The Dragons Arena System: Distributed Simulation of Virtual Worlds

IN4391: Distributed Computing Systems

Philippe Louchtch, 4697685
p.a.louchtch@student.tudelft.nl

Blazej Kula, 4746724
b.kula@student.tudelft.nl

Support cast: Jan S. Rellermeyer, Laurens Versluis, Sacheendra Talluri

April 4, 2018

Abstract

This report discuss implementation of a distributed version of the Dragon Arena System (DAG). The system must by scalable, support multiple concurrent users, be fault-tolerant and performant. Proposed system design is blockchain-inspired with many servers agreeing on one version of the game state. Stateless Rest API is used for communication. We analyzed system with previously listed requirements. Main finding is that the system exhibits good fault-tolerance but is lacking in scalability. It can tolerate $< 50\%$ of servers exhibiting byzantine faults.

1 Introduction

The Dragon Arena System is a an online warfare game developed by fictious company WantGame BV. The company outsourced creating proof-of-concept of the distributed version of their game engine to the authors of this report as they believed this version will be scalable and fault-tolerant. They want authors to investigate feasibility of this approach. The system will be implemented in Java using Spring framework. Communication will be implemented using REST API while keeping in mind that user clients will make requests to server(s) and server will never initiate communication with a client. The report will start by providing background on the game, continue with the system design, which will lead to the results of experiments. Report will end with the discussion on feasibility and conclusions.

2 Background on Application

The specification stated that the system must be capable of supporting hundreds of concurrent players battling AI controlled dragons on a 25x25 grid battlefield.

Players control characters that can move on the grid, heal other players and attack dragons within range. Dragons are stationary, AI controlled units that can attack players. Each participant has AP (attack power) and HP (health points). Game ends when one type of participants is eliminated from the game. Players can issue actions with certain delay. WantGame BV wants their game to be scalable, consistent, fault-tolerant and performant.

As the system needs to support multiple concurrent users playing in the same game session it means that it must be able to scale up in order to handle those users and their actions. To that end multiple servers are used to handle actions in a parallel way.

Inconsistency might be caused by contradicting user actions. In that case the same action must chosen and applied across all server nodes to ensure that game state is the same for all clients (players).

Fault-tolerance means that failures of servers and/or clients does not disrupt game session and connected components are silently reconnected to working copies. Basic fault-tolerance include only faill-stop of components. However system which design is introduced later supports more advanced failure modes such as limited byzantine fault-tolerance.

In massively multiplayer online games response time is important for players. They need to see results of their actions with limited delay, because of that WantGame BV wants to ensure good performance of created system.

3 System Design

3.1 System overview

Implemented system is inspired by the blockchain technology. Each server maintains two complete game (world) states. First world state, called accepted, is the most recent that was accepted by all of the correct servers in the cluster and is sent to the clients when they query for the newest world state. Second world state, called current, starts as a copy of the accepted world state. Actions (commands) received from players are applied in turn to the current world state. If a command is applied successfully, that means it is a correct with respect to the game rules, then that command is added to the server's ledger of applied commands and that command is gossiped to the cluster. Whe another server receives such command it either applies it to its current world state or rejects it as being not invalid with respect to its current world state.

Every 200 ms servers initiate a process called ledger exchange. When a server initiates (or receives information that exchange was started) it freezes its current world state and sends its ledger, possibly signed with the asymmetrical cryptography schema, containing all of the approved commands to every other server. When server received ledgers from all of the servers (or after a certain timeout) it elects the best ledger, that is the one that approved the largest number of commands. Then it applies those commands to the accepted world state and a copy of it becomes current world state. Ledger is now hashed, cleared

and new commands, which have been stored in a cache in the meantime, can now be applied. If for some reason a server becomes disconnected from the rest of the cluster it will apply arbitrary ledgers to its world state. Those can be either its ledger, as it can't connect to any other server, or the best ledger from the subset of the cluster that it has connection to. After communicating with the rest of the cluster it will notice that its world state (based on the last ledger hash) is different than those communicated by majority of the servers during ledger exchange. It will then request the accepted world state from the cluster to synchronize with it. In the same schema if minority of the servers exhibits failure in a byzantine way, majority still will be able to decide on the correct world state and can possibly remove faulty/malicious servers from the cluster.

