IN4391 - Distributed Computing Systems
The Dragons Arena System

March 26, 2018

Apourva Parthasarathy
Delft University of Technology
Delft, Netherlands

Bhavya Jain
Delft University of Technology
Delft, Netherlands
B.Jain@student.tudelft.nl

Laurens Versluis*
Delft University of Technology
Delft, Netherlands

Sacheendra Talluri†
Delft University of Technology
Delft, Netherlands

Dr. J.S. Rellermeyer‡
Delft University of Technology
Delft, Netherlands
J.S.Rellermeyer@tudelft.nl

ABSTRACT

This project report discusses the design and implementation of The Dragons Arena System Game. The aim of the project was to design and implement the game to provide distributed, scalable, consistent and fault-tolerant execution. The game engine must allow multiple geographically distributed players to participate in the game at the same time. The game architecture designed comprises of 2 key layers, the Game Server layer, where the the game state is maintained and actions in the game are executed and the Request Handling Servers layer, where the players interact with the system and submit their actions. The rationale behind this architecture is to load balance players across servers based on their geographic location. Player actions were simulated by randomizing their movement and automating other offense and healing actions, while player joining was simulated using a Game Trace Archive. The communication among servers is implemented using Java Remote Method Invocation (RMI)

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; **Cloud computing**; • **Theory of computation** → *Distributed computing models*;

KEYWORDS

Distributed Computing, Concurrency, Fault Tolerance, Scalability, Game Simulation

1 INTRODUCTION

The *Dragons Arena System (DAS)* is a First Person Strategy Warfare game by *WantGame BV* who want to scale up their infrastructure to make DAS a Massive Multi-Player online game. This is to be achieved by design and implementation of a distributed game engine for DAS. The system supports many concurrent users to collaborate and play the game together. This is achieved by incorporating distributed architecture within the game engine, allowing for high performance, scalability while ensuring consistency and reliability.

The game is implemented in Java and communication between various servers and players is performed using Remote Method Invocation (RMI). This report first addresses the background and requirements for the project followed by the proposed architecture for the distributed game engine. This report discusses how the architecture satisfies the requirements of supporting hundreds of concurrent players and deals with the complexity of running the game engine in a distributed fashion. Finally the report discusses the experiments conducted to test the suitability and validity of the system design and the results will be discussed.

2 BACKGROUND ON APPLICATION

The *Dragons Arena System (DAS)* is a First Person Strategy Warfare game between computer-controlled Dragons and hundreds of virtual knights (avatars or real life humans). The game comprises of a virtual world of 25x25 squares where the participants play. Dragons and Players are spawned in the virtual world or the battlefield, with each participant occupying a single square and no two participants at same location. The aim of the game is for participant to kill the enemy and survive till the end. Hence, the game ends when either all Dragons have been killed or if the Dragons have succeeded in slaying all the players. The game provides specific rules that govern the the movements of Players, the actions that Players and Dragons are allowed to perform and how points are gained or lost based on these actions. Players and Dragons can strike each other if they are a certain distance apart. Players are allowed to heal other players present nearby.

Since there are multiple players logged in to the same game session, the game engine should be scalable to support the players and their actions without a significant loss in performance. The system must also take into account that the players from various geographical regions may be connecting to a single game session. Hence, *WantGame BV* requires a scalable distributed architecture where servers in each geographical region support players in that region. These servers communicate with servers in the other regions so as to provide a consistent view of the game to all players. This architecture poses several synchronization and consistency

*Teaching Assistant
†Teaching Assistant
‡Course Instructor

challenges. All participants in a game session should be able to see same game state at all times. If any participant takes an action, it should be updated at all players as soon as possible.

It is also important that the game experience for participants is not be effected by server failures. In the event of any failure, the participants should be migrated seamlessly to a different server and allowed to resume the game from the most recent state of the players recorded at the new server. The aim is to ensure little to no loss of participant actions.

