Dragon Arena System: A distributed multiplayer board game

Authors: SB Ramalingam Santhanakrishnan (4740270), and Z Sun (4733983)
ICT Innovation, EEMCS, TU Delft
Emails: {S.B.RamalingamSanthanakrishnan, Z.Sun-1}@student.tudelft.nl

Course Instructors: Jan S. Rellermeyer, PDS Group, EEMCS, TU Delft
Email: j.s.rellermeyer@tudelft.nl@tudelft.nl

Teaching Assistants: L Versluis and S Talluri, EEMCS, TU Delft
Emails: {L.F.D.Versluis@, S.Talluri@student}.tudelft.nl

***Abstract*—*Dragon Arena System* (DAS) is an online multiplayer board game service designed for the specific requirements of WantGame BV, implemented as a distributed system. Multiple players from different locations coordinate on the virtual board to eliminate a strong AI adversary, namely the dragons. The service is hosted on Amazon Web Services (AWS) IaaS barebone compute resources, running in multiple instances to allow for scalability and fault tolerance and the system is written in NodeJS in functional reactive style. In our design, we prioritize availability and the overall responsiveness of the service while maintaining a reasonable consistency. We test the system using simulated client bots and show that the system is responsive under load to 100 clients with a majority response time of 250ms and also analyze the fault tolerance capabilities of the system.**

I. Introduction

WantGame BV is a popular online gaming company and are designing a new board game ready to roll out to multiple players. As the company is entering the rapidly expanding market of gaming, they face stiff competition and must maximize the user experience in order to survive in the industry, if not make profit. The gaming server must be able to cope up with peak traffic demands and frequent disconnections from players. Thus, WantCloud BV is evaluating several system designs for their brand new game engine, one of which is presented and analyzed in this report.

The proposed system utilizes Amazon Web Services (AWS) [1] IaaS cloud service provider's compute instance, EC2 (Elastic Cloud Compute) for providing on-demand compute capacity to host the game engine. Apart from using EC2, we do not depend on any other services provided by AWS so that WantGame BV can switch providers or self-host in the future. The gaming engine itself is written in NodeJS [2], on a single threaded event loop as it is a comfortable programming model for a gaming system (traditionally games have been written in while-true loops) and uses TCP/IP socket communication via websockets using the socket.io library [3]. On the servers, the state is maintained on an immutable data store using the Redux library [4] and async events are handled by the RxJS library [5]. The gaming client is a small ReactJS [6] application which visualizes the board and allows keyboard inputs from the player. To simulate several players, we use client bot scripts which send simulated commands to the server, following a simple strategy to attack the dragons aggressively, events are logged both on the client and server side for later analysis.

The overall system design follows the master-slave architecture, where the master server gets events directly from the clients directly connected to it and forwarded events from the clients connected to the slave servers. The master periodically broadcasts the current state of the system to all connected clients and slaves, which ultimately forward the state to their own connected clients.

Prior work on designing gaming systems while maintaining consistency with efficient synchronization has been done in TSS [7] with mirrored server architecture. We have adopted TSS to resolve late incoming events only on the master server. We have used disconnection log data from the Game Trace Archive [8], specifically World of Warcraft's data and have scaled the timeline to our system for simulating client disconnection characteristics in our analysis. The goal of the test workload is to eliminate the AI units from the board.

In section II, additional background information on the application is provided. In section III we illustrate the design of the system in detail with focus on specific aspects pertaining to WantGame's requirements and the rationale behind the decisions and tradeoffs that were considered. In section IV we present the experiments conducted along with the results and analysis. In section V we discuss our findings briefly and list potential improvements. We conclude the report in section VI with a short summary on the main findings and recommendations to WantGame BV.

II. Application

The online gaming server receives commands from end-users to act on their unit via the client application in a pre-agreed data protocol. The game comprises of a 25x25 board with two types of units, Knights (player) and Dragons (AI adversary). Knights can move, Dragons are static and the attack range is 2 squares, while the heal range is 5 squares distance, both non-diagonal distance and only Knights can heal each other. Each unit has health points and attack points, and aim to reduce the health of opponent units by attacking, which results in the opponent unit's reduction of health by the attacking unit's attack points. The system has been written in Typescript [9], targeting NodeJS version 9 using ReactiveX extensions [10], ReduxJS [4] for immutable state in the backend. The frontend and client bot have also been written in Typescript with ReactJS.

