

References

1. Yilmaz, G. and Erdogan, N. (2001) A New Distributed Composite Model for Collaborative Computing, Proceedings of ISCIS XVI, Antalya, Turkey, 3-5 November, pp. 80-86.
2. Yilmaz, G. (2002) Distributed Composite Object Model for Distributed Object-Based Systems, PhD Thesis, Istanbul Technical University, Institute of Science and Technology, Istanbul, Turkey.
3. Mosberger, D. (1993) Memory Consistency Models. Operating Systems Review. 17(1), pp. 18–26.
4. Eddon, G. and Eddon, H. (1998) Inside Distributed COM. Microsoft Press, Redmond, WA.
5. OMG Document Technical Report formal/2002-12-02 (2002) The Common Object Request Broker Architecture: Core Specification-Revision 3.0.2. OMG. Framingham, MA, USA.
6. Wollarth, A., Riggs, R., and Waldo, J. (1996) A Distributed Object Model for the Java System. Computing Systems, 9(4), pp. 291-312.
7. Birrell, A. and Nelson, B. (1984) Implementing Remote Procedure Calls. ACM Trans. Comp. Syst., 2(1), pp. 39-59.
8. Koster, R. and Kramp, T. (2000) Structuring QoS-supporting services with smart proxies. Proceedings of IFIP/ACM Middleware Conference (LNCS 1795), 4-8 April, pp.273-288. Springer-Verlag.

9. Santos, N., Marques, P. and Silva, L. (2002) A Framework for Smart Proxies and Interceptors in RMI. Proceedings of ISCA PDCS'2002, Louisville, KY, USA, September.

10. Mitchell, J., et al. (1994) An Overview of the Spring System. Proceedings of Compcon 1994, San Francisco, California, USA, February 28-March 4, IEEE.

11. Shapiro, M., Gourhant, Y., Herbert, S., Mosseri, L., Ruffin, M. and Valot, C., (1989) SOS: An Object-Oriented Operating System-Assessment and Perspectives. Computing Systems, 2(4), pp. 287-338.

12. Steen, M.V., Homburg, P. and Tanenbaum, A.S., (1999) Globe: A Wide-Area Distributed System, IEEE Concurrency, 7(1), pp. 70-78.

13. Reiser, H. P., Hauck, F. J., Kapitza, R. and Schmied, A. I., (2003) Integrating fragmented objects into a CORBA environment. Proceedings of Net.Object Days, Erfurt, Germany, Sept. 22-25.

14. Hamilton, G., Powell, M. and Mitchell, J. (1993) Subcontract: A Flexible Base for Distributed Programming.Proceedings of 14th. ACM Symp. Operating. System Principles, New York, December, pp. 69-79. ACM Press.

15. Makpangou, M., Gourhant, Y., Le Narzul, J.P. and Shaphiro, M., (1994) Fragmented Objects for Distributed Abstractions. In: Casavant, T.L. and Singhal, M. (eds), Readings in Distributed Computing Systems. IEEE Computer Society Press, USA.

16. Homburg, P., Doorn, L.V., Steen, M.V., Tanenbaum A.S. and De Jonge W., (1995) An Object Model for Flexible Distributed Systems. Proceedings of 1st Annual ASCI Conference, Heijen, The Netherlands, May 30-June 1, pp. 69-78.

17. Odell, J.J. (1994) Advanced Object-Oriented Analysis and Design Using UML. Cambridge University Press.

18. Adve, S.V. and Gharachorloo, K. (1996) Shared Memory Consistency Models: A Tutorial. IEEE Computer, 29(12), pp.66-76.

19. Attiya, H. and Welch, J. (1994) Sequential Consistency versus Linearizability. ACM Trans. on Computer Systems, 12(2), pp. 91-122.

20. Stenstrom, P. (1990) A Survey of Cash Coherence Schemes for Multiprocessors. IEEE Computer, 23(6), pp. 12-24.

21. Carter, J. B., Bennett, J.K. and Zwaenepoel, W. (1995) Techniques for reducing consistency-related communication in distributed shared-memory systems. ACM Trans. Comp. Syst., 13(3), pp. 205–243.

22. Nebro, A. J., Pimentel, E.and Troya, J. M. (1999) Distributed Objects: An Approach based on Replication and Migration, Journal of Object-Oriented Programming, 12(1), pp.22-27.

