# Distributed Dragon Arena System

Achmadnoer Sukma Wicaksana
4517253
AchmadnoerSukmaWicaksana@student.tudelft.nl

Arkka Dhiratara
4437039
ArkkaDhiratara@student.tudelft.nl

| | |
|---|---|
| Due Date: | April 22, 2016. 11:59 PM |
| Lecturer: | Dr. ir. Alexandru Iosup |
| Lab Assistant: | Drs. Yong Guo |
| Class: | IN4391 Distributed Computing System |

**Abstract**

Massively Multiplayer Online Game (MMO) is a computer game which is capable of supporting hundreds or thousands of players simultaneously. In order to deliver the best experience for users on this scale, MMO needs to build based on a distributed system architecture. WantGame BV is a start-up gaming company that aims to deliver their Dragon Arena game as MMO that enable high number of players. As for the implementation, we develop a distributed system by using hierarchical architecture for Dragon Arena that focused on key requirements such as consistency, availability, scalability, and fault-tolerance. Experiments are conducted to test these aspects and we discuss the results. Based on that, we can draw conclusion whether this design is suitable for WantGame BV to deploy.

Keywords: Distributed system, Massive Multiplayer Online Game, MMO, hierarchical.

# 1 Introduction

Playing game with friends is indeed more fun than playing the game by yourself. This experience is the key selling point that massive multiplayer online (MMO) game has to offer. However, massive number of concurrent players surely also raises technical difficulties on how to process all of the client request in timely manner. Moreover, MMO game also require to produce a global game state that ensure game logic working as it should. In addition, MMO also require to ensure availability, scalability and fault-tolerance of the service. WantGame BV is a start-up gaming company that planned to implement distributed system for their game engine to ensure better performance for their promising game, Dragon Arena System.

The distributed Dragon Arena System that has been designed consists of five servers split into coordinators and workers. Coordinators act as the entry point for users to log in and send their actions. Not only that, these coordinators also act as a scheduler and maintain mutual exclusion in order for the game to run smoothly. Furthermore, workers have the task to get the scheduled actions and compute it. The detailed system design, as well as the implementation, will be covered more in the next sections.

In this report, the background about the game is described in Section 2. Then, the chosen system design and its experiments implementation are stated in Section 3 and Section 4, respectively. Finally, the discussion about the chosen design trade-off along the conclusion for it is stated in Section 5 and 6.

# 2 Background: Dragon Arena System

Dragon Arena System (DAS) is a 25x25 grid-based online game that consists of a number of dragons and players which battling each other to be the remaining species. The game is considered to finish when only one species left in the arena. Each grid can only contain one dragon or player at any point in time. The dragons and players both have a certain amount of health points and attack points. Players have the capability of moving, attacking and healing other players. On the other hand, Dragons have higher health and attack points but can only attack players without moving. The effect of attacking is subtracting target health point with attacker attack points. If the health after attacking is zero or lower, unit is dead and removed from the arena.

In order the game to run smoothly, all the players need to have the similar state of the game. Therefore, the consistency of the state of the game is important and required across the servers. The game should be able to handle 100 players and 20 dragons at the same time. This means that the designed system needs to be scalable with at least these numbers of game entities. Along the time, some unfortunate event might happen, such as client and node failures. The proposed system then have to be resilient to these kinds of situations without affecting the ongoing game. In addition, newly fixed servers should be easily added to the system without any disturbances to the service.

# 3 System Design

This section covers all the chosen design aspects of the distributed system. Section 3.1 explains the intention on how the system should reflect real-life scenario as well as the interpretation of that intention to the system architecture for this project. This also includes the methods for handling fault-tolerance, consistency, and also scalability aspect of the system. In the section 3.2, we discuss additional features that have been implemented to the system.

## 3.1 System Overview

For starter, we have a vision that this kind of multiplayer online game could have players from around the world and geographically separated. Based on this idea, we divide the system into two parts; coordinators that act as the main point for the user to connect and the workers that have the task to compute the actions for the system. In addition, this hierarchical approach also helps us to ensure security aspects for the system as fewer servers are exposed directly to the user. The main idea is that the pair of coordinators (active-standby) will be deployed around the world in a region based system while the workers can be shared by multiple coordinators.
In this project, we want to deploy our aforementioned idea with the minimum specification of one region based implementation with Java language and Java RMI library. This would be discussed on the following section 3.1.1 along with the mechanism and scenarios of communications, consistency & scheduling, and fault-tolerance on section 3.1.2, 3.1.3, and 3.1.4, respectively.

