Distributed System of DAS

An in depth description of the design and experiments for the DAS game

Written by

Bas van der Borden
(s.c.vander.borden@student.vu.nl),
Omesh Debipersad (o.v.h.debipersad@student.vu.nl),
Kian Paimani (k.paimani@student.vu.nl),
Felipe Santos Batista de Souza
(f.santosbatistadesouza@student.vu.nl),
Jonas Theis (j.theis@student.vu.nl)

Vrije Universiteit

Course: Distributed Systems
Course instructors: Dr. ir. Alexandru Iosup, ir. Laurens Versluis
Teaching assistant: dr. Alexandru Uta
December 2017

Abstract

WantGame B.V. hired us to develop a distributed game engine for their DAS game. In the paper our design, implementation and analyses are explained. We showed that the distributed game engine has excellent scalability, fault tolerance and fulfills the basic requirements of the DAS game specified by WantGame. On top of this, we implemented and tested repeatability, load balancing and byzantine fault tolerance.
We analyzed the resilience, load balance, scalability and consistency. The experiments regarding these features proved that the decisions we made regarding client-server relations, the peer to peer server setup and synchronization algorithms work efficiently.
Even though the experiments are conducted on a small scale (as per specifications given by WantGame) we are confident that the engine will perform well on a larger scale.

1 Introduction

WantGame B.V. hired us to build a distributed system for their Dragon Arena game, called the Dragon Arena system (DAS). The reason why WantGame B.V. hired us is that the computational complexity of the distributed game engine is high and they believe distributed systems offer an excellent performance-cost trade-off when supporting many concurrent users. In short, the game is about many players versus a few dragons. The game ends when either all players are killed or all dragons are killed.

Many distributed game engines and systems have already been developed. Games which make use of distributed game engines are Farmville, World of Warcraft, DOTA 2 and The Sims. Some projects which make use of distributed systems for processing of big data are MapReduce, Spark and Windows Azure. [4] and [8] described how they implemented fault tolerant distributed data processing algorithms for Windows Azure and Spark respectively.

In this report we describe how we designed and implemented our distributed game engine and show results of different experiments. The goal of the engine is to be able to run the DAS application in a consistent, fault-tolerant and scalable manner. The system is able to handle many clients on multiple distributed, replicated servers with respect to the in section 2 described criteria.

In section 2 we explain the DAS background and its requirements. Furthermore, section 3 gives a detailed overview of our design and section 4 displays experimental results conducted with our system. In section 5 we discuss the results of the experiments. Finally, section 6 concludes the article.

2 Background on DAS

The DAS application is a game in which there are hundreds of virtual knights (avatars or real-life humans) competing against several computer controlled dragons. The goal of the knights is to kill all the involved dragons. Of course there are certain specifications which are applied.
The game is played on a 25x25 grid, each knight and dragon occupy one grid field and they cannot occupy the same grid field. A knight has 10-20 health points (hp) and 1-10 attack points (ap) and a dragon has 50-100 hp and 5-20 ap. In the battle participants can only move one step horizontally or vertically and players can attack a dragon within a distance of at most 2 grids and vice versa. The hp of the attacked participant will be lowered by the ap of the attacker. Knights can be healed by other knights, that have a distance of at most 5 from each other. A player or dragon with zero or less hp will be removed. Finally, there must be a certain delay between two consecutive actions of the same battle participant.

Considering the specifications of the DAS game engine we should, at the very basis, implement the game requirements described in the previous paragraph. Furthermore, there are some engine specific requirements:

- **Fault tolerance**: In case a client or

server crashes, the game must be resilient against these occurrences. Furthermore, all game and system events must be logged on at least two servers.

- **Scalability**: WantGame B.V. wants us to demonstrate the properties of the application when there are 100 players, 20 dragons and 5 servers. The game should end when there is only one class of remaining participants remaining.
- **Consistency**: Make sure that all nodes see all the data and process it in a given order.
- **Performance**: The application should be able to run smoothly, given the fault tolerance, scalability and consistency requirements.

