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.

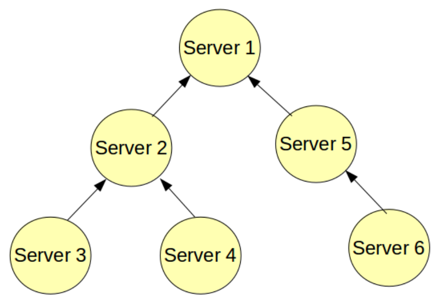


Figure 1: Tree topology

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

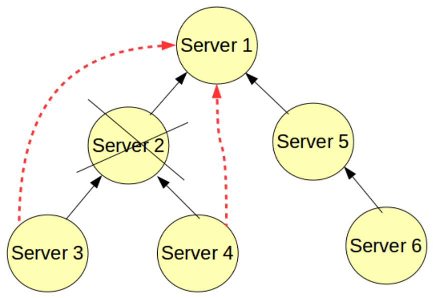


Figure 2: Parent server crashes, reconnect to root server

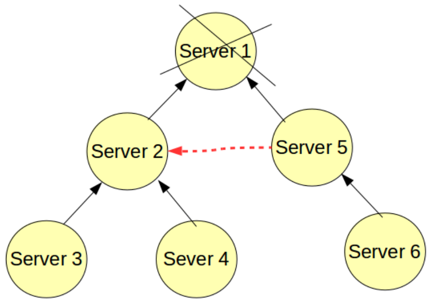


Figure 3: Root server crashes, reconnect to the older sibling

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

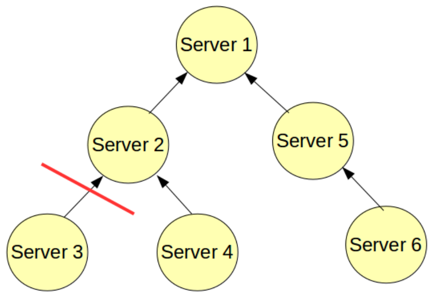


Figure 4: Broken network between S2 and S3

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.

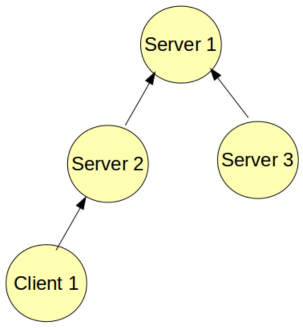


Figure 5: Example of a small network

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 different 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 its user information to server 3 according to its buffer. Server 3 then check its user information compared to server 1's information. If server 3 has the username as in server 1 but a different secret, it will compare the timestamp of registration process. If the user's timestamp in server 3 is less than server 1, it will delete the user from local storage; otherwise, it will send the different user to server 1 and notify that all servers should delete this user.

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.

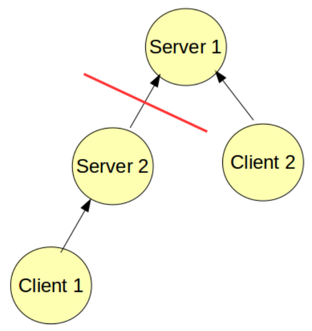


Figure 6: Network is broken during the register process

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 received in the same order as they are delivered. 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.

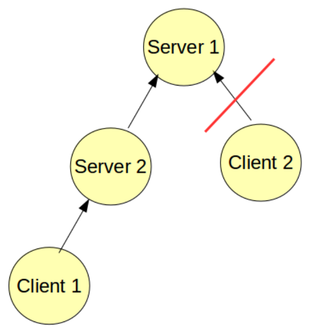


Figure 7: Network is broken during the sending message process

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

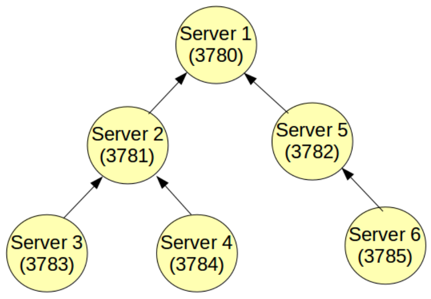
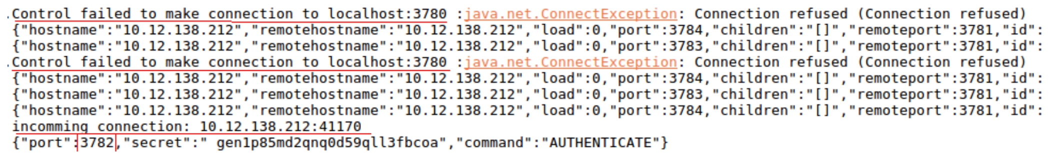


