Distributed Composite Objects: A New Object Model for Cooperative Applications

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. DCO model incorporates two basic concepts: composition and replication. 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 for object composition, distribution and replication of object state, consistency management, concurrency control and dynamic deployment of restructured objects.

1 Introduction

With the increasing use of Web technology for Internet and Intranet applications, distributed computing is being increasingly applied to large size computational problems, especially to collaborative applications. The shared-object model is an attractive approach to structuring distributed applications. In this paper, we propose a new object model, *distributed composite objects (DCO),* that extends the shared object paradigm to meet efficient implementation, transparency, fault tolerance and enhanced performance demands of distributed applications, cooperative applications in particular. We also present the software layer that supports the DCO model, a middleware between a distributed application and Java Virtual Machine, called ***D**istributed **C**omposite **O**bject **B**ased **E**nvironment (DCOBE).* DCOBE offers services that facilitate the development of internet-wide distributed applications based on the DCO model [9].

The DCO model incorporates two basic concepts. The first one 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

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 106–115, 2004.

are replicated on different address spaces to ensure availability and quick 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.

Development of a distributed collaborative application on the internet is not an easy task because the designer has to handle issues of distribution, communication, naming, in addition to the problem at hand. DCOBE middleware facilitates the development and integration of distributed collaborative applications as it hides all implementation details behind its interface. In the following sections of the paper, we present a brief overview of the DCO model and DCOBE. Next, we describe a typical application that can benefit the model we propose: a real-time collaborative writing system. An evaluation of DCOBE performance and related work are described in the following sections.

2 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 SOs. Each SO is an elementary object, with a centralized representation, or may itself be a composite object. Several SOs are grouped together in a container object and form a composite object.

The implementer of the CO distributes its state among multiple SOs and uses them to implement the features of the CO. The clients of the CO only see its interface, rather than the interfaces from the embedded SOs. Therefore, from the client's point of view, a CO is a single shared object, accessed over a well-defined interface. He is not aware of its internal composition and, hence, has no explicit access to the SOs that makeup its state. This restriction allows for dynamic adaptation of COs. The implementer may add new SOs to extend the design, remove existing ones or change their implementation without affecting the interface of the CO. Thus, dynamic adaptation of the object to changing conditions becomes possible, without effecting its users.

The proposed model relies on replication. 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* (DCO).

Fig.1. depicts a DCO that spreads over three address spaces. It is initially created on *Site2* with all of its sub-objects (SO1, SO2, and SO3), and is later replicated on

two other sites, with SO1 on *Site1*, and SO1 and SO3 on *Site3*. 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.

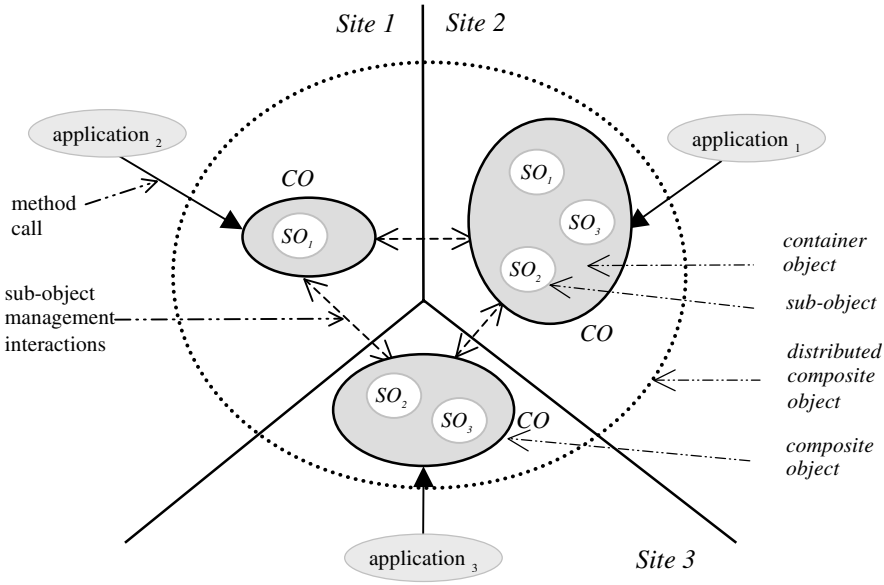

Fig. 1. A distributed composite object that spreads over three sites

The DCO model, with the support of the DCOBE middleware, conceals all implementation details behind its interface, as expected of the object-oriented programming paradigm. Clients of a DCO are isolated from issues dealing with distribution and replication of object state, the underlying communication technology, management of consistency of object state and management of concurrency control.