### 3.1.1 System Operation

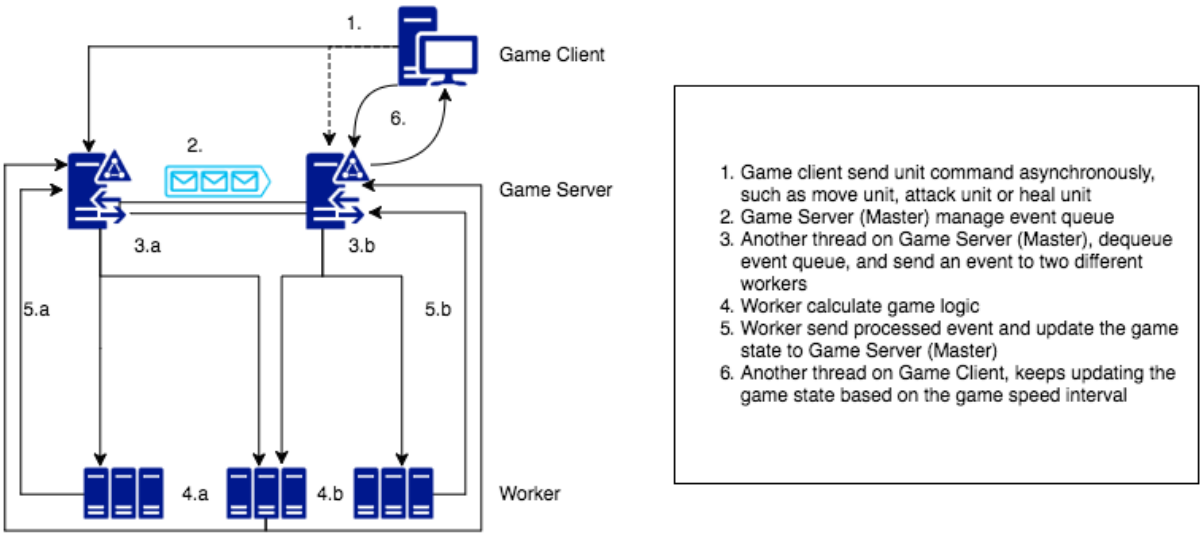


Figure 1: Distributed Dragon Arena System Architecture & Flows

In our design, the system consists of two different nodes which are Game Server/Master Coordinator and Worker. The game server acts as the endpoint for game client to connect and interact with the rest of system. Game client sending events as `EventMessage` that encapsulate any kind of unit actions such as unit move, unit heal, or unit attack. `EventMessage` are sent by client asynchronously to the game server. Game server store the incoming `EventMessage` from game client into `EventQueue`. Using this approach, the game server able to processed `EventMessage` in the distributed system environment and attain scalability. On the other thread of game server, game server dispatch `EventMessage` inside `EventQueue` to workers accordingly.

Next, workers will process the dispatched `EventMessage` and send the result (game state) back to the game server. As a result, Game client is able to retrieve updated game state using their update thread periodically.
Based on this design, we could see that game server has enormous responsibilities and pivotal role for the entire distributed system. Therefore, we also propose replication strategy on game server to ensure high availability using active-standby mechanism. Figure 1 shows overall design architecture as well as communication flows for the system.

