IN4391 – Distributed Computing Systems

Delft University of Technology

Instructor: Jan S. Rellermeyer

Teaching Assistants:

Laurens Versluis

Sacheendra Talluri

The Dragons Arena System: Distributed Simulation of Virtual Worlds

Group 2

Bruno Mateus (4710738, B.A.FrazaoMateus@student.tudelft.nl)

Michail Vrachasotakis (4749189, M.Vrachasotakis@student.tudelft.nl)

April 4, 2018

1 Abstract

Nowadays, online video games are famous and present in our lives, and there is a lot of competition in the market. One important aspect is the optimization and availability of the system and in order to achieve this, it is necessary to have a robust distributed system. Our goal is to implement a distributed game, the Dragon Arena System (DAS), that can handle several concurrent clients, the Players, fighting in an arena against non-player units, the Dragons. The gamestate of the clients and servers needs to be consistent and the system is required to be scalable and fault-tolerant, in order to handle crashes and to have consensus about the validity of actions. In this work, all these are achieved using simple, intuitive algorithms found in the related literature. By observing the logs of the machines and simulating faults, we show that this system, indeed can handle crashes, achieve consensus and provide the same gamestate for all non-faulty machines. The source code has been made open-source and can be found on `https://github.com/baldman005/DistributedComputing_LabExercise`.

2 Introduction

In order to implement a game system that handles large amounts of players, given the requirements of different games genres, it is necessary to have a system that ensures consistency between the machines and that has tolerance for faults such as crashes[4] or byzantine behaviour[16].

Some of the current architectures have only a central server[2] or a peer-to-peer[9] architecture. Although the centralized approach is simpler, it cannot be fault-tolerant due to a single point of failure, which is not sufficient.

Our system is implemented with Java RMI[7] and consists of a main server, mirror servers and the clients, that communicate to ensure the requirements.

The remainder of the report will include a description of the system application and its requirements in section 3 ; section 4 covers the algorithms, protocols and components of the system; in section 5 a description of the experiments and their results; the discussion of the results will be included in section 6 and in the end, section 7 will provide a conclusion of all our findings.

3 Background

The DAS application consists of a game where each client controls a unit, a Player, that can move (1 step vertically or horizontally) in a 25x25 square grid battlefield. They work together to try to kill all the dragons, which are non-playable units that cannot move and are only able to attack. Each unit has a certain amount of HP, max HP and AP (HP means Health Point and AP means Attack Points) and whenever they attack a unit, that unit loses HP equal to the AP of the attacker. The Players can also heal other players, with amount equal to the healer's AP, without exceeding the threshold of max HP. Finally, the game ends when a faction has won or there has been a timeout.

In order to provide this functionality for multiple servers and clients, each unit must process the same information about the battlefield, reaching eventual consistency[6]. Therefore, a protocol which supports communication among different entities and rollback[10] in case of inconsistent states is required.

Another requirement is that the system should be able to handle players and servers in large quantities playing in the same session (same battlefield). To accomplish such a scalable system, there should not be redundant actions in the system, increasing its load unnecessarily.

In case any of the servers and clients fail, such as a crash, the game should be able to continue for all the rest. This requires the implementation of proper fault-tolerance protocols that take each of the possible failure scenarios into account.

Also, it should be fast so that all clients can have the feeling of real-time gaming and also not notice errors from rollbacks or failures. These requirements depend on the game, but for this a rollback of half

a second is acceptable and if the main server fails, it is also acceptable to wait a few seconds (around 5s).

4 System Design

The system includes a main server, the host, other servers that are mirrors of the host and the clients. Initially, the servers are created, each having an id from 0 to the number of servers - 1, and the host is defined (0 by default). Then all the clients are created, having ids larger or equal to the number of servers. Each of the clients and servers use an object, called Machine, to handle the consistency and rollbacks as well as to save most of the variables related to the servers/clients.

