Distributed Systems (400130)
Large Lab Exercise B

The Dragons Arena System

Course manager
Alexandru Iosup

Teaching Assistants
Laurens Versluis
Alexandru Uta
Ahmed Musaafir

Group 9
Group ID: 800
Bjarni Jens Kristinsson (`bjarnijens.kristinsson@student.vu.nl`)
Daniel Edward Galea (`d.e.galea@student.vu.nl`)
Johannes Wikner (`k.e.j.wikner@student.vu.nl`)
Konrad Karaś (`k.karas@student.vu.nl`)
Saidgani Musaev (`s.musaev@student.vu.nl`)
Thanh Long Tran (`t.l.tran@student.vu.nl`)

Due date: December 15, 2017

1 Abstract

We are hired as consultants for a large gaming firm, WantGame BV, to evaluate and implement a Proof-of-Concept (PoC) of a distributed game engine of a sequential game they're almost finished with. We're given the code and are expected to report back whether this is feasible or not, what the gains would be, the drawbacks, and some runnable code to demonstrate. We manage to show that this is indeed feasible but at the cost of user experience (UX).

2 Introduction

The game is called Dragon Arena System (DAS) where players and dragons live together on a rectangular 25x25 square battlefield. Players can move around, heal, and attack dragons while dragons (non-playable characters) are stationary and can only attack. The players are up against the dragons and the game is over when either all players or dragons are wiped out.

The goal when implementing the distributed version is to ensure replication, consistency, and fault tolerance in order to provide the best possible experience to the players. First an overview of the application will be given in which the client-side, server-side, and communication modules will be explained. Then a background on the application will be given, here a more in-depth explanation is given about the approaches taken to achieve replication, consistency, and fault-tolerance. Following this the scope of the system and its limitations are explained so that a better idea of what the system does can be achieved. After the scope has been laid out we delve deeper into the system design, explaining implementation specifics about each component of the system. Experimental results are shown, followed by a discussion, and finally the paper is concluded.

3 Background on Application

The DAS game initializes a 25x25 square battlefield and a given number of dragons at random free positions with randomized attack power (AP) and health points (HP). When clients connect to the game they connect to one of the running servers. That server initializes a player in the same manner, with a random position, a random HP and AP. Players and dragons are allowed to make one action such as move, attack, heal, etc. at a fixed time interval.

With respect to a distributed system that can handle this kind of game mechanics we will now briefly discuss how we can accomplish replication, consistency and fault tolerance.

3.1 Replication

It is important that the game state is replicated in a way such that any node (client or server) at any given time can have a failure without any game state being lost. While replication adds redundancy in terms of memory usage and network traffic, it not only ensures that game state never is lost but it also improves accessibility as any server can be used by any client. We will thus have fully replicated game state on all nodes in the system.

3.2 Consistency

Many approaches of synchronization and consistency were discussed, see Section 7.1. In order to have a working PoC, we opted for a solution with minimum developer complexity that we could improved at a later stage if necessary. Many synchronization algorithms for games such as *trailing-state-synchronization* and *time warp synchronization* are *optimistic*, meaning that rollbacks may be necessary if an older event than the latest applied one is received. A *conservative algorithm* enforces strong consistency and for our particular case where actions are sent at fixed intervals we found it feasible to use.

After discussing with our teaching assistant we came to the conclusion that using a mechanism similar to Baruch Awerbuch's *alpha, beta and gamma synchronizers*[1] would be ideal. The main idea from their work that we will use is that a message can safely be applied whenever all other nodes have acknowledged the message.

3.3 Fault-tolerance

In a distributed system, any node can, at any given time, have a failure. This needs to be taken into account when designing any distributed system. A failure can be either server-side or client-side. If a client fails, it must be able to reconnect to the game and be recognized as a reconnecting player. Similarly if a server fails, its clients need to be able to reconnect to a new server. A failing server will not be able to acknowledge messages from other servers, which is necessary for our consistency algorithm. In order to not hurt the availability of the entire system we also need to mitigate this kind of failures.

3.4 Consistency vs. availability

As the CAP theorem states, given that we have partition tolerance, which fault tolerance implies, we have to choose between consistency and availability for our system. In order to have as predictable results as possible we conclude that having high consistency is more attractive than availability. This will mean that in a real world scenario a player may have a bad game experience but for the purpose of a PoC we are more interested in predictable simulation results than user experience. For the mechanics of our particular DAS implementation we can also allow this as it may take some time from moment when user sends own action to server and moment when action will be approved. If we were to build a network engine for a more interactive game, such as a first person shooter, we would have to come up with a different approach.

4 Scope and limitations

The scope of the project is to provide a distributed system that can scale and perform well. Multiple servers will be utilized and the servers will communicate in a peer-to-peer manner, as this will be a vital component when it comes to achieving the previously mentioned scalability and performance. However, to ensure that scalability, fault-tolerance, and consistency are implemented well we decided to omit certain features with the reason being that they would raise less issues.