Figure 8: Tree topology

A.1 Server topology

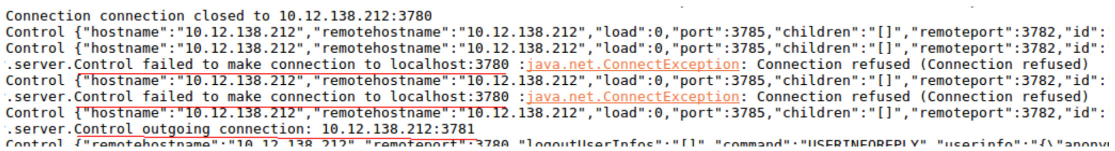
When failures happen, servers will reconnect to another server. This demo is based on the example of section **2. Server topology**. We set up a network of 6 servers as in Fig. 8:

- 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



```
.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"}
```

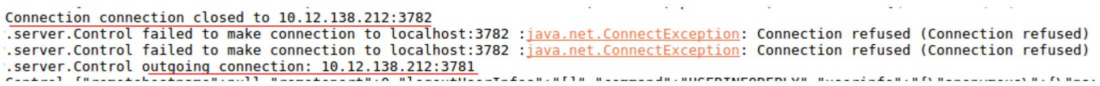
Figure 9: Server 2 (port 3781) reconnected to server 5



```
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) reconnected to server 2

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)



```
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) reconnected to server 2

connects and send an authenticate message to server 2 (port 3781). Server 2 now becomes the root node of the tree.

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

A.2 Registration

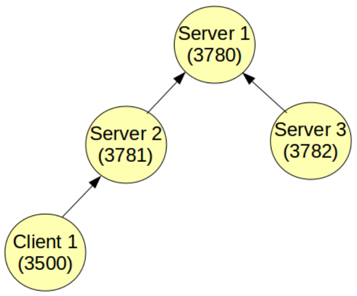
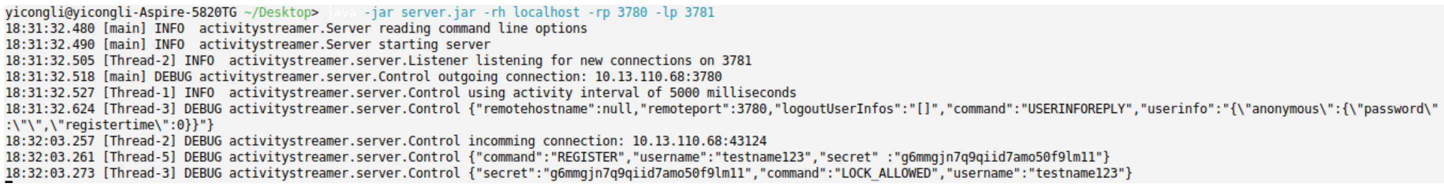
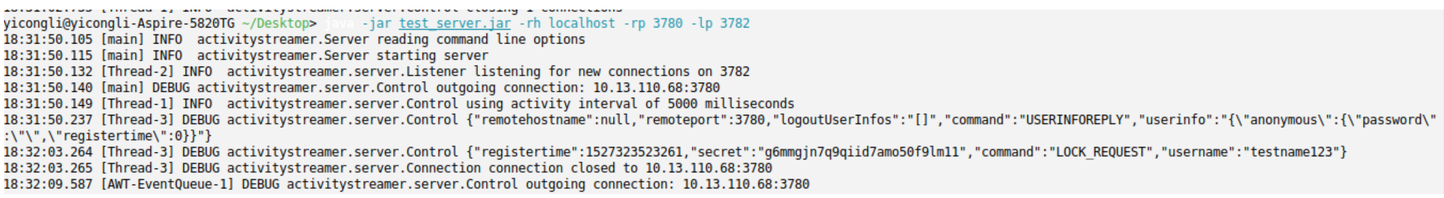


