A New Approach to Multi-User Environments Using Software Distributed Shared Memory *

Erich R. Schmidt, Cristian Ţăpuş
and Daniel M. Zimmerman
Department of Computer Science 256-80
California Institute of Technology
Pasadena, California 91125
{*riky, crt, dmz*}*@cs.caltech.edu*

Abstract

This paper presents an overview of multi-user environments and the application of the *Software Distributed Shared Memory (SDSM)* model to solve scalability problems in such environments. It also outlines a suggested distributed architecture which addresses the drawbacks of single-server architectures.

Games are a very compelling application within the computer community. This led us to choose *Multi-User Dungeons (MUDs)* to illustrate our distributed architecture. Such a distributed server system is necessary to accomodate the fast-growing interest in interactive networked games.

We have implemented our distributed server system using the Java language with a customized communication protocol based on sockets. The result has proven to be more efficient than existing environments written in C or C++, partially due to the hardware and distribution transparency afforded by Java and our framework.

In this paper we discuss the reasoning behind our system design and provide details about our implementation.

Keywords

Software distributed shared memory, distributed systems, scalability, networked multi-user environments, multi-user dungeon (MUD), object-oriented design, client-server model, Java

*This paper is based on work done for Caltech CS141bc, *Distributed Computation Laboratory*, Winter-Spring 1999. Support for this work was provided by the Air Force Office of Scientific Research.

1 Overview

Human nature involves interaction. One of the tools which helps people to communicate is the computer. Therefore, it is not surprising that, very soon after their introduction in the early 1980s, multi-player environments became very popular in the computer community. The first *Multi-User Dungeon (MUD)*, developed in 1979 at Essex University by Roy Trubshaw, was just a collection of inter-connected rooms where people could "virtually" walk and communicate [1]. Given the distribution of these environments, we can categorize them as early SDSM systems.

Early MUDs were highly appealing to the computer community. However, one major drawback of the MUDs was their simple text-based interface. The first MUDs were entirely based on text channels; users could interact only by typing simple commands to manipulate their characters and the environment.

Nowadays, MUDs offer persistent worlds, competition and/or cooperation between users, 3-D graphics, sound effects and other features. The addition of these features has increased the appeal of these games, causing the user base to grow extremely large. This has begun to cause serious problems with traditional system designs. Because of their lack of scalability, existing servers can not increase the number of connections to keep up with the increase in size of the user community and still maintain the same quality of service. The peer-to-peer architecture is not applicable to solve this problem, because there is a single centralized virtual environment shared by all the users in such a system. Traditional client-server architectures are also proving to be insufficient, even in the presence of high-bandwidth interconnections. Thus, it is clear that scalability is more important today than it has ever been before, and this is the focus of our work.

2 System Design

The most popular distributed environments are the role-playing games called MUDs. Since they are very well known and quite easy to implement, we focused our design work on this area.

One obvious "solution" to the scalability problem is to dedicate powerful computers to be used solely as game servers. Unfortunately, the hardware performance we need is not scaling fast enough to the popularity of such environments and, consequently, to the number of users. Some current games allow 32 or more simultaneous connections, but even then, when a large number of players are connected, the communication speed and the game speed decrease dramatically. This sometimes leads to servers crashing under load.

We chose to implement a solution based upon a completely different approach. We distributed the work load among multiple *game-servers*, and tried to keep this distribution as transparent as possible to the player. The *map/world* where the game takes place is divided into *territories*, each one being handled by a separate game-server; this way, only a fraction of all the players in a game are connected to the same game-server. As a result, better scalability and extensibility are achieved. We can add more players without having to change the architecture, and also extend the map with new territories by connecting them to the existing environment.

As mentioned in Section 1, all distributed multi-user environments basically follow the SDSM paradigm. In our approach, the servers comprise a SDSM system.