Management of Sub-objects: We have defined an enhanced object structure to deal with implementation issues and thus provide the object implementer with complete transparency of distribution, replication and consistency management. This new 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 client to object binding, which results in the placement of a valid replica of a SO 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.

The control object is located between the connective object and the local SO and exports the same interface as the SO. It receives both local and remote invocation requests and applies them on the local SO. Consistency problems arise among SO replicas on different address spaces when they want to modify object's state concurrently. The control object is responsible for the management of consistency of object state and concurrency control. It implements certain coherence and access synchronization protocols before actually executing the method call on the SO. The system uses *entry consistency* [1] 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* [5]. 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. Clients ask for updates as they need them. DCO model implements both coherence protocols. The object implementer chooses the one which suits the requirements of his application the best and the control object is generated accordingly by the class generator. The implementer may also specify different coherence protocols for different SOs of a composite object for enhanced performance. The control object implements a method invocation in three main steps:

i) 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. The control object recognizes the type of the operation the method invocation involves, either a write operation that modifies the state of the object or a read operation that does not, and proceeds with this information. It is blocking in nature, and further activity is allowed after 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).

ii) 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.

iii) 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.

A class generator that has been developed in the context of this work is used to generate connective and control objects' classes automatically from interfaces of SOs. Hence, the SO is the only object that the implementer has to focus on. The others are generated automatically, according to the coherence protocol specified by the implementer.

3 DCOBE Middleware

A DCO-based programming environment, depicted in Fig. 2, 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.

DCOBE 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. 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 which participates a DCO-based application in the distributed environment.

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 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 as a backup unit to the DCOBE architecture.

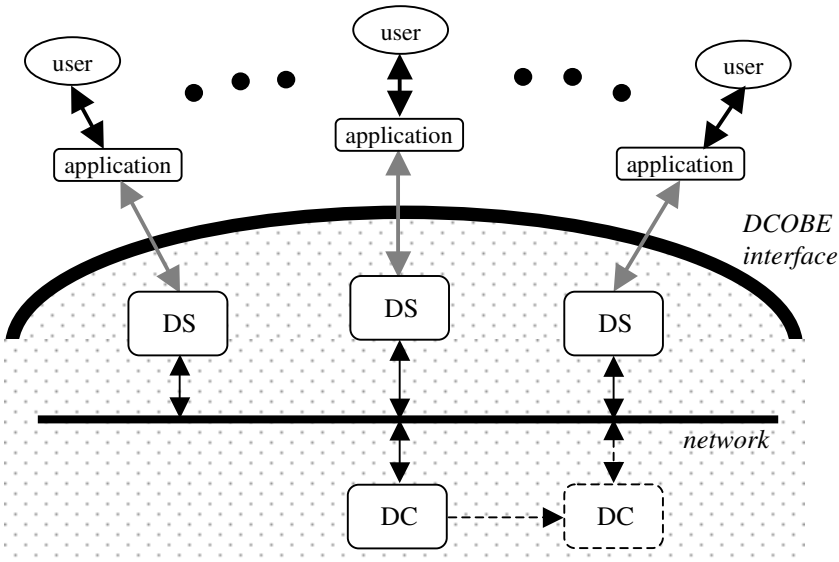

Fig. 2. Schematic view of DCOBE middleware

The main goal of DS is to provide execution support for DCO objects. A DS is actually instantiated within the context of each application and provides facilities that implement the DCO model. As a DS is integrated in each client application, the application can directly perform method calls as both are in the same address space. DSs on remote address spaces cooperate to process application requests and to ensure consistency of the replicated DCO state.

4 An Application: Collaborative Book Writing

Collaborative book writing on the internet is not a new approach. There are several academic and industrial studies on this issue [2, 6, 8]. 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 distribution, replication, consistency, concurrency and communication issues.

The application aims to develop a web-based collaborative environment that allows several physically dispersed people work together to produce a document, to view and to 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 any displayed document content. They also require rapid 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.

4.1 Storage of Text Objects

In this application, the text of a book is a persistent document. It is represented by a single DCO, namely `book`, which is shared among authors. As shown in Fig. 3, `book` consists of a collection of related SOs, each representing a specific part of the book text: its contents, preface, index, references, and chapters denoted by sections and sub-sections. A chapter may include several sections, and these sections may further be divided into sub-sections.

