TU Delft: Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science (EEMCS)

DISTRIBUTED COMPUTING SYSTEMS
(IN4391)

LARGE LAB EXERCISE B
The Dragons Arena System:
Distributed Simulation of Virtual Worlds

Authors
Filipe Fernandes, João Serra, Vasco Conde
{F.M.FonsecaMeirinhosFernandes, J.P.Serra, V.BarrosMendesNazareConde }@student.tudelft.nl

Support Cast
Dr. ir. Alexandru Iosup {A.Iosup@tudelft.nl}
ir. Yong Guo {Yong.Guo@tudelft.nl}

1. Abstract

Since games appeared in early computing systems people became really attracted to them. Seeing that as a business opportunity the industry evolved into more challenging systems that allow other kinds of gameplay, usually online and with a high number of of players, the so called MMOG (Massive Multiplayer Online Games). Although, these systems depending on the game requirements might have different architectures. Our work consists in designing and implementing a distributed game engine that allows several concurrent players to be connected to different servers. Within our findings we find that these kind of systems allow a game to scale, in an order far superior than with just one single non distributed server.

2. Introduction

A game engine that is not distributed can have a hard time handling the requests from a large group of players. Also problems with latency can arise if a server has to deal with a lot of requests. Our approach is to have multiple game engine nodes, each of them handling a subset of the units. This nodes are synchronized with each other so that every player sees the same virtual world.

Each battlefield sends regular updates to all the units that are connected to them. That information will then be used by the units to issue actions. The actions that arrive to a battlefield are then checked to see if they are valid according to the map on the server and also check against the pending actions (actions that are awaiting confirmation from other Battlefields).

Through this paper we present the in section 3 the Background of our application with emphasis in the requirements. Further ahead, in section 4, we explain our System Design. Then the Experimental Results are present in section 5 followed by a Discussion in section 6. We then conclude in section VII and in section VIII the time spent in this assignment is shown in different categories.

3. Background

DAS, Distributed Arena System, is a distributed simulation of virtual worlds in which several battlefields coexist and work together, and that allows units to interact with them. The units, Dragons and Players, that are spread across the several servers battle until only one type of unit remains.

As the virtual world is shared among all the units, and each unit is connected to a specific battlefield, there is the need to maintain the gamestate consistent across the several battlefield servers.

Tolerating failures is important but in this specific case it is essential to recognize them early and take care of them as fast as possible so the game doesn’t get to affected. When a server fails the remaining active servers should be able to detect it and remove the units connected to that server from the map, so that the game can continue. Also Player fails are taken into account and a server should detect one one of it’s players fails and then forward this information to the remaining battlefields.

4. System Design

4.1 System Overview

Our system is composed by servers, the Battlefields, and by units, which are represented by Players and Dragons. The several Battlefields create a network and use the parallelization model mirroring as a way of maintaining the state of the game. This means that while each server handles a subset of the units, they all share the same game world. More about this will approached further ahead.

Each unit is connected to one battlefield server and and receives from that server a periodic game state update with which they will decide the next actions to take. This relationship is simple and prevents cheating by the players since all the game actions are checked and processed in the server.

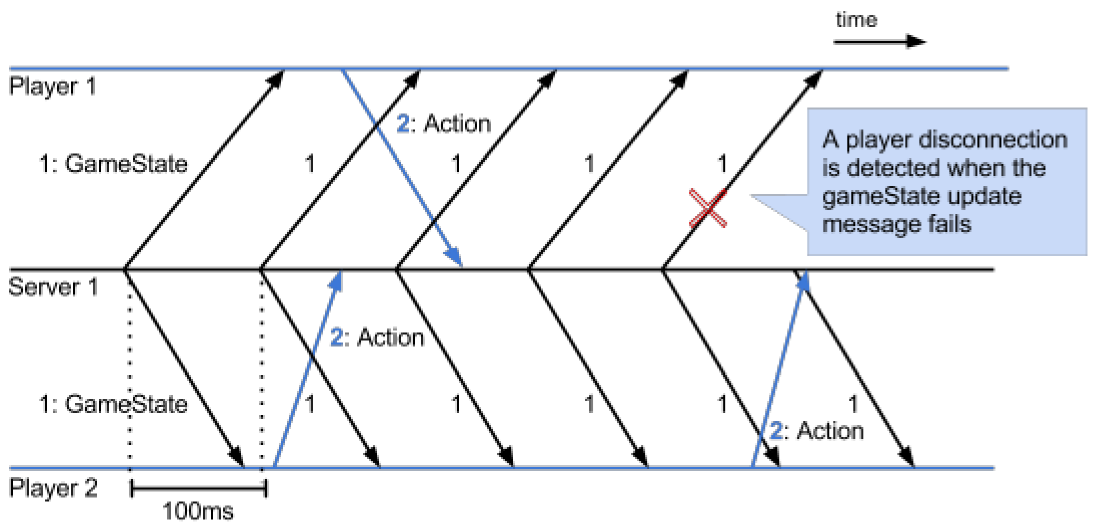

Figure 1. Server updates sends game state to connected units.