A. Requirements

In order to provide a reliable and interactive service, WantGame's requires the following aspects to be handled by the new system.

- *Correct Operation*: At any point in the gameplay, the clients should see nearly consistent view of the board and actions applied on that state should be accommodated as

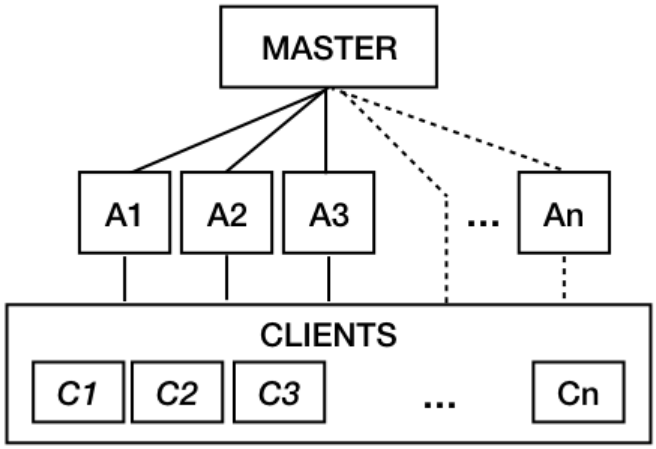

Fig. 1. Overview of system design.

far as possible, without violating the aforementioned rules of the game.

- *Fault Tolerance and Availability*: The system should be resilient to failure of the clients and provide reasonable allowance for reconnection. Failure of a server instance should not result in overall seizure of the gameplay.
- *Scalability*: The system should allow for connection of at least 100 players, supported by at least 5 functional servers.
- *Responsiveness*: To allow for interactive gameplay, the system must remain responsive and update the board on the client as soon as the game reaches the new state.
- *Monitoring*: At least two server instances should maintain a log of the events carried out.

III. System Design

The master node accepts the commands from several clients and also the forwarded commands from slaves. The command is then carried out by the master at the timestamp mentioned in the command if not too old or by rolling back to that state and then replaying all commands that happened on that state following the requested command (determined by a policy). Periodically, the state updates are also sent back to the clients by the server, in turn forwarded by the slaves to their own clients. Election of the master node is out of scope of the current implementation, thus we bootstrap the system with a pre-selected master and a list of successors. In the following sections we illustrate the components of the master and the overall working of the system in terms of consistency, synchronization, replication, fault-tolerance and availability.

A. Overview

The overall architecture of the system is illustrated in Figure 1, which can be seen consisting of a single master node, to which multiples slaves, A1, A2, etc., are connected. Clients can connect to slaves as well as the master directly, as far as the clients are concerned, the role of the server is opaque to them. Ideally, a load balancer is required in such architectures, however in this report we allow clients to randomly select servers from a known list and focus on other design aspects of the system.

The master node, illustrated in Figure 2 consists of the following components:

- *Execution Queue*: The queue which maintains commands pending execution.
- *Replay Set*: An ordered set (by timestamp) which holds the events that were previously executed, to support replays in the future. Older events in the set are periodically removed to recover memory.
- *Command Listener*: The socket server, which accepts client connections and subsequent commands, parses it, and adds it to the pending execution queue.
- *Executor*: The core game engine loop present in master node which executes events in the execution queue, is invoked periodically (every 50ms in our case) and executes the event in the queue head.
- *Broadcaster*: Broadcast the game state periodically (every 100ms in our case) to all the connected clients and slaves.
- *Supervisor*: Responsible for logging all the actions executed in the system, deciding weather the current node is master/slave, self-election as master and maintaining connection with master.

Slave nodes differ from the master node in that they have a command forwarding component with a forwarding queue, instead of the executor, a state updating component connected to the master and the replay set is not utilized until it assumes the role of a master.

B. Working

The components in Figure 2 each have their own life cycle and associated events. Since we follow the reactive pattern, we illustrate the working of the system by tracing the through the execution of the system on receipt of a command from a client, through the system's various components.