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, so updates are first performed at the local address space immediately and then propagated to remote sites, according to the consistency protocol. Multiple authors are allowed to access any part of the shared text. 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 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. Since simultaneous updating operations on shared parts of the books by different authors may conflict, the consistency protocol (write-invalidate is chosen for this application) ensures consistency. While authors carry out these operations, they will not be aware of composite and distributed structure of the object they are working on.

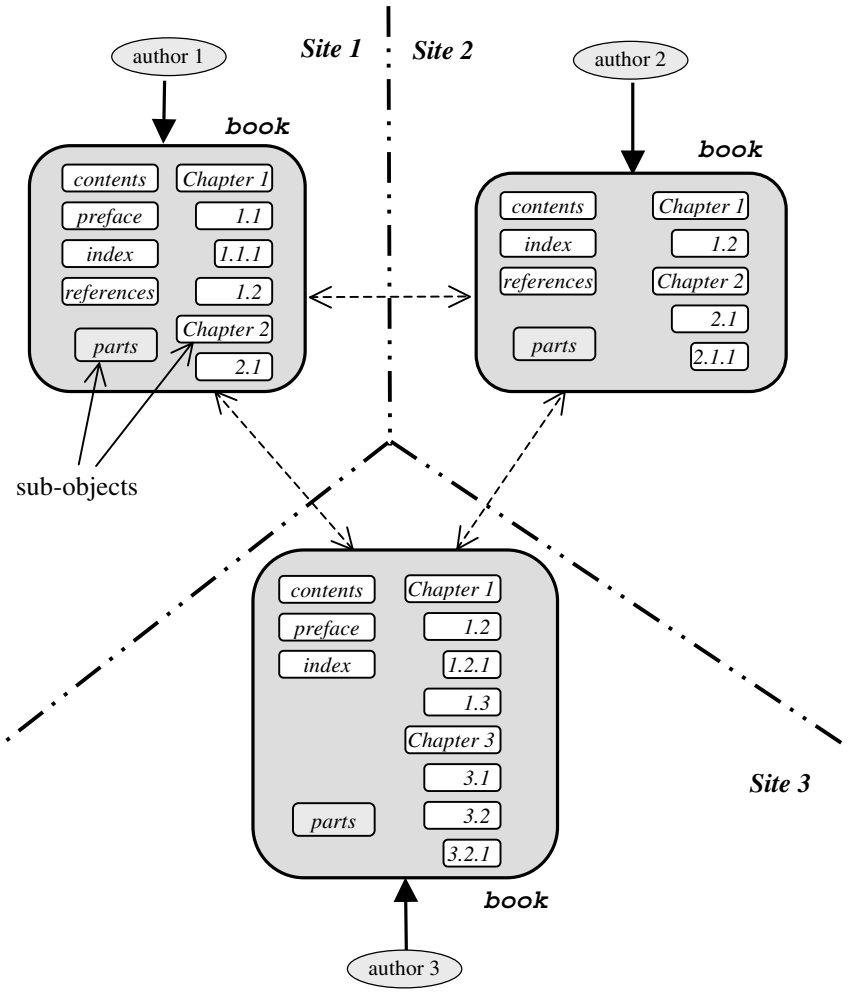

Fig. 3. Composite structure of ***book*** as accessed by three distinct authors

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 special sub-object 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.

4.2 Author Interface

Authors access `book` via a GUI 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 other authors' opinions, last modification date, 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.

5 Performance Evaluation

This section presents an evaluation of DCOBE middleware and its services. We measured the cost of basic operations to better understand system's behavior. The experiments were performed on 5 PC desktops based on Pentium III processors (1000 Mhz, 256 MB RAM), connected through a 10 Mbit Ethernet, running Windows 2000 and JDK 1.3.1. Measurements were repeated 10 times and the reported results are the arithmetic averages of these measurements. Table 1 presents the results.

Table 1. Measured costs of the basic operations in DCOBE

Operation		Time
Registering a DCO with a user defined name on DS and DC.		1.25 ms
Lookup and load operations for container and connective objects of a DCO, respectively, by a new client application.		3.15 ms
Loading of control object and sub-object onto the client site, on the first method invocation request on a connective object.		2.65 ms
Obtain access permission on a sub-object	(from local control object)	0.00002 ms
	(from remote control object)	0.75 ms
Method invocation	(if object is not cached)	0.0006 ms
	(if object is cached)	0.0001 ms
Update and invalidate a sub-object		
invalidate other sub-objects	(for i number of sub-objects)	i*0.75 ms
update other sub-objects	(for i number of sub-objects)	i*0.95 ms

In order to evaluate DCOBE performance, we compared the timing results of basic operations of DCOBE with those of Java RMI. We used the object instance we had used for DCOBE with measurements in RMI, as well. First, an RMI object was

initiated on a node. Then, remote method invocations were performed on that object from a different node. The results of the measurements are illustrated in Table 2.