Since all the processing is done in the servers there is the need of having some system that defines which actions should be carried out and which ones should be refused. One of the key factors to an action to be processed is the fact that that action should be accepted by all the connected servers. For that to happen the action should not conflict with any of the already pending actions.

Each server keeps two lists of pending actions: pendingOwnActions, which is the list of pending actions from units that are connected to that server, and pendingOutsideActions, which is the list of pending actions by units connected to some other server.

When a server receives an action message from a unit, it sends a message to every other battlefield in order to carry on with the processing of the action.

Each Battlefield is responsible for replying whether or not the action is valid accordingly to their state. The same kind of processing is done in the Battlefields that receive an action to synchronize.

To check for conflicts we implemented a system where some actions are given priority over others. The rules that we use are:

- Moves have priority over attacks and heals but not over existing spawns;
- Spawns are not prioritised over existing moves;

This priority is only important when we are dealing with actions that are related with the same position like attacking a player that is moving away or moving to a position where another player is spawning.

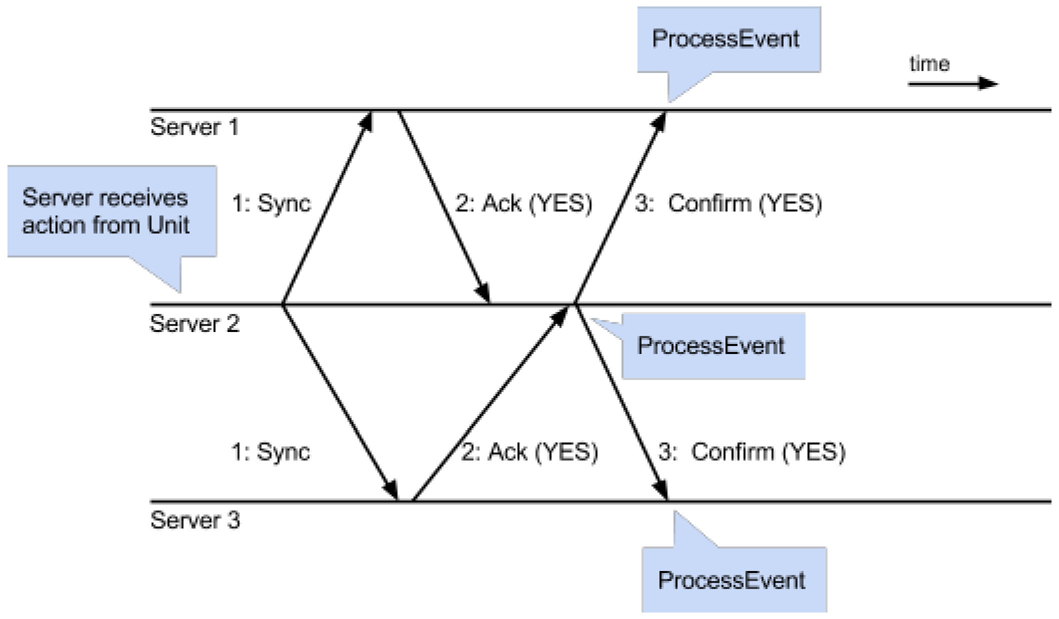

Figure 2. Synchronize action protocol; successful.

If the server receives a positive response from all battlefields it means that the action is valid and should be applied. The responsible server for that action starts by applying it and then it sends a confirmation message to all the connected Battlefields so that they know that they should also apply that action.

A server can also respond negatively to a sync message and in that case the responsible server should send a negative confirm message that indicates that that action should be removed from the list.

