IN4391: Large lab exercise B: Dragons Arena System

Ir. Yong Guo
Delft University of Technology
Yong.Guo@tudelft.nl

Dr. ir. Alexandru Iosup
Delft University of Technology
A.Iosup@tudelft.nl

Michael de Jong
Delft University of Technology
M.deJong-2@student.tudelft.nl
1314793

April 1, 2013

1 Abstract

Massively Multiplayer Online Games (MMOGs) have become very popular in the past few years. MMOGs are games which allow thousends if not millions of players to connect to, and interact with each other. This is a very different gaming experience to scripted games with artificial intelligence and allows people to socialize and form groups to achieve certain goals in the game.

MMOGs also have some of the most interesting and difficult Computer Science problems to solve compared to other distributed computing paradigms. Not only do you have to solve how to share and synchronize data amongst players, you also have to do it very fast. When a game takes too long to react to a players movements or actions the player will experience a phenomena called *lag*. Lag is un unwanted effect, and can cause players to abandon a game if it occurs frequently.

To investigate how distributed computing concepts can improve MMOGs, I have created a system called Dragon Arena System *DAS*. *DAS* is composed of one or more servers which use Trailing State Synchronization (TSS) to sychronize player actions between servers. Each client connects to a single server and sends its actions to that server and receives game updates from that server. Additionally *DAS* has been designed to be fault-tolerant. In case a client disconnects it is removed from the game automatically for all other clients. If a server crashes the remaining servers remove all the clients connected to the crashed server and are able to keep the game going. The clients connected to the crashed server will simply disconnect.

2 Introduction

2.1 Problem statement

As MMOGs become more and more popular, they attracts more and more players. The simplest model is to have a single server to which clients can connect, send and receive game updates. But there are limitations to this model. A computer can have only a limited amount of memory, CPU cores, and concurrently active connections. When the number of players in a single game exceed the memory, CPU or networking capabilities of a single computer, the only logical solution is to scale to multiple computers.

There are also other significant advantages to using multiple computers to act as a single server. The most important advantage is fault-tolerance. A single server is also a single point of failure (SPOF). If the server consists of multiple physical machines it might be possible to keep the hosted game going when one or more of those machines crash.

2.2 Solution

DAS attempts to provide a distributed, fault-tolerant alternative to the single machine server model. The servers will form a fully connected network which is used to synchronize the game state between each other. Each client will connect to one and only one server. Each client is capable of sending actions like MOVE, ATTACK, HEAL, and JOIN to its server. The server timestamps the message using a Lamport clock and then broadcasts it to the other servers. These servers then add the message to a log. Periodically a snapshot is taken of this log. In case an action is received with a timestamp T, which occurred before another action in the log, it means that the log is inconsistent. The last snapshot before T is then used to construct a new snaphot using all the actions which happend at timestamp T and afterwards. When a new snapshot is made the server broadcasts it to all of its clients.

2.3 Structure of this article

The remainder of this article is structured as followed: In section 3 I will explain some of the requirements that have been established to which a system has to adhere. In section 4 I will discuss the design and implementation of the created distributed system *DAS*. In section 5 I will present some of the experiments and results that were done with the *DAS* system. In section 6 I will elaborate on the experiment results and properties of the constructed *DAS* system. Finally in section 7 I will briefly state my conclusion of this report. Additionally in appendix A, a description of the time it took to construct the *DAS* system can be found.

3 Background on Application

3.1 Requirements

- **Operation** Clients should be able to connect to a single server. Once in the game a player should heal other nearby injured players. Alternatively they player should attack the nearest dragon.
- **Fault-tolerance** The system should be able to handle both client and server crashes. If a client crashes or disconnects it should be removed from the game (from the point

of view of every other client). If a server crashes all the connected clients should be removed from the game (from the point of view of every other client).

- **Scalability** The system should be able to handle 100 players and 20 dragons on a system of 5 server nodes.

4 System Design

4.1 Messaging

One of the suggested means of messaging between nodes in the distributed system was Java's Remote Method Invocation (RMI). Each node can call a remote method with a Message object and a Address object, describing the sender. The sender can either wait for the method call to complete, which in essence signifies an acknowledgement, or ignore the result and continue onwards right after sending the Message.

4.2 Trailing State Synchronization

Trailing State Synchonization is a mechanism for dealing with delays in between actions performed on different physical machines. For example three machines might be sending each other actions which are to be executed on their own internal current game state. If one action is delayed during transport across the network it might be executed later than it should be and can cause inconsistent game states between the three machines. In TSS every message or action which affects the internal game states is timestamped before it is sent to the other machines using a Lamport clock. This simple mechanism ensures that all the received actions can be ordered exactly the same across all machines. This does however not solve the problem entirely.

The network might delay the broadcast of an action to other machines. In this case these other machines might have already processed actions which should have occurred after this delayed action. TSS's solution is to take snapshots periodically of the game state. If such an inconsistency is detected, all snapshots since this discrepancy are discarded and a new one is generated based on the actions which have been received since then. This new snapshot is then used from that point onwards when making decisions about the game.

