The *Dragon Arena System*[*]

A mirrored server online gaming architecture

Christopher Esterhuyse, Axel Koolhaas, Zakarias Nordfäldt-Laws,
Roy Overbeek, Dalia Papuc, Rink Stiekema
Faculty of Science, Vrije Universiteit Amsterdam
{c.a.esterhuyse, a.e.koolhaas, z.o.nordfaldtlaws
r.overbeek, d.papuc, r.stiekema}@student.vu.nl

Abstract

This report details the requirements, design and evaluation of the *Dragon Arena System* (*DAS*) game engine. The DAS can be used to support games which have many concurrent players participating in the same game session. System properties that are considered particularly important for the DAS are consistency, scalability and fault-tolerance. The DAS is implemented as a mirrored server architecture to meet these requirements. System evaluation was performed with simulation experiments, in which real-life game trace data from the Game Trace Archive was utilized. Results show that the DAS is highly fault-tolerant and consistent. While the system scales flexibly by design, results indicate that adding servers does not yield performance gains.

1 Introduction

As demonstrated by the popularity of multiplayer games such as *World of Warcraft* and PLAYERUNKNOWN'S BATTLEGROUNDS, there is considerable market interest in games that allow a large number of users to participate in a single game session [1–3]. Relative to small-scale online multiplayer games, this type of game puts new demands on online gaming architectures. Apart from having to deal with increased network traffic, it can e.g. prove challenging to efficiently synchronize all players.

There exists a wide variation in architectures that address these challenges. *Second Life*, for example, uses multiple servers with distinct dedicated purposes to spread the system workload [4]. By contrast, *Warframe* is an online action game that relies on p2p technology [5]. An approach that combines elements from both of these examples is the mirrored server architecture [6]. Like the *Second Life* model, in this model multiple servers are used to distribute the client-handling workload. Unlike the *Second Life* model, however, these servers have near-identical functionality, and they intensively communicate with each other in a decentralized fashion. Because this server-server submodel resembles the p2p model, it also shares some of its nice properties, such as the absence of a single point of failure.

The present report describes the requirements, design and evaluation of a mirrored server architecture that implements a game engine called the *Dragon Arena System* (*DAS*). The DAS will be used by the (fictional) Dutch company WantGame BV to run a large-scale online warfare game, in which players control knights and fight dragons on a square grid battlefield. Requirements related to consistency, scalability, fault-tolerance were prioritized, in part through the application of replication methods. These properties are therefore emphasized throughout this report. The eventual evaluation of the system implementation was performed using simulation experiments, using real game trace data from the Game Trace Archive [7].

The remainder of the report is structured as fol-

[*]This report was submitted for the lab assignment of the 2017-2018 edition of the Vrije Universiteit Distributed Systems master course, under supervision of prof. dr. ir. Alexandru Iosup (a.iosup@vu.nl), dr. Alexandru Uță (a.uta@vu.nl) and ir. Laurens Verluis (l.f.d.versluis@vu.nl).

lows. In Section 2, more detailed background is provided on both the DAS and its requirements. In Section 3, we present an overview of our high-level system design. Section 4 briefly describes the system implementation and the experimental setup, and summarizes the experimental results. These results are interpreted and discussed in Section 5. Finally, we make some suggestions for future work in Section 6.

2 Background

The DAS must support large-scale game sessions, in which player-controlled knights fight computer-controlled dragons on a square grid battlefield. The repertoire of actions is limited. A player-controlled knight can move to an adjacent empty square, hit a dragon, or heal a player, provided that its target is in range. Computer-controlled dragons can only hit nearby players. A session starts when a player connects, and ends when either only knights or only dragons are alive. While a session is in progress, players can connect and disconnect at any time. Since the game progresses in real-time (rather than in a turn-based fashion), the game is moderately time-sensitive. For administrative purposes it is finally important that all game and system events are logged.

