Scalability issues in MOOs: the role of object migration and replication[*]

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Abstract

Typically Multi-User Object-Oriented Environments (MOOs) are implemented using a client-server architecture, where a centralized server maintains the MOO state. Such architecture may limit the scalability of the MOO. This paper discusses the role of object migration and replication in the implementation of scalable MOOs.

1 Introduction

A Multi-User Object-Oriented Environment [5, 3] (MOO) architecture that is based on a single central server to manage all the MOO state is inherently non-scalable. When many simultaneous sessions are maintained, the processing power and the available network bandwidth of the the central server may become a bottleneck. Additionally, MOO users may be geographically dispersed and may experience quite different connectivity when connecting to a single server.

A more scalable approach would be to implement the MOO as a set of cooperating servers, and to use migration and replication techniques to place the relevant data near to its users. Having several servers can also be the basis to increase the availability of the environment. In this paper we discuss some of the challenges of building scalable MOOs and suggest some basic architectures to address the problem.

2 Static partitioning

We now assume that we implement a MOO server has a set of distributed cooperation servers. In this case, the objects managed by the MOO will be distributed among the servers. We will start by discussing a simple static partitioning scheme.

In the following discussion, we also assume that the end users interacts with the MOO using a *client* program. To support distribution, the client must be able to contact one or more servers during a session. The client program may also support the user interface, executing presentation tasks in the client machine.

Form the point of view of distribution, migration and replication, it is useful to cluster the objects managed the MOO into several categories. In this paper we distinguish two categories:

- objects associated with a given user;
- objects associated with a given "room".

A simple way to scale the system is to distribute the "rooms" among the servers. For the moment, we will assume that the mapping between "rooms" and servers is defined when a "room" is created and remains static for the lifetime of the system. Such system needs to implement a directory service to preserve these mappings. Before interaction with a room, the client needs to obtain the identification of the server hosting that location.

In a similar way, the persistent information about users can also be distributed among User Storage Servers. However, when the user interacts with a room, for performance reasons, it is useful to locate the object associated with that user in the server hosting the room. This means that the system must be able to migrate user data among the room servers. This architecture is illustrated in Figure 1.

3 Dynamic partitioning

The previous scheme does not optimizes load balancing, since in run-time, the distribution of users among servers is not necessarily even. For performance reasons, it make sense to locate the user objects in the same server that maintains objects of the room being visited. Since the main load comes from the user interaction with the system, one should try to distribute the rooms among the servers.

In the following discussion, we will assume that a room is *inactive* when no user is interacting with it and *active* oth-

[*] Selected sections of this report will be published in the Workshop Reader of the 13th European Conference on Object-Oriented Programming Brussels, Lisbon, Portugal, 14-18 June 1999. Copyright Springer-Verlag.

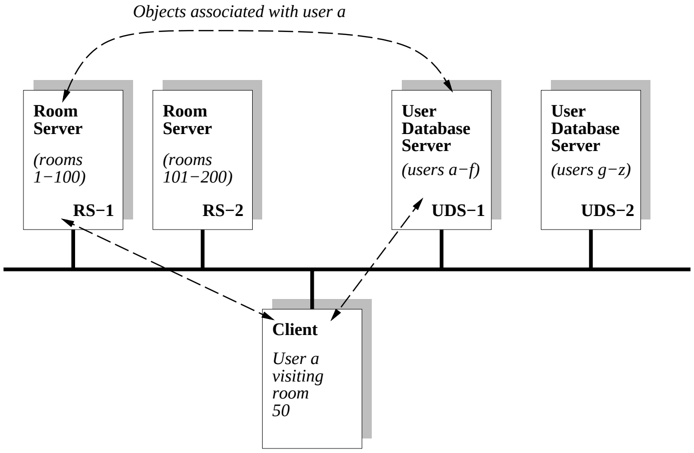

Figure 1. Static partitioning

erwise. At this point, it is worth to distinguish Room Storage Servers from Room Execution Servers, as illustrated in Figure 2. Storage Servers keep the persistent state of rooms. Execution Servers support the interaction of users with the rooms and the interaction among users.

The simplest form of dynamic partition consists in establishing the mapping between the room and the Execution Server at activation time. A load balancing algorithm would evaluate the load of each Execution Server and decide the best location for the room. In this approach, the location of the room will remain unchanged until the room is deactivated (i.e., until all users leave the room). This provides a limited form of load balancing since it may be impossible to predict the number of users that will move to a given room. Thus, a load balancing decision that is appropriate at activation time may become sub-optimal with the passage of time.

A more sophisticated form of dynamic partitioning is to allow rooms to be relocated in run-time from one Execution Server to another server. This may be a complex task since all objects associated with the room, including those associated with users in that room have to be migrated.

