Final Report

Marauder's Map, CS 5412

Haotian Pan(hp343), Yize Li(yl2376), Hang Chu(hc772)

1. Introduction

1.1 Motivation

To provide a distributed gaming platform for Zombie City and other similar online webpage games for any portable devices:
a) Simple framework on web page for real-time interactive multiplayer games.
b) Service guarantees for availability, fault-tolerance, speed.

1.2 Main content

1.2.1 The cloud framework

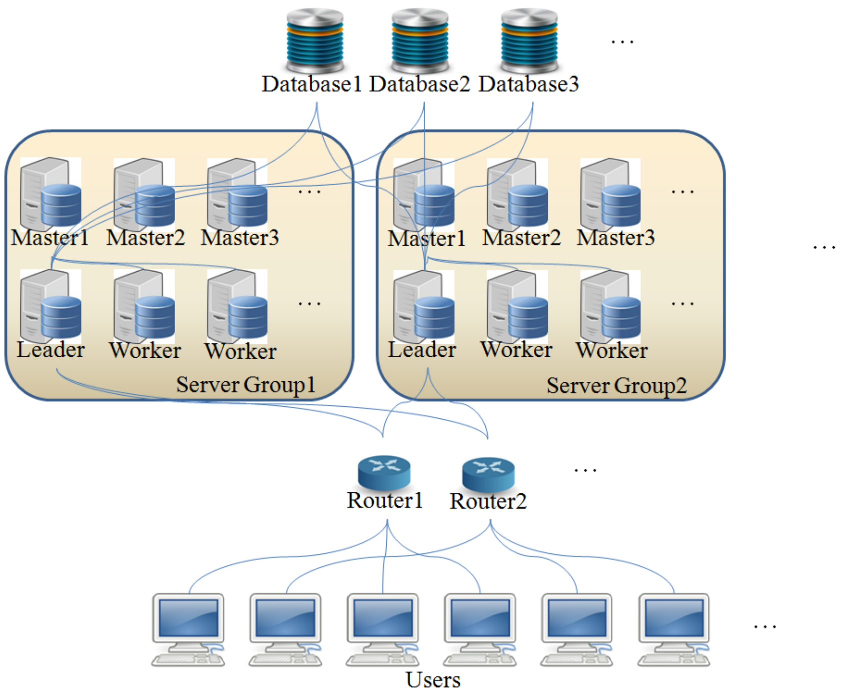


Figure 1. The framework of our cloud game hosting system

Figure 1 shows an overview of our cloud framework. From Figure 1 it can be seen that users can enter our cloud system through routers. Our system supports more than one routers, so that even one of them fails the game will be still functioning if the user visit other routers. In an more practical situation, a DNS-alike structure can be used to wrap all routers into one unified domain name, but this is already beyond the concern of the cloud system itself so we did not implement it.

The next level of our system are server groups, our routers will automatically balance the load and assign users to different groups. Each group contains three types of servers: master server, leader server, and worker server. The master server oversees the whole group, we have more than one master servers in one group such that even the current working master fails, other sleeping masters will automatically take its place, guaranteeing the consistency of the game. The leader server is the server that hosts the game. The worker servers will remain sleeping until a leader fails and receives an instruction from the current master to be the new leader. Our group structure guarantees that when single server (of any type) failure occurs, the game still continues as usual.

The third level of our system are databases. The leader server will report its own potion of user data to the databases. When a leader fails and a worker becomes the new leader, the worker will first retrieve the part of user data it needs to run the game from the database. To guarantee the reliability of our cloud system, we also implemented multiple databases such that when one of them fails, the system still functions normally.