#### 3.1.2 Communication: (Dis)connecting clients and servers

As mentioned in the previous sections, the system is designed so that clients can only communicate with coordinators. Figure 1 shows this client-coordinators communication configuration on the step (1). Each client has a list of IP addresses of both coordinators to connect. If a client wants to join a game, it should initiate connection towards master coordinator to authenticate itself using username and password. Upon successfully connected and authenticated, the client then can occupy one free random grid of the game and start playing. In addition, the master coordinator then registers the client to the player list and send this information towards backup coordinator. In parallel, the client also creates another thread to periodically pulling the newest game state from the coordinator as shown in Figure 1 step (6).
If the connection to the master coordinator is failed, then the client will divert its connection toward the backup coordinator. In this way, the process of user connecting is fault-tolerance and still provide a good user experience towards the user. Please note that for this kind of situation we do not account for network issue, so it means this unreachable situation only happens when the master coordinator is unreachable due to crash failure.
The communication flow when the user disconnects from the system also similar to the connecting flows. The user sends a message to the coordinator that it wants to log out and then the coordinator remove the player from list player and remove the unit from the arena. This information is also sent towards backup coordinator for replication purposes.
Beside clients connect and disconnect mechanism, the system also incorporates the mechanism to handle such events between servers, both coordinators and workers. This is done by using multicast for communication so that every time a server change status to online, it will inform other servers so that itself can be included in the system to do its job. This particular feature is useful when servers back to online after failure and then immediately can join the others.

#### 3.1.3 Scheduling & Consistency: Clients performing actions

User can perform actions after it successfully connected to the system. Each dragons or players have a random waiting time which mean our system would have to process actions that come in non-deterministic ways. The full procedure for this is described Figure 1 with complete flows from user sends request until it is processed by workers. The user sends the request asynchronously towards server without the needs of reply as we can see on step (1) on the Figure 1. This way we could avoid blocking in the client side in the case of the request is lost or no reply from server. This mechanism also maintains the consistency of the game state is always on the server side instead of relying on state within each client, although supposedly client will have periodically recent game state thanks to its pulling request for game state on another thread.
On our design, the game logic of Dragon Arena is processed by game worker. Each event that received by game server from game client (users) will be inserted into `EventQueue` based on the arrival time. This way the game server can maintain event ordering and smoothly dispatch the event with FIFO mechanism to the active workers using push method on different thread. Game worker responsible on determine whether each event is comply with the game logic and updating the game state, which cover unit movement, unit attack and unit heal. This is also the case when there is situation for mutual exclusion such as players want to move to the same (free) grid at the same time. Using this pattern, our system is able to cope and perform under an increased workload with horizontal scaling by adding more workers.

#### 3.1.4 Fault-Tolerance: Server(s) crashes

We have identify at least two connection type that need to be established in order our system to work, which are game client to game server and game server to workers. First, game client register any known game server address and connect to the game server. If the connected game server is crashed or not available, game client will move its connection to other game server. This mechanism is possible because each of game server replicate the same `EventMessage` and current game state which are required to continue its role. We can achieve smooth fail-over with this method and ensure that the client can still play and maintain good user experience. Fail-over mechanism for this system is shown in the Figure 1 with the step (3b), (4b), and (5b) which means the backup game server now handling the traffic due to master game server failure.
Second, if the workers have failure on their side, game servers get notification about current worker nodes availability whether it is *available*, *crash*, or *busy*. Using this approach, game server able to determine which worker is suitable to receive the actions of users. For example, game server will only choose workers that have state as ready. As soon as workers change its status to ready, the worker have the possibility to be chosen by game server. Using this design, we are able to achieve fault-tolerance for our distributed system despite of having workers failure.

### 3.2 Additional System Features

#### 3.2.1 Advanced Fault Tolerance

As we mentioned on previous chapter, our main challenges on this distributed system is handling massive number of concurrent users. Our system is required to process events sent by game client in timely matter. With this requirement, there is no room for any fault on the system. We implement extra precaution on dispatching events from game server to workers. As a minimum, game server dispatches one `EventMessage` to two different workers on the same time. Next, game server just required to retrieve only one result from workers to update the game state. This mechanism is able to overcome fault possibilities whether worker fail to completed the job or in the case when the request is missing in the network. In addition, this hierarchical design combined with our design for fault-tolerance enable multiple failure for server and workers happen at the same time, but with a cost of high-load on the remaining server and worker.

## 4 Experiment

In this section, we will provide simulation of connected players that sending different set of event messages to game server. We have developed bot client that mimic real-player behavior interaction to the game, such as moving unit randomly in the arena, attacking nearby unit, and also heal other unit (Knight) if necessary. Actions from units are written to log files. Furthermore, simple GUI can be used to see the progress of the game. The game is started when the first players enter the game and will end after one remaining species left. Using this simulation, we are hoping to simulate the real online game situation on the distributed system.

