The Dragons Arena System: Distributed Simulation of Virtual Worlds

D. Diomaiuta 2553339 | V. Gkanasoulis 2617203 | F.M. van Beest 1620746 | W. Rens 2528808 | M. Went 2507013

email: {d.diomaiuta, v.gkanasoulis, f.m.vanbeest, w.rens, m.m.went}@student.vu.nl

Distributed Systems - Vrije Universiteit Amsterdam
Group 1 — Team 1300

Course Instructor: A. Iosup

Lab Assistants: A. Uta, L. Versluis

December 15, 2017

Abstract

Online gaming is very popular in the present day. Providing a platform that serves a large number of concurrent users requires a distributed system that is replicated, consistent, scalable and fault-tolerant. In this report we present an attempted solution for the Dragon Arena System where we address the previously mentioned properties that make distributed systems such complex systems.

1 Introduction

Our team was hired by the fictive company WantGame BV to develop a distributed game engine for the The Dragon Arena System (DAS). Distributed game engines have several benefits over the traditional client-server architecture, however there is a trade-off with development and management complexity. One of the benefits of a distributed game engine is that in case of a server failure the game can continue by having another server take over. Furthermore there is Geo-locating, which may facilitate lower latencies by placing server(s) close to players.

A fundamental challenge in distributed systems is synchronization, Cronin et al. [2] propose a new mechanism, which they call trailing state synchronization (TSS). By using an optimistic algorithm together with rollbacks at the time of detecting inconsistencies, TSS performs well in low latency environments such as first person shooters. Although the game-engine could benefit from this technique, having rollbacks in a board game may be more frustrating than in faster paced video games.

This paper will describe a distributed game engine that is resilient to server node crashes by multi-casting client actions in total order between five servers. Note that the amount of five servers has not been chosen arbitrarily but is a mandatory assignment requirement.

The next section will provide background information on the DAS and its system requirements, after which the high level architecture is introduced. More detailed information of the architecture is described in the subsections server and client. In Section 4 the experimental set-up is presented, which are followed up by a discussion in Section 5 and conclusion in Section 6. The paper concludes with recommendations for future work.

2 Background on Application

The DAS is intended as a game where real-life players fight computer controlled dragons. The virtual world is a 25×25 grid battlefield, each square in the battlefield can by occupied by at most one contender, be it a dragon or a player. All participants are initialized with some health points (hp) and some attack points (ap). Players can move on the battlefield where dragons cannot. When within striking range, players and dragons can attack each other, where the attack points of the attacking party are subtracted from the remaining health points of the receiving party. Whenever a participant has lost all its health points, it is removed from the battlefield. Players have the additional ability to heal other (nearby) players.

2.1 Requirements

WantGame BV consulted our team to design and implement a distributed, replicated and fault-tolerant version of the DAS. The following subsections describe the mandatory requirements [1] with regards to consistency, fault-tolerance and scalability to achieve such a version.

2.1.1 System operation requirements

In order to be able to assess the system, all players actions must be emulated. The in-game strategy of players is simple: they heal other players in range having less than 50% of their initial hp or move towards the closest dragon and attack otherwise. To simulate connection events it is required to use data taken from the Game Trace Archive [6].

2.1.2 Fault-tolerance

As stated in the contract description WantGame BV strives for an online warfare platform servicing many concurrent users. The DAS must therefore be designed such that it is resilient against failures from both client and server nodes. To ensure a consistent execution of the game all events must be logged in order of occurrence on at least two server nodes.

2.1.3 Scalability and Performance

To assess the designed system, the DAS must be demonstrated when running a replicated game state on 5 server nodes, containing 100 players and 20 dragons. The game should run until either class, dragons of players, eliminates the other class.

3 System Design

This section is divided into two parts. The first part outlines the system's architecture and communication mechanisms, while the second part describes the system operation, fault tolerance, and scalability components in detail.

3.1 System Architecture

For the main game server setup, five homogeneous game servers, that are fully connected using TCP, are setup. To keep the game state in sync the servers make use of a total-ordered multi-casting (TOM) pattern, which will be explained in more detail in Section 3.4. Furthermore, each server has a publish socket to push the game state to clients on a synchronized interval.