23. Java Remote Method Invocation Specification (1997). Sun Microsystems. Palo Alto, CA,.

24. Halloway, S. D. (2001) Component Development for the Java Platform, Addison-Wesley, New York.

25. TUMI9524. (1995) The Collaborative Multi-user Editor IRIS. Technische Universitat Munchen.

26. TR-01-13 (2001) NetEdit: A Collaborative Editor. Computer Science, Virginia Tech.

27. Pacull, F.,Sandoz, A. And Schiper, A. (1994) Duplex: A Distributed Collaborative Editing Environment in Large Scale. Proceedings of ACM Conf. on CSCW, Chapel Hill, North Carolina, USA, 22-26 October.

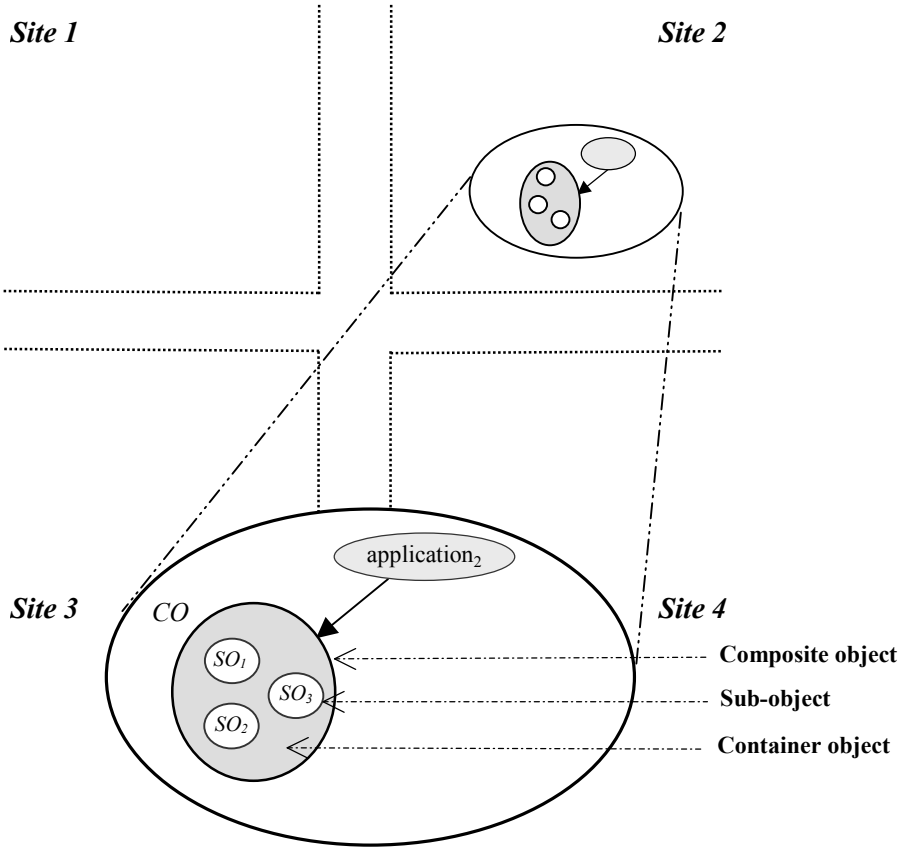

Figure-1. A composite object created on a single site with its three sub-objects

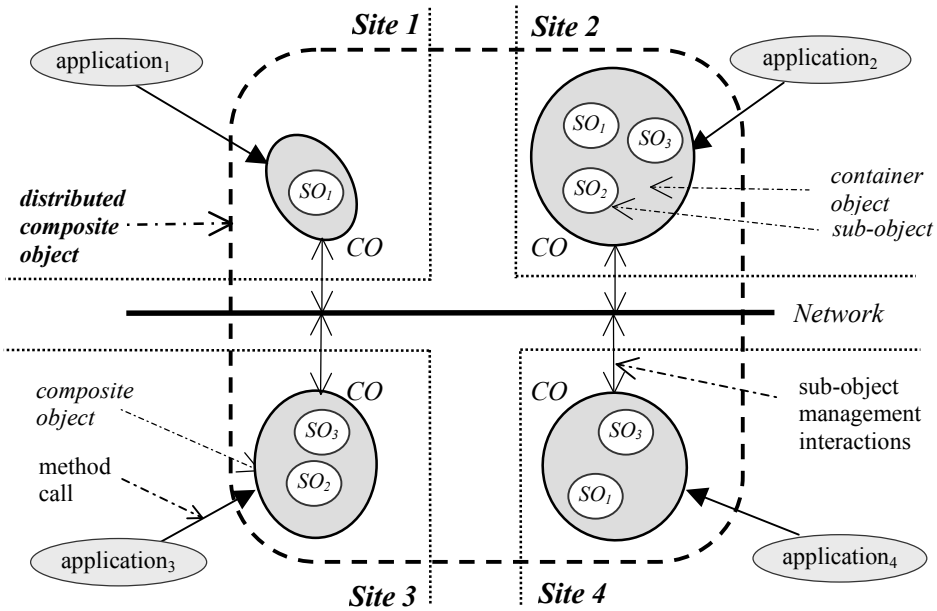