Figure 12: Demo - Register at any time



```
yicongli@yicongli-Aspire-5820TG ~/Desktop> java -jar server.jar -rh localhost -rp 3780 -lp 3781
18:31:32.480 [main] INFO  activitystreamer.Server reading command line options
18:31:32.490 [main] INFO  activitystreamer.Server starting server
18:31:32.505 [Thread-2] INFO  activitystreamer.server.Listener listening for new connections on 3781
18:31:32.518 [main] DEBUG activitystreamer.server.Control outgoing connection: 10.13.110.68:3780
18:31:32.527 [Thread-1] INFO  activitystreamer.server.Control using activity interval of 5000 milliseconds
18:31:32.624 [Thread-3] DEBUG activitystreamer.server.Control {"remotehostname":null,"remoteport":3780,"logoutUserInfos":"[]","command":"USERINFOREPLY","userinfo":"{\"anonymous\":{\"password\"
:\"\",\"registertime\":0}}"}
18:32:03.257 [Thread-2] DEBUG activitystreamer.server.Control incomming connection: 10.13.110.68:43124
18:32:03.261 [Thread-5] DEBUG activitystreamer.server.Control {"command":"REGISTER","username":"testname123","secret" :"g6mmgjn7q9qiid7amo50f9lm11"}
18:32:03.273 [Thread-3] DEBUG activitystreamer.server.Control {"secret":"g6mmgjn7q9qiid7amo50f9lm11","command":"LOCK_ALLOWED","username":"testname123"}
```

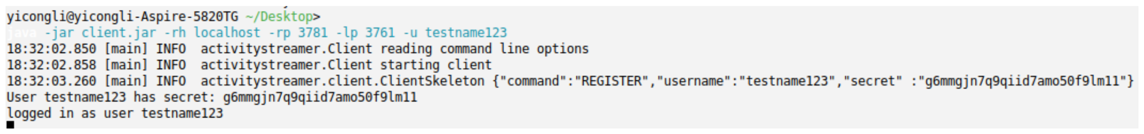
Figure 13: Server 2 sent the lock request after getting the register request from the client



```
yicongli@yicongli-Aspire-5820TG ~/Desktop> java -jar test_server.jar -rh localhost -rp 3780 -lp 3782
18:31:50.105 [main] INFO  activitystreamer.Server reading command line options
18:31:50.115 [main] INFO  activitystreamer.Server starting server
18:31:50.132 [Thread-2] INFO  activitystreamer.server.Listener listening for new connections on 3782
18:31:50.140 [main] DEBUG activitystreamer.server.Control outgoing connection: 10.13.110.68:3780
18:31:50.149 [Thread-1] INFO  activitystreamer.server.Control using activity interval of 5000 milliseconds
18:31:50.237 [Thread-3] DEBUG activitystreamer.server.Control {"remotehostname":null,"remoteport":3780,"logoutUserInfos":"[]","command":"USERINFOREPLY","userinfo":"{\"anonymous\":{\"password\"
:\"\",\"registertime\":0}}"}
18:32:03.264 [Thread-3] DEBUG activitystreamer.server.Control {"registertime":1527323523261,"secret":"g6mmgjn7q9qiid7amo50f9lm11","command":"LOCK_REQUEST","username":"testname123"}
18:32:03.265 [Thread-3] DEBUG activitystreamer.server.Connection connection closed to 10.13.110.68:3780
18:32:09.587 [AWT-EventQueue-1] DEBUG activitystreamer.server.Control outgoing connection: 10.13.110.68:3780
```

Figure 14: Server 3 received the lock request and then crashed before sent back the lock deny

To demonstrate the function of clients register at any time in the network. We followed the structure as shown in Fig. 12. In this example, client 1 registered to server 2. However, during the lock request process, the connection between server 1 and 3 crashed so server 3 cannot send the lock deny message. Without waiting for the recovery, server 2 responsed to client 1 as register success.



```
yicongli@yicongli-Aspire-5820TG ~/Desktop>
java -jar client.jar -rh localhost -rp 3781 -lp 3761 -u testname123
18:32:02.850 [main] INFO  activitystreamer.Client reading command line options
18:32:02.858 [main] INFO  activitystreamer.Client starting client
18:32:03.260 [main] INFO  activitystreamer.client.ClientSkeleton {"command":"REGISTER","username":"testname123","secret" :"g6mmgjn7q9qiid7amo50f9lm11"}
User testname123 has secret: g6mmgjn7q9qiid7amo50f9lm11
logged in as user testname123
```