Clients have a list of hard-coded addresses for TCP and publish socket endpoints. They arbitrarily connect to a single server i.e. the client submits game actions to the same server as which it is subscribed to for receiving game states.

Figure 1 shows an overview of the proposed high level architecture. In the following subsections individual components are explained in more detail.

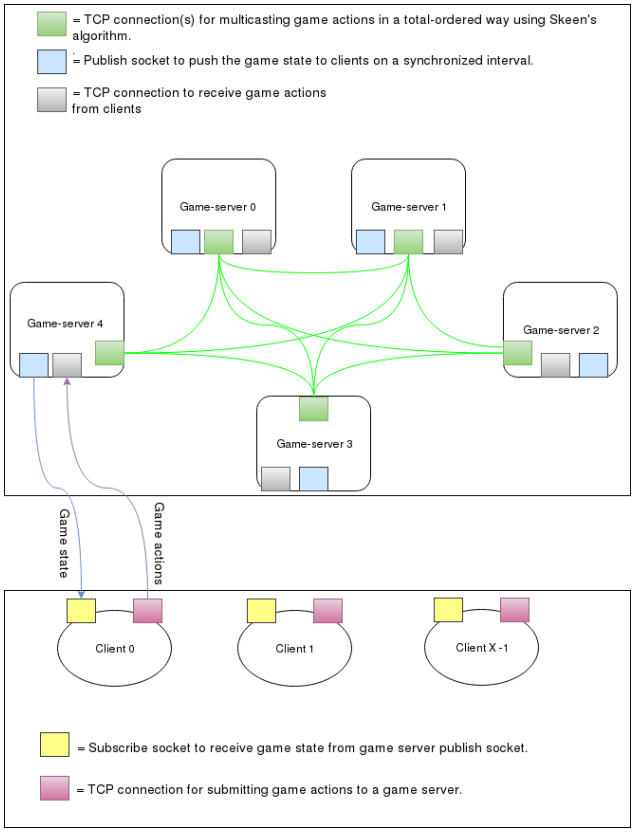

Figure 1: High level architecture. The top box shows the fully connected servers. The bottom box shows the clients and how they are connected to a server using a publish/subscribe principle.

3.2 Communication Mechanisms

The communication layer is where the Server-to-Server, Client-to-Server and Server-to-Client communication takes place. This was implemented using Java asynchronous TCP sockets.

A description of the three types of communications, follows:

- Server-to-Server (STS): The STS communication is a complete graph where the number of nodes represents the number of servers that are alive. The game servers need to communicate with each other in order to ensure game consistency. Every time a server receives a move from a client it initiates a multi-cast (which is implemented as a serial unicast) transmission to the other game servers. Section 3.4 describes the exact message exchanging required to ensure consistency.

- Client-to-Server (CTS): The game client needs to communicate with the server in order to issue the player's moves. This is a unicast transmission where the sender is the client and the receiver is the server to which the client is connected to.

- Server-to-Client (STC): The game server needs to notify the game clients whenever a move, that changes the battlefield, takes place. This is made by a multi-cast (which is implemented as a serial unicast) transmission where the server is the sender and the clients subscribed to that server are the receivers.

At the thread level both game servers and clients are implemented with two threads, one for sending and one for receiving packets. The game servers have an additional third thread, used during the bootstrap phase, that discovers the other alive servers. When a new server is discovered, a one-to-one communication channel is stored which can be re-used to send messages.

3.3 System Operations

To improve the AI that simulates a client move, a recursive search was implemented to find a player within the clients healing range (5 steps). The search pattern is a diamond shape as diagonal moves count as two steps. To improve the search efficiency of this recursive method, close-by dragons are searched as well. When found within the attack range (2 steps) the player can immediately attack if no other player needs healing.

If no player is found within 5 steps, or a dragon within 2, the player will move towards the closest dragon. Which is either found within the 5 step healing range, or a new search is started. This search is outside the 5 step range, to find the closest dragon. The player will move in that direction until he is either within attacking range or healing range of another player.

