Distributed Computing Systems: Dragon Arena System

Authors:
Mitchell Lester
m.lester@student.tudelft.nl

Wesley van der Lee
w.vanderlee@student.tudelft.nl
Supervisors:
prof. ir. Alexandru Iosup
Alexey Ilyushkin

March 22. 2016

Abstract

The ever growing popularity of online gaming takes its toll. The massive number of online connected and concurrent players in one game requires a new system solution to ensure a functional and pleasant gaming experience. WantGame BV recognises this problem and wants to investigate on the feasibility of a distributed system, which ensures a scalable, fault-tolerant and consistent gameplay. This report concludes the feasibility of such a system by providing a working and tested proof of concept.

1 Introduction

The number of people who participate in online computer or mobile games is ever growing. Moreover new enjoyable games for such platforms are being released each day. The notion of online gaming intensifies the gameplay, as people can now interact, collaborate or compete with each other. On the other hand, a massive number of concurrent players introduces new technical challenges, such as "How to design a fault-tolerant system for that community?" and "What kind of system architecture would scale and support the ever growing demands best?".

WantGame BV recognises the high demands placed on the game engines, and wants to investigate whether a distributed system architecture would provide a solution to the technical shortcomings in the current system. This distributed

system architecture would distribute computation onto multiple different machines, further referred to as nodes, and ensuring key requirements of a distributed system, i.e. consistency, scalability, fault-tolerance and performance [5].

The distributed game engine has been designed, implemented and tested and this report functions as a full evaluation on the feasibility of a distributed system for WantGame's required environmental settings in which the non-distributed version of their current game runs. Section 2 elaborates on the background of the current application as currently deployed by WantGame BV. This section contains an analysis of the game's functional requirements and behaviour as concluded by us. Section 3 proposes the design of a distributed system which acts as a proof of concept for the feasibility of the distributed game engine. This section will moreover discuss how the key requirements of a distributed system: consistency, scalability, fault-tolerance and performance have been tackled. In order to provide ground for the established requirement, section 4 discusses experiments and their results performed on the implemented system. Section 5 and section 6 will respectively discuss and conclude this report.

2 Background on Application

This section provides an overview of the application which is currently deployed by WantGame BV in a non-distributed setting. The following section provides a summarised functional description of the game. The continuance of this section elaborate on the game requirements respectively consistency, scalability, fault-tolerance and performance for a distributed system.

2.1 Functionality

The game which is currently deployed on a non-distributed game engine is called the Dragon Arena System, or shortly abbreviated as the DAS. The DAS is a service of online warfare, where on a battlefield of fixed length a number of dragons and warriors combat. Dragons are computer controlled bots which may or may not attack nearby warriors, and warriors are human controlled avatars which may attack dragons, heal one another or move tactically around the battlefield. Consecutive actions of the same player are invoked with a certain delay, which causes the player to choose non-deterministically an action, e.g. a warrior can not run away and attack a dragon at the same time. However the action frequency is determined randomly for each dragon and warrior, the delay can be interpreted as players alternating turns.

2.2 Consistency

The distributed system proposed in section 3 provides a solution to the massive number of players. The players independently invoke actions to either move, heal or attack. Each action causes the game state to change, as moving around

the map causes a repositioning of the warrior, and healing or attacking causes an increase or decrease in life health. Concurrent players should all see the same game state, which leads to the notion of consistency. In a distributed setting, game states are often computed on different nodes (machines as recalled from the introduction) and may diverged if they are not synchronised properly. To retain consistent states, nodes need to agree on the validity of actions and synchronise commands properly.

2.3 Scalability

The scalability requirement states that the system should behave as expected, even when compounded with a large number of player entrants. As stated by WantGame BV, the minimum number of entrants to proof the distributed concept consists of 100 warriors, 20 dragons and 5 computing servers. In this setting, warrior behaviour will be emulated by a computer.

2.4 Fault-tolerance

The fault-tolerance requirement states that the system should behave normally in adverse situations. These adverse situations can be the crash of a server, the loose of connectivity of a client, etc. Different tests have been set-up to test the fault-tolerance of the distributed DAS, and will be discussed in section 4. Note that it is practically impossible to create a system which is fault-tolerant against all adverse situations, such as a massive earthquake destroying all locations of all running servers.