1.2.2 A simple game for testing

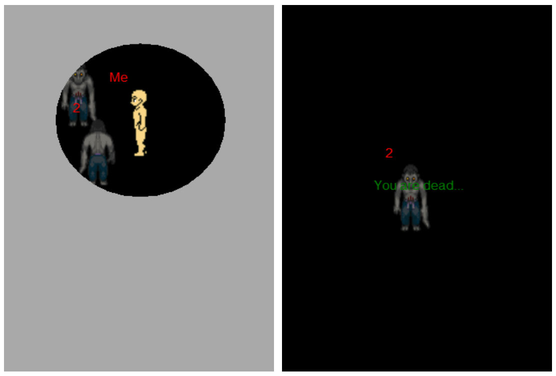


Figure 2. Screen captures of the game we use for testing our cloud framework.

Figure 2 shows screen captures of the running game. We divide all players into two types: zombie and human. Players can move around and kill other players that belongs to the other fraction.

It should be noticed that although our game design is fairly simple, but it’s enough to include all types of basic gaming requests (spawning player, loading map, receiving and updating player data). We set our focus on the cloud framework, rather than waste too much time on implementing a game that just looks fancier but doesn’t change much essentially.

1.3 Final status

As a high-level summary of our project, we have designed and implemented a cloud architecture, which has good scalability and robustness. Our cloud is capable of handling failures of any possible single server, and can even handles certain failures of more than one server. We also implemented a simple but playable game, to test and demonstrate our cloud framework.

We have achieved all the goals that we have proposed previously.

2. Approach

In this section, we will describe our project implementation in detail. We will focus on discussing our approach of realizing our own cloud system, and also introduce the techniques we used for our example testing game.

2.1 A cloud system with high reliability and fault-tolerance

Figure 1 shows the framework of our cloud system. There are several important issues we want to introduce about our cloud system. In the rest of this subsection, we will first explain the key design factor in our system: the master-leader-worker server group strategy. Then we introduce our design of assigning a new leader from workers, and the master replication strategy, which renders our server group robustness towards failure of any type of single node. After that, we will also describe our approach for implement the databases and routers, which are also robust to single node failures.

2.1.1 The master-leader-worker server group

The simplest way to host a game, is to use just only one server and let it handle everything. Obviously, this is not a good idea if we want our game hosting system stable and scalable. Inspired by Azure and Zookeeper, we decided to implement a master of the group: a server that oversees the group. Except the master, there are two other types of servers: the server that is actually spinning and processing game data, which we call the leader; the servers that are sleeping and are standing by to take the leader's position once the leader fails, which we call the workers. The master coordinates the group by constantly check the leader, once it detects the leader is no longer working, it immediately promotes a available worker to be the new leader. The promoted worker will do two things after it receive the master's order: first it checks which users are connected to the previous leader, second it retrieves data of those detected users from the database and start hosting. When a leader hosts a game, it receives and broadcast user status change to all connected users, and in the same time save the changes to the database. Furthermore, a group can have more than one master. Only one master will be in charge and the others will remain sleeping but keep an eye on the working one, such that one of them can take up the place once the current master fails.

In our framework, there are no upper limits for: 1. number of workers one group can have, 2. number of backup masters one group can have, and 3. number of groups the entire system can have. Those are all signs of high scalability of our cloud system. There is a unique leader among all the workers in a group. This is because for an application like our online instant battling game, users submit operations and receive data in high frequency. Thus makes it not a wise choice to spread users to different workers, as the communication and data passing among workers would be fairly heavy and time consuming.