Every client sends commands to the central server which is a load-balancer that evenly distributes them across servers. Those commands can be signed with a key in the asymmetrical cryptography schema to ensure that commands are sent by the authorized player. Communication protocol is stateless and is exposed in the form of the Rest API. Client can issue one of the game action together with the spawn action to begin the game and a command to get the newest accepted world state. This means that if a server fails clients never see it, in the worst case clients can see that one of their action had no effect if it has been sent to the failing server. Issue of the central load-balancer failure can be fixed by introducing a schema where server upon first command to a server receives addresses of all the servers in the cluster and client sends its commands to a random one, possible different each time.

Counter-intuitively adding additional servers to the cluster does not help to scale system up but it mainly helps with fault-tolerance. While designed system can supports 100 players with 20 dragons on 5 server nodes, in order to enhance scalability a change in its protocol would have to be made. First option is sharding, so given subset of cluster is responsible for given area in the grid and interacts with different subsets only if their areas intersect. Second option is a change in the ledger exchange protocol. Servers would not gossip received messages but still add them to their ledgers. During the ledger exchange they would try to apply received ledgers on top of theirs. Each server would broadcast this sequence and how many messages had been applied using it and one with the most messages applied would become accepted world state.

All major system events are logged in the cloud via Rapid7 provider. This way all servers data is logged in once place for accessibility and retrieving data for experiments. For testing purposes system runs in the docker environment and requires some time for the cluster discovery. This is achieved by simply scanning adjacent local IP addresses (as docker assigns them in such a way) before first client sends first command which is a sign to start ledger exchanging. Docker additionally acts as the load-balancer as it relays commands to servers.

3.2 Additional System Features

3.2.1 Advanced fault-tolerance

As the system is blockchain inspired it support limited byzantine fault-tolerant. Limited means that if an adversary can take control of majority of the servers system will be faulty. Moreover system does not support excluding malicious servers from the cluster however this can be added in future development.

3.2.2 Security

As the system uses cryptographic hashes and is designed to support asymmetrical cryptography for commands exchange it is secure against malicious external agents that can intercepts messages.

4 Experimental Results

4.1 Experimental setup

Unfortunately, most of our time went into making the system operational on an acceptable level, from the distributed part down to our re-implemented game logic. Therefore, we did not manage to prepare a lot of experiments.

Our set-up consists of the servers running within a docker swarm on one machine as a *docker-compose* service. The configuration file can be found in our github repository (see the Discussion section). The scaling of servers was done by manipulating the `deploy.replicas` property with the desired amount of servers. The load-balancing is handled by the docker *overlay* network.

Furthermore, we have three tools available.

1. A simple game board html+js based visualization tool. This tool continuously gets a world state and draws the locations of the units on a grid.

2. A simple html+js based latency calculation utility. Just like the visualizer it continuously gets a world state, but merely computes the rolling average latency between each successful world state get.

3. A java based client simulator. This tool is a very trivial AI to play the knights. Used to simulate a load on the system. Can easily scale to hundreds of concurrent players.