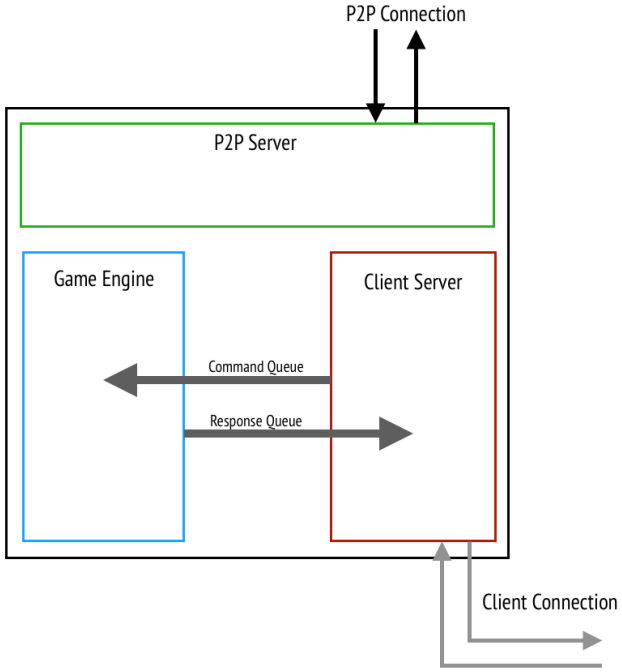

Figure 1: Overview of the system architecture

3 System Design

We started our design with a basic client-server architecture. Henceforth, we iteratively enhanced this inital design to meet more of the requirements. In the basic client-server model, each player is a client with in total one authoritative server. The benefit of an authoritative server is that it prevents cheating, which is essential in a multi-player game. To take the system to the next level, we replicated our servers to have *multiple mirrored servers*. In the new distributed version, the system consists of multiple servers and each client connects and communicates with the closest server (lowest latency). Indeed, the servers broadcast all game commands to each other and all clients will execute a loosely consistent game state. Next, the system was further enhanced to support *strong consistency* and *fault tolerance*. These traits are further discussed in the next sections. Due to our clear and concise inital design and incremental enhancement, we were able to add such expensive features to our system with minimum effort.

The overview of our design for **each server** is as depicted in Fig 1. The Server module is divided into three main components, each performing isolated tasks. The server component consists of a *Game Engine*, a *Server for clients* and a *Peer to Peer server* that other servers can use to establish a connection to. The Game engine executes all game commands in an isolated process. It will receive commands from the server using an IPC[1] queue and will put their responses respectively in the *Response Queue*. The *Client server* will accept connections from clients and receive their game commands. After being executed by the server, the response status of commands are sent back to the client. To be more specific, the executed commands are broadcasted to all clients, so that they can update their game state. The *Peer to Peer* component will accept connections from other servers or connect to them. It will broadcast all of the commands received by the *Client server* to other peers and put the commands received from other peers in the *Request Queue*.

3.1 Scalability, Reliability & Availability

The server architecture of our design enables us to deliver multiple valuable traits such as scalability, reliability and availability. The servers

[1]Inter Process Communication

connect to each other in a peer-to-peer manner. Consequently, there are no *master nodes* in the peer-to-peer system. In other words, the architecture of the servers is flat and therefore there are no server bottlenecks. The only exception is that at least one of the servers must have the initial map of the dragons in the field. Henceforth, any of the servers can leave the cluster and this will not have any significant effect on the other servers. Furthermore, we can dynamically increase the number of servers to be able to handle more clients. These newly added servers can be turned off after peak workload is passed to save resource consumption. Furthermore, the servers will all maintain a list of peer connections and will iteratively send heartbeat messages to them to ensure they are still available. These heartbeat messages are also used to exchange meta-data which is used in other aspect of the system. A peer is deemed *kicked* if it fails to respond to a configurable number of heartbeat messages. The kicked server can later re-join the cluster by sending an *INIT* message to *any* of the active peers. This will allow the kicked server to fetch an updated game map and continue to serve clients.

Note that we suggest Docker to deploy server nodes and the Docker Engine can be configured to remotely restart the server's container if it crashes.

From the client's point of view, the game server cluster can be seen as one unified, reliable component. In other words, if a client is disconnected from its own authoritative server, it will automatically re-connect to another server from the peer-to-peer cluster. Furthermore, if a server crashes, the clients associated with that server will immediately identify the failure and invoke the same re-connection process again. This theoretically ensures that in a cluster with multiple servers and hundreds of clients, even if all game servers crash except for one, the clients can continue.

