COMP90015 Project 2
High Availability and Eventual Consistency

Thu Thao Le (`thaol4@student.unimelb.edu.au`)

Yicong Li (`yicongl2@student.unimelb.edu.au`)

1 Introduction

The goal of this project is to implement new server protocol to address the server failure issues. The server can provide availability and eventual consistency among the servers and clients when network partitioning happens.

There are many challenges in this project. The hardest part is to rebuild the server protocol to handle the failures in the network. We have used tree protocol with some improvements. Moreover, handling the register request as well as the activity messages is a major problem.

2 Server failure

According to CAP theorem [1], if there are failures in the servers and the connections, we can only achieve either consistency or availability. If we choose availability, we can always return the messages even it is not up-to-date data. If we choose consistency, we have to update the latest data before returning the messages to clients.

There are three kinds of failures can happen in the system:

- Server crashes suddenly
- Client crashes suddenly
- Network is temporarily broken, then it can eventually be fixed.

The main problem in this project is how can we maximize the availability and consistency when network fails. There are three steps to handle the failure of the system [2]:

- Detect the beginning of the failure
- Limit some operations: There will be a trade-off between the consistency and availability. In this project, we focus on the availability during the partitions.
- Recover the network: After recorvering the failures, we can eventually reach the consistency.

3 Server topology

We use the tree structure to implement the server protocol in this project. However, we have some improvements to handle the network partitions. We set the hierarchy of the tree as in a family. As seen in the Fig. 1, server 1 is the root server, which has two children: server 2 and server 5. Server 3,4 and 6 are the grandchildren of server 1. If the servers have the same parent, they are the sibling. For example, server 3 is the older sibling of server 4.

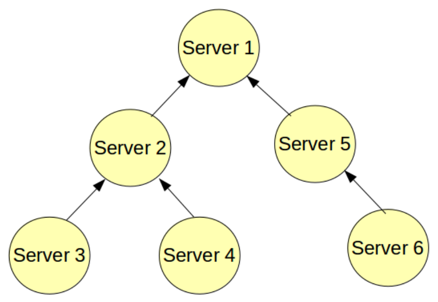


Figure 1: Tree topology

Fig. 2 and 3 describe two failure circumstances. In the first figure, server 2 crashes and server 3 and 4 will connect to their grandparent, which is server 1. In the second situation as shown in Fig. 3, root server crashes and server 2 and server 5 have no grandparent; therefore, the older sibling, which is server 2 will

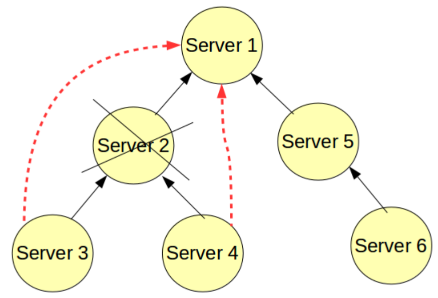


Figure 2: Parent server crashes, reconnect to root server

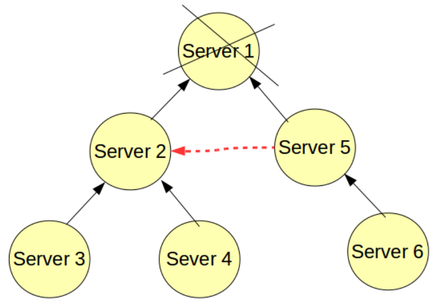


Figure 3: Root server crashes, reconnect to the older sibling

become the root server and server 5 will connect to its older sibling.

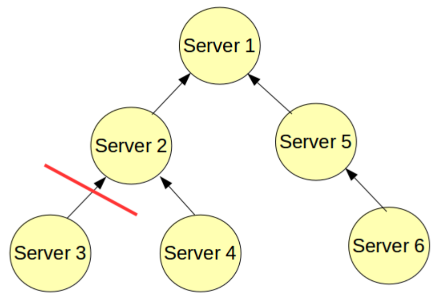


Figure 4: Broken network between S2 and S3

Another failure that can happen is the broken network due to many reasons (E.g. turn off the wifi, etc). In this situation, we handle by following the process described below.