### 4.1 Simulation of connecting/disconnecting clients

In order to simulate connecting/disconnecting clients, we use data from the GTA (Game Trace Archive) that is resemblance of MMORPG [4]. We can conclude that amount of players at the morning is lower than at the night. This means that for our system, we started with the initial small value of connected bot for Dragon and Knight. Later, number of Knight bot is gradually increase and fill the arena. We also do some adjustment on how Knight bot spawn, this is important because if we spawn all dragon first before knight, most likely every each spawned knight will will be killed instantly by dragon and log out from the game. Therefore, we also take into account the proportion of Dragon and Knight accordingly to balance the game.

## 4.2 Experiment Setup

The simulation to test our distributed system fault-tolerance, availability and consistency capabilities is prepared as follow:

a. Normal Case: Using two laptops to emulate distributed system in different physical devices. One laptop runs one game server and two game workers. Another laptop runs one game server and one game worker. Player and Dragon increasingly added to arena while other existing units are still playing. Adding of units stop until number of maximal dragons and maximal players have reached.

b. Game Server failure: Normal game still running. One game server (main) crashed and have to be handover to the backup game server and game still running smoothly. Faulty game server back to work state and synchronize with the current master and now become backup game server.

c. Game worker failure: Normal game still running. One game worker failure. Game is still running normally. Faulty game worker back online, connected to the system, and ready to pull task from game server.

d. Two game workers failure: Normal game still running. Two game workers failure. Game still running normally. Faulty game workers back online, connected to the system, and ready to pull task from game server.

## 4.3 Experimental Results

### 4.3.1 Scalability

During the experiments, we introduce 100 players and 20 dragons as the game requirements to the system. As it turns out, the designed system can handle this specified number of players and their actions without outstanding issues. We also believed that the scalability of the workers can be easily increased by adding more game workers in the future to handle increased load of the games. However, for the Game Server we need different approach to handle increased load in the future. Due to the nature of the design that handled by single Game Server and have backup to ensure consistency and high-availability, we need to add another pairs of Game Server if we want to expand the service, let say to another region, like our original approach to the system. However, for the basic requirements of the game we believe this kind of standard setup have met its purpose.

### 4.3.2 Consistency & Fault-Tolerance

**Game Server failure**

In this experiment, we simulate condition whether there is a failure on the master game server. As we have described, game server has a pivotal role for the entire system. Therefore, we have implemented fault-tolerance and consistency capabilities to them by redundant communication channel from client and synchronization of game state between game servers. Figure 2 shows our result to emulate this kind of situation. As we can see from the graphic, the red line is increased when the normal game is still running and the server-1 still serves all the traffic towards the system.
When the server-1 crashed at around t-100, the server-2 is ready to take the job and the game still run smoothly towards the end. The workload that before routed to the master game server now switched towards backup game server as we can see with the rapid increase of green line on the graph. This scenario is possible because the consistency on both servers is maintained and client immediately reroute its request towards the backup server when the master server is failure. Later on when the faulty game server-1 is repaired and online, it will synchronize with the current master and immediately act as the new backup game server.
Another thing that we can observe from the graph is that there are fluctuation of workload over the time. This happens because the randomness of waiting time among users before each action. In addition to that, there are also the random aspect of the progress of the game, such as the movement, action, and also the remaining participants of the game over the time.