4.1 Player integrity

The integrity of players will not be taken into account, meaning that a player could be impersonated by a third party. The issue is possible to mitigate however, we imagine that a client that enters the game will get a unique authentication token associated with her controlled unit.

4.2 Auto scaling

In a real world scenario, it would make sense to automatically deploy and take down servers depending on the current load. For this project we limit ourselves to use a fixed number of game servers.

5 System Design

In this section we outline the design decisions made for our system. We describe the whole life-cycle of a game from initialization to finish. Finally we present our solutions for server and client failures.

[1]`https://en.wikipedia.org/wiki/Synchronizer_(algorithm)`

5.1 State synchronization

For our design we opt for strong consitency across all nodes. When a client wants to issue an action, its server node first gives the message a timestamp. Timestamping is used to give other servers an idea about when an action was requested. For this prototype we make the assumption that servers have synchronized clocks. Incoming messages will then be placed in a priority queue so that they are ordered by their timestamp. Next the server node will send an *ask* message to its server peers, they will verify that the action can be applied on their replica and either repsond with an *accept* or *rejcet* message. Because we have to assume that network partitions can happen, we can't assume that all peers will respond to the *ask* message. We therefore introduce the rule that the message can be considered accepted if no remote sever has rejected it after 2 seconds. When a message has been accepeted, the server will broadcast to all connected nodes a *commit* message.

5.2 Game servers

The game state is managed by 5 game server nodes, each holding a replicated copy of the entire game state. Clients can connect to any of these but will choose the server that is most appropriate for them. An appropriate server is typically one with fast response time, if it has fast response time we can assume that is closely situated and probably has a low load. As a secondary heuristic we can also take into account the number of connected clients so that if we have two or more servers which all have fast response time the client selects the one with the fewest connected players.

For the sake of constructing a completely distributed system the the game servers are hard-coded. Nodes thus discover each other by connecting to the list of given servers. In Section 7.1.4 we discuss how we can use a *bootstrapping node* as a centralized source for retrieving these nodes and the benefits and disadvantages with this.

5.3 Communication

Client-server and server-server communication is done using TCP sockets on which messages are sent and received. When a client connects to a game server, it will either (1) join the game as a new player or (2) rejoin with an existing player due to a failure. For (2), a newly connected client initially transfer its player id. In case (1) it signals the server that it's a newly connected client that needs a player associated with it by simply omitting any id field in the message. If the server accepts the request, it responds by transferring its current game state and the player id. After the client has recieved this respond we consider the client as initialized and ready and synchronized with the game state, meaning that it can start sending messages on behalf of its player as well as receiving state updates over the socket connection.

Server-to-server connections are two ways meaning that one server will have $2n$ server-to-server sockets open where n is the total number of servers. While this add some extra overhead it allows server nodes to be symmetrical in the context of setting up server-to-server communication. An early implementation attempted to use a single socket, this however raised the question about which server should initiate the connection. In order avoid answering this question we decided that all server nodes are responsible for setting up sockets with all other server nodes.

5.4 Client interaction handling

Client participates in the game by exchanging actions with server. Client can send messages to inform server about desired actions to perform and also receives messages from server with game state updates that should be applied locally.

For testing purposes all clients are being controlled a simulation algorithm. Each player unit tries to approach the nearest dragon unit and attack it. Player unit can also heal other nearby players if their HP are below 50%.

Each action that player wants to perform is being sent to server. Player cannot perform any actions locally without acceptance of the server. This approach guarantees consistency of battlefield states in all clients participating in the game so that rollbacks never will be necessary.

When an action request was sent to server, client waits for response with updated game state. If such a response was received, it means that requested action was accepted by all servers and client can apply it locally as well. Client also receives actions that were performed by other players.

Client Application has to be able to deal with lost connections. When connection with server is lost, client tries to reconnect to another server in the same pattern as it was done initially by choosing the best available server to connect. After successful reconnection, client receives current game state from the new server and is able to update its unit by the ID that remains unchanged through the whole game.

6 Experimental Results

During the development phase the system was only tested locally on a single computer running a small number of servers and clients. This made debugging the system easier. Throughout the development we slowly scaled the number of servers and clients for testing purposes.

6.1 Experimental setup

We have created two working environments for our experiments to monitor the properties of our system. One smaller setup involving only one computer (a personal laptop) running 3 servers and 20 clients (15 player clients and 5 dragon clients). For the larger setup none of our computers were powerful enough to run a huge amount of clients. So we have created a local network between our computers, each computer running a few number of servers or some clients that connect to the server.