To test it's workability an execution experiment was devised to run locally only, and show how the AI handles the rule-set.

3.4 Consistency

To implement a consistency model for the system, total-ordered multi-casting (TOM) was selected. Specifically a TOM implementation with Skeen's algorithm utilizing Lamport's logical clocks [5] explained in [4]. The algorithm steps

are outlined in the list below. Upon receiving a client message, a game server will:

1. Accept the received client message and append the local Logical Clock time-stamp.
2. Multi-cast the message to the rest of the game servers and move it to the local un-deliverables queue.
3. Every other server stores the message to its local un-deliverables queue and sends back their proposed times-tamps(local logical clock) for that specific message.
4. The initiator of the process gathers the proposed times-tamps, and selects the maximum between them as the final time-stamp for the message.
5. The initiator then multi-casts the elected time stamp to the rest of the servers and moves the specific message from the local un-deliverables to the local execution queue.
6. The rest of the servers receive the maximum time stamp, mark the specific message as deliverable and move it to the execution queue (which is ordered according to times-tamps).

These steps constitute the TOM procedure. This procedure is decoupled from its input and output stages, meaning that a call to the procedure is not blocking for the server. The decoupling is achieved by using queues (and threads) both for the input and the output. When a client message is received by the server, a 'client-receiver' thread takes the message, inserts it into the input queue, and keeps on listening for other incoming messages. Similarly for the output of the TOM procedure, there is a separate thread continuously checking on the output queue for available executable actions to be applied to the battlefield.

Essentially the TOM procedure exists as a software layer between a server's receiving socket and the battlefield. This implementation induces overhead for each message that needs to be processed by the servers but in return it ensures that messages are totally ordered within the whole system. In other words the replicated execution queue will contain the same messages in the same order across all the game servers. The servers are then able to detect if an action is (in)valid locally.

This design choice is based on what players, in this case the end users of the system, would expect from an online multi-player game. Rollbacks and inconsistency are among the top factors that contribute in a player's decision to keep or stop playing a game. Therefore a strongly consistent model would be best suited for this kind of game.

3.5 Fault-tolerance

In order to provide proper fault-tolerance detection of failing nodes is implemented on both client and server-side. The mechanism are explained in the following sections.

3.5.1 Server failures detected by servers

Game server fault tolerance is ensured at the server level by the implementation of TOM. Each server, as described in Section 3.2, has all the one-to-one communication channels with servers that are alive. When a server stops communicating a timeout occurs and all the servers, that have a one-to-one channel open, are notified. Then the other servers remove the dead server from the alive-server-list and stop communicating with it. In the case it becomes alive again the discovery thread (described in section 3.2) adds it to the list of alive servers, restoring the communication. This makes the game servers structure responsive to fault tolerance by adding and evicting server nodes dynamically.

3.5.2 Server failures detected by clients

If the TCP connection over which the game actions are submitted, times out, clients will try to subscribe to a different publish socket. If successful the client will use this new servers TCP endpoint for sending game actions. This process is illustrated using an UML activity diagram, which can be seen in Figure 2.