Figure 15: Client got the register success response

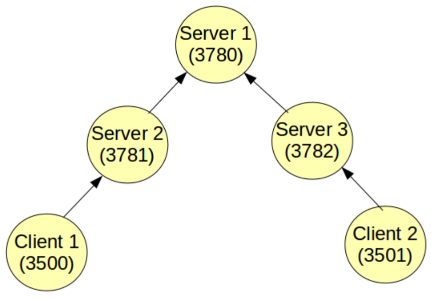
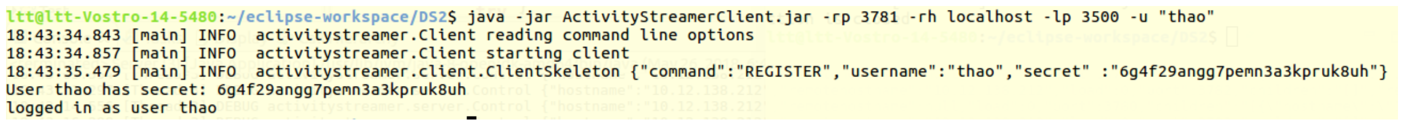
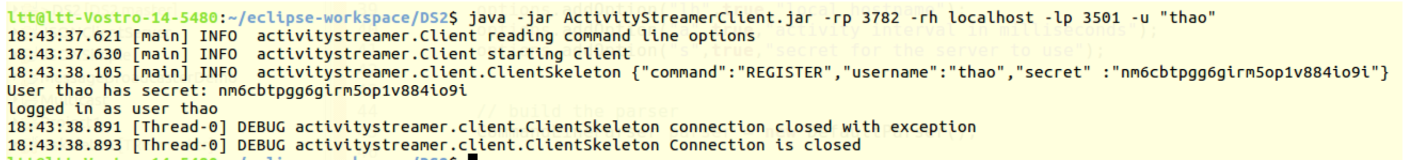


Figure 16: Demo - Register at the same time



```
ltt@ltt-Vostro-14-5480:~/eclipse-workspace/DS2$ java -jar ActivityStreamerClient.jar -rp 3781 -rh localhost -lp 3500 -u "thao"
18:43:34.843 [main] INFO  activitystreamer.Client reading command line options
18:43:34.857 [main] INFO  activitystreamer.Client starting client
18:43:35.479 [main] INFO  activitystreamer.client.ClientSkeleton {"command":"REGISTER","username":"thao","secret" :"6g4f29angg7pemn3a3kpruk8uh"}
User thao has secret: 6g4f29angg7pemn3a3kpruk8uh
logged in as user thao
```

Figure 17: Client 1 (3500)



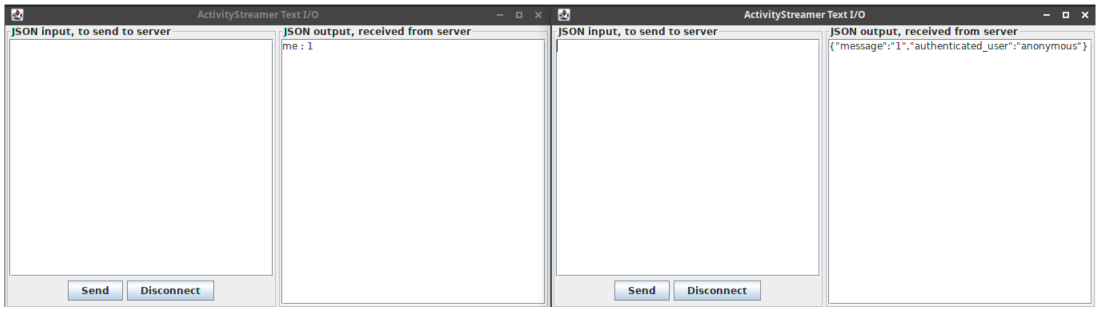
```
ltt@ltt-Vostro-14-5480:~/eclipse-workspace/DS2$ java -jar ActivityStreamerClient.jar -rp 3782 -rh localhost -lp 3501 -u "thao"
18:43:37.621 [main] INFO  activitystreamer.Client reading command line options
18:43:37.630 [main] INFO  activitystreamer.Client starting client
18:43:38.105 [main] INFO  activitystreamer.client.ClientSkeleton {"command":"REGISTER","username":"thao","secret" :"nm6cbtpgg6girm5op1v884io9i"}
User thao has secret: nm6cbtpgg6girm5op1v884io9i
logged in as user thao
18:43:38.891 [Thread-0] DEBUG activitystreamer.client.ClientSkeleton connection closed with exception
18:43:38.893 [Thread-0] DEBUG activitystreamer.client.ClientSkeleton Connection is closed
```