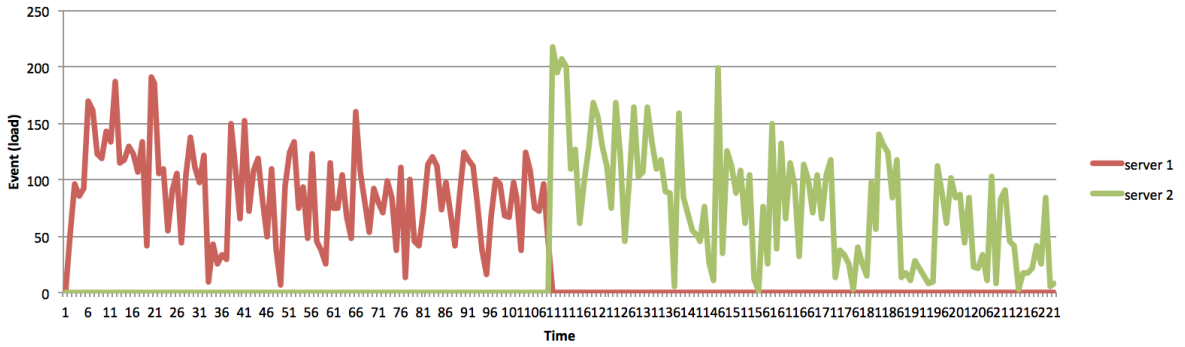


Figure 2: Scenario Game Server failure

### Game Worker(s) failure

In condition where one of the game worker fails, the whole system can still continue to work. This condition is possible, because game server is being notified or aware that there is a worker fail/crash and dispatch `EventMessage` to other workers. As we can see on Figure3, the game is running normally and the load is evenly distributed between the three game workers. However, worker-3 stop working on t-48. We could see that after that point of time, worker-4 and worker-5 still process the same share of workload dispatched by game server but with higher responsibility because they also handle the workload that previously should be handled by worker-3. We also simulate when the faulty game worker-3 come back to active and ready to serve. This happens at t-94 and as we can see that worker-3 has successfully integrate back to the system and as the result the successive workload now is evenly spread between three workers, instead of only two workers.
In the case of space-correlating failure where there is only one remaining worker available, the system still could handle the workload but with the cost of one workers have to handle all of the workload until the faulty game workers back to ready.

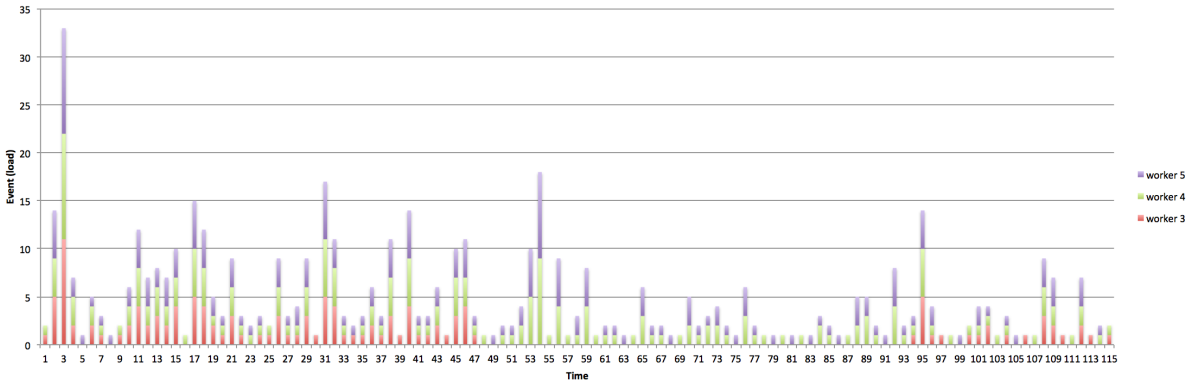


Figure 3: Scenario Worker failure

## 5 Discussion

In this section, we discuss about trade-off of our choice of design and the resulting experiments from Section 4. Then we would give recommendation to WantGame BV whether they should adapt their system to the designed distributed system.

### 5.1 Scalability

As discussed earlier, we choose hierarchical approach for the design of the distributed system to accommodate world-wide player base. This design successfully meets the basic requirements from the WantGame BV. If the client wish to expand this system, they can either expand the shared workers or introduce new pairs of game server on different region. Adding new worker is rather easy because it immediately can connect to the system and get the workload for the system. On the other hand, this system is designed so that the client should only point to one particular server within the region. The increased of workload pointing towards that server could cause bottleneck and affect entire system if not carefully planned. The solution of game server scalability then is by adding new pairs of game server. Although it is doable, it has drawback in term of simplicity because every time the workload can not be handled by the current server, WantGame BV needs to add pairs of game server instead of add gradually from one.
On the bright side, this design enables sharing of workers among several regions so that it can obtain maximize utilization of game workers instead of dedicated game workers for each pair of game server. Another thing to be concerned when deploying this distributed system is that traffic dimensioning and engineering are highly needed because when the game gain its popularity, the combination of workload from several regions need to be adapt to the shared workers capacity. These concerns and further directions are not within the scope of this document and should be tested in the future.