For the experiments we did not use any special workload or monitoring tools. The applications are pure Java applications that were launched on a JVM on the given operation system of the current computer. To analyze the features of the system we simply inspected the time-stamp of important events during the simulation, such as the response time of a network request or the performance of our synchronization algorithm. It is important to mention that the system contains randomized elements and to ensure the reproducibility of simulations the random number generation is seeded. However, since the network conditions cannot be guaranteed to be the same throughout the experiments, the simulations might yield different end results.

6.2 Experiments

There were three features of the system that we wanted to analyze: scalability, fault-tolerance and consistency. For each feature we made different experiment scenarios to test our system. The following subsections will detail the experiments conducted to analyze the feature of the system.

6.2.1 Scalability

In order to test scalability we would add and remove clients and/or servers so that we can observe how the system reacts. The maximum amount of clients that could be connected to each server was set by a threshold which changed depending on the total amount of players connected to all servers and the number of live servers. This meant that the threshold was not strict. The reason being that if a server currently has a threshold of ten clients it can then change and have a threshold of eight clients instead. What would happen in this scenario is that the ten clients remain connected but no new clients can connect to that server.

When removing and adding clients we noticed that the clients would in fact try to connect to the server with the best ping, but would not connect to it if the client-threshold for that server was reached or exceeded. Furthermore, the threshold also increased and decreased depending on the amount of clients connected and live servers available. However, we noticed that the threshold changed frequently (as frequent as the amount of clients and servers that would connect or disconnect) and so it was often the case that the threshold of each server was exceeded but the server would still function as it should.

6.2.2 Fault-tolerance

There are two scenarios that system needs to handle to keep the game going and not result in system failure that prevents the continuation of the game. The system needs to handle the situation when one of the game servers crashes and disconnects from the network. As described in Section 5.4 the disconnecting clients will try to reconnect to the network in case a node failure. The redistribution of the clients when a crashed server comes back on-line is not implemented, as a result, the reconnection of the server was not tested.

In the experiment we were normally running the simulations, but at any given time we intentionally and manually shut down one of the servers to see how the system responds to such an event. In this situation the clients that were connected to the failing server quickly noticed the unresponsiveness of the server and immediately started the reconnection operation. Since the clients ping all the addresses for available servers, this process is slightly slower in the large scale setup. However, the clients were able to reconnect to a new server and continue the game without affecting the game of other clients.

6.2.3 Consistency

By using the synchronization method described in Section 5.1 it can be proven that at any given time a server will be at most $n-1$ steps behind the current state of the game (n is the number of servers). This will not be proven in this report, but the experiment results support the previous statement and we can imagine the case where each server has a pending message that has just been accepted by all other server peers. In that situation each server will have its own message applied while the commit messages still are being transmitted over the network. In both experiment setups we simply ran the simulations without any interference. We monitored the state of the clients manually by selecting a number of clients as checking all the clients would require large amount of time and work. After the simulations have ended we could see if all the clients had reached the same end state or not.

In the small scale work environment the system performed very well keeping the state of the game consistent throughout the simulation and all the clients reached the same end state.

7 Discussion

In this section we discuss any trade-offs in system design, and why the distributed version of the DAS should be used over the non-distributed version. The system was designed to be robust and provide a good user experience. It offers scalability, consistency, and fault-tolerance which on their own should be enough to choose it over the non-distributed version.

Throughout the entire design process the features changed, and one important feature that we decided to drop was the "Bootstrapping node". This feature was removed as it would have been a single and important point of failure whilst also limiting the user experience and acting as a bottleneck. Hard-coding the list of servers ensured that clients could connect to servers without issues.

7.1 Alternative implementations

Different methods of synchronization were discussed during the design phase.

7.1.1 Transactional updates with rollback and reapplication

The main contesting idea involved achieving *eventual consistency* of the game-state between the servers and clients at the cost of user perceived fluctuation of game state.

Every client would take actions based on its current view of the game state. The client sends its action to its server *with an attached time stamp* based on the client's system clock.[2] The server responsible for said client broadcast the action to the other servers along with the client's time stamp. The core idea is that the broadcasting server *doesn't have to wait for an answer from the other servers* whether the move is accepted or not. All servers run the same logic and will therefore reach the same conclusion given they have the same set of actions. And all servers will eventually have the same data as broadcast action messages may be delayed and arrive at servers at different times in different order but eventually they will all arrive at all destinations.

When a server receives an action message with an older time stamp than that of its newest message, the server will rollback, apply it and then reapply the newer messages. This may cause some changes of game state to the players. An old message may even render newer actions invalid causing substantial changes to the game history and perceived game state for the players. This is the biggest drawback of described synchronization approach. This might be litigated with some sort of periodic all-hands server synchronization where messages that have not been received by all servers are rendered invalid (i.e. setting timeout to messages). The servers having the invalid messages discard them and restructure game state and the server not having them mark the fingerprint and discard them upon arrival when they arrive after the synchronization points.