1) *Bootstrap*: This phase marks the start of the system, before the arrival of commands. The server nodes start and one of the nodes assumes leadership based on prior configuration and starts listening for client as well as slave connections. The game state consists of the board state with units and their health/attack points and also a scalar timestamp, set to 0 at the start. The timestamp is advanced on execution of each command. slaves make

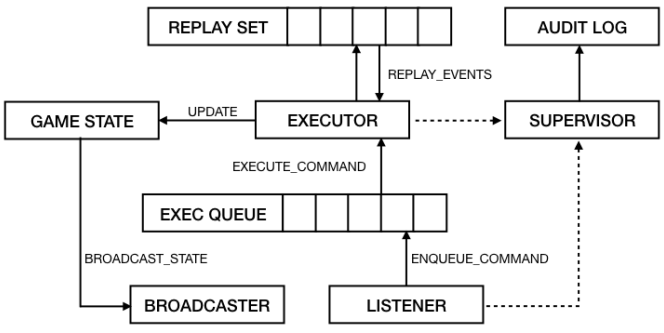

Fig. 2. Master components with message flows

their connections to the master and listen for client connections as well as state updates from the master.

2) *Command Receipt*: The listener listens for incoming commands from end-users and adds it to the execution queue. The executor periodically picks tasks from the execution queue, looks at the current timestamp and determines if the action can be executed on the current state if it is not too old (based on policy MIN_EXEC_WINDOW), if above the policy window, it determines if the timestamp is below an executable window (policy, Max_Exec_Window) and attempts the rollback and execute operation illustrated in [7], provided the state is available in replay set. If the action is not executable at this point, it is simply discarded. If exists, then the executor picks the corresponding state from the replay set and executes all the stored commands that occurred after the given timestamp and finally updates the resultant state and also stores it in the replay state. We try to maintain weak consistency, on the assumption that most of the actions requested by the player are local to his/her position on the board, thus limiting the chances of overwriting other parts of the board. In case there are conflicting events, then we follow the policy of last-write wins. We do not differentiate actions leading to weak/strong consistency unlike in [7].
3) *Game State Progress*: As can be seen in command receipt handling, we try to make the best effort to accommodate as many client commands as possible for a point in time. But this could lead to stagnation of clients at some arbitrary timestamp until the server makes further progress due to rollbacks, which leads to more events queueing up in the same timestamp, leading to very slow or no progress at. To mitigate this, we do not send out any state updates or accept moves while the rollback is under execution. The slaves may still continue to accept actions from their own clients but they are dropped by the master until the rollback-replay is complete.
4) *Master Failover*: When the master fails, all of its connected clients attempt to connect to one of the other known servers. The slaves which were connected, recognize the event via socket disconnection. One of the slaves become a master and once the new master is alive, the master prioritizes all its buffered client commands first and then makes its best effort to execute the fresh forwarded commands. In case of drastic state changes, the clients which are connected to the slaves will see a jump on their screens. It must be noted that the newly elected master is not aware of the replay set possessed by the previous master nor about its rollback/replay status, thus effectively everything is restarted from the last known stable state.
5) *Monitoring*: The role of the supervisor is to keep track of the executor and the listener. We explicitly log the execution queue each time an event is added to it, making the system repeatable to a large extent. We use the Bunyan [11] library for formatting our log messages in Log4J [12] type of format.

C. System Capabilities & Limitations

The major limitation of our architecture is that the clients connected directly to the master may enjoy better responsiveness if their connection paths offer lower latency compared to the clients connected to slaves. However, this could be mitigated by adding an artificial delay on the clients after a latency measurement step before spawning the units.

1) *Synchronization, Consistency and Replication*: Synchronization and replication between the servers are pretty much achieved out of the box as a result of our architecture. We follow a weak consistency model, partly due to the fact that the game we are designing is not mission critical, contrary to LockStep [13], which would greatly reduce the responsiveness. Since we only guarantee weak consistency, we ensure the best effort to accommodate as many commands from a client as possible.
2) *Availability and Responsiveness*: Firstly, we make the system responsive by allowing clients to send commands asynchronously from the state updates. However, this could lead to inconsistent or updates made with a stale state assumption. This problem is mitigated by the mechanisms discussed in the previous sections. During failover, we continue to collect commands which makes the server available, but there is no guarantee if those messages will see light, unless the target server becomes the master.
3) *Fault Tolerance*: Our current implementation, can tolerate fail-stop on both the server as well as the client side. When a client fails, we wait for a timeout period until which if the player reconnects and claims the unit id, is allowed to continue, else the unit is removed from the board (UNIT_REMOVE_TIMEOUT).