3.2 Load Balancing

The *Client Server* component also provides a simple UDP socket so that clients can ping all our servers and then connect to the server who responded the fastest. In a simple approach, this usually results connecting to the geographically closest server which does not provide a good load balancing among the servers if there are many users connecting from one region. Therefore the servers exchange meta-data about how many clients are currently connected to them with the previously mentioned heartbeats. Based on this data, the average amount of users a server should host is calculated and the ping response time is artificially delayed. Thus, clients receive ping responses of less busy servers first and connect to them. Furthermore, in a certain interval servers evaluate their currently connected clients to the total amount of clients and kick some clients if there are too many. These clients will reconnect to another server, according to the previous described process.

3.3 Synchronization

To synchronize servers we use a timebased approach with the Network Time Protocol (NTP). NTP intends to synchronize all servers to within a few milliseconds of Coordinated Universal Time (UTC). It uses the intersection algorithm, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions [6].

This synchronized time is combined with the concept of game-state tics to reach full consistency among servers. Game tics are certain iterative points in time in which an accumulated number of commands from clients is executed. In other words, all servers will receive different commands from their clients and other servers for an amount of time but *will not execute them immediately.* Instead,

in a synchronized interval, all of them will execute the commands at once and broadcast the responses to all clients connected to them. Two further details elaborate how this approach can be a sophisticated consistency mechanism. First, at the beginning of each execution tic, all servers will sort commands based on their timestamp. This ensures that all servers will execute the commands in order. Secondly, to ensure that commands received by a server actually belong to that specific tic, we prevent servers from executing commands which were received very late within each tic interval. Instead these commands are executed in the next tic. In other words, this ensures that no server executes a command before it has been broadcasted to the other servers. Thus, it ensures consistency. The threshold of postponing messages to the next interval can be configured based on cluster network delay in which the servers are deployed.

3.4 Simulation using a GTA File as input

As part of the project implementation a mechanism to read a GTA (Game Track Activity) file was created, the idea was to use it to analyze the system behaviour in different situations based on real data.

The GTA file used as input (GTA-T3)[2] contains approximately 1,5 Million events of the game World of Warcraft, there it is possible to identify which action a given player took at a specific timestamp. The events of interest were LOGIN, LOGOUT and QUITING.

Given that a simulation has a specific execution timeframe, this parameter was taken into consideration to normalize the timestamps collected from the file according to the formula:

$$\frac{origEventTimestamp \times simTotalTime}{origElapsedTimeFromGTAFile}$$

With each login/logout event having an associated normalized timestamp it was possible to trigger events according to the defined simulation time frame. This was done by creating one thread per event, this thread would then at that specific time execute (or kill) a client process (this concept was further extended to simulate servers joining/leaving the simulation). For development and test purposes a reduced version of the GTA file was created so it would be easier to track the behaviour of the system in different scenarios. Although the simulation was useful to understand the concept behind the GTA file, we understood to be more efficient to conduct most of the experiments by triggering the clients and servers via command line according to the scenario which we wanted to test/validate.

[2] obtained at gta.st.ewi.tudelft.nl/datasets/GTA-T3

3.5 Additional System Features

In this section we describe three additional system features and how they have been implemented:

- **Byzantine fault-tolerance**: Our system is able to cope with arbitrary failures. For example, if a dragon dies while it has plenty of hp or a client moves more than one step (maliciously sent information), the system will be able to cope with this. This is achieved by basically whitelisting valid movements within the game on the authoritative server.
- **Repeatability**: In situations where two events happen at the exact same time, we make sure that the outcome of the simulation is always the same for the same input by making use of priority numbers as described in [3] (p.85-86).
- **Load balancing**: This feature is extensively discussed in section 3.2.

3.6 Implementation details

We chose *Python* and its basic sockets, threading and multiprocessing libraries to develop the game and its distributed system. This decision was made because we wanted to have full

control and knowledge about the systems fundamentals rather than relying on a high-level library.

The communication between clients and peers has been implemented with TCP. With providing a message length in the first 4 transferred bytes in Big Endian order we created our own message protocol on top of the stream based TCP. We assume that TCP is a reliable protocol, because it has been used for decades and guarantees delivery of data in order [2].

The source code and instructions how to install and run our system with help of our console tool can be found online under Apache-2.0 license[3].

4 Experimental Results

We have conducted multiple experiments that depict the various aspects of the system, most of which being the requirements of the system as a scalable distributed system. The experiments can be categorized as follows:

- **Reliability and Availability**: The outcome of the system in case of server and/or client crash.
- **Load Balancing**: How the clients are evenly distributed in different scenarios.
- **Scalability and Elasticity**: How can the system cope with an increasing number of clients?
- **Full Consistency**: A demonstration of the game state and history is exactly the same on each mirrored server at any point in time.