Since players interact in a shared, fully-inspectable virtual world, it is important that their view of the world is highly consistent. Moreover, consensus must be reached on what the current game state is in case inconsistency is detected. When a single client-independent server is involved, the resolution is simple: the server at all times defines the game state. If multiple servers are involved, however, server-server consistency must be assured as well.

With regards to scalability, the initial target for the DAS is that it can support at least 100 players fighting 20 dragons on a 25×25 size battlefield, hosted on 5 servers. The system design should ideally account for the fact that the number of players are expected to increase in the future. Moreover, the DAS should be capable of handling players and servers connecting and disconnecting at any time throughout a session.

The system must be fault-tolerant in the event of client or server failure. When either a client or a server crashes, the game should continue to run. Moreover, server crashes should be detected by the other servers, which then take over its workload. This server crash recovery process should be transparent to the client. In addition, a complete log should be constructible post-game even when server crashes occurred.

All these requirements must be met without causing severe performance degradation, since games that perform badly affect player experience [8]. What level of performance is acceptable is highly game-dependent, however, and must ultimately be assessed empirically [6].

3 System Design

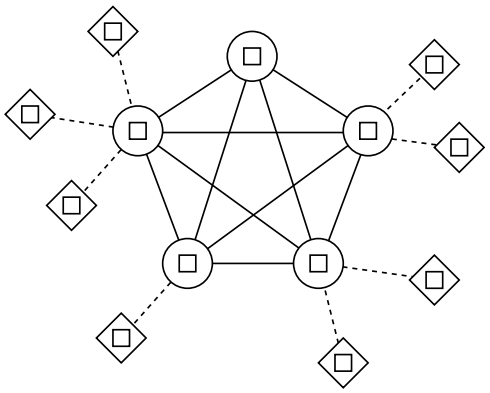

Figure 1: The DAS architecture. In this example configuration, server nodes are represented by a circle, client nodes by a diamond and game objects by a square.

3.1 Architecture

The DAS is implemented as a mirrored server architecture. Servers are arranged in a completely connected, symmetrical flat group, where each server handles its own set of clients, and each client is handled by at most one server (Figure 1). All of these clients participate in the same game session. While a client can at any time submit an action request to its server, it relies completely on the server to update its local game state. Servers process batches of requests in lockstep at set intervals, and then flood updated game states to their respective clients. To simplify our communication protocols, we rely on

consistent use of TCP for its guarantees of in-order delivery and retransmission protocol.

3.2 System operation

We first describe how the system behaves in a stable configuration: this is the *standard protocol*. Then we describe how client and server joins and crashes are handled relative to this protocol.

3.2.1 The standard protocol

At a start of a tick (lockstep round), a server fixes its batch of client requests that has to be processed for that tick. The server floods the batch of requests to every other server, followed by a **DONE** message. It then waits at a barrier until it has received a **DONE** message from every other server. When this happens, it has received all of the requests that must be processed in that tick. The server orders the requests in a way that is guaranteed to be consistent among servers, and then applies these ordered requests to a local copy of the game state. Finally, it sends a copy of the updated game state to its clients, and sleeps until the next tick.

3.2.2 Server join

A joining server S_1 sends its request for integration to the active server with the lowest server *id* S_2. S_2 acts as in the standard protocol, except that it withholds its **DONE** message. This ensures that the other servers wait at the barrier. Upon receiving **DONE** messages from all the other servers, S_2 computes the new game state and shares it with S_1. S_1 then establishes connections with all other active servers, and sends an acknowledgement to S_2. When S_2 receives the acknowledgement (or a time-out occurs), it releases servers from the barrier by sending out its pending **DONE** message. Only one server join is handled per tick, in order to prevent deadlock.

3.2.3 Server crash

For simplicity, we assume that communication links do not fail. Thus a crash of server S is detected by another node A when A loses connection with S. If A is a client, A will automatically begin to migrate to another server. If A is a server, A discards any requests of S that were not yet succeeded by a **DONE** message. To handle the unfortunate case in which not all the servers have received the **DONE** message from S, servers compare hashes of their game state copies every n ticks. In the case of a discrepancy, servers declare the game state with the highest hash to be the new game state. This defensive consistency mechanism also protects against failures that may have escaped analysis. A server that rejoins after it has crashed follows the same mechanism as a new server joining, maintaining its previous logs.