Figure-2. A DCO with three sub-objects that spreads over four sites

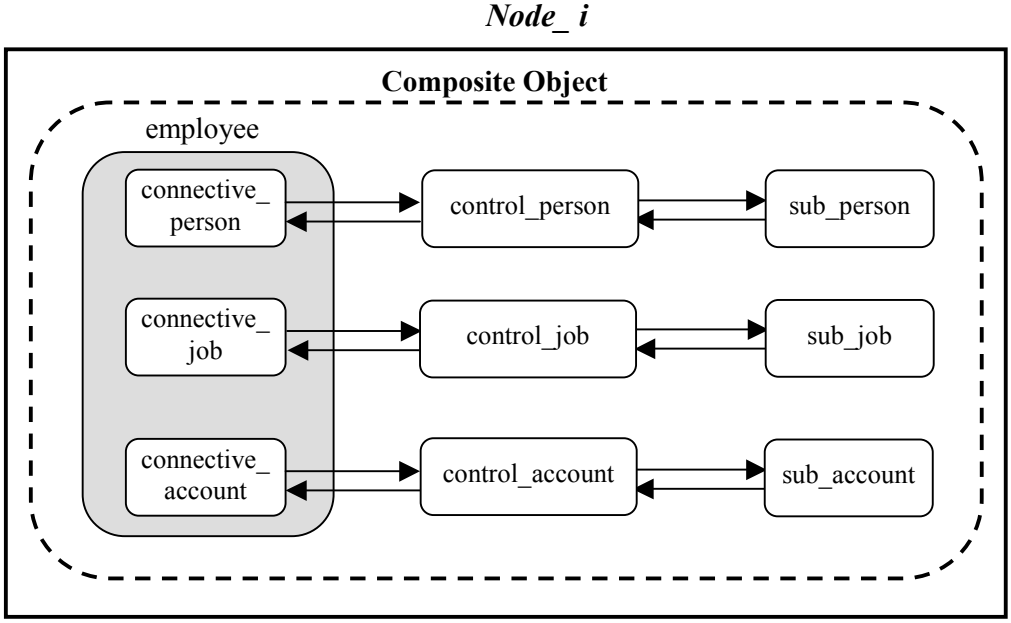

Figure-3. Access pattern in a composite object

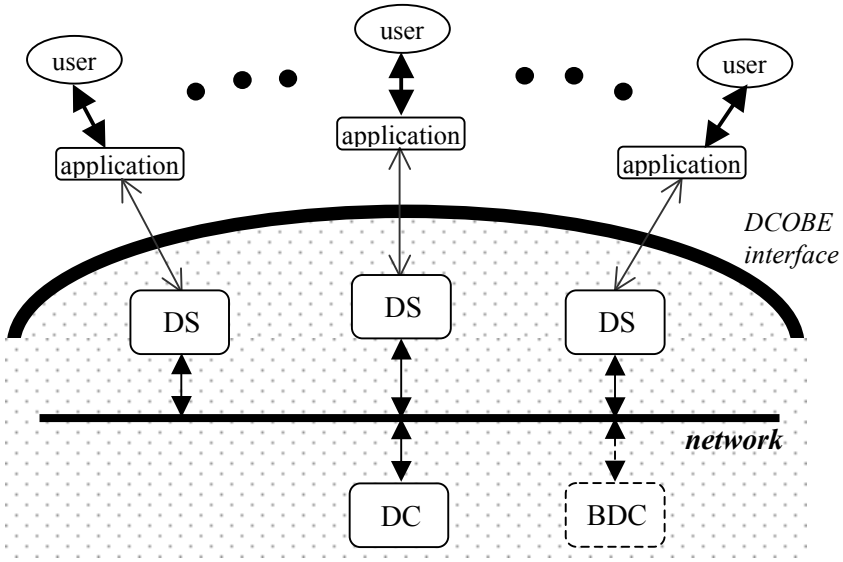

Figure-4. Schematic view of DCOBE middleware

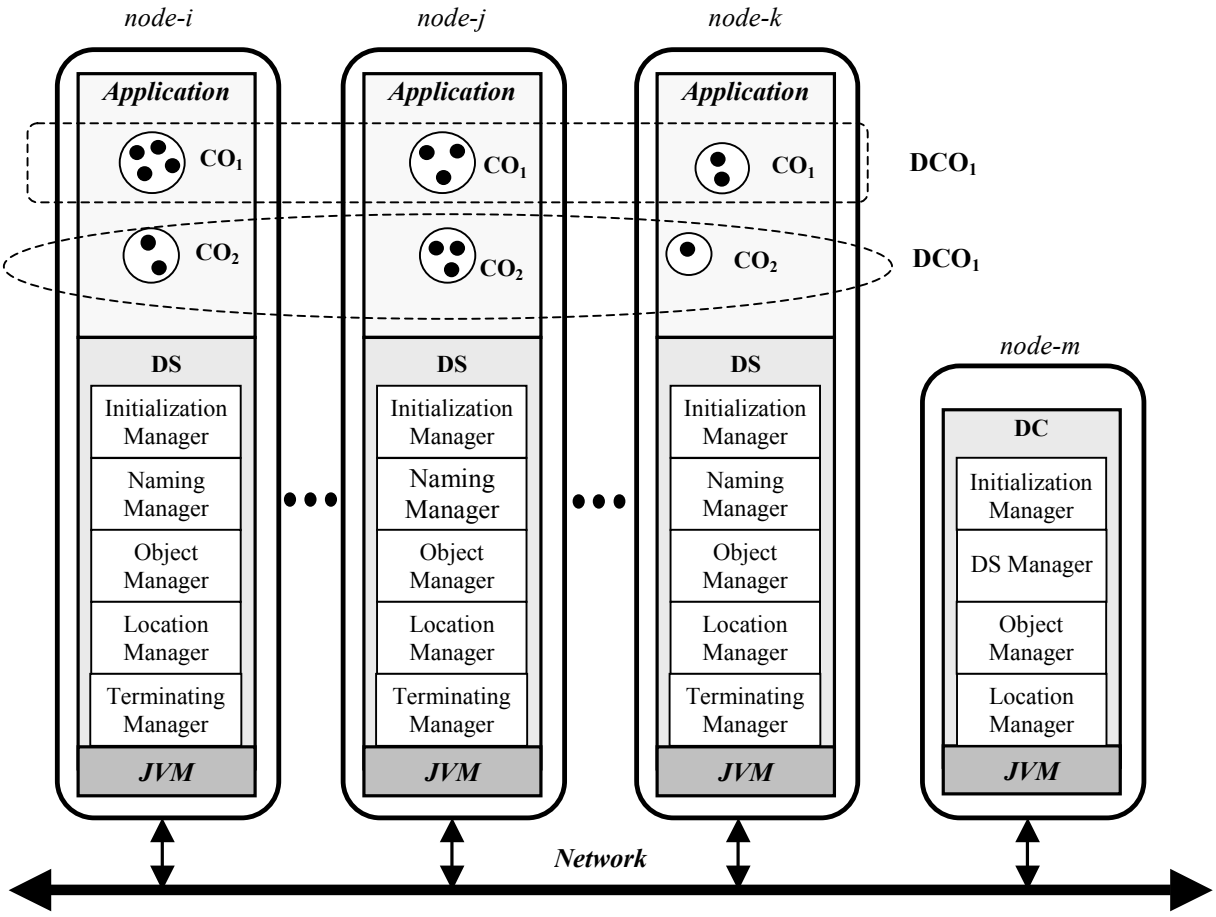

Figure-5. General view of DCOBE architecture

```
public class Employee {
     //Definitions of the sub-objects and other variables
      Connective_Person      person;
      Connective_Account     account;
      Connective_Profession  profession;
      String                 name;
      float                  amount;
      int                    professionId;
      public Employee() {
            account      = new Connective_Account();
            person       = new Connective_Person();
            profession   = new Connective_Profession();
            name         = "";
            amount       = 0;
            professionId = 0;
      }
      public void setName(String nm) {
            name = nm;
      }
      public String getName() {
            return name;
      }
      public void depositAccount(float amount) {
            account.deposit(amount);
      }
      public void withdrawAccount(float amount) {
            account.withdraw(amount);
      }
      public float balanceAccount() {
            amount = account.balance();
            return amount;
      }
      public void setProfessionId(int pid) {
            professionId = pid;
      }
      public int getProfessionId() {
            return professionId;
      }
}
```