2.1.2 Leader election from workers

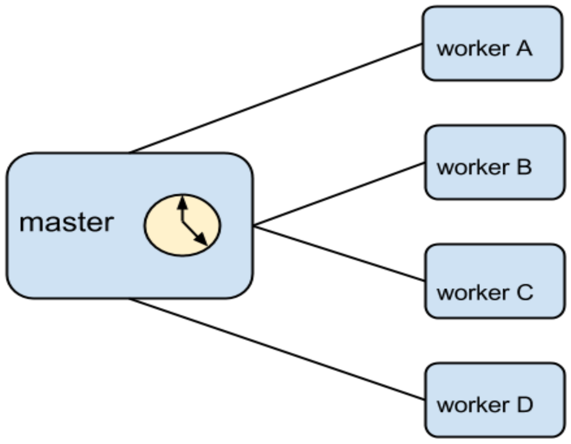


Figure 3. Diagram of promoting a worker to be the new leader

We now introduce the strategy we use to elect a new leader from workers. There are already some existing solutions to this problem (with different presumptions), such as the Chang-Roberts algorithm[1], the Franklin's algorithm[2], the Dolev-Klawe-Rodeh algorithm[3] for ring election, and the tree election algorithm[4]. However, what we want for our system, is a simple, robust, and efficient solution for election. Thus we let another process, the master, to hold a global logical lock and choose the worker from the queue that has the highest rank to be the new leader. The logical locks is a number that increases when a worker connects to it or upon worker disconnection/failure..

2.1.3 Master replication

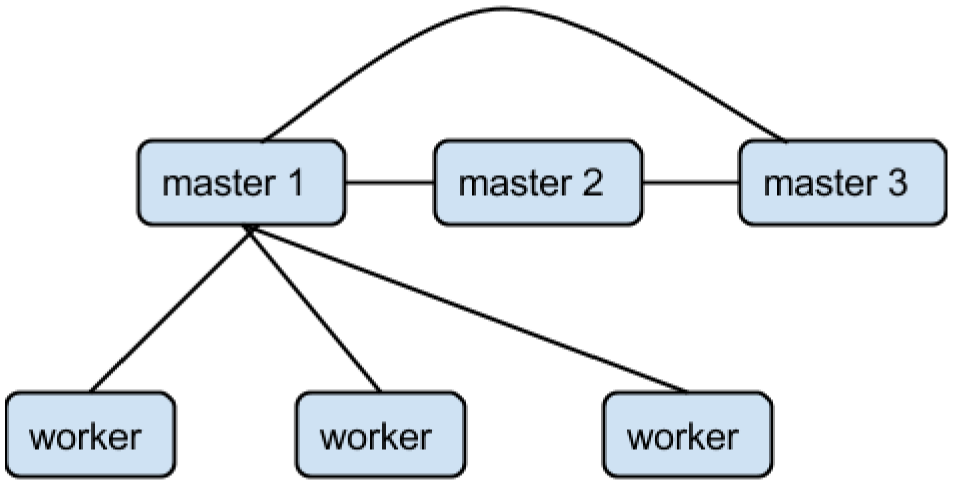


Figure 4. Diagram of master backup strategy

With the above implementations, our server group is able to handle single point failure of leader and worker. To further extend the group's robustness and make it able to continue serving under single point failure of the master, we use a triple-master framework. We set up three master servers when the group initializes, before any worker starts. The three masters follow the following protocols:

[1] Ernest Chang; Rosemary Roberts (1979), "An improved algorithm for decentralized extrema-finding in circular configurations of processes", Communications of the ACM (ACM) 22 (5): 281–283, doi:10.1145/359104.359108

[2] Franklin R. On an improved algorithm for decentralized extrema finding in circular configurations of processors[J]. Communications of the ACM, 1982, 25(5): 336-337.

[3] Dolev D, Klawe M, Rodeh M. An< i> O</i>(< i> n</i> log< i> n</i>) unidirectional distributed algorithm for extrema finding in a circle[J]. Journal of Algorithms, 1982, 3(3): 245-260.

[4] R. G. Gallager, P. A. Humblet, and P. M. Spira (January 1983). "A Distributed Algorithm for Minimum-Weight Spanning Trees". ACM Transactions on Programming Languages and Systems 5 (1): 66–77. doi:10.1145/357195.357200

1. Master1 has the highest rank, then master2's, and master3 has the lowest rank. All of them know others' ranks.

2. Master1 will turn itself the actually working master when it detects master2 or master3 connects to it successfully. Master2 and master3 will remain sleeping during master1 works.