Network latency, is another important factor that may be desirable to take into consideration when defining a room location policy. To minimize the latency experienced by users it is desirable to place the rooms “close” (in term of network delays) to their users. For instance, consider a distributed MOO with servers in Europe and North-America. If a given room is being accessed exclusively from the USA, the room should be located in a server on that continent. Unfortunately, this type of policy may be very difficult to apply. It requires the system to collect the delays experienced by its users. Furthermore, it also requires the system to estimate the delays that those users would experience if the room would be migrated to another server.

4 Replication and performance

Data replication and data caching are well known techniques to increase the performance of distributed systems (by minimizing access times). Having different replicas of the same object requires the system to preserve replica consistency. Needless to say, the only case that is simple to manage is the case where objects are immutable (i.e., they support read only operations).

For scalability reasons, is fundamental that immutable objects can be differentiated from those that can be updated. Immutable objects can be easily cached in the client machines, alleviating the servers from unnecessary load.

Mutable objects can be replicated using a primary-secondary scheme [1]. The execution server that hosts the room (or the user) to which the object is associated plays the primary role. Clients that interact with the object may keep secondary copies. Read operation can always be performed locally but updates would be executed in the server and propagated to the clients. Note that this architecture also simplifies the implementation of access control. The server only propagates to a client copies of those objects that can be read by that client. Additionally, the server can locally verify the permissions for update operations and does not need to trust the client program.

5 Replication and availability

Another motivation for using replication is for fault-tolerance. In its most demanding form, fault-tolerance would provide complete fault-transparency, i.e., users will continue to interact with the MOO even if one of the servers would fail. Techniques to achieve this level of service, such as active replication, are usually very costly. It is unlikely that such techniques will be used in MOOs. For instance, the state machine approach [9] requires the duplication of

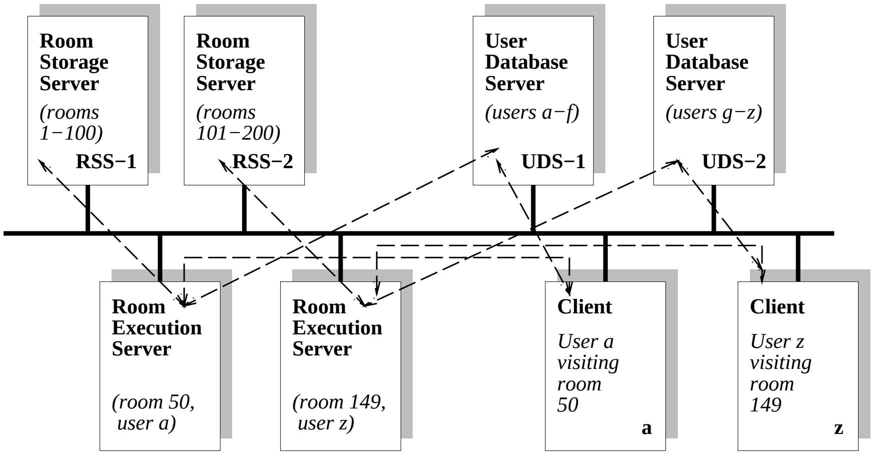

Figure 2. Dynamic partitioning

all resources and the use of an atomic multicast protocol [8] to interact with the replicated servers.

On the other hand, if fault-tolerance is not considered, the crash of a server may leave the MOO unavailable for a long period. This may discourage users or even defeat the purpose of the MOO. The state-of-the-art in terms of commercial solutions for high-availability servers is the use of clusters [7]. From the users point of view, the cluster can be seen as a single machine, that may crash but recovers in a short period, thus being available most of the time. In terms of architecture, the use of clusters does not change our previous design. It just makes servers less prone to long down-time periods.

6 Replication and network partitioning

Clusters make servers more available but do not solve the problem of network partitioning, as illustrated in Figure 3. A partition occurs when the system is split into groups of isolated nodes. The nodes in each partition can communicate with each other, but no node in one partition can communicate with any node in any other partition. Partitions are not a rare event in the Internet.

In our architecture, in the presence of network partitions each user would be limited in the number of rooms available to visit. Replication can also help to alleviate this problem. If the information is replicated among several physical locations, it is more likely that a copy is reachable from the client machine.

Unfortunately, availability and consistency are conflicting goals when partitions may occur. If users can perform independent updates on each partition, the state of the replicas may diverge to an inconsistent state. For instance, consider a room that holds an object which is supposed to be unique in the MOO. If this room and its objects are kept available in two distinct partitions, different users can pick the object, something that is inconsistent with the uniqueness property.