IV. EXPERIMENTAL RESULTS

We conduct a total of 3 experiments and analyse the results in the following sub-sections.

A. Setup

The goal of the experiments is to analyse the impact on consistency and responsiveness as a function of total client connections and server failures. The test system is deployed on the AWS cloud platform using bare bone EC2 compute instances. We use the m3.medium type (vcpus=1, memory=3.75Gb based on Intel Xeon E5-2670 v2) for both the master and the slave nodes. We restricted the deployment to a single region, North Virginia (US-East-1) with default availability zone placement policy and the default virtual private cloud settings provided by AWS. A custom security group is used for exposing the required ports.

The workload consists of client bots, managed by a runner which incrementally spawns upto 100 bots. For simulation, the bots follow a simple game strategy of attacking a dragon if healthy, else move towards another knight. If another nearby

TABLE I
FACTORS AFFECTING GAMEPLAY

Factor	Value
STATE_UPDATE_INTERVAL	1000 ms
CLIENT_MOVE_INTERVAL	1000 ms
MIN_EXEC_WINDOW	250
MAX_EXEC_WINDOW	500

knight is not health, then heal else move towards a dragon. The logs are recorded on the local AWS disk and analysed separately. We use python libraries pandas [14], scipy [15] and numpy [16] to analyse the time series log data and matplotlib [17] for visualizing the same. We also run the client bots on the same AWS region and availability zone subnet on a separate EC2 instance to minimize network disruptions. Unless specified, all the experiments use 5 server nodes. In order to merge the timelines of various clients, we perform interpolation of time series data so that we can approximate the values at each real-world clock second. Table I lists the factors of our experiment that directly affect our gameplay. The rationale behind the chosen values is to have an observable resolution on the timeline and exec window so that we can accumulate a reasonable amount of replay sets and also discard very old events in our experiments.

B. Experiments

1) Overheads: The latency between EC2 servers was found to be an average of 3ms. To measure client consistency, we make them send the their board state explicitly back to the server at random timestamps, this leads to an extra command, which is however just for logging purposes but not executed by the server, leading to a negligible overhead. Another source of overhead is logging, but negligible. We did not find any other significant overheads that could affect the experiments on the client bot or any other part of the system.

2) Consistency Deviation: The metric for consistency deviation is the number of square differences in the board position at a given timestamp. In order to measure this, the clients periodically send their board state at 10 timestamps in the past and the server compares the received board with its own board at the same timestamp and logs the deviation. Figure 3 shows the plot of a 180 second time period, starting at a stable state, where a an average of around 5 squares deviation is observed. However, since during rollback-replay the server shuts off any interaction, we observe spikes that last upto 10-15 seconds where the board differs by more than 20 squares. In terms of user experience, the clients would have experienced several resets and a jumpy game experience during this period.

3) Responsiveness: The metric for responsiveness that we use is the response time, defined as the time taken for the server to accept the client's command, sent at the current timestamp, execute it and reflected it on the next state update. We measure this end-to-end on the client side by checking the next received board state and see if the expected change had occurred. We restrict our probes to consider only the move commands because it is difficult to ascertain the same for attack and heal if any other player unit's action is affecting the same target square. To this effect, we keep only one dragon on the board and keep changing its location randomly so that the knights keep moving as much as possible.