2.1 Implementation Issues

In order to implement the system described above, we had to take many factors into consideration. The implementation language, and the communication substrate to be used, were two of these factors. The system, as described, fits with an object oriented approach, and this is why we decided to implement it in an object oriented environment. The two main options for this were C++ and Java, and we finally decided upon the latter. Some of the main reasons for preferring Java to C++ were its portability, which allows the system to be used on multiple physical platforms, and its friendlier interface for socket-based communication and object serialization. In addition, Java's thread model, with its convenience and built-in synchronization mechanisms, was more appropriate for our work than any of the thread models available in C++.

Previous applications have proven that Java and SDSM are a very good combination, making hardware support and data and load distribution transparent. The same Java program is executable without modifications on any system, provided that a Java Virtual Machine is present; by distributing the data and work load on multiple servers, the user perceives the entire server system as a single data and computational unit [2].

As in all MUDs, the most important components of our system are the servers, the clients and the map. The servers we use are our game-servers, which manage the game on regions of the map, and a *meta-server*, which plays the role of a "nameserver" for the territories. Each territory consists of *rooms*, and each room is a directed graph of *cells* which are indivisible units of the map. The division of the territory into rooms and cells makes the system flexible in that it can be easily adapted to more complex applications than just MUD games.

We adopted a multithreaded design, because previous studies had shown that parallelizing SDSM systems can drastically improve application performance. Since most of the information that the clients request from the server involves *read* operations, these can be executed simultaneously by multiple server threads on the shared data. In the case of multiple *write* operations, the performance is not diminished and the system behaves in a manner similar to a single-threaded approach. Also, this design allows us to write simpler and cleaner code; each server thread handles one client connection at a time, which eliminates the need for a request queue [3].

The architecture used in our system is completely transparent to the user. The user can walk anywhere on the map, interacting with the environment and with other players, without being aware of the location of the current room and cell on the game-servers. The system solves the problem of server overload by a mechanism of delegating responsibilities to new servers. We now discuss the mechanisms for player movement and load balancing in greater detail.

2.2 Player Movement

A basic requirement of any MUD is player mobility - the users wants to walk around on the map, find other characters, objects, and creatures, and interact with them. On a single central server, this task is quite simple to achieve - the server simply keeps track of the position of the player on the map and takes care of the logistics.

Conceptually, the situation is similar in the distributed approach. Using a unique ID for each room and cell on the map, we can simply keep track of the position of the player on the map by keeping the IDs of the room and cell where he/she resides. When a player moves from one cell to another, we update the player's location. We also need to keep track of the positions of the rooms, since we allow them to be moved to different game-servers (see territory division in 2.3). In order to do this, we use a meta-server which manages a database containing the location of each room on the network. The actual information associated with a room in the database is the hostname of the game-server which manages the territory containing the room. This informa-

tion is most important when a player wants to join the game, as illustrated in Figure 1.

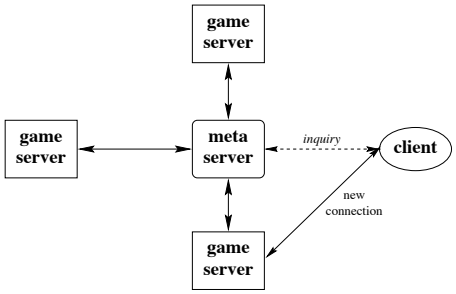

Figure 1: Joining a Game

First, the client sends a join-game *inquiry* to the meta-server, specifying the last location of the player (room, cell) from the previous game session. The central server looks up the current location of the room and returns this information to the client. After this, the client connects to the specified game-server.

The information managed by the meta-server is also used when the player moves on the map. If the new position of the player is part of the same territory, we simply update the position information on the game-server and all the logistics are handled locally. If the new position is not part of the current territory we find ourselves in the case presented in Figure 2.