2.5 Performance

The performance requirement states that all actions should be processed in real-time or semi real-time in case of an operations queue which does not diverge over time. When operations are communicated among servers on a given time t, the second flush of operation communication at $t + 1$ should not increase the queue, that is as t reaches infinity (or an advanced period of time in the game) the queue size should also not reach infinity, meaning that the delay of action processing increases over time. We plan to implement a real time system, with direct meessage broadcasting and processing.

3 System Design

The setup of the system is based of a mirrored server client design [4], as depicted in figure 1. The mirrored server-client architecture is similar to the client-server design, the difference is that there is now a cluster of servers instead of one main server. In this architecture each server maintains its own copy of the game state, each copy of the game state is identical. Clients are allowed to connect to any server, the server then balances the number of clients between

the number of servers, this reduces the load on one server and distributes it over all servers. The server cluster is connected by a private network, the private network between the servers is used to mirror the servers. The client "plays" the game by sending requests (game action) to the server, the server then validates the request then passes this request to the rest of the servers. When a server receives a request from another server it updates it current game state then passes the request to every connection.

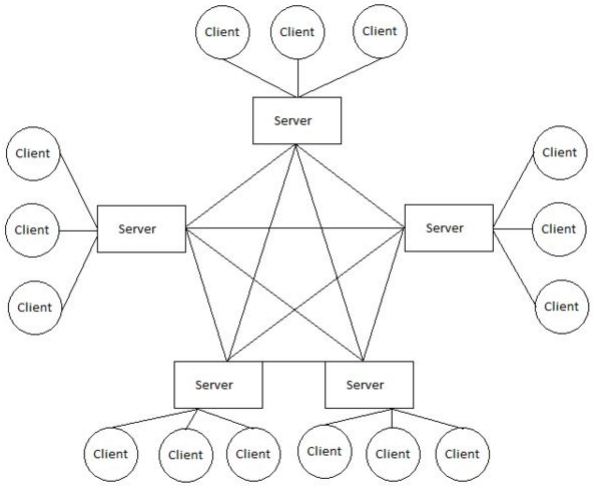

Figure 1: Mirrored Server client model for system design

3.1 Fault-tolerance

In the system, the game running on the servers must not fail if there is a server crash or a network failure. In the event of a client failing either by crash or loss of network connectivity the player will be removed from the game while it continues for all the other players. These clients can join the game again by connecting to another server.

As mentioned in the system design, each severs passes any new request to all connection after updating the local game state. The function introduces a redundancy measure where if a server is not updated by the first recipient of the request it will be updated by the next, allowing the system to continue in the case of a server crash. For example, in the case of a server crashing during broadcasting a new event, fault tolerance is achieved because all the servers broadcast any newly received event the message from the crashed server only needs to reach one server to be sent to the rest.

3.2 Scalability

The mirrored Server component of the system allows high scalability, this is due to each server maintaining a local copy of the game state. The design allows for extra servers to be added to the network if necessary, additional resources may be needed to be added to allow for fast processing capability on the server side.

3.3 Consistency

The system will tolerate and network delays keeping the game state up to date across all servers. In the design for this system, we considered several different approves to deal with consistency, in the end, we decide to use the Trailing State Synchronisation method. Trailing State Synchronisation method maintains multiple instances of the game state that are staggered from the preceding state by a couple of milliseconds, the leading instance has no latency as shown in figure 2. To identify inconsistencies, every synchronise analyses the game state for changes that an execution of a command produced and compares it with the immediate preceding state.

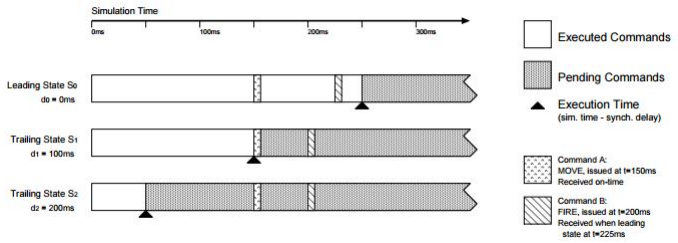

Figure 2: Trailing State Synchronisation Execution [4]

When an inconsistency is found, a change from the trailing state to the leading state is performed called a rollback.

Due to the rollback being performed during the gameplay it could be perceptible to the user that there has been a change in the game state since the user's game state is the leading state. To reduce the change of the user seeing any change, the delay of the trailing state would be of 100ms or less. The Trailing State Synchronisation method results in a system that has a high degree of consistency with a low latency and the ability to be scaled.