3. Master2 will turn itself the working master server when it losts connection with master1 and receives a message from master3 saying that master1 is lost.

4. A working master quit itself when it finds itself being the only left master.

5. The working master is responsible for providing the address of the leader server to clients.

2.1.4 Database replication

We choose MongoDB 2.7.6 to be our ultimate database to store player's information on disks, which includes player's name, location, orientation, live/dead flag and the total distance that player has walked. We send these data to three different databases located in distinct areas every 20ms (the update rate can be adjusted according to the quality of the network). Basically, setting update rate contains a trade-off between more recent game records and less network congestion. Since we are sending the data very rapidly, we do not consider consistency of the game records in those databases at one certain time as long as most of the time the databases will get newer consistent data very soon. Note that the storage can also be extended by MongoDB itself, since MongoDB already provides data replication scheme.

When something bad happens, e.g. a worker goes down and a player want to retrieve from the game, the system will try to retrieve players' data from these three MongoDBs. Again, due to the very fast recording of game data, we assume MongoDBs available at the time of retrieval should include approximately(most of the time exactly) the same data for a certain player. This simplifies the consistency problem and guarantees a faster response and a better feeling to players. In our scheme, we first try to retrieve data from one MongoDB, if this fails, we then try to reach our second MongoDB, and goes on until the last MongoDB. Though this retrieving strategy of ours seems quite intuitive, it works quite well. Also we want to stress that a more sophisticated retrieving strategy can be easily added to our current one. Finally the player will be dropped out only when all MongoDBs fail. In most of the cases, our first attempt will succeed, thus making the game immediately able to continue.

2.1.5 Router replication

Now we have introduced all replication/backup strategies except for the router. As a matter of fact, our router is self replicable due to our design of the framework. This is because in our framework, the router has only two functions: 1. balancing the overall load by redirecting newly connected users to different server groups, 2. send the html and static script files used in the page to the users' web browsers. Thus the router does not handle actual game data stream and everything it involves are static. That means in our framework, routers can be replicated by simply copying the related script files. Different servers have different ip addresses, they can be unified into a certain domain name by any DNS protocol. We consider DNS as another issue and beyond the scope of our cloud system, so it is not implemented in our current version.

2.1 A webpage-based instant battling game

As we have mentioned in Section 1, we did not put too much energy on making the game look fancy. We only seek a minimal, however, general game example that can be used for testing, evaluating, and demonstrating our cloud system. It should also be noted that though our game does not have the best visual effect, the amount of work needed to implement such a minimal game from scratch is still considerable.

To guarantee our game's availability in different platforms (Windows pc, Linux, Android, ios, etc.), we choose to implement our game in a webpage (html+javascript). Thus any device that has a html5-available browser installed is able to play our game.

The game webpage has two functionalities: 1. detect the users action, and send this action to the cloud system (i.e. the leader server of the the group the router has chosen for this user). 2. receive new game status and paint the game according to the received status. We implemented this in a script file and it will be loaded automatically to the user's browser when the user visits the html file.

In the current version of our game, we divide all users into two types randomly: zombie and human. All the users are alive initially, they can move around and only have the adjacent area visible, the rest of the map will be covered by a "war shade". Users can also attack by pressing the spacebar, the visual effect of attacking is a yellow rectangular (it might look ugly, but the point is it works!). If there is any enemy players in the attacked region, then she will be marked as killed. We also added another attribute for every players: the total walked distance. The total walked distance is not used in the game, but we implement it as an example of permanent user data, so if anyone wants to extend the game, making users having levels, experiences, items, armors, or weapons, they can follow our implementation of walked distance.

3. Project Progress

3.1 Milestones

3.1.1 Milestone 1

a) Create the game map

According to our game model, the map is 2D. It can be either manually crafted, or can be retrieved from real-world map such as Google Maps.

-We seeked advice from our TA, as he said: ”focus on the cloud, the game doesn’t matter”, so we use a manually crafted map.