Figure 18: Client 2 (3501)

To demo a network that a username can only be registered once over the network. We set up the network as in Fig. 16. First, we crashed server 1 and during the partitions, we registered client 1 to server 2 and client 2 to server 3 with the same username (”thao”). After server 3 reconnected to server 2, it compared the timestamps of two users and removed the user with bigger timestamp. In this demotration, Fig. 17 and 18 show our result. Two clients both register successfully to the server during the failure. However, after server 3 connected to server 2, the user from client 2 is deleted and user ”thao” can just login again with secret of client 1.

A.3 Activity messages

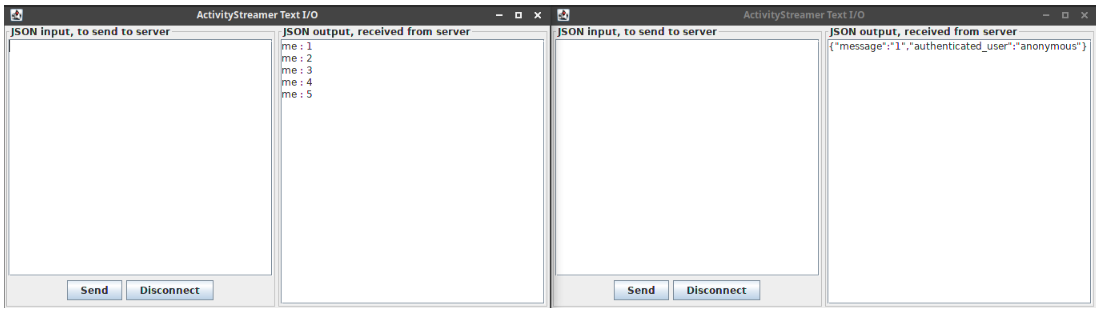
In this part, we demonstrate that the messages sent by a client are delivered in the same order as follow. This example shows that client 1 tried to send messages to client 2 (Fig. 19). After that, a failure happend and client cannot send messages to client (Fig. 20). Finally, after the network is recovered, client 2 received the messages in order.


ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect
ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
```

```
{"message":"1","authenticated_user":"anonymous"}
```

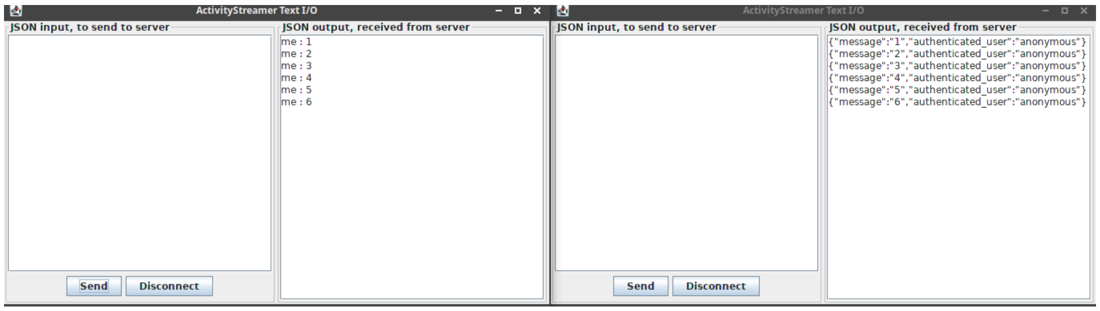
Figure 19: Client 1 sends first message to client 2


ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect
ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
me : 2
me : 3
me : 4
me : 5
```

```
{"message":"1","authenticated_user":"anonymous"}
```

