Creation of a Distributive Server

Alberto Pareja-Lecaros
TJHSST
6/14/06

June 14, 2006

1 Abstract

The goal of this project is to create a distributive server able to handle a large application. It requires being able to handle multiple users, the security of those users, and be able to have any client act as server at a moment's notice. A lot of this will require Kerberos for security, RMI for remote access of code (clients accessing server code on the main server), and JAAS, which is a Java Authentication and Authorization service allowing secure connections between clients and servers and also setting permissions of those clients. No results or conclusions as of yet.

2 Introduction

This project will concern itself with the development of a distributed server to be able to handle multiple clients connecting to an interactive environment. This environment will be grand enough to merit needing a division of CPU between multiple computers. I expect the entire project to be completed by June, and if not at least a regular server should work.

3 Background

Distributed servers are used today as a way of dividing up CPU time between different computers in order to reduce the load any one computer has to

process. Usually this would consist of one main server, and then as clients connect to the server and the server starts to experience some slow down in performance, a client will then also be made as a server, and between the two computers the work will be divided up. Once the load for both of those two computers becomes too great, yet another client would join and act as a server, and so on.

The advantages of having a distributive server are numerous. First of all it allows for a virtually infinite number of client connects without sacrificing much performance in any one computer. Therefore, huge environments can be run without any user experiencing lag, a term meaning that the user is behind in receiving data or information. Another benefit of this distributed server is that if any computer in the server were to crash, another computer could easily be assigned to take over as a part of the server. Having everything concentrated into one computer would be horrible if that computer were to all of a sudden crash.

Java is a useful language for server development because of its built-in classes that deal with connections. Java has its own sockets which allow computers to open one of these sockets in order to transmit data. A server would have its own kind of socket called a ServerSocket which accepts a connection from a regular client Socket. As for creating a client, all it has to do is try to open a socket on the specified port to a certain destination, the IP address or name of the server computer. Once the connection is established, data streams can be opened on the socket and as long as these streams are never closed the socket should stay alive. The server usually works by continuously listening for clients trying to connect to it, and this is normally done on an infinite loop. Threads would have to be used if the server has other operations it needs to do.Distributed servers are used today as a way of dividing up CPU time between different computers in order to reduce the load any one computer has to process. Usually this would consist of one main server, and then as clients connect to the server and the server starts to experience some slow down in performance, a client will then also be made as a server, and between the two computers the work will be divided up. Once the load for both of those two computers becomes too great, yet another client would join and act as a server, and so on.

The advantages of having a distributive server are numerous. First of all it allows for a virtually infinite number of client connects without sacrificing much performance in any one computer. Therefore, huge environments can be run without any user experiencing lag, a term meaning that the user is

behind in receiving data or information. Another benefit of this distributed server is that if any computer in the server were to crash, another computer could easily be assigned to take over as a part of the server. Having everything concentrated into one computer would be horrible if that computer were to all of a sudden crash.

Java is a useful language for server development because of its built-in classes that deal with connections. Java has its own sockets which allow computers to open one of these sockets in order to transmit data. A server would have its own kind of socket called a ServerSocket which accepts a connection from a regular client Socket. As for creating a client, all it has to do is try to open a socket on the specified port to a certain destination, the IP address or name of the server computer. Once the connection is established, data streams can be opened on the socket and as long as these streams are never closed the socket should stay alive. The server usually works by continuously listening for clients trying to connect to it, and this is normally done on an infinite loop. Threads would have to be used if the server has other operations it needs to do.

In addition, Java has numerous already built-in libraries that greatly speed up production of server applications. A Kerberos library exists that allow the usage of that technology. Also, Java has a built-in JAAS, which stands for Java Authentication and Authorization Service. It allows for the secure connections of clients to servers as part of its authentication protocols, and then uses authorization to determine the permissions a particular client has. The benefit of this is that now different types of user accounts can be created, for example administrators and the just regular clients. Kerberos itself can be used in order to provide maximum protection for using passwords and establish secure connections. Kerberos will make sure a user's password is never stored or sent to the server. It instead uses a hash function over the password, erases the password, and sends the hashed password as well as other information for further protection. Java has built in RMI library as well. RMI allows for the remote execution of code. This is vital for a distributive server because no client should have any of the server code, and in order for a client to become a server it will become necessary for that client to access the main server's remote server code. This accessing of code must also be protected, which Kerberos combined with JAAS can address.