-Completed in Intermediate Report I (IR1).

b) Create game GUI

We will implement a webpage-based GUI for gamers, which includes: a map display with real-time player status, buttons for taking actions such as move or attack (we will also try locating by GPS to substitute manual moving control).

-Same as the previous one, the cloud is the point, game is only for testing. Thus we don’t include the GPS feature (though this can be easily added).

-Completed in IR1.

c) Traditional single server-client framework

We will build a simple gaming framework, using only one server. Every user is connected to this server.

-Completed in IR1.

d) Testing maximum response speed

We will test the real-time performance of the simple framework.

-Completed in IR1.

3.1.2 Milestone 2

a) Improve communication efficiency

In the previous stage we only test response speed, in this stage we will improve communication efficiency to meet the 100 ms requirement.

-Completed in IR1.

b) Multi server support

Only one server cannot be called a cloud. We will implement strategies to enable our framework working at multiple servers. Different game instances will be hosted on different servers.

-Completed in Final Report.

c) Load balancing

We dont want to exhaust one server while other servers are barely used, so we will design and implement strategies for assigning servers, such that the work will be distributed to servers averagely.

-We use the router to balance the load globally, as described in Section 2.

-Completed in Final Report.

3.1.3 Milestone 3

a) Error tolerance (replicating servers)

In this case, we will duplicate the data of each server to two other redundant servers. Changes in any one of the servers would cause the other two to synchronize. This ensures that clients would not receive bad data after one server crashed.

-We tried multiple strategies for this. We first tried just storing the data in the user, and a new server retrieving the data from all users, this will arise the problem of game cheating so we gave it up. We then tried shared json file, memcached, and MongoDB, and finally set up to use MongoDB for its stability.

-Completed in IR2.

b) Clients should witness changes in a consistent way

For this part, we may actually implement with the isis tool. We may come

across some protocols to guarantee consistency between clients.

-The above paragraph is from our initial proposal. We do have considered using Isis2 for realizing this, and we admit that Isis2 is a very powerful and robust solution. However, we think it would be more interesting if we implemented our own protocols. Certainly, our protocol won't be as powerful as Isis2, but at least it works and it is able to provide the features we want. We gained a lot of valuable experiences from the process.

-Completed in IR2.

c) Maintain 100 ms delay limit

Assume that we have made several changes from previous milestones, in order to meet the requirements, we still want to check and maintain a 100 ms delay limit.

-Completed in IR2.

3.1.4 Milestone 4

a) Recovery protocol for failed server replicas

We will ensure that the recovered replicas get all the information that the current

operational replicas have to keep consistency.

-We use a MongoDB-based database and the worker election strategy to realize this. Detailed descriptions can be found in Section 2.

-Completed in Final Report.

b) Improve efficiency

We then will try to decrease the delay in response as possible as we can.

-This can be done by adjusting the updating rate of the leader servers to the database.

-Completed in Final Report.

c) Adjusting gaming model

Making the game funnier and more vivid to play with.

-Completed in IR2.

3.2 Other attempts

3.2.1 JSON files

We had been using JSON files to store players' data. Basically, we transformed every player's information into a json object and then write it to the disk frequently. When retrieving game, we read the json file and transform it back into an object containing the player's location, orientation, live/dead flags….This finally turned out to be a naive way to store data. The main problem of this is that we could not write robust code by ourselves to deal with large amount of queries, modifications and insertions to json files. Later on, we decided to use MongoDB instead.

3.2.2 Memcached

We also explored using memcached for data replication. We expected memcached to be able to share and replicate data automatically among machines, but this turned out to be not true and what memcached can provide is just a distributed hash table. There was a replicating-enabled version of memcached called repcached, but the project had been closed. After all these attempts, we decided to use MongoDB.

4.Demo Plan

4.1 Game interface and basic logic

We would like to show on the final presentation that how our game looks like and how could players play the game.

4.2 Fault tolerance for leaders/workers

We would like to show on the final presentation that our system will not be affected when the leader or a worker goes down.