Performance of the game must be high, in terms of time it takes to implement users' desired actions and responses from other participants and updates of their actions. This is required in current fast paced environment and any delay between instruction and execution of an action is not desirable as it can lead to participants' dissatisfaction.

3 SYSTEM DESIGN

This section discusses the system design and architecture proposed as a solution for *WantGameBV's* requirements. This has been developed keeping in mind operational requirements, fault tolerance, multi-tenancy and scalability requirements. This section also discusses how the game is deployed across various servers and game status is constantly synchronized.

3.1 System Operation Requirements

The proposed solution comprises of 3 types of servers: Game Servers, Request Handling Servers and Backup Servers as seen in 1

The central Game Server maintains the state of the Battlefield and synchronizes the state on all the Request Handling and Backup servers. Request Handling Servers are deployed in various geographical regions. The participants (Human players) connect to the Request Handling Servers which process the request and forward the action to the central Game Server. The Game Server updates the state of the Battlefield based on the user action and sends the updated state to all other Request handling Servsrs and Backup Servers. Backup Servers are present for fault tolerance and are constantly backed up with latest server state.

The game is designed to support 100 players and 20 dragons per game session or game instance. All participants play the game on a common battlefield which is unique to a game session. The game communication is done by various types of messages.

3.1.1 Communication Messages. There are 12 different types of messages used for communication with the participants and within the game engine. 2 provides an overview of the message passing during the game.

The *spawnUnit* message is sent to initialize participants, both Dragons and Players while the *removeUnit* message removes the participants from the *battlefield*. The *getUnit* message fetches the state of the unit at a given location on the *battlefield*.

moveUnit message transfers the request of player to move to a new location. The new location requested by the player is within the game constraints. The *dealDamage* message is used by a Player to deal damage to a Dragon and vice versa. Players use *healDamage* message to heal a nearby player.

The *setup* message is used to perform initialzation on various servers and bootstrap the game. *sync* messages are sent by the Game Server after every action that is executed for any participant. This message ensures that all the servers have updated game state. The *heartbeat* message is used to constantly check for the server status in order to implement fall-back or hot-swap in server failure scenario. The *changeServer* changes the server in use to one of the *backup servers* in case of server failure.

3.1.2 Game Servers. The Game Server implements the logic of the game engine and is responsible for executing player actions, updating game state and synchronizing the game state on local Request Handling Servers. Each game session or game instance is run on *one* Game Server. The system architecture however easily allows for multi-tenancy by implementing a centralized resource manager and scheduler. The resource manager will be responsible for starting a new game instance when the number of players playing in the existing session reaches the limit. This new instance can be started on one of the already running servers or a new server can be provisioned depending on resource availability.

The Game Server maintains global state of the game, including the Dragon and Player states. Complete information about participants like Health Points, Attack Points, Strike Power and Location is maintained here. The game servers accepts players' requests via various types of *request* messages from the Request Handling Servers. The Game Server executes the specified action on the Battlefield thus updating the state of the Battlefield, the Players and Dragons involved in the action. Section 3.2 on consistency discussed how the Game Server deals with conflicting actions from different players. Once the action is executed, the Game Server immediately sends a synchronization message to all other Request Handling Servers to ensure they all have updated game state.

3.1.3 Request Handling Servers. Players send their game actions to their allocated Request handling Server. The current implementation of the system does not provide a service to match a new player to the correct Request Handling Server. However, the system has been designed keeping this requirement in mind. A new player that wants to join the game sends a Spawn request to a central routing service (this can be deployed on the same machine as the game server). The routing service- depending on the geographical location of the player and the load on the currently executing Request Handling Servers in that region- matches the player to the most suitable server. The player then connects to the Request Handling Server and begins playing the game.