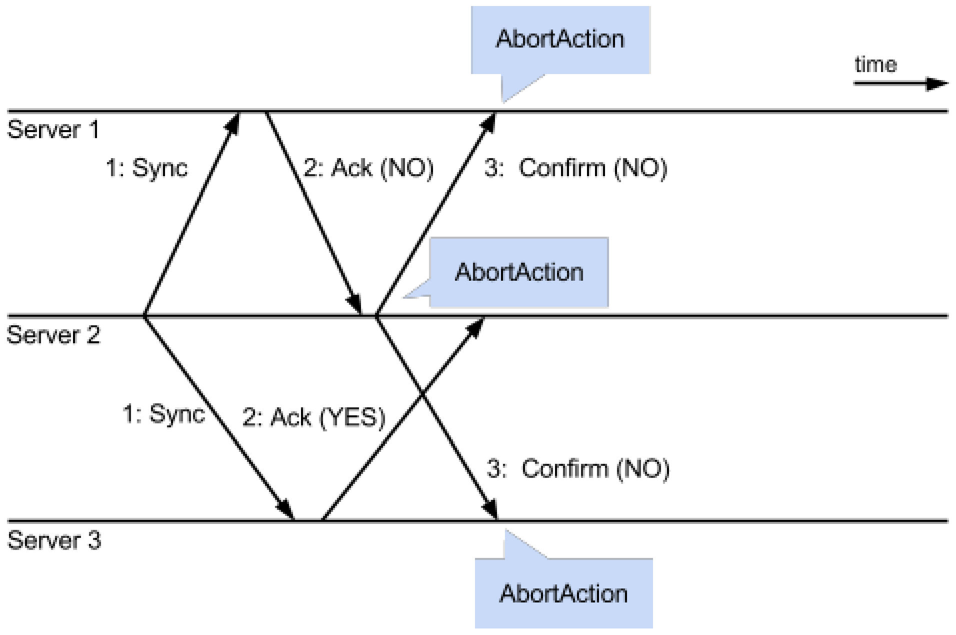

Figure 3. Synchronize action protocol; unsuccessful because of negative acknowledge.

In case that the server does not receive an acknowledge by every other server in a reasonable amount of time, the server will send an abort message. The a failure of server is recorded and after a predetermined number of failures that server is removed from the list.

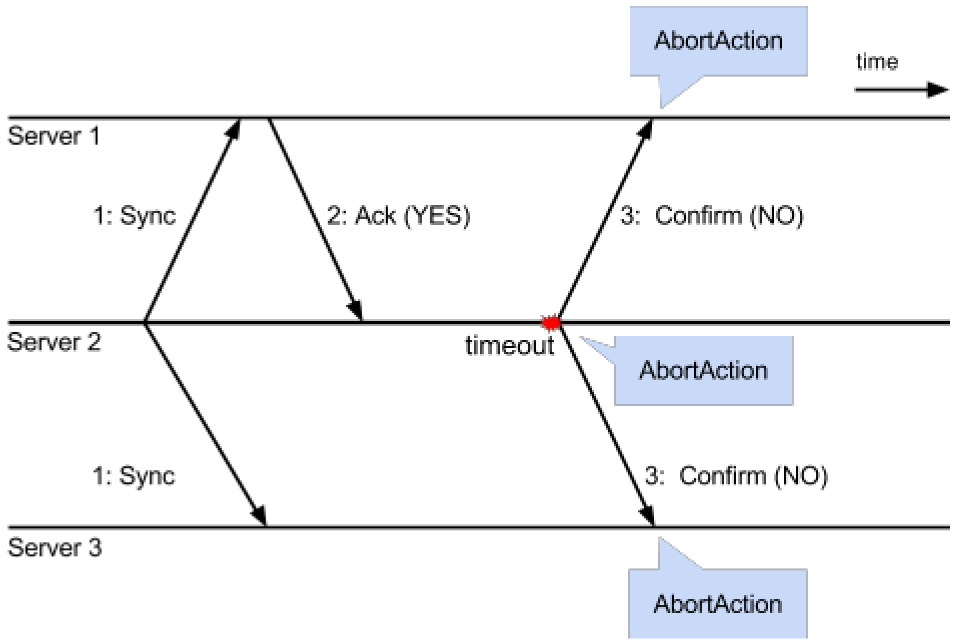

Figure 4. Synchronize action protocol; unsuccessful - missing ack messages.

4.1.1 Fault-tolerance

The servers keep a list of all the servers that are connected at a certain point in time and they are responsible for managing this list meaning that they should detect when any other server stops responding. We use a mechanism base on consequent number of failures to detect when a server crashes. This number can be predefined within the system and it's not relevant right now for the explanation.

If the one of the servers detects that another server stopped responding it removes it from the list and removes also his Players from the map. All the servers should be able to detect another server crash so no synchronization messages are exchanged at this point.

When a unit crashes it is up to that unit's server to detect the failure and inform all the other servers that this unit is no longer part of the world.