As shown in Fig. 4, there is no connection between server 2 and server 3. We assumed that the network partitions is temporary and can eventually be fixed in less than 12 seconds. In this situation, server 3 will try to reconnect to server 2 in every 6 seconds. After two attempts of trying (exceeds 12s) and it still cannot connect to the server 2, server 3 will try to connect to its grandparent server, which is server 1. If server 2 does not have grandparent server, it will try to connect to its sibling server. If all attempts are failed, the connection of server 3 is closed and it will be removed out of the server's system.

4 Implementation

We implemented many functions on the server side to get the highest accuracy and consistency as possible, as well as an improvement of the first project. These functions will be described as follow.

4.1 Connection

The first improvement is that the servers can join the network after clients have joined. To fix this problem, we implemented a function to synchronize all user information from the remote server to the new server. For example, in Fig. 5, we can assume that Client 1 has joined the network before Server 3. When server 3 connects to server 1, server 1 will broadcast its user information from local storage to server 3. Therefore, if client 1 logout, it can login to server 3 again in the future.

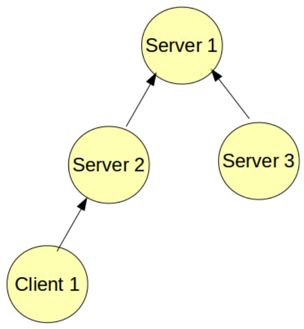


Figure 5: Example of a small network

Furthermore, the clients should be evenly distributed over the servers. We made the servers broadcast the number of its clients to all other servers. After that, we checked if the difference number of the clients between two servers are equal or more than 2, we will redirect the client to another server.

4.2 Registration

One of the hardest problem in our project is handling the registration process at any time. In the first project, if a client is trying to register to one server and one server crashes during the lock process, the client cannot register to the network and the server will return an exception. To deal with this issue, we come up with the idea that the server will response as soon as possible during the lock process without waiting for the network is fixed. An example of the process based on Fig 5 will be described as below:

- Client 1 sends the register request to server 2
- During the lock process, server 3 crashes and cannot send the lock deny to server 1.
- Server 1 will know that server 3 has crashed, instead of waiting for the recovery, server 1 will response lock allow immediately to server 2. After that, server 2 responses register successfully to the client. However, server 1 has a buffer to store the register information of the user.
- After the failure is fixed, server 1 will send the lock request to server 3 according to its buffer. Then, server 3 sends the user information in its local storage. In this case, server 1 will check if the secret of client 1 matches the secret in server 3, there is nothing to do. However, if the secret does not match, server 1 will broadcast a message to all other servers to notify that the username of client 1 should be deleted.

Another issue is that a username can only be registered once. We fixed this problem by giving a timestamp (`registertime`) to the JSON object that store the username. We got the value of `registertime` by `Data().getTime()` method. An example of this process is shown based on Fig. 6 below.

- Client 1 and client 2 register at the same time. Before sending the lock request, the connection between server 1 and server 2 is broken.
- Server 1 and 2 send register success to each client without waiting for the recovery of the connection. This is the trade-off we made to get the high availability
- When connection is fixed, both servers will compare the timestamps of both users. Then servers will keep the username with the smaller timestamp and delete the other one.