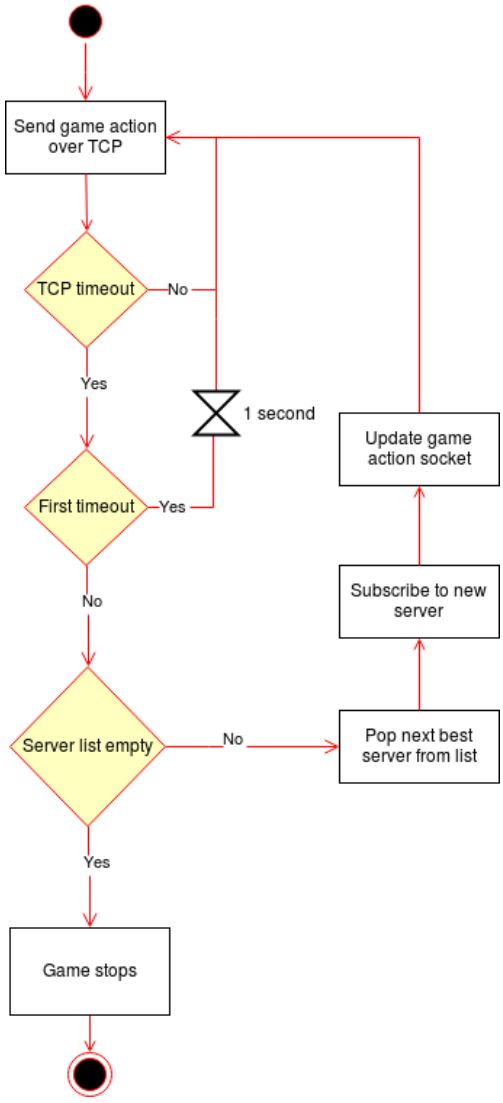

Figure 2: UML activity diagram that shows the client's logic regarding server connections. Clients send their game actions to the server (and receive game updates via the subscriber sockets described in Figure 1). Whenever there is a time out, the fault-tolerance mechanism makes sure clients find a new server if necessary as long as the game is still running.

3.5.3 Client failures

Since there is a publisher/subscriber model used between Server-Client communication, then the server will know when a client gets disconnected, possibly due to a failure. When this happens, the server will be able to locate the Unit that belonged to the client and remove it from the battlefield. Since disconnecting has a similar effect as dying, the process for a dead unit can be reused. A message is propagated using TOM to remove the unit, which frees up space for a new client to connect.

3.5.4 Logging

To further enhance fault tolerance, a logging mechanism is implemented on 2 of the active game servers. Every server is aware of all the game events due to using a synchronized queue (TOM). Every server is also aware if any of the clients or servers have crashed since those events. This can be deduced from failed communication (server does not send ack or server doesn't publish game state to clients, or no response is received within the specified timeout). All important messages, states or communications are then logged into a file, which can be reviewed at a later time to find and solve any issues that might come up.

3.6 Scalability

The scalability of this distributed system is tightly related to the selected consistency model, which is described in Section 3.4. While the sheer amount of players that can possibly be connected to a single game server depends entirely on that server's capacity, the amount of game servers that exist within the system depends on the consistency model. Considering TOM, for each client message received, the system will have to send that message to $n-1$ servers, receive $n-1$ timestamps back, and resend $n-1$ final timestamps back, where n is the number of servers in the setup. This communication pattern leads to $3(n-1)$ messages exchanged within the system, per client message.

Adding a new server to the system, results into higher communication overhead per client message, since all servers need to agree on its place in the ordering. This does not mean that the amount of game servers to be used should be capped, rather that adding more servers introduces additional latency between the time that a player issues a move and the time that the move is executed. A detailed testing method for generating a rough estimate on scalability is included in Section 4.1.4.

3.7 Trace

Two python scripts were written to adjust the GTA-T3 data set of The Game Trace Archive [6] to our needs. This was needed a) because a typical game session for the Dragons Arena is shorter than a WoW session; b) only connect and disconnect events are of interest. The first script adjusts the GTA-T3 data set as such that:

- Only PLAYER_LOGIN and PLAYER_LOGOUT events are stored.
- The earliest event as per epoch timestamp will get the timestamp 0.
- All other events get a timestamp which is relative in seconds to this first event.
- The relative timestamp will be the difference in epoch with the first event, and decreased by a factor x.
- Events are sorted on timestamp.

And the second script adjusts the GTA-T3 data set as such that:

- Only a single PLAYER_LOGIN and PLAYER_LOGOUT event is stored of each unique player.
- Players that do not have **both** a PLAYER_LOGIN event and a PLAYER_LOGIN event are removed.
- For each unique player, the earliest PLAYER_LOGIN event as per epoch timestamp gets the timestamp 0.
- The PLAYER_LOGOUT event gets a timestamp which is relative in seconds to its PLAYER_LOGIN event.
- The relative timestamp will be the difference in epoch with the first event and decreased by a factor x.
- Events are sorted on timestamp.