Figure-6. Class definition for the container object `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;
	}
}
```

Figure-7. Code for sub-object class `Sub_Account`

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

Figure-8. Interface description for class `Sub_Account`

```
public class Connective_Account {
     int obj_id;
     Control_Account controlObject;
     public Connective_Account() {
           controlObject = new Control_Account();
           obj_id = controlObject.get_id();
     }
     public void deposite(float amount) {
        .......
     }
     public void witdraw(float amount) {

        .......
     }
     public float balance() {
           if (controlObject == null) (1)
                controlObject = (Control_Account)
                     dcobeServer.get_controlObject(obj_id);(2)
           return controlObject.balance();(3)
     }
}
```

Figure-9. Class definition for the connective object `Connective_Account`

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

     public Control_Account() {
           subObject = new Sub_Account();
           server_id = dcobeServer.get_serverId();
           obj_id = dcobeServer.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;
     }
}
```

Figure-10. Class definition for the control object `Control_Account`

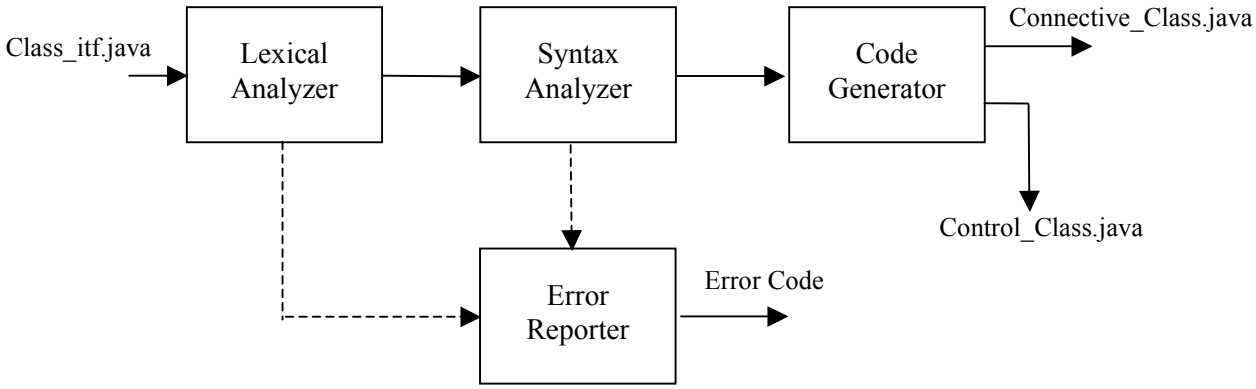

Figure-11. Automatic Class Generator Modules

1. list_of_itf → ∈ | itf list_of_itf
2. itf → "**public**" "**interface**" **id** "**{**" method_list "**}**"
3. method_ list → ∈ | method method_list
4. method → "**public**" return_type **id** "**(**" parameter_list "**) ";**"
5. parameter_list → ∈ | parameter | parameter "**,**" parameter_list
6. parameter → parameter_type **id**
7. parameter_type → "**int**" | "**String**" | "**boolean**" | **id**
8. return_type → ∈ | "**void**" | "**int**" | "**String**" | "**boolean**" | **id**