Figure 20: Client 1 sends multiple messages to client 2 during the failure

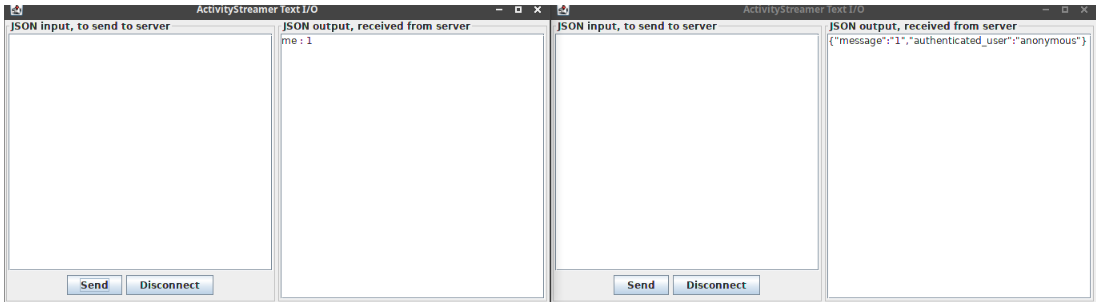

ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect
ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
me : 2
me : 3
me : 4
me : 5
me : 6
```

```
{"message":"1","authenticated_user":"anonymous"}
{"message":"2","authenticated_user":"anonymous"}
{"message":"3","authenticated_user":"anonymous"}
{"message":"4","authenticated_user":"anonymous"}
{"message":"5","authenticated_user":"anonymous"}
{"message":"6","authenticated_user":"anonymous"}
```

Figure 21: Client 2 receives messages after the failure in order

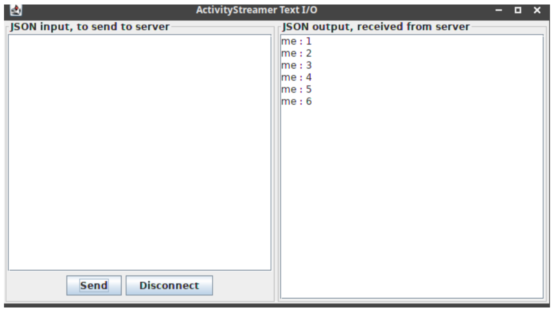
The second example in this part is messages that are sent at time X can be delivered to clients connected at time X.


ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect
ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
```

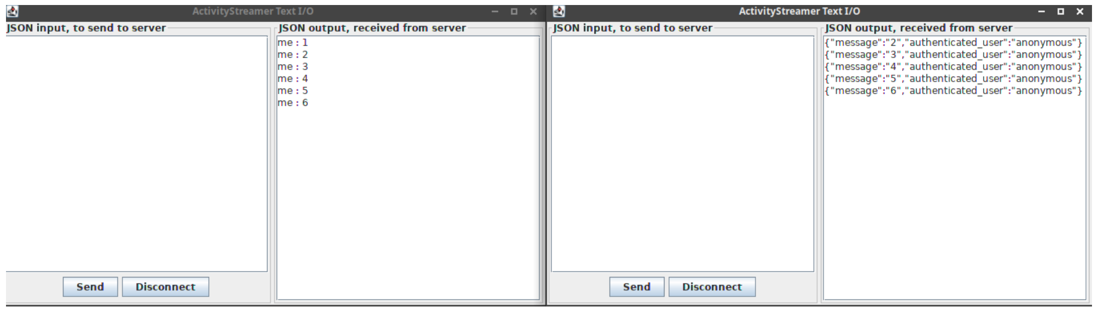
```
{"message":"1","authenticated_user":"anonymous"}
```

Figure 22: Send first message from client 1 to client 2


ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
me : 2
me : 3
me : 4
me : 5
me : 6
```

Figure 23: Client 1 sends message but client 2 has been logout before receiving the message


ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect
ActivityStreamer Text I/O
JSON input, to send to server
JSON output, received from server
Send
Disconnect

```
me : 1
me : 2
me : 3
me : 4
me : 5
me : 6
```

```
{"message":"2","authenticated_user":"anonymous"}
{"message":"3","authenticated_user":"anonymous"}
{"message":"4","authenticated_user":"anonymous"}
{"message":"5","authenticated_user":"anonymous"}
{"message":"6","authenticated_user":"anonymous"}
```

Figure 24: Client 2 receives the messages after login again