4.1 Some important variables

Some of the most relevant and important variables are the battlefield and the list of units. The battlefield class holds a double array of units, the map, that has the units in their position (the ones free are null). The list of units contains the same units, but it is used to provide access to actions that require all the units (such as finding if a faction has won), more quickly. The machine also has the Servers_URLs array, so that they can communicate with the servers. The order of this array is also equal for all the machines. Also, whenever a new client is registered, its URL is saved on Clients_URLs.

4.2 Consistency

Whenever an action is taken, a command is processed. These commands can be: spawn, remove, attack, heal, move, gameStart and falseStart, the last one only used for rollback.

Each machine has a buffer (steps) that saves the steps that it receives and processes them. These steps contain a normal and an inverse command, so that one can easily do a rollback and also include the timestamp associated with it, that is based on Lamport clocks[15]. Whenever a step is added, it first finds its correct location in the buffer, which happens when a step with a timestamp that is lower is found, by comparing its timestamp and id with the previous ones. If the timestamp is equal, the origin id (saved in the normal command) serves as a tiebreaker, where the lower the id, the higher the priority. When it reaches a position and it finds an inconsistency, it does the inverse of the commands previously processed, to restore the last known consistent game state. Then, it continues normally processing the normal commands.

There are some special details in the protocol. If a unit dies, it is almost certain that it received more damage than the HP it had initially (overkill) or when a unit is healed, it is possible that its final HP is equal to the max HP, but the AP of the healer was higher (the threshold of the max HP was exceeded). Therefore, in the inverse command it is necessary to take into account the difference between the "apparent" AP of the attacker and the real one.

Another detail is the remove command. Whenever a machine updates a unit's HP to 0, it adds directly to the steps buffer the remove command, right after the attack command that led to the kill. Additionally, when a rollback happens and it goes through a normal remove command, the step associated is removed from the buffer, because this command may not be valid anymore.

4.3 Fault tolerance

To simulate a crash fault every machine has a flag "crash". When it is on, the machine does not process the steps buffer and neither receives nor sends messages.

To handle the crashes, the PBFT[12] 2-phase version is implemented. For this, a machine has an additional buffer, "val_buffer", that contains a new object called Commit. This object contains different counters for valid commits, aborts and checking if there has been a timeout and a step that is

has received or sent. In order to make a decision, it is necessary to do a majority between the decisions of the servers, for which a number of servers greater than $3f + 1$ is needed, where f is the number of faulty servers.

Whenever a client wants to perform an action, it first sends a request to the host, waits for $f + 1$ replies from the servers, and computes the majority. If the previous condition is satisfied, then the majority is equal to the decision of the replicas. If the decision is abort, then it does not add the step to the steps buffer while if it is commit, the step is added to the buffer.

When the host receives a request, it adds the step to the val_buffer, checks if it is valid (by incrementing the corresponding counter), sends prepares to the replicas and commits (with the corresponding decision). When a replica receives a new step, it adds it to the val_buffer and if it receives a prepare, it checks the validity before sending to the other servers a commit with that value. Whenever a server (either replica or main) knows $2f + 1$ validity values from the servers, it can make a decision with the majority and send the reply to the client.

If a replica crashes (and the bound of f is not exceeded) the execution of the game continues with the previous protocol. However, if the main server crashes, the client can find out if it has not received replies and therefore notices a timeout. In this case, it changes the host to the next one ($new\_host\_id = host\_id+1 \mod serversN$) and sends an election to the replicas. Upon the replicas receiving the election, they also change the host to the next one and notify the clients. If a machine is the new host, then it starts the threads to process the dragon units.

Also, when the host changes, the gamestate is equal between the different machines, given consistency is already ensured.

4.4 Additional System Feature - Byzantine Behaviour and cheat commands

It is pretty straightforward to ensure that invalid steps are not committed in the non-faulty machines. Using the 2-phase protocol, it is only necessary to make sure that there are, at most, f byzantine servers, and to decide how does a server check the validity of a command. Whenever there is an invalid command (e.g. dragons are healed or heal; players move more than 1 block) a server decides that that command is invalid, and chooses the abort option. Given that the consensus is already implemented, these commands are not committed in the non-faulty machines.