3.4 Performance

The requirement of having a system that has good performance is apparent with every multiplayer game, the latency of the system should be reduced to making gameplay as smooth as possible. The performance of the system is comprised of several other aspects of the system including, client-server network delay, server-server network delay, consistency method as well as several other factors

Since our system used a low latency private network between the servers the server-server network delay is small, the main delay from the network will be the connection between the client and the server, this delay is mainly due to the bandwidth of the client. Besides the private network reducing the latency problem, the private network also allows the game company to have complete administrative control of the mirrored servers. As mentioned in the consistency section the system will utilise the Trailing State Synchronisation method which allows for a low latency system.

4 Experimental Results

4.1 Conducted tests

During the implementation phase, we tackled many challenges. One of the challenges we couldn't tackle successfully was to distribute the system on the DAS4 [1].However since we were not able to implement our system on the DAS4 we performed the experiments on two connected machines, both running a game server with multiple clients connected to it.

4.1.1 Fault-tolerance tests

The goal of this test is to analyse the fault-tolerance of the system in the event of a server crash. To check that the system was tolerable to faults of the server crashing, we manually caused server crashes and network failures. After performing the experiment several times and analysing how the system dealt with the failure, it was apparent that when one server crashes the game continues on the other server with the clients that are connected. The clients connected to the crashed server are disconnected from the game.

4.1.2 Stress tests

The Stress test was conducted from the server's view, that is how many clients could connect to the server before the server would slow or crash. This experiment was performed on two machines running the game server, we then connected 50 clients to each server. In the end, there was a total of 100 clients playing the game, the game continued as normal with the players attacking, moving and healing.

The main problem with this test was that the battlefield has several problems, the main one being as players moved around the map they would leave "ghosts" behind, ghosts are images of the player stuck on the map. Even though in the experiment it appeared that the system handled the stress of 50 clients per server, it is hard to be certain due to the problems of the battlefield.

4.1.3 Consistency tests

The Consistency test was connected again on two servers running on two machines, we then connected a small number of clients to the server. By using the BattleFieldViewer to observe the game we could see if there were any inconsistencies between the servers. The observation was achieved by comparing the two battlefields from the viewer. After conducting the test numerous times, we can to the conclusion that there were no inconsistencies between the servers that was visible to the client.

4.2 Future tests

During the implementation phase, we tackled many challenges. One of the challenges we couldn't tackled successfully, was to distribute the system on the DAS4 [1]. This mainly impacted the tests we wanted to conduct, especially with regards to performance. Instead of describing the test results, this section will describe the tests we wanted to execute if the Dragon Arena System had successfully been installed on the DAS4.

Hypothetical Test 1: Load Distribution

As stated in the scalability requirement, the system should be scalable to support a total of 100 warriors, 20 dragons and 5 main servers. An evenly distributed solution would then logically aggregate 20 client nodes and 4 computer controlled bots per main server. This means that a server would communicate in total with 24 clients (warriors and dragons) and 4 other servers. This number seems far from an ideally computed result and moreover has never been justified at all. The test proposed in this section would determine the exact amount of connected nodes to a server in order to achieve an optimal performance.

This test wants to quantify two unknowns: the number of warriors W and the number of dragons D ideally aggregated to one single servernode. Say that each game could have some measures which function as measurement for performance. Then for each possible number of warriors and dragons up to a certain maximum the gameserver performance is measured. Gameserver performance could be measured among others by:

1. The average message queue size over a fixed time span
2. The average latency for a message to arrive and the corresponding action to be performed.
3. The number of seconds a gameplay takes normalised by the amount of performed steps in the gameplay and also normalised by the ratio $\frac{W}{D}$. A gameplay with a high $\frac{W}{D}$ is usually pretty quickly won, whereas a gameplay with a low $\frac{W}{D}$ is pretty quickly loss. This fraction therefore also influences the gameplay time.