4.2 Experiments

4.2.1 Latency of getting a *valid* World State

In this experiment we have measured the rolling average latency between successful world state retrievals from the server(s). The setup procedure was as follows:

1. The docker service is started we wait for the startup and discovery to be completed.

2. Start 100 client bots to simulate a load on the system and possibly cause some servers to de-sync/fork off. The bots stop playing after 8 seconds

3. During startup of the bots, the latency calculation tool is started.

Below, the results per number of servers. The latency is measured at ledger number ≈400

	1 server	2 servers	3 servers	4 servers	5 servers
Latency	≈ 500 ms	≈ 750 ms	≈ 750 ms	≈1650 ms	≈2100 ms

Table 1: Time sheet

The results are interesting and somewhat unexpected. With more than two servers, it was clear from the logs that at least one server was de-synchronized, this also clearly impacts the latency from a technical perspective. But the result for a single and two servers is higher than expected. Might be caused by a faulty calculation in our tool and/or the technology used (ajax calls, jquery and FireFox).

4.2.2 Finishing Game with Five Servers, Hundred Players and Five dragons

This was the simplest experiment, but difficult to prove. The experiment was done to show two things, one to show our system capable of handling the load and two, at the same time showing that it is fault tolerant. The faults are due to the unsolved issues and implementation deficiencies of our system, here our fault-tolerant architecture, however rough, shows its potential.

Please see the logs and a picture in our repository, under the directory `results`.

5 Discussion

Unfortunately implementation of the system didn't go smoothly. We had troubles deciding on the final architecture and settled for quite complicated design which in the hindsight was overkill. This combined with the fact that we didn't start until late in the quarter caused problems during final presentation of our system. Since then we have finished implementing it in order to run experiments. Moreover we didn't test it in the distributed way over the network. All our tests where done in a docker swarm. Many of the implementation problems were discovered and fixed during last-minute tests which additionally hindered our progress. It is also important to notice that large chunk of development time was performed with the *extreme programming* methodology. This was caused by the fact that initial implementation phase required good knowledge of the entire

code-base and was rather one-man-job. We also observed that due to the random anomalies our solution, when scaled to five servers, often de-synchronizes, or *forks*, when bombarded with requests from many clients. This casuses the clients to receive different world states on consecutive calls. We managed to mostly alleviate the issue by using a simple solution where the client rejects a world state that has a lower version than it has last seen. By no means is this a perfect solution, but it manages to work well enough in practice. A better solution would be to utilize hashes and verify, client or server-side, that the new world state is indeed a child of the last seen state.

As stated in the system design there are a lot of vectors which can be used to make system more viable, i.e. sharding of the battlefield or asymmetric cryptography for validating commands. To that end we decided to open-source the project, the repository can be found here: `https://github.com/PhilipeLouchtch/in4391-2018-team17`.

Main question still stands: whether the designed system is useful for WantGame BV. Performed experiments show that even with corners cut implemented system is still able to handle 100 players and 20 dragons with good performance. Failing one or more of the servers did not affect clients in any significant way and those fails were handled gracefully internally. Because of the time constraints we were not able to perform further tests with larger number of players but we assume that, because of the relatively long ledger exchange delay (200 ms), the system would still be performant. We ran experiments on the cluster that consisted of 5 server nodes so we don't know how larger one would behave. Overall we think that traditional design with mirrored backups would be a better option for MMO game such as DAS. In the situation where fault-tolerance is more important than scalability proposed system might be good option, but if the game would require more resources this design is not ideal.

6 Conclusion

The implementation of a distributed system is challenging with many pitfalls along the way. We can safely say that a simpler design, in the form of multiple reflected servers with a master node would be more suitable and significantly easier to implement. However, the project and implementing our solution was an interesting experience and we learned a lot, that was the whole point wasn't it? Our only regret is that we started our work too late. Performed experiments show that the system shows promise of being performant and scalable under the assignment requirements. Blockchain-inspired design shows great potential for fault-tolerance of the system. Additional features and enhancement proposed throughout this report provide nice starting point for further work.

7 Appendix

7.1 Time sheet

Time/Who	Blazej Kula	Philippe Louchtch
think-time	12	12
dev-time	46	75
xp-time	2	2
analysis-time	4	1
write-time	8	1
wasted-time	4	10
= total-time	76	101

Table 2: Time sheet