5 Experimental Results

5.1 Experimental Setup

The system was analyzed by running the game in the separate computers of each group member, although the option of Amazon EC2[1] was considered and tested briefly. The workload consists of an incremental set of servers and clients in order to first test the consistency and fault-tolerance of the game and then its scalability. The main monitoring tool used is VisualVM[11]. VisualVM is essential for a more in depth analysis of memory usage and the load the system imposes on machines. Eclipse[5] allows a glimpse into the thread creation and garbage collection or termination of a program through its debug interface, however it is unreliable.

To check if the execution is exactly the same, every client and server writes the main steps of their execution on the terminal and on a file associated with them. These main steps are the processing of a command (either normal or inverse), the start and end of the game (where it is specified which faction won or if there has been a timeout) and the start and end of a rollback.

Additional facts are that when sending a message, a thread is created to simulate a delay and when simulating crashes, the Ironforge 1d data from the GTA-12 dataset[14] are used to simulate the behavior of real players.

5.2 Experiments

5.2.1 Consistency

First, consistency was checked when there are no faulty nodes. To perform this experiment, only one server was used and the dragons to players ratio is 0.2. So, the test configurations are 1 dragon and 5 players(dubbed C1), 3 dragons and 15 players(dubbed C2) and 5 dragons and 25 players(dubbed C3).

It was observed that if the time it takes for the machines to process steps is compared to the time between actions of the units, (by low is one order of magnitude lower), then the results and steps buffers are equal between machines, as intended.

Test Configuration	Runtime(s)	Threads	CPU(%)	Memory(MB)	# Loaded Classes
C1	25	53	42	75	2512
C2	40	300	47	218	2513
C3	stuck	426	36	261	2530

Table 1: Service and Usage Metrics for Consistency tests

Table 1 contains service and usage metrics for scenarios C1, C2 and C3. The main service metric is the duration of the game measured in seconds, with stuck meaning that the game seems paused(no further actions) and terminates due to a timeout. Usage metrics are comprised of the maximum number of live threads created by the system, the average load of the CPU in % percentage, the maximum amount of memory allocated by the game and the number of loaded classes.

5.2.2 Fault-tolerance

To test the fault-tolerance, the number of faulty servers considered is $f = 1$, the number of servers equal to 3 and the same number of dragons and players as before. As expected, given that the necessary condition mentioned before was not satisfied, whenever a server failed the others could never have the sufficient number of responses to commit or abort the steps. This leads to the game simply not running and a timeout happening.

However, by using a number of servers equal to 4, the condition is satisfied and therefore, the game runs. When only one replica fails, the game continues smoothly, since the clients only send the request to the host. But, if two or more servers crash at the same time, f should be 2 which causes the game to hang as the condition is not satisfied anymore.

When a host crashes, the game stops while the clients are waiting for the timeout to happen. When it does, the replicas change the host id(as mentioned before), warn the clients and the game continues to run as normal.

If a client dies, the game simply continues (with their unit on the battlefields idle and not taking actions) and it finishes normally (either the players win or dragons kill the idle unit) or a timeout happens (dragons killed the remaining clients but could not kill the idle player because it was far from all dragons).

There were some issues related with restarting a machine, that will be discussed in the Discussion section.

5.2.3 Scalability

To test the scalability component dragons and players were added to the previous experiments, but still maintaining the 1:5 ratio. After around 30 players (and 6 dragons), the program would get slow or stuck, and exception of OutOfMemory[8] would appear. This will be discussed in the Discussion section. The game can scale up to the requested target which is 100 players, 20 dragons for 1 server, named S2. We provide the usage metrics for 50 players, 10 dragons and 1 server which will be referred

to as S1; S3 is the case with 5 players, 1 dragon and 3 servers and S4 5 players, 1 dragon and 5 servers. Table 2 showcases the different behavior of the system for these cases of scalability.

Test Configuration	Runtime(s)	Threads	CPU(%)	Memory(MB)	# Loaded Classes
S1	stuck	2000	53	261	2513
S2	stuck	4138	56	350	2583
S3	29	149	39	143	2527
S4	26	270	32	176	2509

Table 2: Service and Usage Metrics for Scalability tests

5.2.4 Performance

It was observed that if there is only one server, the removal of delay threads increases the performance, since it is more scalable and quicker when there many units. However, with more than one server, the program would not start and get stuck, which will be discussed later.