Both server and unit crashes are logged in the server logs. The first kind is logged by their own and the second one is issued by the unit's server and is the logged in the other server when they receive the message informing that a certain unit disconnected.

A unit may connect during the simulation.

4.2 Additional Features

The additional features that our system comprises are:

- Advanced fault-tolerance: As explained above in sections 4.1.1 reconnects from units are possible in our system. A player can also disconnect from one server and connect to another.

5. Experimental Results

5.1 Setup

For carrying out our experiments we used one of the Clusters of the DAS-4, The Distributed ASCI Supercomputer 4. To launch the application through the several nodes we used human input. We didn't use any libraries

5.2 Experiments

To prove the operation of our system we carried out several experiments with different environments. The time was not measured in these experiments since that was not a key factor. We checked processor usage and this was steady around 200% which means that 2 of the 16 processors of a node where being used by the servers to simulate the experiment.

5.2.1 Experiment 1

In this experiment we tested our system with the minimum requirements: 100 players and 20 dragons which were uniformly distributed across 5 server nodes.

Every server was able to detect when the game ended. By analysing the log files we concluded that state was kept synchronized across all servers. On average there were 77 players remaining. The players manage to win with such a margin because they have the ability to heal each other.

5.2.2 Experiment 2

This experiment was designed to test our system's response to unbalanced number of units across the servers. While we still have 5 server nodes and 20 dragons distributed equally by the servers now we have the extreme case in which all the 100 players are connected to the same server.

The experiment ran successfully. On average there were 81 players remaining at the end of the game.

5.2.3 Experiment 3

This experiment tested the system response to a server failure. We used 5 server nodes and during the game execution one of them was disconnected.

The experiment ran until there were only 59 players remaining. This number of players at the end is what was expected considering that the previous experiments showed higher number of players at the end. This could also be proven by the logs of the servers where the disconnect from the server was reported.

5.2.4 Experiment 4

How our system responds in the event of a client crashing or disconnection was the intent of this experiment. With the system up and running with 5 servers and 20 players connected to each we suddenly disconnected 10 of the 100 players originally connected and it was visible the drop in the number of active players, also confirmed by the DISCONNECTED_UNIT event in the logs of the servers.

6. Discussion

When deciding whether an action should be performed or not we have to take into account the state of the other system entities. We can either chose a simpler and faster method that can lead to some inconsistencies in the state of the battlefields, or a more complex algorithm which does a better job of maintaining the game state consistent. In this system we chose to give a bigger importance to the consistency, so we make sure that before a server applies a client action it checks that it is valid according to its local state as well as with the state of all other servers.

From our experience developing the distributed DAS we believe it to be a good solution for WantDS to implement their game engine. The non-distributed engine has some constraints related to fault-tolerance, scalability and performance that the distributed version, to some extent, solves.

Although our system performed well for the requirements, if the number of entities in our system would be considerably higher, our decision for the server to synchronize a client action as soon as it arrives might decrease the system performance. To improve this issue we believe that the server could group the actions into groups based on time intervals and then synchronize them as a whole. This would decrease the amount of messages exchanged between the servers which is the most problematic factor.

7. Conclusion

In this paper we presented our implementation details of the distributed arena system. Our approach is based on simple actions that we try to carry out as soon as possible relieving the server from long queues of actions to be performed sometime in the future. This approach has proven to be feasible but also challenging due to the fact that actions reach the servers at an high rate and a lot of computing is required to deal with the computation needed to compare the temporary actions. As future work we would be interested in allowing servers to connect to the system in runtime. This would be possible by installing a new group view where the information of the game state would be shared with this new server and this server would be made ready to the normal execution of actions. We think our work can serve as a launch pad for the development of a more complex distributed game engine.

8. References

[1] Tanenbaum, Andrew S., and Maarten Van Steen. "Distributed Systems: Principles and Paradigms." (2006).

[2] Brewer, Eric. "CAP twelve years later: How the." Computer 45.2 (2012): 23-29.

9. Appendix

This report is public and is available with the source-code at the following GitHub repository: https://github.com/vascoconde/DCS-DAS

We apply for the following bonuses:

- Advanced fault-tolerance (section 4.2);
- Open-source source code and public report;
- Excellent report;

Time spent in the assignment:

- the **total-time** = 159 hours;
- the **think-time** = 15 hours;
- the **dev-time** = 98 hours;
- the **xp-time** = 15 hours;
- the **analysis-time** = 12 hours;
- the **write-time** = 14 hours;
- the **wasted-time** = 5 hours.