3.2.4 Client join

A new client C that wishes to join pings the active servers and orders them by ascending latency. C asks the first server in this ordering to be welcomed. The server may refuse if its workload is high relative to other servers: in this case, C retries with the next server in the ordering. Suppose server S accepts C. S adds C's spawn request to its request buffer. When the spawn request is processed in the standard protocol, it sends the resulting game state to C, along with a fresh (and provably non-colliding) token identifier. S also sends a hash that was generated using a secret supplied by C and C's address. In case S crashes, C can later send this hash and its token identifier to another server in order to authenticate itself and reclaim its token.

3.2.5 Client crash

If a server loses connection with a client, it is assumed to have crashed. The server adds a request to its request buffer for removing the client's token. A client that rejoins after it has crashed is treated like a new client.

3.3 Properties

3.3.1 Consistency

Client-server consistency is easily guaranteed, since clients do not tentatively update their local copies of the game state, and TCP ensures not only that updates from the server arrive, but also that they arrive in chronological order.

Reliable server-server consistency is achieved through the blocking nature of the standard lock-

step protocol. We note that any random effects are also assured to be consistent among servers, since at all times seeds are locally derived from game state properties shared by all servers. Thus, consistency is preserved under operations that involve randomness.

Finally, should an inconsistency occur due to an aberration, then the defensive consistency mechanism will detect and correct it within a bounded number of ticks.

3.3.2 Scalability

The design imparts no *a priori* constraints on the number of clients and servers. Given a large enough grid, joins can in principle always be handled. The identifiers will moreover never collide.

Of course, there are practical limitations. A single server can be overburdened by clients. This scenario is avoided as much as possible through both static and dynamic load-balancing of clients per server. Static load balancing was already described in the subsection detailing how clients joins are resolved. For dynamic load-balancing, proportionally busy servers will periodically let their clients migrate to other servers.

Also, we note that scaling up servers does not necessarily increase performance, since the message complexity will increase, and the slowest server will always delay the other servers in the lockstep.

3.3.3 (Advanced) fault-tolerance

The system is highly fault-tolerant. Because the servers form a flat group and clients have the ability to (transparently) migrate to another server, the system will continue to function properly if at least one server stays alive. Server crashes do not affect the quality of logs either: all servers log the game and system events they observe, and at the end of a session a unified script can be created. All communications are finally highly reliable due to the handshake protocol inherent in TCP. Failures are further hidden by using time redundancy achieved by TCP retransmission.

3.3.4 Security

Some security features have been added as a proof of concept. A server that wishes to join the lockstep has to provide a secret for authentication. In the event of a server crash, clients can only reclaim their token if they authenticate themselves (see Subsection 3.2.4). However, we note that the client secret hash does not prevent a man-in-the-middle or replay attack.

Table 1: Average server work times per tick (ms).

	experiment		
server id	1 server	3 servers	5 servers
0	55.80	48.19	63.32
1	–	44.93	46.80
2	–	46.98	48.45
3	–	–	45.17
4	–	–	50.55

3.3.5 Repeatability

As described in the Subsection 3.3.1, the outcome of a random operation is always defined by game properties. Thus, if care is taken that requests arrive during the same ticks, a game is completely repeatable.

4 Experimental results

4.1 Experimental setup

The system implementation was programmed in Python 2.7. Apart from `msgpack` for serialized data transmission, only standard Python libraries were used in the implementation [9]. The experiments were run on the *Distributed ASCI Supercomputer 4* (*DAS-4*) [10]. In case multiple servers were used, each server was run on its own compute node. Clients always shared a compute node.

Client activity was completely simulated by bots in the experiments. Each bot submitted requests according to a simple strategy loop, in which heals were approximately prioritized over attacks, and attacks prioritized over moves. Global connection and disconnection events were simulated using the `GTA-T11` (*WoT*) dataset from the Game Trace Archive [11]. This dataset was chosen as it provides all the necessary information (client connection time and lifetime), and its overall client lifetime is shorter compared to other similar traces such as `GTA-T10` (*SC2*). The data was rescaled to shorten the experiments.