Also, with the increase of clients the occurrence of rollbacks would increase too, as it is expected. This aspect will also be discussed in the Discussion section.

Examples of the change in performance when no delays are issued on messages are presented on Table 3.

Test Configuration	Runtime(s)
C1	16
C2	30
C3	stuck

Table 3: Service Metrics for Performance tests

6 Discussion

It is observed that the consistency and consensus work as intended, although there are some issues as verified, mostly related with the scalability.

Currently, there is also a bug that does not allow for a machine that crashed to register again. This is due to the fact that the host needs to send a list with the units in the battlefield. However, an exception ConcurrentModification[3] appears when trying to do it.

Furthermore, to test for a greater number of units it is necessary to use several virtual machines, for example using platforms as the Amazon EC2. Another aspect that can make it more efficient is to not write the message outputs to so many files and to the console and also to remove the delay threads, since their number grows exponentially with the number entities used by the system.

Another issue is that, rollbacks also increase when there are more players. There could be an improvement, by only forcing close events (events that happen in a near zone in the battlefield called Region of Interest) to be exactly consistent, since far away events are independent to a certain degree (e.g. two battles happening in opposite sides of the battlefield are independent).

Moreover, it is observed that the main server has a high workload, since it processes all the dragon units. To improve this, some dragons could be processed by the other servers.

Ad From the metrics, we can conclude that the memory that the system needs does not increase to a level that is unacceptable for large workloads and the same result holds for the CPU utilization. On the other hand, the number of threads that each scenario spawns seems to have a steady increasing trend and may lead to problems. The number of loaded classes is just a sanity check to ensure that

when changing the game settings, there are some classes that are not used. Finally, the results indicate that for larger workloads than the ones used for testing here, the gaming experience will be unpleasant. Therefore, extensions upon this game require redesign of various parts of its architecture.

7 Conclusion

The final product, based on a main server, replica servers and clients communicating, is able to support a moderate number of players and a small number of servers. The requirements for consistency and fault-tolerance are satisfied but the current system is only adequate for a small-scale online game, not a massively multiplayer online one yet. It is evident that straightforward algorithms derived from the standard literature of distributed systems are unhelpful for such a task. The solution is to devise protocols that are specialized for the game system that needs to be implemented and are optimized for a certain combination of consistency, performance and availability[13].

References

[1] Amazon EC2. `https://aws.amazon.com/ec2/`.

[2] Centralized Client-Server Model. `https://en.wikipedia.org/wiki/Client-server\_model\#Centralized\_computing`.

[3] Concurrent Modification Exception. `https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html`.

[4] Crash Error. `https://en.wikipedia.org/wiki/Crash\_(computing)`.

[5] Eclipse. `https://www.eclipse.org/`.

[6] Eventual Consistency. `https://en.wikipedia.org/wiki/Eventual\_consistency`.

[7] Java Remote Method Invocation. `https://en.wikipedia.org/wiki/Java\_remote\_method\_invocation`.

[8] Out Of Memory Exception. `https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html`.

[9] Peer-to-peer computing. `https://en.wikipedia.org/wiki/Peer-to-peer`.

[10] Rollback. `https://en.wikipedia.org/wiki/Rollback\_(data\_management)`.

[11] VisualVM. `https://visualvm.github.io/`.

[12] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[13] Gilbert, S., and Lynch, N. Perspectives on the cap theorem. *Computer 45*, 2 (Feb. 2012), 30–36.

[14] Guo, Y., and Iosup, A. The game trace archive. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games* (Piscataway, NJ, USA, 2012), NetGames '12, IEEE Press, pp. 4:1–4:6.

[15] Lamport, L. Time, clocks and the ordering of events in a distributed system. 558–565.

[16] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4/3* (July 1982), 382–401.

A Time Sheets

This table shows the time spent on this project.

Total	Think	Dev	Xp	Analysis	Write	Wasted
127	10	95	9	2	5	6

Table 4: Time Sheets