4 Project Development

To begin this task, I needed to make sure I had the knowledge base necessary to create a server. I began this process by creating a simple chat program which will help train me in how to use Java's sockets and will also help me to come up with the techniques necessary for the server to be able to perform its function. For this chat I also had to create the client, but this wont be necessary for the real project as the rest of the team is concentrated on that. So far I've had to look up how a server handles multiple clients on a server, and more recently I looked up how FTP connections worked, and I decided that that would be how I in the end will have to approach the creation of the server.

At first I created a simple interface that allowed a single client to connect to the server, but I had problems keeping the connection alive. I later discovered that closing any data streams that are open on a particular socket also close the socket, so what I ended up doing was keeping the stream alive indefinitely. This seemed to work for the time being. Later on when I moved into working with multiple clients, I realized that each of the clients would have to have their own thread on the server in order for the server to handle all of the data it is being sent while in the meantime still accepting connections from new clients. I also created another thread which would handle sending back the data to other clients.

Keeping the connection alive the way I had been doing will not work because the server continues to receive data from the BufferedReader I am using. The only way I see to fix this is to close the BufferedReader (and therefore close the socket), but this closes the connection between the client and server. So, what I have to do is create two connections per client. One connection keeps the client actually connected to the server and manages the connection. The second connection will open when data is ready to be sent, and then it will close when the data is received by the server. This will actually make the server have to wait instead of continuously looping for data, so that will help CPU time significantly. I still have to implement this new strategy but it should prove promising and will in the end be the only way to manage the huge environment I will eventually need to host.

Objects called messages were created to handle the different types of data the server will eventually be required to interpret. As of now there are a few different types of Messages, the Login message which takes care of the user name and password of a client, the Universe message which sends a

replacement of the universe along the stream, the Chat message which stores text data, and the Command message which will contain its own codes that tell the server what type of command to execute, such as moving a ship, destroying an object, changing the velocity/acceleration of an object, etc. Clients send these objects as needed and the server receives and executes them. The results are then sent back to the client. Any object passed that is not a type of message is discarded by the server immediately. I have recently begun using Darkstar, a new Java Sun Server built to handle exactly what I have been trying to do all year. It consists of transparent architecture which handles data securely and at a level that the developer need not worry about so much. Darkstar consists of 4 tiers, the 3rd tier is the game logic and engine. The 2nd tier is found server side and handles the event driven code, or the actual processes that occur when different sorts of messages are received from the client. The 1st tier is the communications tier which controls messages which are sent to the client or server. Tier 0 simply controls the API for the clients, or what is needed for a client to connect to a server and actually submit commands. All Game Objects are called GLOs, and these represent anything from an actual object (a ship or a planet) to a logical method to a player. They are built in such a way such that they work as if synchronized between different VMs so that there are no hangs, or in other words so that tasks are performed smoothly and one after the other.

5 Results

As of now clients can connect to the server and appear in a lobby sort of screen where they will soon be able to chat with other players. Connections to actual sectors are yet to be accomplished due to faults on the graphics side. Meaning, players would be able to connect but they would see nothing.

6 Conclusion

A main server working with authorization and authentication in addition to various other servers that are mapped by the main server with similar security protections should allow for secure client connections while at the same time managing what server a client would go to. RMI will allow clients to perform certain server functions when strain is put on the system. A third

program managing connections to establish a balance between keeping the clients in real-time and reducing latency may also be implemented to improve performance. In addition, game servers will go through a similar process and will be capable of handling any object data sent to it and then will be able to send results quickly back to all of its clients.