Figure-12. Grammar rules for interface description

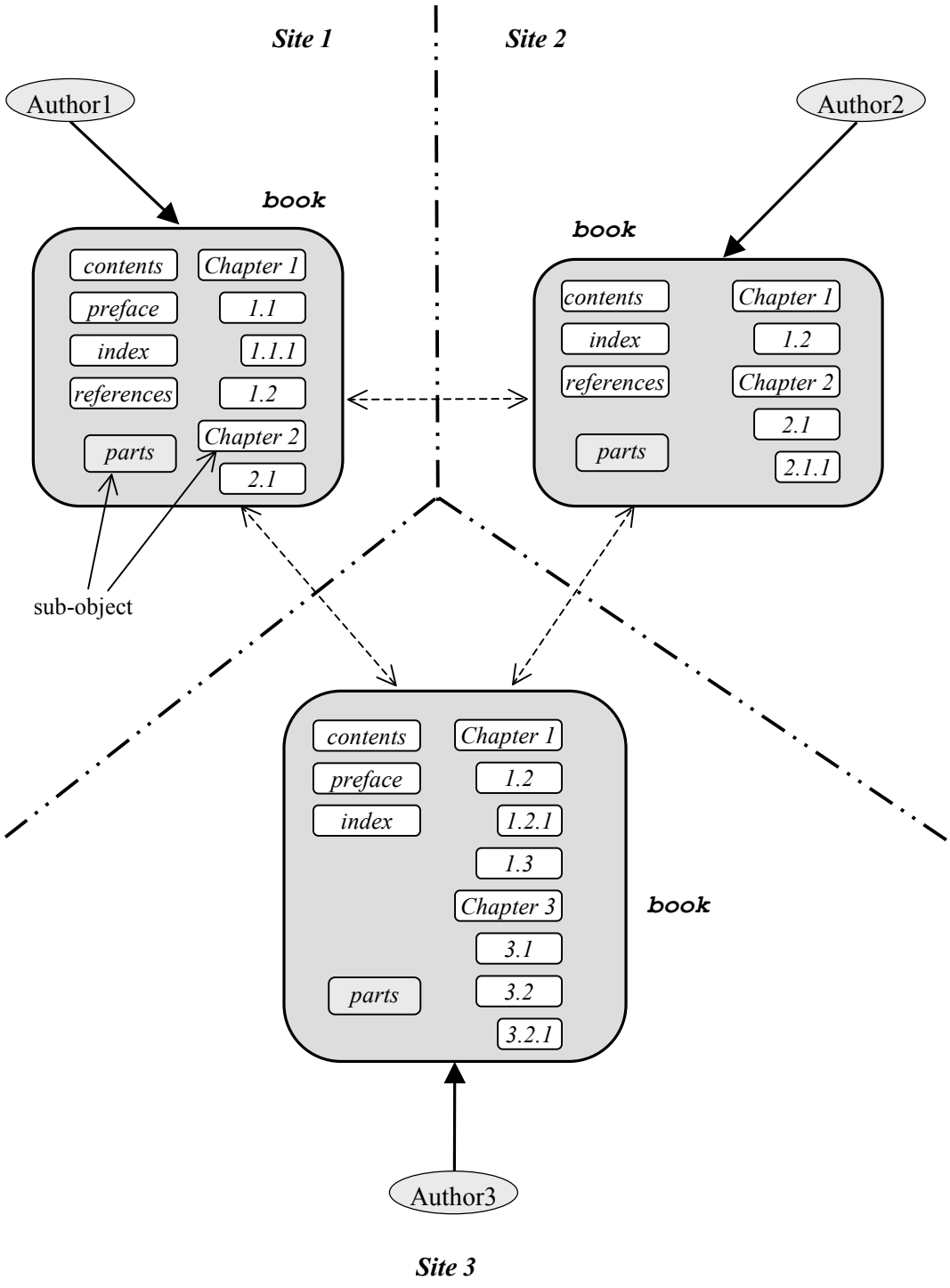

Figure-13. Composite structure of `book` object as accessed by three distinct authors