4.3 Fault-tolerance

Replication is a very important requirement in this system. Both servers and clients can disconnect or crash. In case a client crashes its server should (eventually) notice this and remove that client from the game state. It should then broadcast a DISCONNECT action to the other server to instruct them to do the same. In case a server crashes one of the other servers will (eventually) notice and remove all the clients related to that server from its game roster and then send a SERVER_DISCONNECT action to the other remaining server to instruct them to do the same.

This way the remaining servers are able to continue the game. Disconnected clients will simply be removed from the game as soon as possible, but the game itself will continue as if nothing happened.

5 Experimental Results

Because the development of *DAS* took considerable time for a one-person team. The time left to experiment with this system was limited.

5.1 Experimental setup

DAS is lightweight enough to run simulations of 5 servers and 120 clients on a single MacBook Air 2011 (1.8 Ghz i7, 4 GB memory). Nor the DAS cluster, nor Amazon's EC2 platform was used to run the experiments. Since the system can easily run on a single laptop, this was preferred to running the experiments on multiple physical machines since experiments are easier to monitor and execute. The system was however also tested on multiple physical machines (2 laptops and a dual-core desktop computer) as well to demonstrate that it works.

DAS employs Java's Remote Method Invocation (RMI) to call methods on remote machines with locally specified parameters. On top of RMI a small layer of code has been built which abstracts it into a messaging system with acknowledgements. More details on this have been described in the *System Design* section of this report.

The system is packaged into .jar files using Maven's package command. This .jar file is then distributed to all the physical machines which will be running this software. It can be configured to either run (multiple nodes) locally on a single machine, or a single node on a single machine connected to a network of other machines.

5.2 Experiments

To test if the system meets the requirements specified earlier, I ran multiple simulations of 5 servers and 120 clients of which 20 were dragons. The game simulation starts as soon as two clients have connected via one or more servers, and allows other clients to join when they are ready to do so. The simulations stopped when either all dragons were killed or when all players were killed.

5.2.1 Consistency and fault-tolerance

In order to test consistency and fault-tolerance, I ran the simulation and killed one or more nodes in the system (both servers and clients), and checked that the game kept running as long as one server remained in working order. I also checked that the appropriate clients were removed from the game when a server or client was terminated.

5.2.2 Scalability

The requirements stated a minimum scale of 5 servers and a 120 clients (of which 20 dragons). This is easily achievable but the current system has not been tested beyond this scale. The debugging UI stops updating at this scale because of an unknown problem. It is most likely that there are too many updates to the game board. My intuition is that the system will be able to cope with bigger scales that the specified scale, but at the cost of latency. The more servers are added to the system, the more actions need to be broadcasted between servers. It also becomes more likely that actions will be received out of order because of delays, which will require each server to revert back to old snapshots. It is probably more rewarding to only add servers to the system when all servers in the system have reached

their maximum amount of clients. What the maximum amount of clients a single server can support is yet to be determined.

5.2.3 Performance

The *DAS* system has shown that it is able to cope with a high volume of concurrent actions in a distributed setup. The debugging UI does not show any signs of lag or latency. I have not tested the system beyond this scale, and I am sure that the system could be tweaked so it can support an even larger number of clients.

6 Discussion

6.1 Main findings & tradeoffs

- There's an inherent issue with the volume of messaging when multiple servers are used. *DAS* replicates all actions to the other servers which then in turn (if required) recalculate the new game state from an older snapshot. The requirements listed 5 servers as a minimum, but 2 or 3 servers are enough to keep the game running in case one crashes. This could greatly reduce the number of messages to be sent.
- Another improvement might be not to send every game state update to the client, but only ones which are 100ms newer than the last sent game state for instance. This does affect the lag / latency of the game a bit, so this should only be used sparingly and only when lag/latency is not crucially important to the game.
- Adding servers at run-time seems a bit harder than in the *DVGS* system since here the new server will need to get a few game state snapshots from other servers and the actions which were performed since then. Then all the other servers need to be notified of the new server and replicate actions to that server as well when they receive an action. Only once this has been established can the new server start accepting new clients.

6.2 Recommendations

DAS is a system which meets the requirements of WantGame. Because using multiple servers will mean that the system will send more messages to replicate all actions amongst all the servers, it is advised to keep the number of servers as low as possible for the best performance. I would suggest limiting it to 2 or 3 server. This gives the game enough servers to continue if one or two servers crash, whilst it only minimally impacts the network capacity.

7 Conclusion

DAS is a system which has met all the stated requirements and allows a multitude of clients to connect to a distributed server architecture. This architecture is fault-tolerant and can keep the game running in the event of both client and server crashes. This is a really interesting model for MMOGs to experiment with because it potentially allows them to scale beyond the limitations of the single server model.

8 Appendix A

- Total-time: 65h
- Think-time: 12h
- Development-time: 36h
- Experiment-time: 6h
- Analysis-time: 1h
- Wasted-time: 3h
- Write-time: 7h