Because there is no specific dataset for the DAS, the factor x, which is the divisor for game session length, was arbitrarily chosen and set to 10.

4 Experiments

Even though different experiments are in place to test every major mandatory requirement. The lack of manpower and time, means that these experiments can't be carried out, for the reason that the implementation has bugs that cause the system to deadlock. The feature that can be successfully tested is fault tolerance of the system. The game servers were able to detect one of their own crashing and remove them from the group. Therefore the experiments outlined in the following section were partially carried out, meaning that a few of them were completed while others are theoretical.

4.1 Experimental Setup

For the experiments, as outlined in the following sections, to have any statistical significance, experiments would be repeated multiple times, and the average results derived. All of the experiments were run on personal computers (localhost) using different ports for each server. Specifically, the hardware specs of the laptop are the following: Intel Core i7-7500U CPU @ 2.70GHz (4 cores), Intel HD Graphics 620, 4 GBs of RAM memory, and a standard HDD at 7200 rpm. Experiments were run under the Ubuntu 16.04 TLS 64-bit operating system using Java version 8.

4.1.1 Parameter Tuning

For all experiments described below regarding specific requirements, the system is tested with (input) parameters. The parameter values can have a large influence on the performance on the system. Take for example the timeout values that are set after which we consider a server offline. If the set value is too long, processing time is wasted by waiting in case of a

failed server. The overall performance will drop. If the timeout value is set too low, a slow server might be considered offline while messages just take a bit longer too arrive. The result would be that clients are redirected to other servers, while the current server behaves correctly but slow. The redirection of players affects the performance as well. To find the optimal values a grid search can be performed where results are averaged.

4.1.2 Consistency

To effectively test the implementation of TOM, a series of experiments is run of gradually increasing complexity. The first experiment is to connect only a single player to the game, allowing to easily determine whether if the underlying system infrastructure is working and messages are being sent between the servers correctly.

The next experiment tests the consistency of the system. Only a small number of players are allowed to connect and play the game. By logging every single action these player make, from the moment of issuing an action up to the moment that the action is applied to the battlefields on all servers, the consistency of message ordering among the replicated game servers can be determined. Based on the previous experiment, more players can be added iteratively which will further test that the system is consistent.

4.1.3 Fault Tolerance

To be able to effectively test the fault tolerance of the whole system, nodes with different roles are forcefully crashed, on repeated experiments. The list of experiments is the following:

- Run experiments where 1 or more players disconnect from the game.
- Run experiments where 1 server crashes at random.
- Run experiments where 1 server who is also a main logger, crashes at random.
- Run experiments where multiple servers and clients crash

4.1.4 Scalability

The consistency model, does affect how scalable the system is since it acts as a software layer between the game server's receiving module and the battlefield. When more game servers are added to the system, then more time is required before a message is fully processed and applied.

A derived test measurement of scalability is 'mean time to execution'. This term refers to the average time it takes from the moment a client message is received up to the moment it is executed. By keeping track of this time, additional overhead can be determined every time a new server is added to the system, in other words how scalable the system is. Since in realistic scenarios, players also have latency between them and the game servers, increasing the overall time it takes to execute issued commands would result into poor user experience (performance- wise) for the system.

5 Discussion

As was already stated in Section 4, we did not manage to implement the working system as we designed it in Section 3. Implementing TOM in a distributed system turned out to be too difficult in the given time. Despite warnings from the lab assistants regarding the difficulty, we decided to implement TOM as it would be our best system feature. In hindsight, a simpler synchronization mechanism would have been the better choice, since it had probably allowed us to finish the mandatory requirements.

Thinking of more realistic scenarios than the mandatory requirements, total-ordered multicasting seems to be a bottleneck with respect to scalability. As explained in Section 3.6 for n servers we need to send $3(n-1)$ messages among them to ensure the correct order for a single client message. All servers are required to reach agreement on the total ordering of these messages, which introduces more delays in processing a message. This can reach a point where the interaction delay is not acceptable to users anymore.