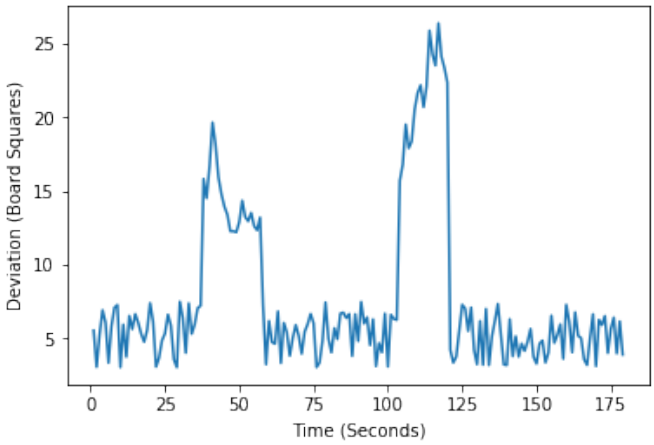

Fig. 3. Average consistency deviation of all the clients for a sample time slice of 180 seconds.

The average response time over a period of 120 seconds is plotted in Figure 4. We gradually increased the client connections from 0 to 100 at the rate of 5 clients joining every second. We see some dips in the average because as soon as the client connects, the current state is sent by the server synchronously, leading to a fast response time for the first state update, which skews the average a little bit. We observe that although the simulation finished adding 100 clients at the 25 seconds mark, the system actually spawns it only at around the 28 second mark. At around 40 seconds, we see a spike, corresponding to that, we observed rollback and replay occurring on the master. The rollback and replay causes the system to not accept any events and also creates further thrashing as an avalanche of events reach the server right after the rollback-replay ends. At around the 80 second mark, the system reached a stable state again and continued to maintain an average response time at about 300ms despite occasional

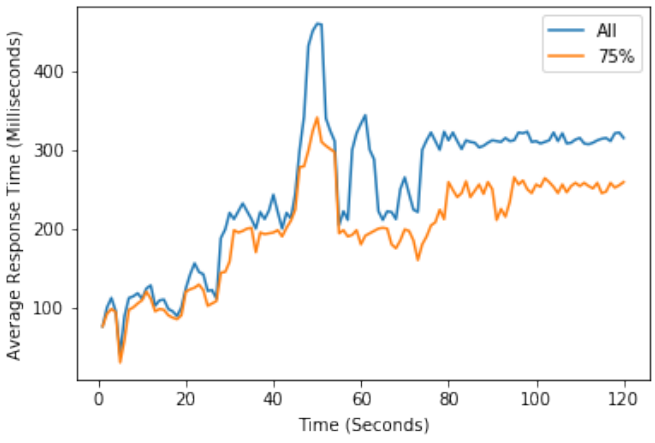

Fig. 4. Average response time of all the clients, over a period of time.

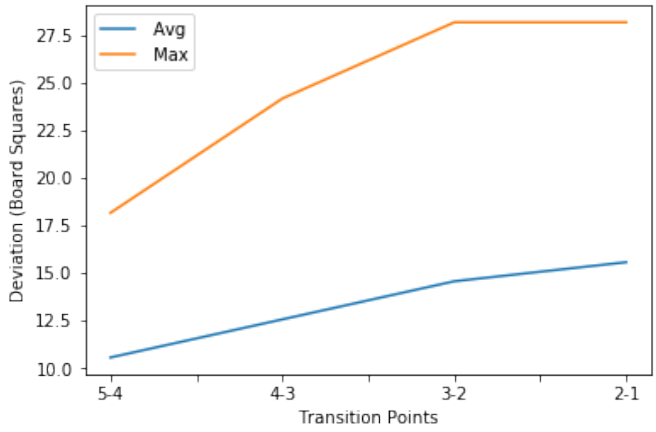

Fig. 5. Consistency maintenance during server failures

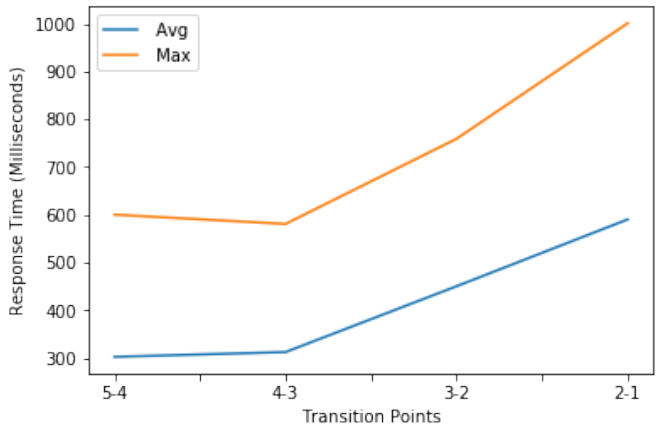

Fig. 6. Responsiveness maintenance during server failure

replays. Further we found that more than 75% of the clients got the response within 250ms after reaching stability. Considering that in the real-world that many knights will be killed and clients be disconnecting, the response times are reasonable.