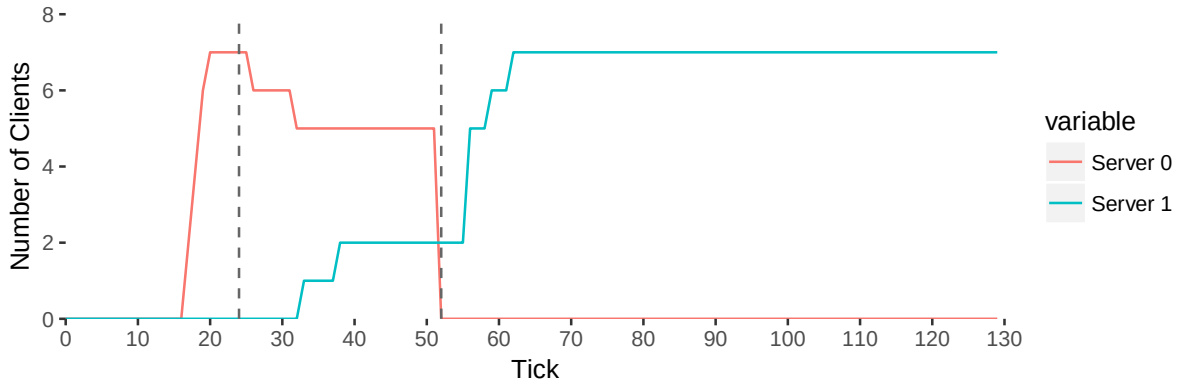

Figure 2: Server 1 dynamically takes over clients from server 0 while server 0 is still alive. The first dashed line indicates server 1 finding server 0. When server 0 crashes (dashed line 2), its clients migrate to server 1.

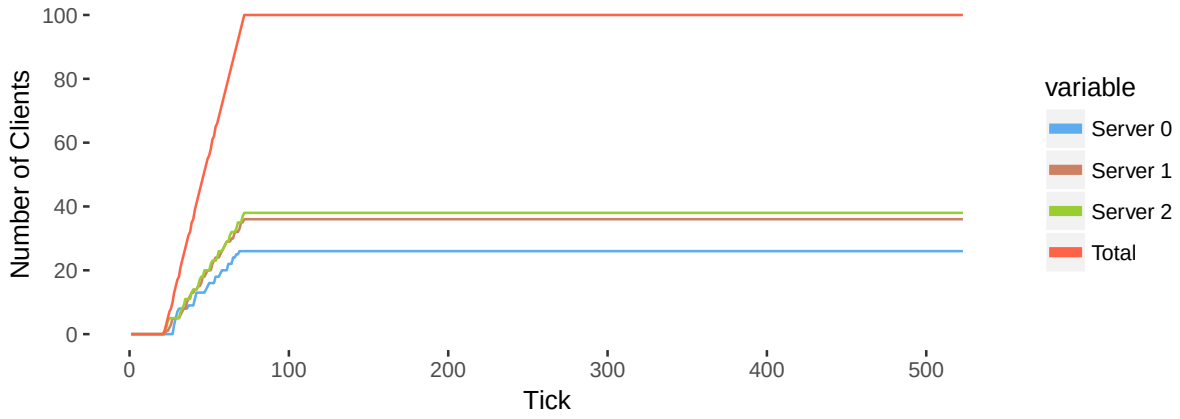

Figure 3: Clients joining in the experiment involving 100 clients and 3 servers.