### 5.2 Consistency & Fault-Tolerance

In order to ensure consistency, the system use replication for entire game field among game servers. Moreover, this mirrored design enables the system to maintain high-availability and fault-tolerance aspect because whenever the master server is down, the backup server can immediately handle the traffic without any disturbance. This is really good because the fail-over is transparent from user perspective as there is nothing wrong happening in the system. If the faulty server back online then it automatically synchronized the game state and ready to be the new backup.
However, for each successful user action, the master server needs to send the update of game state to the backup server. Despite only sending the update and not the entire field, this mechanism require a lot of message to be sent. Another thing is that the backup server has low utilization because it only gets the current game state and wait for the fail-over to happen. One solution to achieve better utilization is by using active-active mechanism instead of active-standby. However, this also need to be followed by different strategy of consistency & fault-tolerance and also need to be tested whether it provides better overall performance or not.
We use multicast for group management and also redundant action dispatch to ensure fault-tolerance aspect of the workers. Game servers know which game worker that is ready to be used by maintaining multicast communication among the group. This way, if the worker is not available because of failure, there will be no event that is sent to it. Also, with this method the system still could handle the game even with only one remaining worker but with the cost of increasing workload to that particular worker. Although the experiment was successful, there could be the case when at some point of time the workload really high because of the random nature of waiting time of users before action and also random progress of the game. This kind of scenario should be reproduced and analyze in the future if the system want to handle more and more player base.
Moreover, we also incorporate redundant event request to two workers so that if one worker is failure when the request is being set, the request can still be replied by another worker. Of course it could still be a problem if two of the chosen workers have failure during the transmission of the request, but this mechanism still minimize error for a single worker transmission failure.

## 6 Conclusion

Using the design that we have proposed, we have successfully achieve Dragon Arena game that runs on distributed system. Moreover, based on our simulation and testing we have ensure that our distributed system also has capability to perform on massive amount of users, maintain consistency of game state,

good availability of service, scalable, and provide fault-tolerance. Overall, we recommend that WantGame BV should use the distributed system as presented on this project because it meets all the criteria and also have a room for improvement in the future.
On top of that, based on our experience on implementing Distributed Dragon Arena System, we have learn a lot regarding difficulties and challenges on developing distribution system in general. We found out that, solid and detail system design in the early phase of the project is very essential. We also learn that debugging a distributed system is very challenging and an extensive logging could overcome this problem.

# 7 Appendix

## 7.1 Opensource Code and Public Code

Our implementation of Dragon Arena in a distributed system is available as open-source code and public report on our Github repository: `https://github.com/arkka/in4391`.

## 7.2 Time Sheets

### 7.2.1 Achmadnoer Sukma Wicaksana

Table 1: Sukma time sheets

| Type | Hours |
|---|---|
| Total-Time | 73 |
| Think-time | 7 |
| Dev-time | 26 |
| xp-time | 12 |
| Analysis-time | 12 |
| Write-time | 16 |
| Wasted-time | 0 |

### 7.2.2 Arkka Dhiratara

Table 2: Arkka time sheets

| Type | Hours |
|---|---|
| Total-Time | 75 |
| Think-time | 7 |
| Dev-time | 28 |
| xp-time | 13 |
| Analysis-time | 12 |
| Write-time | 15 |
| Wasted-time | 0 |

# References

[1] E. Cronin, et al, "An Efficient Synchronization Mechanism for Mirrored Game Architectures", 2002.

[2] R. Fujimoto, "Parallel and distributed simulation systems", John Wiley & Sons, 2000.

[3] S. Shen, O. Visser, A. Iosup, "RTSenv: An experimental environment for realtime strategy games". NETGAMES 2011: 1-6.

[4] Yong Guo, Alexandru Iosup: The Game Trace Archive. NetGames 2012: 1-6. [Online] Available: http://dx.doi.org/10.1109/NetGames.2012.6404027 and http://www.pds.ewi.tudelft.nl/~iosup/game-trace-archive12netgames.pdf