When the Request Handling Server receives the layer action, it forwards the request to the Game Server. The Game Server executes the action based on the game logic and sends a reply to the Request Handling Server. The reply informs the Request Handling Server if the action was executed successfully and provides the updated game state. The Request Handling Server then updates it's own local state to be consistent with the Game Server. At the same time, the Game Server also sends *sync* messages to all other Request Handling Servers so that they may update their local state. The

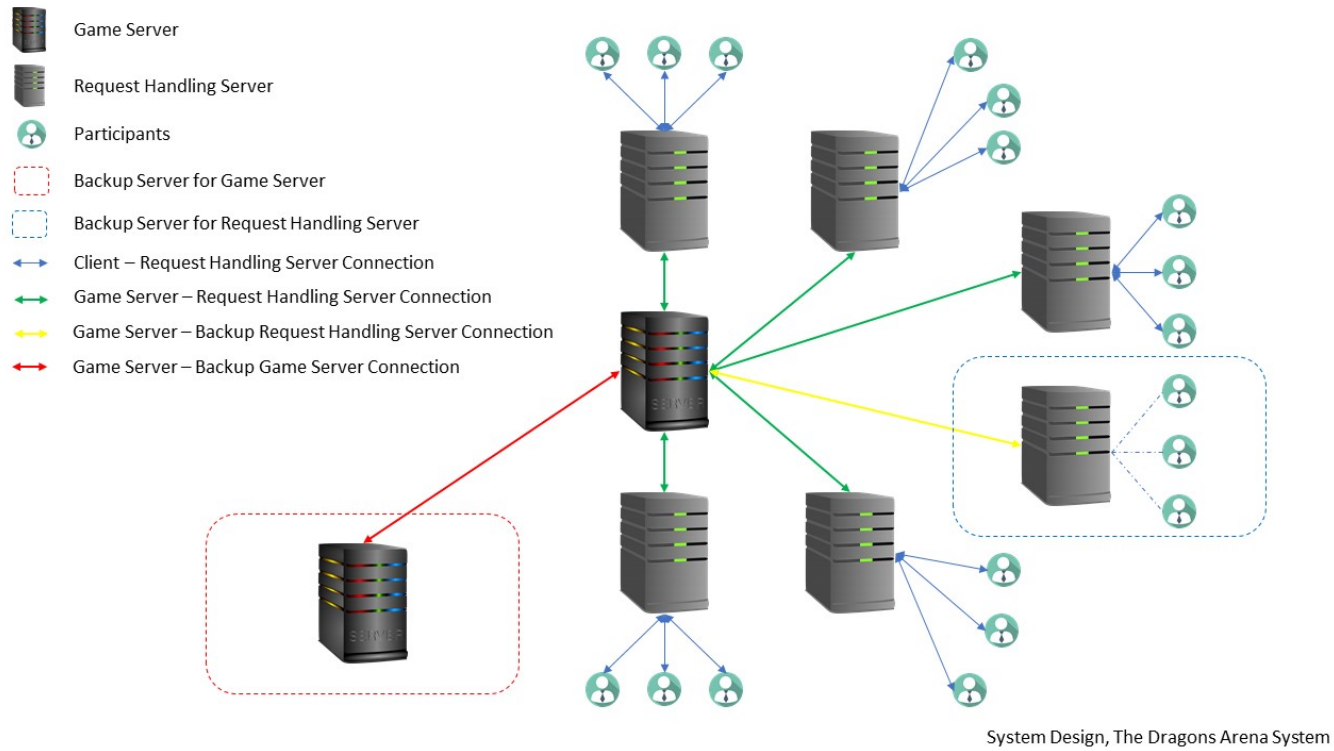

Figure 1: The Dragons Arena System

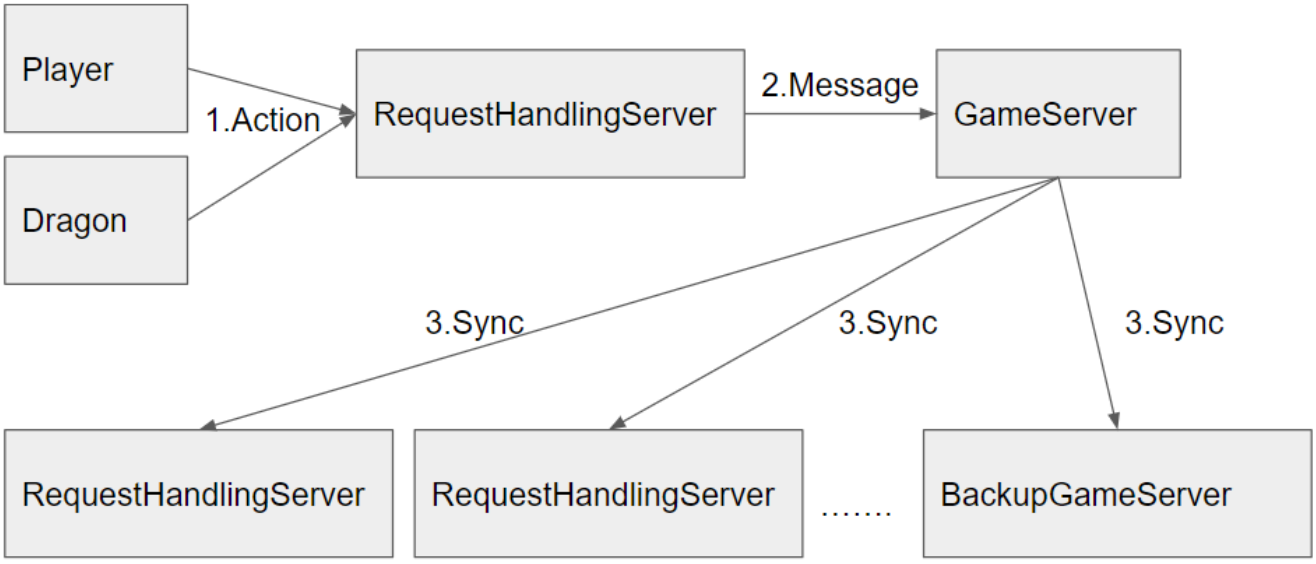

Figure 2: Message passing between servers in Dragons Arena System

Request Handling Servers periodically send the updated game state to the players.

In our design, we decided to implement multiple intermediary Request Handling Servers instead of a single central Game Server. This was to allow for the game to be truly distributed in realistic

scenario where players from all around the globe will be logged in a game session. Multiple intermediary Request Handling Servers would be distributed geographically to reduce the latency that user might face. For example, if there is a game instance with main Game Server located in Europe, then a player joining in from California will not have to connect to the Europe server, but to the nearest Request Handling Server with available spots for new players. This architecture also leverages the fact that

3.1.4 Backup Servers. The *Backup Servers* are used to ensure availability in case of server failure. The proposed solution has one *Backup Server* for the *Game Server* and one for all the *Request Handling Servers* in a region. Every update to the game state is synchronized at the *Backup Server*. The game server periodically sends heartbeat messages to the backup server. When the backup does not receive a heartbeat from a server for a specific period of time, it assumes that the server has failed and assumes the role of the game server.

3.2 Replication

All the participants and dragons are assigned hit points and attack points when they are spawned. A player sends his action to the Request Handling Server which forwards the message to the Game Server. The action is executed at the Game Server, the state is updated and this updated state is send as a *sync* message to all the Request Handling Servers. Hence, all local Request Handling Servers maintain an up-to-date replica of the game state so that they can service read requests from the local players with low latency. The game state is also replicated at the Backup servers in order to provide high availability in case of server failures.

3.3 Consistency

The system maintains consistent state across all servers through the use of *sync* messages which carry updated game state from Game Server to Request Handling Servers and Backup Servers. At the Game Server, the consistency of the game state is maintained by making all updates to the game state synchronized. Hence, only one thread at a time can update the game state and all updated to the game state are atomic.