The upside of this synchronization method is less server-to-server communication (no acknowledgement messages) and smooth game play (high availability) as we never need to lock or freeze the game state to synchronize servers.

7.1.2 Bucket synchronization

To have a consistent state across all servers we could also be using bucket synchronization. This means that on a fixed interval, all game servers will exchange their batch of messages that they have recorded from their clients with all other game servers. All game servers have the same strategy of merging these batches so the resulting game state will be consistent across all nodes. In order to mitigate conflicting actions we introduce the following heuristics. The conflicting message with the higher time stamp will be rejected in favor of the one with the lower time stamp. In the case where the time stamps are equal, the message from the server with the lower index will have precedence. We will briefly discuss how a server is assigned an index in the next section.

We want synchronization to happen at fixed intervals across all server nodes but in a real world scenario we can not assume that we have any globally synchronized clock. Instead, a newly deployed server will wait for an incoming synchronization message to be transmitted and will on that point reset its timer. It will be an empty batch because the newly deployed server will not have any clients connected to it yet. After this initial synchronization the server knows when the next synchronization will occur.

There will be a short time window when all servers broadcast their recorded messages. Any batch not delivered during this time window, has to be discarded to maintain a good game experience for the majority of the players.

7.1.3 Zone Allocation

Another idea involved achieving less network overheads and better performance of the application. We came up with following idea (which is little bit more difficult to implement).

Every server allocates some *zone* on a global map. Then only one server is responsible for this zone. So when Client sends own move to own server, this move needn't be approved by other servers. All servers can easily take new move from any other server, because they can just trust

[2]We make an assumption that the server's and client's clocks are fairly synchronized, i.e. within tens or hundreds of millisecond from each other. Even if they are not this method of game state synchronization will work but it will be unevenly skewed towards the player with the fastest clock.

each other. With this approach we remove our heavy synchronization protocol. But not fully, because of squares located on the edges of zones should be synchronized between servers. This synchronization can be done by the same *strong synchronization protocol* which we have in our current implementation. Also we can have another problem here *load imbalance*. Of course *zone allocation* could be dynamic and should be redone periodically.

The upside of this synchronization method is less server-to-server communication (transactional updates and acknowledgement messages) and smooth game play (high availability) as we never need to lock or freeze the game state to synchronize servers. Such approach will reduce network overheads and improve performance.

7.1.4 Bootstrapping node

To continuously check the availability of the game servers and to expose this information to new clients we could introduce a *bootstrapping node*. It would expose an interface where game servers register themselves when they go online and unregister as they go offline. Upon registration an index is generated for the game server, the index is incremented every time a server registers to ensure different servers have different indices.

Clients would also use the bootstrapping node as their entry point to the game. A client will retrieve a list of available game servers and select the best suited one to connect to. We'd use the server with the fewest clients connected. This service can be implemented using a tiny HTTP REST API. For checking server availability, our bootstrapping node also features an HTTP client which queries the registered game server nodes for their status.

This idea was rejected because of it causes the *SPOF (single point of failure)* problem which is very bad for distributed systems.

8 Conclusion

The strong synchronization that we have doesn't scale very well. Each action requires $O(n)$ messages (n is the number of servers and clients). If we have a huge amount of servers with long distances between themselves (long pings) we can face real problems. However, our current application implements *Mirrored-Server Architecture* very well. Game state is replicated between servers and failure on one server doesn't have a cascading effect through the whole system apart from causing higher loads on the other servers as clients reconnect to the game.

A Time sheets

Rough worklog:

- 2017-11-16: 6 x 3 hours of planning
- 2017-11-21: 5 x 2 hours of planning, creating GitHub issues and dividing work amongst ourselves
- 2017-11-22: 6 x 4 of planning, working on the report, networking design, bootstrapping node implementation
- 2017-11-23: 3 x 6 hours of discussion about design and coding
- 2017-11-25: 2 x 4 hours of coding
- 2017-11-26: 6 x 6 hours of discussions about design and some coding
- 2017-11-28: 6 x 6 hours of coding and discussions
- 2017-11-30: 4 x 4 hours of meeting with TA and designing synchronization protocol
- 2017-12-01: 2 x 4 hours of discussions and coding
- 2017-12-02: 4 x 4 hours of discussions and coding
- 2017-12-03: 4 x 4 hours of discussions and coding
- 2017-12-04: 6 x 10 hours of various coding style (single, double, multi, reviews and discussion)
- 2017-12-05: 6 x 10 hours of various coding style (single, double, multi, reviews and discussion)
- 2017-12-06: 5 x 6 hours of bugfixing before and after mandatory presentation
- 2017-12-13: 5 x 6 hours of running experiments and writing report
- 2017-12-14: 5 x 6 hours of running experiments and writing report
- 2017-12-15: 3 x 6 hours of completing report

This equals to *434 hours* total work.