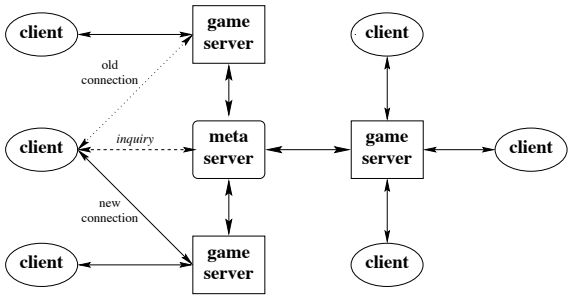

Figure 2: Player Mobility

The game-server redirects the player's request to the meta-server, and the meta-server follows the same procedure as in the case of a client joining the game, illustrated in Figure 1.

2.3 Territory Division

The load balancing mechanism mentioned earlier consists of splitting the territory managed by a game-server into two parts. One remains on the initial game-server, while the other is sent to another game-server. The players situated in the region which migrates to another game-server are redirected to the new-host, through the meta-server, in a manner similar to the one presented in Figure 1. Before this happens, the game-server updates the information contained in the meta-server's database. An example of this procedure is illustrated in Figure 3.

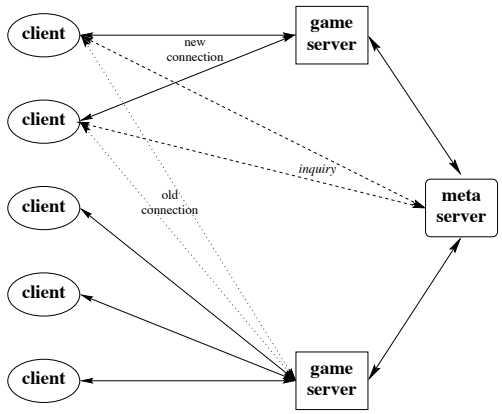

Figure 3: Territory Division

All the clients were initially connected to the bottom game-server. The load on the bottom game-server becomes high, so the territory is divided into two new territories. In the current implementation of the system, rooms cannot be split; division is done only at the territory level. The top game-server accepts one of the new territories, while the old server keeps the other. The players are not aware of the actual change, since they see the entire system behaving as a single server.

3 Future Directions

Our work so far has focused mainly on the issue of distributing the computation seamlessly to the user; issues like security, enhanced game logistics, performance maximization, and complete decentralization have not been of primary concern. However, we have considered each of these issues, and plan to extend the system to address them in the future.

As far as security is concerned, we believe it is necessary to add password-based access, as well as object grouping based on trust levels. Game information will be kept consistent among a group of trusted objects (i.e. servers) while untrusted objects will not be able to alter it.

Another component of the system that could be improved is game logistics. We are certain that one could imagine a more challenging environment and a more detailed command list, which could be implemented by making minimal changes to the system.

In order to increase the performance of the system,

we could introduce territory overlapping: the rooms on the "edges" of the territories could be managed both by the local game-server and the game-server of the neighboring territory. This would require very careful consideration of consistency and persistence issues, but it would provide more seamless distribution transparency to the client.

The current system is built around a meta-server, which has a key role in the system's functionality. Although it does not limit the scalability and extensibility of the system in any way, it could be argued that the use of a meta-server is not exactly in the spirit of pure distributed systems. One way to eliminate the meta-server would be to store more data on the game-servers and use IP multicast for queries. However, the elimination of the meta-server brings consistency and persistence issues to the forefront, as well as increasing the communication load in the system.

4 Conclusion

The use of multi-user environments has expanded dramatically over the last few years, and their ability to meet the demands of large user bases is extremely important. There is high demand for reliable solutions to this scalability problem, and the system described in this paper meets most of the requirements. Our implementation demonstrates the viability of our approach to the problem. This system opens new directions for the enhancement of multi-user applications in distributed environments and, as discussed in Section 3, its design allows extensions with minimal changes to the core system.

References

[1] Bartle, R.: *Early MUD History*, <http://www.apocalypse.org/pub/u/lpb/muddex/bartle.txt>

[2] Yu, Weimin and Cox, Alan : *Java/DSM: A Platform for Heterogenous Computing*, ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.

[3] Thitikamol, Kritchalach and Keleher, Peter : *Multithreading and Remote Latency in Software DSMs*, in The 17th International Conference on Distributed Computing Systems, May 1997.