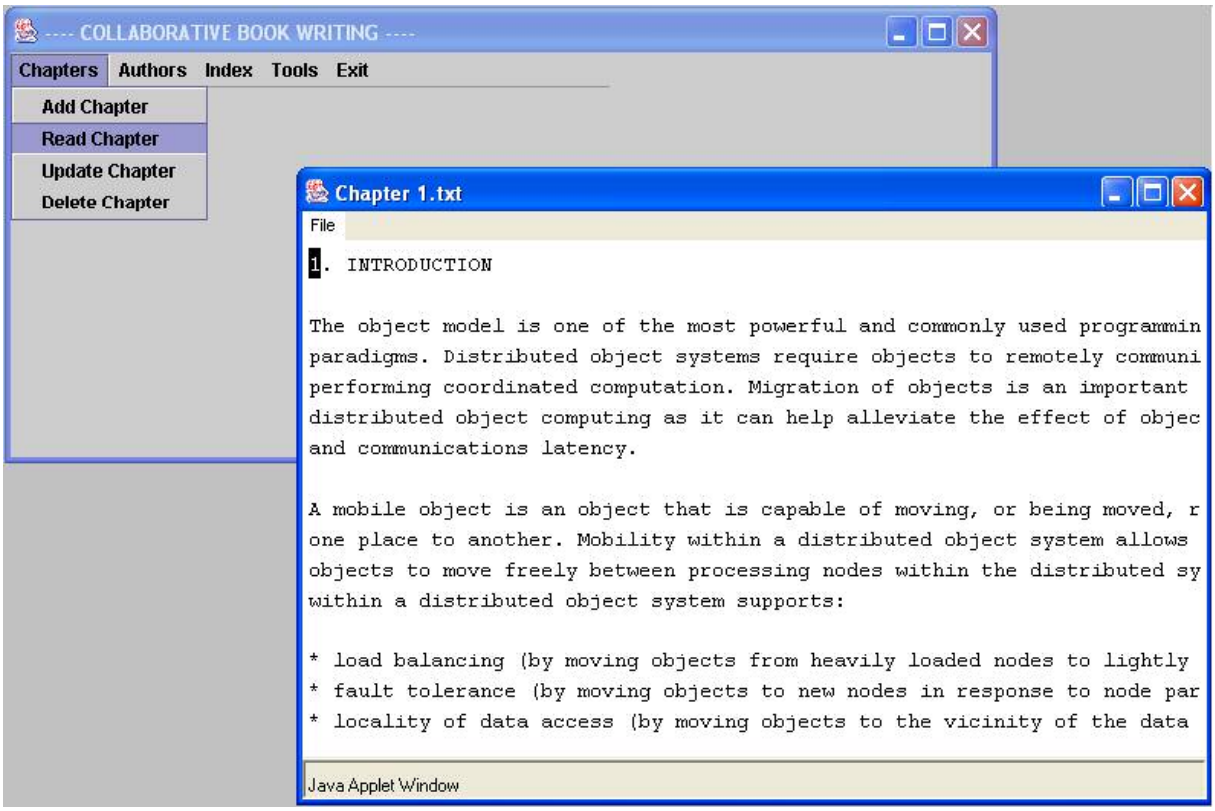

Figure-14. Graphical user interface for authors

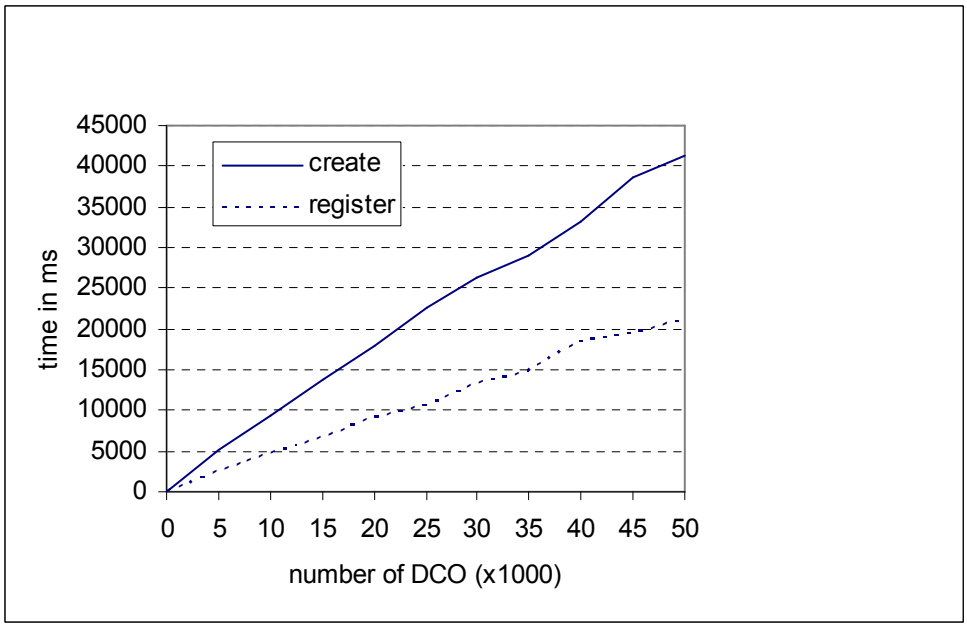

Figure-15. DCO creation and registration costs for increasing number of DCOs

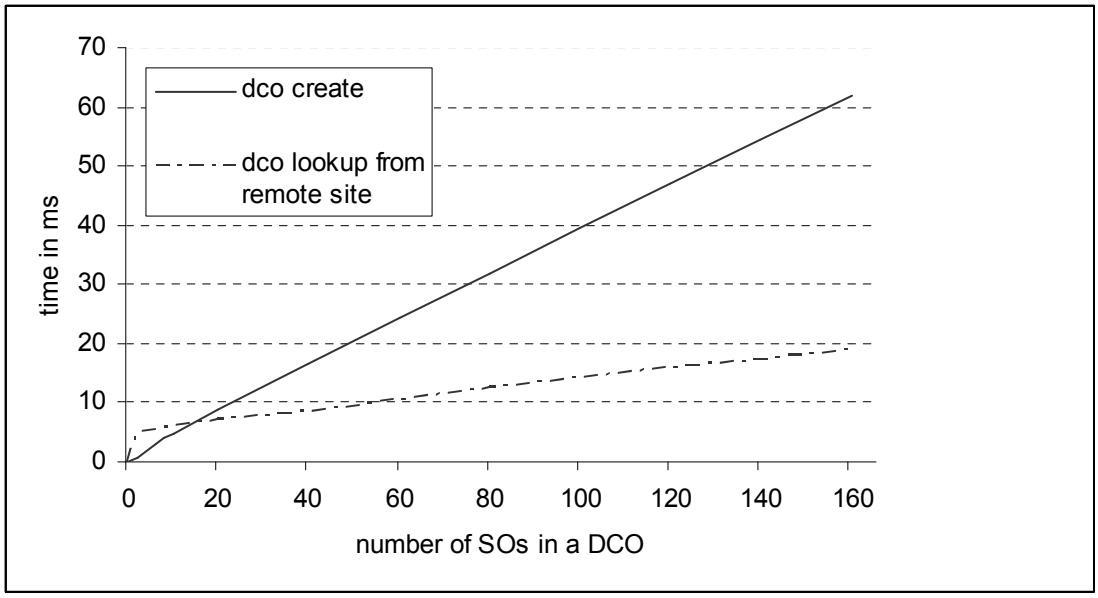

Figure-16. DCO create and lookup costs for increasing number of SOs

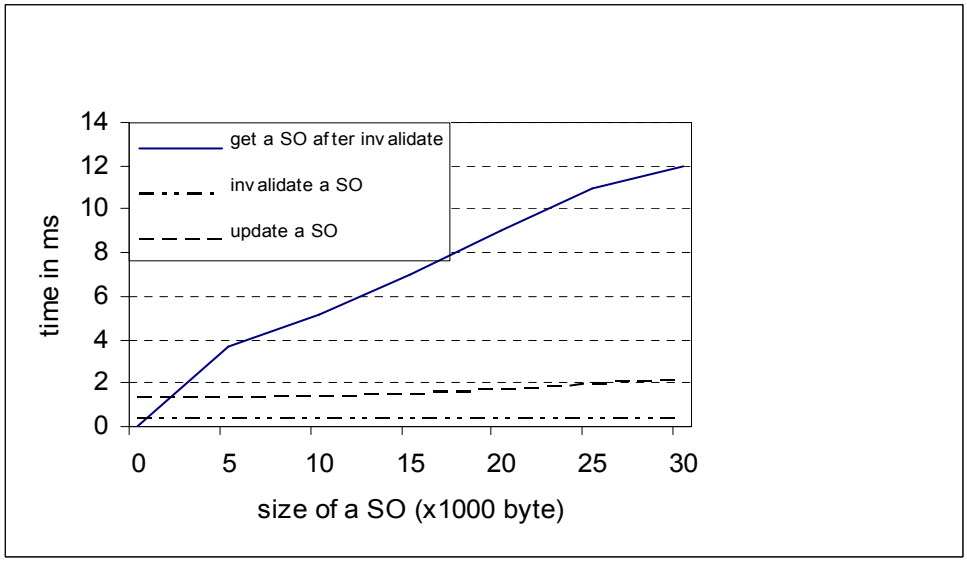

Figure-17. Update and invalidation costs for increasing sizes of SOs

Table-1. Comparison of basic operations in RMI and DCOBE

Basic RMI and DCO Operations	**Time in RMI**	**Time in DCOBE**
Object creation on a local site	0.26 ms	0.82 ms
Object registering	3.32 ms	0.42 ms
Lookup and binding	3.51 ms	6.02 ms
Initial method invocation	0.42 ms	5.75 ms
Successive method invocations	0.42 ms	0.001 ms
Update and invalidate a sub-object's replicas • invalidate replicas (for i number of replicas) • update replicas (for i number of replicas)	---	 i*0.41 ms i*1.30 ms