Binding operation of the RMI object on a server registry and the lookup operation of that object's stub from the remote server and similar operations in DCOBE are executed only once. Measurements show that the durations of these operations are almost similar in both systems. With RMI, every method invocation is forwarded to a remote site. However, in DCOBE, an invoked object is replicated on the requesting site with the target sub-object(s) and then invocations are carried out locally. Therefore, if the read access to write access ratio of method invocations in an application is high, DCOBE is expected to perform better than Java RMI.

Table 2. Measured costs of the basic operations in Java RMI

Operation	Time
Binding of the RMI object on a server with a user defined name	2.15 ms
Looking up of the RMI object's stub from the remote server	1.75 ms
A remote method invocation (with parameter)	0.5 ms
A remote method invocation (no parameter)	0.4 ms

6 Related Work

Our work has been influenced closely by the SOS [3], Globe [7], and Jgroup [4] 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 objects*. Fragments export the FO interface, while connective objects implement the interactions between fragments. The lowest level of the FO structure, the connective object, encapsulates communication facilities. 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.

One of the key concepts of the Globe system is its model of *Distributed Shared Objects* (DSOs) [7]. A DSO is physically distributed, meaning that its state might be partitioned and replicated across multiple machines at the same time. 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 developers to concentrate on the semantics of the object. Jgroup [4] extends Java RMI through the group communication paradigm and has been designed specifically for application support in partitionable distributed systems. ARM, the Autonomous Replication Management framework, is layered on top of Jgroup and provides extensible replica distribution schemes and application-specific recovery strategies. The combination Jgroup/ARM can reduce significantly the effort

necessary for developing, deploying and managing partition-aware applications. None of these projects 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.

7 Conclusion

This paper presents a new object model, *distributed composite object*, for distributed computing along with the design of a middleware architecture, DCOBE, that provides basic mechanisms to deal with issues related to activity on distributed environments. The proposed model, with the support of DCOBE, allows for collaborative design and control of distributed applications. DCOBE, being implemented on JVM, provides an environment that works on heterogeneous platforms. 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. We plan to enhance the model by introducing data persistency and capability-based access control policies to prevent unauthorized access to objects.

References

1. Carter J.B., Bennett J.K. and Zwaenepoel W., Techniques for reducing consistency-related communication in distributed shared-memory systems. ACM Transactions on Computer Systems, Aug. 1995, 13(3), pp. 205–243.
2. Fish R.S., Leland M.D.P., Kraut R.E., Quilt: A Collaborative Tool for Cooperative Writing, In Proc. of ACM Int. Conference on Office Information Systems, vol. 9, pp. 30–37, 1988.
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, pp. 170–186, 1994.
4. Meiling H., Montresor A., Babaoglu Ö., Helvik B. E., Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing, Technical Report UBLCS-2002-12, Dept. Comp. Sci. Univ. of Bologna, Italy, Oct. 2002.
5. Mosberger D., Memory consistency models, Operating Systems Review, 17(1), pp. 18–26, Jan. 1993.
6. Pacull F., Sandoz A., Schiper A., Duplex: A Distributed Collaborative Editing Environment in Large Scale, In Proc. of ACM Conference on Computer Supported Cooperative work (CSCW), North Carolina USA, 1994.
7. Steen M.V., Homburg P. and Tanenbaum A.S.: Globe: A Wide-Area Distributed System, *IEEE Concurrency, 7(1),* Jan.-Mar., pp. 70–78, 1999.
8. Tammaro G., Mosier J., Goodwin N., Spitz G., Collaborative Writing is Hard to Support: A Field Study of Collaborative Writing, The Journal of Collaborative Computing 6, Kluwer Academic Publisher, pp. 19–57, Netherlands 1997.
9. Yilmaz G., Distributed Composite Object Model for Distributed Object-Based Systems, PhD Thesis, Istanbul Tech. Univ., Institute of Science and Technology, Istanbul, Turkey, May 2002.