Technical details of these experiments and how to reproduce them is omitted in this document and can be viewed in the source code repository of the system [4].

4.1 Reliability and Availability

Clients are designed to be fully functional without any dependency on a specific server. This means that at the initial startup, clients have full autonomy to choose one of the servers to connect to. Furthermore, if a server crashes, all clients connected to that node will automatically choose a new server to connect to. Additionally, servers can have the exact same behavior. Our experiments clearly show that in scenarios where one of the servers is intentionally killed, neither any of the other servers nor the clients will be affected. Furthermore, similar to clients that can re-join if they lose their connections, servers can be spawned again and join the cluster again.

4.2 Load Balancing

Load balancing of the clients in between the servers is a unique and ubiquitous trait of our DAS implementation because it can be seen in almost any experiment. Keeping the mechanism of our client distribution in mind, the simplest case that we can come up with to demonstrate load balancing is just two servers and two clients. In this scenario, the first client will be connected to an arbitrary servers. Due to the dynamic initial handshake delay injected into connection by each server, the second client will almost certainly be connected to the second server with no client[5]. Furthermore, the experiment can be grown to the scale of the full game. As an example, if the game is deployed using the main configuration of 5 serves and 100 clients, in all experiments our dynamic distribution ensured that each server will get around 20 clients. Fig 2 demonstrates the number of clients connected to an arbitrary server with the horizontal axis showing the total number of servers. It can be clearly seen that as the number of servers increases, the number of clients connected to the initial server decreases on the fly. Note that in our ex-

[3] Source code under Apache-2.0 license: https://github.com/jonastheis/das

[4] All experiments can be seen here: https://github.com/jonastheis/das/wiki/Experiments

[5] Note that this does not mean that geographical distribution is irrelevant. The geographical delay will still be significant. This will force it to be fair with regards of server load too

periments, we inserted a small delay (less than a second) between clients joining. This is a realistic assumption because in most real cases clients will gradually join a field game.

4.3 Scalability and Elasticity

An interesting appendix to the same load balancing mechanism, based on the number of clients, brings us to the scalability properties of the system. With our design, servers will iteratively communicate over heartbeat messages and exchange their number of clients and try to evenly distribute their clients. As explained in the first experiment, since clients can seamlessly migrate to other servers, this operation is relatively cheap and insignificant for the client. As an example, when a system is operating with 2 servers and 50 clients, if a third server joins the peer to peer network, almost immediately, a third of the client will connect to this new peer to reduce the load of the other peers. As explained in the previous sections, this operation can also happen in the reverse order. If a server leaves the server cluster, its clients will never crash and will immediately connect to other candidate nodes based on the number of clients that each node has.

The overall conclusion of the first three experiments is that *within a cluster of low latency servers*, our implementation can have a near perfect distribution of clients. In other words, we can guarantee that the difference of the number of clients in each server will not exceed a certain threshold.

4.4 Consistency

Each DAS server uses two logging systems, one for *system* logs and another one for *game* logs. Furthermore, loggers have different levels that can be configured at runtime. To avoid unnecessary complexity, we have set both log levels to **INFO** which is relatively high and outputs only the important information. For proving the consistency of the system, we deployed a full-scale game with 5 servers and 100 clients (note that clients will be automatically distributed evenly among different servers). After all dragons are dead, all clients are removed from the field at once. Consequently, all servers are stopped at the same time. At this point, the game log of each server will contain a sequence of statements that contain information about the parameters of each executed command and their respective result (e.g. was the execution successful or not). All log statements are stored in order and they contain the timestamp of the command. Afterwards, we use a custom *diff* script to check the difference of logs in different servers. Our tests for tens of experiments with the same configuration, explained as above, clearly show that there is no inconsistency between the logs in any of the servers. To further emphasize this, we should also mention that the average size of each game log file for each server is about 10MB of raw text data. This clearly justifies that for a network with predictable delay, our elegant, yet simple consistency model is completely functional and efficient.