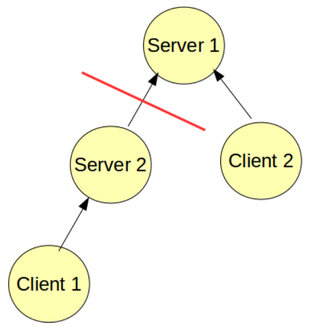


Figure 6: Network is broken during the register process

JSON object of user will be stored in JSON file as below.

```
{"my_user":{"my_secret":"","registertime":0}}
```

4.3 Activity messages

The first goal of activity messages is that when a client sends a message at time X, all clients connected to the network at time X should receive the message. The model failure causes a problem as shown in Fig. 7. When client 1 sends the message to server 2 at time X, server 2 then broadcasts the message to server 1 at time X' ($X' > X$). However, the message cannot go client 2 because the connection between server 1 and client 2 is shut down or client 2 crashes. We propose a solution for this problem by saving the message in the buffer of both server 1 and 2 when the partition is detected. When client 2 connects to the network again, it can get the message from the buffer of the server. The server sends the message to client 2 will then broadcast a notification to all other servers to delete the buffer.

Another problem we have to fix is the activity messages should be sent in the same order. To solve this problem, we put the index to the message and send the message in order based on the index. Given the example shown in Fig. 6, client 2 receives the first message sent from client 1. However, the connection between two servers is broken and the remaining messages cannot be sent. In this case, server 2 will save the messages in its buffer and re-sent it to client 2 when the connection is fixed.

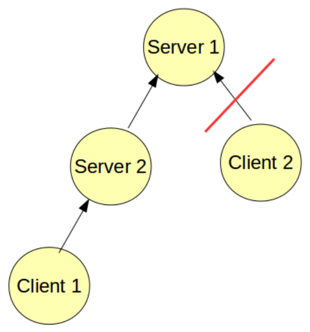


Figure 7: Network is broken during the sending message process

5 Availability and Eventual Consistency

In this project, we focus on the availability so every message should be sent as soon as possible without waiting for the network is recovered. The registration process indicated a high availability of the system where the server's response to the clients without waiting for the network is fixed. Moreover, since both registration and sending activity message stored the information into a buffer, consistency will eventually be reached when the connection is fixed.

In some cases, it is more important to achieve consistency than availability. When we have to keep the order of the messages, we have to mitigate the availability so the server has to wait for the recovery of the network to get all the messages in the same order as when they are delivered.

However, there are some issues in our system. The clients cannot always be able to send the messages. For example, in Fig. 7, if server 2 crashes, client 1 can neither send the message nor receive a response from the server. Since we have to keep the structure of client-side as in the first project, we did not come up with any ideas to fix this problem.

6 Conclusion

In summary, even though we successfully implemented the tree structure as proposed above, the availability was not maximized and consistency is only achieved after the recovery process of the network. Improving the approaches is necessary when applying the system in a large scale network or getting the highest availability.

References

[1] Eric Brewer *Towards robust distributed systems.* Jan, 2000.

[2] Eric Brewer *CAP Twelve Years Later: How the "Rules" Have Changed.* Feb, 2012.

A Demonstration

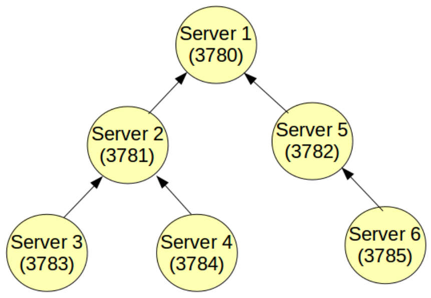


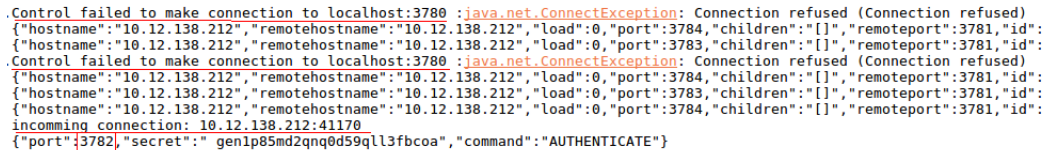
Figure 8: Tree topology

A.1 Server topology

When failures happen, servers will reconnect to another server. This demo is based on the example of section **2. Server topology**. We set up a network of 6 servers as in Fig. 8:

- Server 1: -lp 3780
- Server 2: -lp 3781 -rp 3780
- Server 3: -lp 3783 -rp 3781
- Server 4: -lp 3784 -rp 3781
- Server 5: -lp 3782 -rp 3780
- Server 6: -lp 3785 -rp 3782

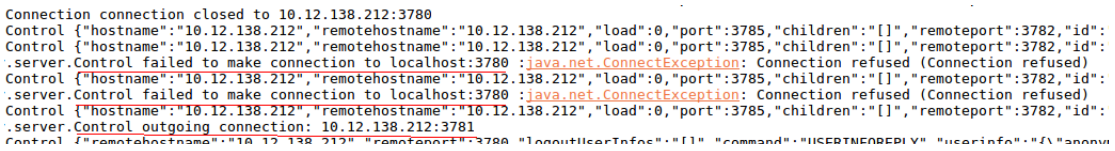
In the first demo, server 1 crashes, then server 2 and server 5 tried to connect to each other. Fig. 9 and 10 show the result of server 2 and 5, respectively. We can see that in these two servers, they tried to connect to server 1 again in every 6 seconds. After two attemps failed to connect to server 1, server 5 (port 3782) connects and send an authenticate message to server 2 (port 3781). Server 2 now becomes the root node of the tree.



```
Control failed to make connection to localhost:3780 :java.net.ConnectException: Connection refused (Connection refused)
{"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3784,"children":"[]","remoteport":3781,"id":
{"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3783,"children":"[]","remoteport":3781,"id":
Control failed to make connection to localhost:3780 :java.net.ConnectException: Connection refused (Connection refused)
{"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3784,"children":"[]","remoteport":3781,"id":
{"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3783,"children":"[]","remoteport":3781,"id":
{"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3784,"children":"[]","remoteport":3781,"id":
incomming connection: 10.12.138.212:41170
{"port":3782,"secret":" gen1p85md2qnq0d59qll3fbcoa","command":"AUTHENTICATE"}
```

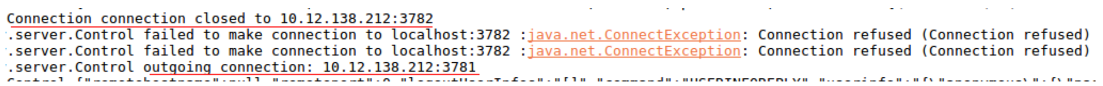
Figure 9: Server 2 (port 3781)

After server 1 crashed, we crashed server 5. Therefore, server 6 will try to reconnect to server 2. The results of server is shown in Fig. 11.



```
Connection connection closed to 10.12.138.212:3780
Control {"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3785,"children":"[]","remoteport":3782,"id":
Control {"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3785,"children":"[]","remoteport":3782,"id":
.server.Control failed to make connection to localhost:3780 :java.net.ConnectException: Connection refused (Connection refused)
Control {"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3785,"children":"[]","remoteport":3782,"id":
.server.Control failed to make connection to localhost:3780 :java.net.ConnectException: Connection refused (Connection refused)
Control {"hostname":"10.12.138.212","remotehostname":"10.12.138.212","load":0,"port":3785,"children":"[]","remoteport":3782,"id":
.server.Control outgoing connection: 10.12.138.212:3781
Control {"remotehostname":"10.12.138.212","remoteport":3780,"logoutUserInfos":"[]","command":"USERINFOREPLY","userinfo":"{\"anony
```

Figure 10: Server 5 (port 3782)



```
Connection connection closed to 10.12.138.212:3782
.server.Control failed to make connection to localhost:3782 :java.net.ConnectException: Connection refused (Connection refused)
.server.Control failed to make connection to localhost:3782 :java.net.ConnectException: Connection refused (Connection refused)
.server.Control outgoing connection: 10.12.138.212:3781
```

Figure 11: Server 6 (3785)