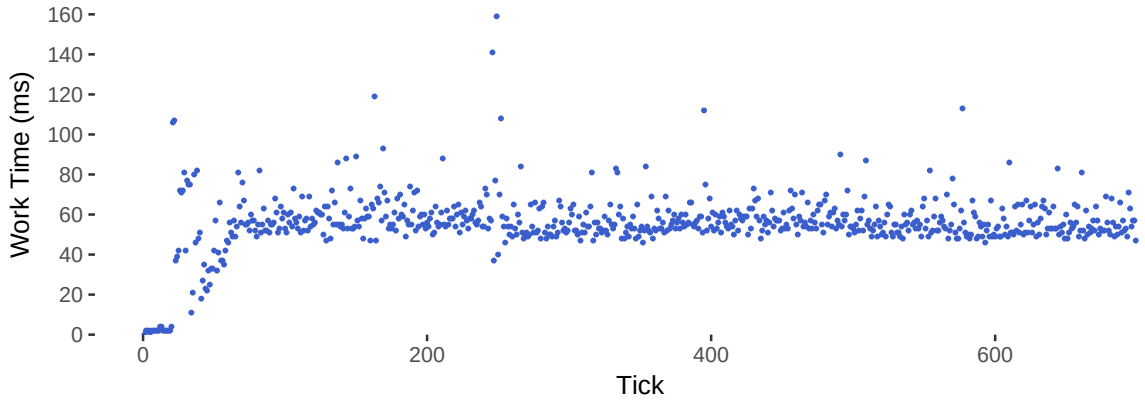

Figure 4: The work times for 1 server handling 100 clients.

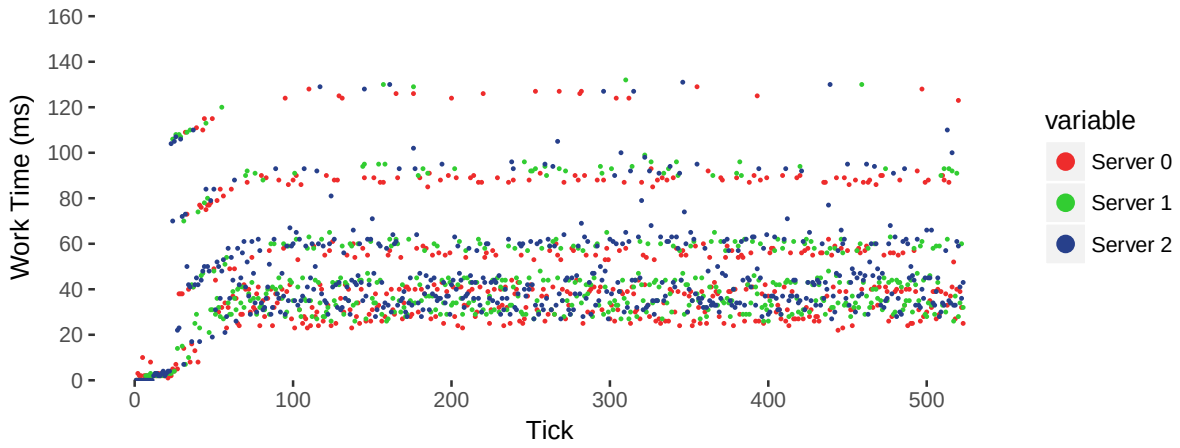

Figure 5: The work times for 3 servers handling 100 clients.