The complete graph used in the communication between servers becomes too expensive when we increase the number of servers. If we imagine geo-located servers, the RTT for a single message between servers can add up to 100-200 ms due to network delay. If we consider the possible loss of messages we need to wait for a set timeout value before we can decide on the final timestamp of client messages. All this combined with the delays from the TOM procedure lead to large delays which will not be tolerated by the clients. Instead of a total ordering, partial ordering or some majority vote mechanism can reduce the processing time of messages and keep the total delay at an acceptable level.

The current implementation only demonstrates fault-tolerance between communicating servers before the game will get stuck in a deadlocking TOM procedure. A possible cause for the deadlock is the loss of exchanged messages between servers. In its current state the system does not tolerate failures in the TOM procedure. A first step towards resilience would be the use of multiple receiving threads listening for incoming server messages. The publisher/subscriber sockets to push game states back to the clients were issued as future work. With the addition of these missing parts in our system, we would at least be able to start testing a working DAS.

The absence of a working system yields the absence of experimental results. Although we did describe our intentions for the experimental setup in section 4 we could have taken a more iterative approach where we implement, test and analyze smaller fraction of our system. Our chosen approach did not really allow us to test individual parts of the system, since everything depends on a working TOM procedure.

6 Conclusion

In this paper a solution to the DAS assignment has been demonstrated. By using a TOM procedure of submitted game actions between servers, the game state stays synchronized. Together with the proposed fault-tolerance procedure, server crash failures can be masked. Unfortunately its performance could not be tested due to the difficulty of implementing the system.

References

[1] A. Iosup, A. Uta, L. Versluis. LARGE LAB EXERCISE B: The Dragons Arena System: Distributed Simulation of Virtual Worlds, 2017. [Online: `https://canvas.vu.nl/files/277099/download?download_frd=1`; accessed 07-December-2017].

[2] Cronin, E., Filstrup, B., Kurc, A. R., and Jamin, S. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of the 1st workshop on Network and system support for games* (2002), ACM, pp. 67–73.

[3] D. Diomaiuta, V. Gkanasoulis, F.M. van Beest, W. Rens, M. Went. Github repository: ds_group1. [Online: `https://github.com/Dunky13/ds_group1`; accessed 07-December-2017].

[4] Garg, V. K. Elements of distributed computing. 215–216.

[5] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[6] Yong Guo, Alexandru Iosup. The Game Trace Archive, 2012. [Online: `http://dx.doi.org/10.1109/NetGames.2012.6404027`; accessed 08-December-2017].

Appendix A Time sheets

Table 1 shows the time sheets for this project. The initials used correspond to the author names at the top of this report. Figure 3 shows the time spent per component proportional to the total time spent of 386 hours.

	D	V	FM	W	M	Total
think-time	14	16	14	20	16	80
dev-time	42	51	48	10	40	191
xp-time	0	1	0	0	0	1
analysis-time	0	0	0	0	0	0
write-time	5	7	10	10	8	40
wasted-time	12	13	14	20	15	74
total	73	88	86	60	79	386

Table 1: Timesheet

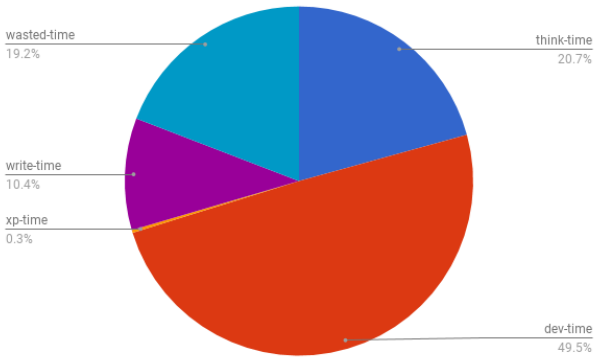

Figure 3: Pie chart showing the relative amount of time spent for each component listed in 1.

Appendix B Repository

Our source code and report are publicly available at Github [3]. A `README` is provided with instructions how to build and run the system.