In case of conflicting *MoveUnit* actions by players, the thread that executes first wins. The players whose action was successful gets a success message while the other player gets a null reply. All players receive the updated game state from a *sync* message so that all players have the correct information about the result of the action.

Similarly, if multiple players perform *HealUnit* on the same unit at the same time, the unit is healed until it reaches it's maximum health value. Subsequent heal messages are ignored and the senders receive a null reply in this case.

3.4 Fault Tolerance

Fault Tolerance is incorporated in the system design using periodic server status check by the *Backup Servers*. When a server failure is detected, the backup servers automatically take over.

3.4.1 Backup for Game Server. If the *Game Server* suffers a failure, all the individual *Request Handling Servers* have to point to the *Backup Server*. The backup sever implements a Failover Service which is responsible for seamless transfer of control to the backup server. The failover service sends a message to all the Request Handling Servers to change their Game Server connection the to backup server. The backup server then resumes execution from the last synchronized state available at this server. Hence, only the messages that were queued for execution when the game server crashed are lost. The Request Handling Servers can re-transmit the lost messages to the backup server and the game continues.

3.4.2 Backup for Request Handling Servers. When a Request Handling Server failure is detected, the backup informs all the clients connected to the Request Handling server to now connect to the backup server. As the backup server maintains the latest Game State, it can resume the game from the latest synchronized state. Hence, the positions of the players and their health and attack points are not lost and the player continues the game without any awareness of server failure. In the current implementation of the system, client connections are simulated using Game Trace Archive (GTA). The failover service thus creates the lost client connections on the backup server from the GTA and the game resumes.

3.4.3 Fault tolerance for Backup Servers. When a sever sends a heartbeat to the backup, the backup responds with it's own heartbeat to one of the designated game servers. If the heartbeat reply is not received within a specific period of time, the backup server is assumed to have crashed. The game server then brings up a new backup server and synchronizes the game state on this server.

3.5 Scalability

The proposed solution is highly scalable. Each game supports 100 players and 20 dragons which is the expected load for each game instance. Based on the capacity of the Request Handling Server, we assign each server a maximum of 20 players. When more players join, new Request Handling Servers are provisioned such that each server handles a maximum of 20 players. Each Game Server services a maximum of 5 Request Handling Servers (corresponding to a maximum of 100 players). Once the number of players exceeds 100, a new instance of the game is started on a new Game Server.

Each game instance runs on a separate server, hence our system can be extended to any number of game instances based on the demand. As players leave, Request Handling Servers may be de-commissioned so that we do not pay for resources that are not needed. This elastic scaling allows us to support variations in demand at a low cost by provisioning resources per demand.

3.6 Logging

Every server implements a logging service which logs all actions performed at the server along with the timestamp. These logs can be used for debugging purposes and also for analyzing system operation.

4 EXPERIMENTAL RESULTS

Multiple experiments were conducted to ensure the system operates as required and intended and is truly scalable, reliable and fault-tolerant.

4.1 Experimental Setup

The multi-threaded code was deployed on AWS EC2, running Amazon Linux on t2.micro instances. Each instance has 1 GB of RAM, 1 CPU and 8 GB of storage. The servers and run on JDK 1.8.0. Secure Connection was established using Putty and files were transfered using WinSCP. Security groups for each EC2 instance were modified to allow for communication between instances. Communication between instances was done using JAVA RMI (Remote Method Invocation), and port 1099 was used to register RMI service.

The number of players and dragons were passed as arguments to the main functions of Request Handling Servers and Game Servers respectively. Network Configuration file was added to the system to allow for registry of individual servers with RMI. The Game Server bootstraps the game by setting up the battlefield and communicating this to the other servers. Each request handling server then makes use of the Game Trace Archive to simulate players joining and leaving the system. The Game trace consists of player ids, the time at which the player joins and the time at which the player leaves the game.