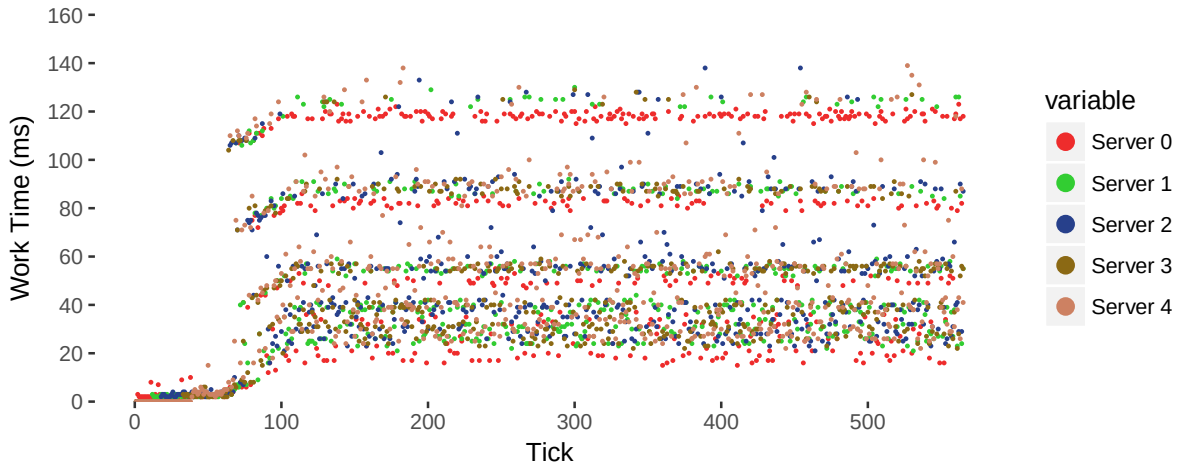

Figure 6: The work times for 5 servers handling 100 clients.

The tick period for the performed experiments was set to 0.2 seconds, with an average of 1 client request sent per tick.

4.2 Experiments

4.2.1 Fault-tolerance

To check whether the system tolerates the crash of all but one server, we manually crashed servers and analyzed their logs. Multiple set ups were run and provided us with consistent results. Figure 2 shows one of the runs with 2 servers and 7 clients, in which the servers were started and crashed at different moments.

The figure shows that the clients connected to the crashed server (server 0) reconnect to the available server (server 1) allowing the game to transparently continue for all clients. Crashing clients are not of interest for this experiment, as the game continues in case of client failures.

4.2.2 Scalability

We tested the system's scalability using three experiments. In these experiments, the number of clients was kept constant at 100. The number of servers were fixed at 1, 3 and 5, respectively.

An example of static load balancing is shown in Figure 3. When all the three servers are started at the beginning of the game session, the client joins occur according to the described procedure in Section 3.2.4. The dynamic load balancing of DAS is shown in Figure 2: here we see that server 1 takes over clients from server 0 while it is joining, even though server 0 did not crash.

Let the work time of a server in a tick be defined as the time not spent waiting at the barrier or sleeping. The average work time of each server is exhibited in Table 1. Figures 4–6 detail the work time for each of the run experiments with respect to the tick number. They consistently show a linear increase in the work time as related to the occurrence of client join events.[1] The layers in the work time graphs can be partially explained by periodic events, such as hash comparisons.

[1] All clients joined linearly before tick 100, as illustrated in Figure 4 illustration.

4.2.3 Consistency

To check the consistency property, the logs of all the aforementioned experiments were checked with respect to the defensive consistency mechanism. In neither of the experiments this mechanism was triggered. Moreover, we did not detect any obvious gaps in the logs.

5 Discussion

Does the current design and implementation meet the desired properties of consistency, fault-tolerance and scalability? How well is the system expected to scale beyond the workloads that have been tested in the experiments? And is it ultimately worth implementing the DAS as a distributed system? These questions we discuss in order.

Strong consistency is certainly provided. This is attested to by the fact that the defensive consistency mechanism (described in Section 3) has never fired during any final experiments, and the fact that servers ended with the same game state. We are moreover confident that the defensive consistency protocol is actually functioning as intended. During one development stage it would fire frequently, and this exposed a bug in the function that computes the order in which updates ought to be applied. After the bugfix, the defensive consistency mechanism never fired again.

Strong fault-tolerance is also provided. Figure 2 shows that clients of a crashing server can quickly migrate to another server. We have also not discovered any omissions in the generated logs, although this is of course difficult to check exhaustively. It is true that the DAS is not tolerant of client crashes: players have to rejoin the game as a new player by design. This is intended, as it ensures that clients gain nothing by crashing intentionally. However, an alternative design decision is to add a grace period for crashing clients.