4) Fault tolerance: In order to understand how the failure of master impacts the system, we start with 5 server instances and kill the master one after the other after the system has stabilized. During this time, we collect the data on consistency deviation and the average response time for the clients.

In Figure 5 and Figure 6 we plot the average consistency and average response times 10 seconds after the event of master failure, starting from 5 servers down to 1. We observe a general trend towards increased consistency deviation and decreased responsiveness. Although the next master starts processing events within 500ms (due to an available list of successors), we observed that the clients who were previously connected to the master take upto 5 seconds to retry and successfully connect to the remaining available servers. Since the clients do not have the knowledge of which server died and due to what reason, they randomly keep trying out servers, which causes maximum delay when the last server is remaining.

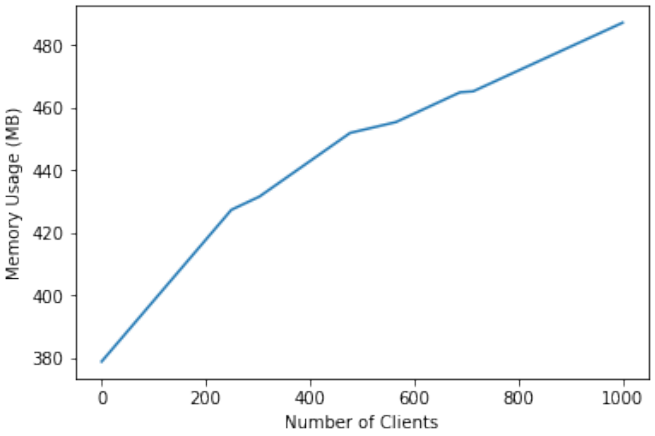

Fig. 7. Overall Memory Usage, benchmarked on a single server for upto 1000 client connections

C. Server connection Benchmark

In order to aid future decisions on load balancing and capacity allocation, we benchmark a single server node running on an EC2 instance and we measure the memory and cpu load using NodeJS standard libraries as a function of clients connected, upto 1000 clients. For this experiment, the AI dragons were disabled, the board size was expanded to 50x50 to accommodate 1000 clients and the clients were setup to make random moves.

From Figure 7, initial overall system memory usage was around 380MB. With the number of connection increasing to 1000, the memory usage reaches around 489MB. Considering the total memory usage and the number of connections, the average memory consumed by each connection is around 0.109 MB. The CPU usage when there are constant socket connection arriving at the rate of 5 per second Figure 8, kept an upward tendency with the number of connection increasing and it reached a peak of around 8.5% when there are approximately 900 connections with moves incoming every 1 second. Of course, the consistency model was easy here since there are no attacks and hence removal of units. The benchmark proves that for the current game rules, a single master is able to handle incoming moves at a reasonable resource utilization.

V. Discussion

The experiments and analysis suggest that replay-rollback policy, combined with our architecture has a significant impact on the gameplay. Although stable response times are maintained most of the time, careful selection of the factors are required to provide the best user experience at the same time maintain a reasonable amount of consistency. The benchmarks show that our setup is capable of handling a large number of clients and that the whole system is I/O bound, with a high requirement of memory relative to CPU.

For future work on the implementation, we would like to work and present on a more accurate time series analysis by varying the factors and identify how the system responds to the specific problems we encountered in our current experiments.