A typical approach to prevent divergence in partitionable environments is to allow updates on a single partition (usually called the primary partition). Translated to our architecture, this means that a given room could be available in both partitions but only users in the primary partition would be allowed to change the state of the objects (in our example, to pick the object). In this case, users in the non-primary partitions would nedd to be aware of network conditions [2].

Much better availability can be achieved if the semantics of the object are taken into consideration [4]. For instance, a bag of items could be filled concurrently in different partitions. When the partition disappears, consistency is re-established by merging both bags. MOOs are an excellent environment to apply object-specific replication policies and much of the work that is being done in the area of disconnected operation can certainly by applied here [6].

7 Fully distributed architecture

A step further in our design would be to incorporate an execution server in every client program. In this approach, rooms will be activated in the client of the user entering the room. Additional users would be re-directed to contact directly the client hosting the room. In this architecture, Execution Servers would be unnecessary, as clients visiting the same room would synchronize directly. Room Storage Servers would still be required to preserve the persistent state of inactive rooms and to maintain the location of each room.

By eliminating the need for the Execution Servers, this approach eliminates a potential bottleneck in the system. Unfortunately, this architecture also permits an unreliable or malicious client machine to jeopardize the operation of the system. This can be achieved simply by disconnecting

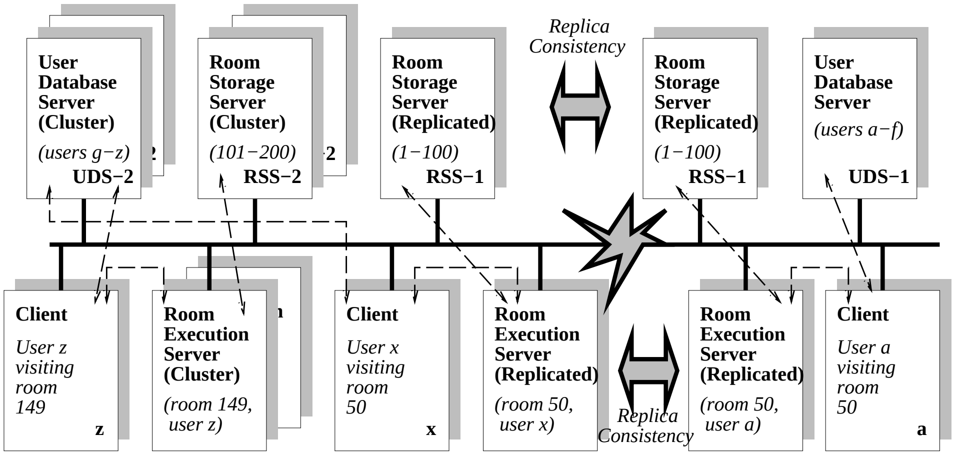

Figure 3. Replicated operation

the client, taking with it important system state. At this point, it is not clear which types of MOO functionalities would benefit from allowing clients to interact directly with each other.

8 Conclusions

In a MOO, the use of a single server is an obstacle to scalability. Implementing the MOO as a cooperating set of servers provides the ground for implementing load balancing policies. Since the load distribution changes in time, dynamic load balancing schemes are desirable. In turn, these require the ability to migrate objects between servers. Replication is a powerful tool to preserve the MOO availability in the presence of crashes or network partitions. The use of semantic-aware replication policies is key to balance availability and consistency.

References

[1] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 8. Addison-Wesley, 1993.

[2] F. Cosquer, P. Antunes, and P. Veríssimo. Enhancing dependability of cooperative application in partitionable environments. In *Proceedings of the 2nd European Dependable Computing Conference*, Taormina, Italy, Oct. 1996.

[3] A. Díaz and R. Melster. Designing virtual WWW environments: Flexible design for supporting dynamic bahavior. In *First ICSE Workshop on Web Engineering*, May 1999.

[4] A. Downing, I. Greenberg, and J. Peha. Oscar: A system for weak-consistency replication. In *Proceedings of the Workshop on the Management of Replicate Data*, pages 26–30, Houston - USA, Nov. 1990. IEEE.

[5] S. Evans. Building blocks of text-based virtual environments. Technical report, Computer Science Department, University of Virginia, Apr. 1993.

[6] J. Heidemen, T. Page, R. Guy, and G. Popek. Primarily disconnected operation: Experiences with ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, Monterey, California, Nov. 1992. IEEE.

[7] G. Pfister. *In search of clusters. Second Edition*. Prentice Hall, 1998.

[8] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of the Seventh International Conference on Computer Communications and Networks (IC3N'98)*, Lafayette, Louisiana, USA, Oct. 1998.

[9] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 7. Addison-Wesley, 1993.