4.2 Experiments

4.2.1 Consistency. The system was initially tested on our local machines in order to verify that the game state remains consistent as the game makes progress. The system runs one Game Server, two Request Handling Servers, One Backup Game Server and one Backup Request Handling Server.

The battlefield is implemented as a singleton on the Game Server. Writes to the battlefield states are synchronized so that only a single thread can update the state at a time.

Consider the scenario where two players perform a conflicting action at the same time. For example, two players want to move to the same position. In this case, the move actions are executed by two different threads on the main server and the thread that executes first wins. Hence, the player whose move was successful will receive a success message while the player whose move was denied will receive a null message. All other Request Handling Servers (and hence the players connected to this server) receive a sync message so that they may update the move action in their local state.

During our experiment, we verified that this behavior was working as intended by examining the log files which record all actions in the game.

4.2.2 Fault Tolerance. The system was tested for fault tolerance based on two aspects: Failure Detection and Automatic Recovery from Failure. In order to test these features, we run one Game Server, two Request Handling Server, One Backup Game Server and one Backup Request Handling Server. First the Backup Request Handling Server was started, followed by the Request Handling Servers, Backup Game Server and Game Server. The system runs 20 players per Request Handling Server (Total 40) and 2 Dragons.

Failure Detection: From the log file, we observe that based on the configured time interval (5 seconds in this experiment), heartbeat messages were exchanged between the Game Server, Request Handling Server and the respective backup servers. When the Request Handling Server or the Game Server process is killed during the game, the log file shows that the backup detects the missing heartbeat and reports the failure of the server to the Failover Service.

Automatic Recovery from Failure: In case of Request Handling Server failure, the failover service spawned new threads for the players and dragons on the backup server and they resumed the execution of the game from the latest state present at the backup server. Hence, the players are in the same position and have the same health state as they did when the Request Handling Server crashed. When the Game Server fails, the backup server assumes the role of the Game Server and send a message to all Request Handling Servers and Backup Server to point to this server.

4.2.3 Scalability. The system was tested to meet the required scalability requirements, running one Game Server, five Request Handling Server, One Backup Game Server and one Backup Request Handling Server. Each of these servers was implemented on an AWS instance. The system ran successfully for 100 players (20 per Request Handling Server) and 20 Dragons.

5 DISCUSSION

When designing the architecture of the system, our main aim was to make the game engine highly scalable and fault tolerant. Though not currently implemented, the architecture supports the extension of the system with a resource managers and scheduler that can provide multi-tenancy. As we have already investigated the threshold of the number of players each game instance can support, the Resource Manager can be designed to provide a simple service to start and kill new servers and allocate the appropriate number of players to each server based on these thresholds.

One of the important design considerations in our system is the user of local Request Handling Servers and global Game Server in order to provide a trade-off between performance and system complexity. While this architecture leads to a large number *sync* messages to be communicated, it provides the important benefit of load balancing players from different geographical regions on closely located Request Handling Servers thus improving scalability and providing low latency for read operations.

6 CONCLUSION

In this report, we have discussed the requirements for the Distributed Dragons Arena System and the design and implementation of the system. We discuss how the system satisfies the basic requirements of consistency, replication, fault-tolerance and scalability and the trade-offs considered in oder to build a system that best satisfies these requirements. Finally, we discuss how the Dragons Arena Game was simulated using a Game Trace and the results of the experiments involving 100 players and 20 dragons.

The code and report is open-source and is available at https://github.com/BSJAIN92/IN4391_LAB_B

7 APPENDIX A: TIME SHEETS

The implementation of the game was completed within the stipulated time frame and all the mandatory requirements were met. Table 1 shows the breakdown of the amount of time spent on the project.

Table 1: Time Spent on the Project per Activity

Activity	Time
Total Time	114
Think Time	16
Dev Time	50
XP Time	14
Analysis Time	4
Write Time	20
Wasted Time	10