4.3 Fault tolerance for masters

We would like to show on the final presentation that our system will not be affected when a master goes down.

4.4 Fault tolerance for databases

We would like to show on the final presentation that our system will not be affected when a database crashes.

4.5 Fault tolerance for routers

We would like to show on the final presentation that our system will not be affected when a router goes down.

5.Poster

Cloud System for Marauder's Map
CS5412 Course Project, Spring 2014

Yize Li (yl2376@cornell.edu) , Haotian Pan (hp343@cornell.edu) , Hang Chu (hc772@cornell.edu)

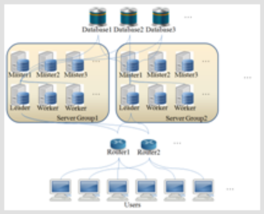
Motivation

We want to provide a cloud gaming platform for Zombie City and other similar online webpage games for any portable devices:

a) Simple framework on web page for real-time interactive multiplayer games.

b) A cloud game hosting system that guarantees availability, fault-tolerance, and speed.

The cloud system



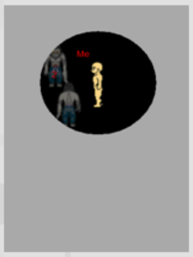
Users can enter our cloud system through routers. Our system supports more than one routers.

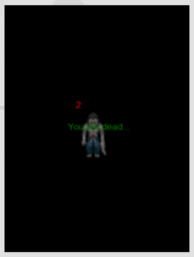
The basic unit of our system are server groups. Each group contains three types of servers: master server, leader server, and worker server. The master server oversees the whole group. The leader server is the server that hosts the game. The worker servers will remain sleeping until a leader fails and receives an instruction from the current master to be the new leader.

We implemented a multiple database.

With our designed cloud system, the game hosting service won't be affected under any type of single point failure.

A simple game for testing



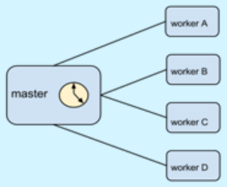


We divide all players into two types: zombie and human.

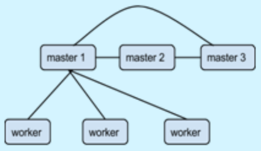
Players can move around and kill other players that belongs to the other fraction.

The game is minimal but it includes all factors a more advanced game should have. New gaming features can be easily added.

Approach



The master holds a global logical lock and choose the worker from the queue that has the highest rank to be the new leader.



1. Master1,2,3 have ranks 1,2,3.
2. Master1 will turn itself the actually working master when it detects master2 or master3 connects to it successfully.
3. Master2 will turn itself the working master server when it lost connection with master1 and receives a message from master3 saying that master1 is lost.
4. Aworking master quit itself when it finds itself being the only left master.
5. The working master is responsible for providing the address of the leader server to clients.

mongoDB mongoose

We use tripple MongoDBs to guarantee the reliability of the database level.

In the retrieving phase, we try retrieving from MongoDBs one by one until succeed. This guarantees short response time.

Our framework also supports multiple servers. The router has only two functions: 1. balancing the overall load by redirecting newly connected users to different server groups, 2. send the html and static script files used in the page to the users' web browsers. The router does not handle actual game data stream and everything it involves are static. Routers can be replicated by simply copying the related script files. Different servers have different ip addresses.

Reference

[1] nodejs.org/
[2] S. Fulton et al., HTML5 Canvas
[3] memcached.org/
[4] mongoosejs.com/
[5] L. Lamport, Paxos made simple
[6] docs.google.com/

Contact information

Yize Li, yl2376@cornell.edu
Haotian Pan, hp343@cornell.edu
Hang Chu, hc772@cornell.edu

Acknowledgements

We would like to thank Prof. Kenneth Birman and our TAs Stavros Nikolaou, Theodoros Gkountouvas for their excellent lectures and helpful suggestions