Because the performance would be best for a W and D as low as possible, i.e. only one warrior and one dragon in the game would give an message queue as low as possible, the ideal number which needs to be investigated is to determine the best amount of clients (warriors and dragons) per server (*). Therefore for each possible combination of warriors and dragons, we want to depict the $\frac{W+D}{GS}$, where GS is the amount of game servers. An algorithm to provide a structure to conduct the described test, is stated below.

```
for i=1..m do
    for j:=1..n do
        List¡GameServer¿ servers := null for k=1..o do
            servers.add(new GameServer(k));
              InstantiateWarrior().linkToGameServer(servers.get(j % k));
        end
    end
    InstantiateDragon().linkToGameServer(servers.get(i % servers.size);
    results.add(gs.performance, i, j, k);
end
write(results);
```

Algorithm 1: How to write algorithms

At the end of the test, one has a large table of performance combined with the triple (i, j, k) where i is the number of warriors, j is the number of dragons and k is the amount of servers. From the (i, j, k) one can compute $\frac{W+D}{GS}$ by computing $\frac{i+j}{k}$, and link the performance even so. This can be visually graphed, and one could find an local optimum, which represents the ideal number of clients per nodes. We emphasise on the locality of this optimum, because we expect this data to be skewed to the left, because of (*).

The result of the described experiment would refer to the notion of load balancing, where additional servers are spawned after they are swamped with a maximum summation of load. In our implementation we would then spawn a new server once this optimal maximum has been reached.

5 Discussion

We realise that our system is far from finished, and as stated in the testing section, far from tested correctly. The notion of distribution so far has only been distributed on two connected laptops and not on a multi-cluster system such as the DAS4 at all. On the other hand we got introduced to the notion of distributed computing and always kept in mind that the program should operate on a distributed system. This follows from the fact that we continuously programmed as if the program were deployed and tested on a distributed system, by for example working small on a different local IP + ports, keeping in mind that during a later stadium the IPs could be changed to the IPs of the DAS4. The later seemed unsuccesful, as it appeared to be more difficult.

In our opinion, the way we started off was correct. We think that we've created an ideal architect (theoretically) and got accustomed to RMI accord-

ingly. The latter becomes apparent in our implementation, as one could identify multiple RMI interfaces, for multiple instances and multiple purposes. We discovered the true functioning and importance of RMI especially in distributed computing systems, even on two sole connected laptops.

If we were to continue on this lab, we would first of all focus on implementing the multi-cluster way by integrating our system to the DAS4. This however requires a lot of time, as the documentation is scarce, and even after following this [2] tutorial, we are still left questioned on how to properly distribute the system on the DAS4.

6 Conclusion

In conclusion we can state that we have delivered a proper proof of concept, showing the advantages of using a distributed system for the Dragon Arena Game. This answers our main research question whether or not a distributed system would be feasible, with yes, a distributed system for the Dragon Arena Game is thus feasible. This system shows a lot of advantages, especially with regards to the notions of scalability and performance. A distributed architecture also arises new challenges, for example how to tackle consistency between different servers, and how to create a fault-tolerant system. The architecture and implementation as we proposed does tackle the challenges of inconsistency and fault-tolerance by effective communication between multiple servers, each keeping an up to date state.

Although our implementation could not have been finished on a real multi-cluster system, concepts of Remote Method Invocation and the Grid Engine batch queueing have been learnt in order to conclude with an extraordinary distributed gaming system.

Appendix A: Time sheet

Because of the consultancy plot in which this lab took place, it is of utmost importance to keep track of time for invoicing. Note that this lab accounted for 40% of a 5 ECTS course, thus accounting for 56 hours per person, or 112 hours in total. We exceeded the 112 hours with a total of 60 hours which is justified below per person. We can also state that bot group members have equally contributed to this project, and should therefore be credited accordingly.

Type of Time	**Time (hours p.p.)**	**Remark(s)**
total-time	86	Total time
think-time	8	Discussing different system architectures, reading papers
dev-time	48	Developing the System as present on here [3]
xp-time	4	Performed tests as described here 4
analysis-time	1	Discussed and analysed results
write-time	5	The writing time of this report
wasted-time	25	Mainly on non-implemented development features (dead ends), exploring the DAS4, exploring tutorials that did not contribute to the end result like [2].

Table 1: Time distribution

Furthermore we think that we are eligible for the following bonuses:

1. Open source code and public report (+100 pt.)
2. Excellent report (writing, graphing, description of system) (+500 pt.)
3. Excellent analysis (design of experiments, analysis of results) (+500 pt.)

References

[1] `http://www.cs.vu.nl/das4/users.shtml`.

[2] `http://www.cs.vu.nl/das4/jobs.shtml`.

[3] `https://github.com/wesleyvanderlee/DragonArenaSystem`.

[4] Eric Cronin, Burton Filstrup, Anthony R Kurc, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of the 1st workshop on Network and system support for games*, pages 67–73. ACM, 2002.

[5] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*, volume 2. Prentice hall Englewood Cliffs, 2007.