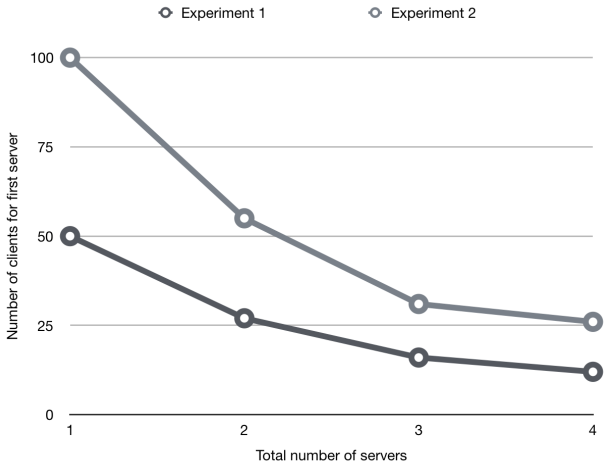

Figure 2: Load balancing experiment

5 Discussion

We have designed and implemented a distributed game engine which fulfills all of WantGame B.V.'s requirements. By implementing our design we managed to build a game engine which executes all of the basic requirements of DAS, is fault tolerant and scal-

able.

The results from our experiments, as mentioned in the previous section, are very positive.
The results for reliability and availability showed that if a server crashes, the client will automatically reconnect to another server. Our experiment showed that when a server is intentionally killed, the clients nor the other server will be affected and once the killed server is restarted it can re-join effortlessly.
The load balancing experiment showed that the clients are evenly distributed amongst the servers. In case of the test with 100 clients and 5 servers, each server had almost 20 clients. This is due the dynamic initial handshake delay injected into connection by each server.
When looking at the combination of resilience, load balancing and scalability, our experiments show that the distribution of clients is approximately equal to $\frac{clients}{servers}$.
Finally, the last experiment we conducted is about consistency. The game was tested with 100 players, 20 dragons and 5 servers and goes on until either the dragons or players are dead. Once either group of participants is game over the game will be cleared. The created game logs for the servers were tested with a difference function and showed that their are no inconsistencies between the logs of any server.

Now the question is: *should WantGame implement our distributed game engine to run the DAS?* We think that given our engine it should be possible. We showed that our system is reliable and load balances near perfectly. Furthermore, we also showed that our system scales extremely well. This will be the case for 100 players or 100000 players. However, there are some aspects which have to be further explored.
The first point we would advise to explore is the consistency on a much larger scale. Now the game is tested with 100 players, 20 dragons and 5 servers. In real life scenarios we expect that there will be tens or even hundreds of thousands players playing DAS which are geographically spread.
Coming to the notion of geography, the experiments were conducted on low latency servers. In real life we expect that servers will have a relative large geographical distance from each other. The current limitation in our synchronization mechanism, described in section 3, is that if the latency is larger than a tic interval the servers might not be synchronized anymore. This could easily be fixed by implementing TSS [1] or lamport timestamps [5]. When deploying DAS on our game engine these notions will have to be further researched.

6 Conclusion

We implemented the design for the DAS game and have fulfilled all of WantGame B.V.'s requirements. Not only did we meet the requirements of WantGame, we also implemented extra features to make a stronger distributed game engine. The extra features we implemented are byzantine fault-tolerance, repeatability and load balance. We are delighted with the result and are confident that WantGame B.V. will be delighted as well.

7 Appendix

Table 1: Time spent on building DAS

Omesh	**Subject**	**Time spent (hours)**
	Design	12
	Development	0
	Experimenting	8
	Analysis	8
	Writing report	30
	Wasted time	10
	total time spent	68
Kian	**Subject**	**Time spent (hours)**
	Design	15
	Development	60
	Experimenting	10
	Analysis	5
	Writing report	5
	Wasted time	25
	total time spent	120
Jonas	**Subject**	**Time spent (hours)**
	Design	25
	Development	70
	Experimenting	10
	Analysis	10
	Writing report	10
	Wasted time	15
	total time spent	140
Felipe	**Subject**	**Time spent (hours)**
	Design	8
	Development	15
	Experimenting	5
	Analysis	5
	Writing report	1
	Wasted time	2
	total time spent	36
Bas	**Subject**	**Time spent (hours)**
	Design	
	Development	
	Experimenting	
	Analysis	
	Writing report	
	Wasted time	
	total time spent	

8 Bibliography

References

[1] E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):7–30, 2004.

[2] K. R. Fall and W. R. Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[3] R. M. Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. 2012.

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

[6] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[7] S. D. Webb, S. Soh, and W. Lau. Enhanced mirrored servers for network games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 117–122. ACM, 2007.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.