Is the system scalable? It depends. On the one hand, any number of clients and any number of servers can in principle be supported by the DAS. Moreover, the system uses both static and dynamic load balancing (Figures 2–3), to ensure that servers are about equally burdened. On the other hand, adding servers does not necessarily result in a per-

formance gain, as is evident from Table 1. In fact, the average work times seem to be of roughly the same order in each of the three experiments. This can be explained as follows. Suppose n clients are uniformly distributed among m servers. While a server S handles only n/m clients, S still has to forward requests to $m-1$ servers and receive forwarded requests from $m-1$ servers. With the current design, the number of requests S has to process is moreover completely independent of m: S will ultimately process every client request. On top of that, S might have to wait for slow, crashing or joining servers. It also has the obligation to periodically communicate with other servers to verify consistency, and to check whether workloads are still fairly distributed.

For supporting a much larger number of users, then, a redesign would probably be needed once performance starts to negatively affect player experience.[2] Since the world would also grow, an obvious solution would be to partition the world into connected zones. These zones could be processed semi-independently from each other, reducing both server load and communication between servers. The DAS architecture could then be used for each individual zone, to ensure fault-tolerance and consistency at the zone level. In addition, communication could possibly be made more lightweight.

Extrapolation from the experiments is furthermore difficult for two reasons. First, clients were all run on the same machine, while in a real-life scenario they would be running on different machines. Secondly, clients and servers were extremely close geographically. How geo-scalable would the DAS be? Not very, we expect, since the slowest server will slow down all other servers.

To conclude, it makes sense to implement the DAS as a distributed system, because it provides high fault-tolerance without sacrificing consistency. However, if the player base were to grow, the architecture would ultimately need to be modified. In this modification, subcomponents could still use design elements from the current DAS implementation.

[2] In the experiments so far, game states were typically updated in about 50 ms. Thus no more than 20 frames per second could be generated on the client-side. For many games this figure would already be insufficient. Whether it is sufficient for a simple two-dimensional game like this would have to be assessed empirically.

6 Conclusion

In this report a mirrored server architecture was described for the DAS game engine. The most emphasized system properties included consistency, scalability and fault-tolerance. Experiments confirmed that the design is highly consistent and fault-tolerant. In addition, the system was shown to scale flexibly. However, experiments also showed that performance cannot be improved by scaling up. Whether the current level of performance is actually sufficient depends on currently unknown variables, such as the eventual geographical distribution of the system, the projected number users, and the relation between game performance and player experience.

References

[1] Number of world of warcraft (wow) subscribers from 1st quarter 2005 to 3rd quarter 2015 (in millions), 2015. `https://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/`, last accessed 15 December 2017.

[2] Steam & game stats, 2017. `http://store.steampowered.com/stats/`,last accessed 15 December 2017.

[3] F. Brown. Playerunknown's battlegrounds beats dota 2's highest concurrent player record, 2017. `http://www.pcgamer.com/playerunknowns-battlegrounds-beats-dota-2s-highest-concurrent-player-record/`, last accessed 15 December 2017.

[4] Server architecture, 2011. `http://wiki.secondlife.com/wiki/Server_architecture`, last accessed 15 December 2017.

[5] Devstream 12, 2013. `http://warframe.wikia.com/wiki/Devstream_12`, last accessed 15 December 2017.

[6] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Proceedings of the 1st workshop on Network and*

system support for games - NETGAMES 02, 2002.

[7] G. Yong and A. Iosup. The game trace archive. *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, 2012.

[8] K. T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games. *Multimedia Systems*, 13(1):3–17, 2007.

[9] G. van Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995. `http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Aercim_cwi%3Aercim.cwi%2F%2FCS-R9525`.

[10] Das-4: Distributed asci supercomputer 4, 2012. `http://www.cs.vu.nl/das4/`, last accessed 15 December 2017.

[11] A. Iosup, R. Bovenkamp, S. Shen, A. Lu Jia, and D. Epema. An analysis of implicit social networks in multiplayer online games. *Internet computing*, 2014.

Appendix

Code for the implementation for this project is publicly available in the repository located at the following link: `https://github.com/Fladderman/distributed-systems-dragon-arena`.This repository is released under the GPL.