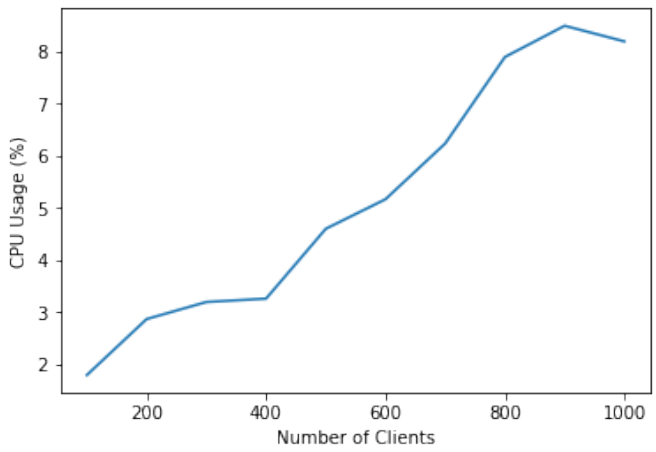

Fig. 8. System CPU Usage, benchmarked on a single server for upto 1000 client connections

In addition we would like to work extensively to improve in the following areas.

1) Master restart and detecting network partition. Currently, only fail-stop of master node is supported.
2) Scale up and scale down server instances dynamically. This is mainly to save cost but requires a leader election algorithm in place.
3) Implement a load balancer so that the client connections are distributed fairly and low latency servers are selected by the clients. Also prevent clients for retrying all the servers since load balancer already performs independent health checks.
4) Build replication and synchronization for hosting multiple masters (mirrored architecture). A more complex improvement could be to add a master to manage a separate region of the game map.

Clearly there are many other drawbacks in our implementation which we wish to overcome. However the experiments gave some insights into the various tradeoffs in building a distributed system which has tight constraints on responsiveness.

VI. Conclusion

In the report, we have designed and analysed a multiplayer gaming engine, aligned to the chief requirement of WantGame BV, to maintain responsiveness at peak traffic demands, which is clearly met by the system, as seen in the experimental results, where a majority of the clients are able to see their action executed in under 250 milliseconds, while maintaining a consistent view of the game. The tradeoff made in fault-tolerance and scalability has to be noted though, mainly in that the system is not elastic and has to have scaling mechanism in place without disruption of gameplay. In addition we found that in some cases, optimistically processing commands in the client and slave can greatly improve the responsiveness as well as offload the master server. After analyzing the tradeoffs, we think that it is a worthy endeavour for WantGame BV to build a system with the proposed design and keep improving as new features are to be met. On the other hand, for a game with larger squares we recommend an alternate architectures due to the memory load on the master.

References

[1] Amazon Web Services (AWS). [Online]. Available: https://aws.amazon.com/
[2] NodeJS. [Online]. Available: http://www.nodejs.org/
[3] Socket.io. [Online]. Available: http://www.socket.io/
[4] Reduxjs. [Online]. Available: http://redux.js.org/
[5] Rxjs. [Online]. Available: http://reactivex.io/rxjs/
[6] Reactjs. [Online]. Available: http://reactjs.org/
[7] E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin, "An efficient synchronization mechanism for mirrored game architectures," *Multimedia Tools and Applications*, vol. 23, no. 1, pp. 7–30, 2004.
[8] Y. Guo and A. Iosup, "The game trace archive," in *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*. IEEE Press, 2012, p. 4.
[9] Typescript programming language. [Online]. Available: https://www.typescriptlang.org/
[10] Reactive extensions. [Online]. Available: http://reactivex.io/
[11] Bunyan. [Online]. Available: https://github.com/trentm/node-bunyan
[12] Log4j. [Online]. Available: https://logging.apache.org/log4j/2.x/
[13] T. A. Funkhouser, "Ring: A client-server system for multi-user virtual environments," in *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM, 1995, pp. 85–ff.
[14] Python pandas. [Online]. Available: https://pandas.pydata.org/
[15] Python scipy. [Online]. Available: https://www.scipy.org/
[16] Python numpy. [Online]. Available: http://www.numpy.org/
[17] Python matplotlib. [Online]. Available: https://matplotlib.org/

Appendix

A. Source Code

The project source code is available at https://github.com/captainsano/das-game.

B. Time Sheets

TABLE II
Time spent on the project per activity.

Activity	Time (hours)
Total	170
Think	20
Dev	80
XP	15
Analysis	15
Write	20
Wasted	20

TABLE III
Time spent per experiment.

	Time (hours)		
	Dev	Setup	Total
Consistency Deviation	8	4	12
Responsiveness	1	0.5	1.5
Fault Tolerance	1	0.5	1.5
Benchmark	1	0.5	1.5