COMP90015 Project2 Multi Server Network
--High Availability and Eventual Consistency

Group Name: Fantastic Four
Group Members:

Name	LMS Login	E-mail
Ning Kang	ningk1	ningk1@student.unimelb.edu.au
Yiru Pan	yirup	yirup@student.unimelb.edu.au
Nannan Gu	nannang	nannang@student.unimelb.edu.au
Wenyi Zhao	wenyiz4	wenyiz4@student.unimelb.edu.au

1. Overview

This project is an improved multi-server system based on project 1, which implements high availability and eventual consistency with several enhancements in the presence of possible server failure and network partition.

2. Outcome

- Clients can join (register/login) and leave (logout) the network at any time, Servers can join the network at any time
- High Availability: system can re-connect automatically after network partition
- Unique Register: a given username can only be registered once over the server network
- Message ensure: a message sent by a client can reach all clients that are connected to the network at sending time
- Message order: all activity messages sent by a client are delivered in the sending order at each receiving client
- Load balancing: clients are evenly distributed over the servers

3. System Architecture

As *Figure 1 Layered Structure* illustrates, the server contains 3 layers, namely application layer, data layer and network layer. Network layer is the communication layer for the whole system and is responsible for maintaining connections, sending or receiving data and delivering different types of message to different data consumers. The reconnection (high available) will be conducted in this layer and will not impact the other layers.

Data layer is used to store all local data and sync its data with other servers' data layer, which means there is a distributed database across the whole server system. Specific information is stored in tables. There are 3 tables designed in this project, as *Table 1 Tables of Data Layer* shows:

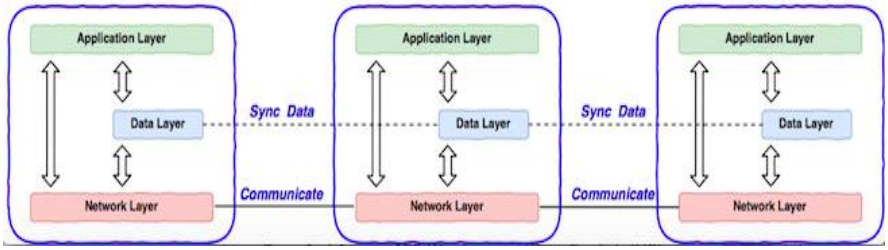


Figure 1 Layered Structure

Table 1 Tables of Data Layer

Table	Row	Columns
ServerTable	ServerRow	serverId, host, port, load, online flag
UserTable	UserRow	username, secret, online flag
ActivityTable	ActivityRow	owner All activities that should be sent to this owner.

In this case, application layer can access global data just like access its local data. All data sync tasks will be conducted by data layer.

Application layer is a high-level layer, it can request network layer and data layer to execute their tasks respectively. Besides, application layer also needs to process activity message, authenticate, user login/log out and user register. Diagram is as *Figure 2 Communication between layers.*

Formats of specific messages for each layer are included in Appendix.

4. High Availability

The failure model suggests servers can crash at any time or network connections between servers or servers-clients can be broken at any time, but failures are transient and broken network connections will eventually be fixed. Per CAP theorem, in network partition scenario, high availability and high consistency cannot achieve at the same time, system has to choose one between availability and consistency. This system focuses on availability so that once network partition occurs, each sub area of system can still work normally, clients are able to send messages and receive message. Messages/operations that are conducted during partition period will be synced among all parts after reconnections are done, which will be discussed later in this report.

Two mechanisms have been used to achieve high availability:

- Backup List

Each server has a backup list in its network layer. Backup list contains host and port information of neighbor servers. Backup list of a server is broadcasted to its neighbor servers. For example, if one server fails, other connected servers will try to reconnect to a server in backup list in order, until reconnection is successful or the first available backup server is the server itself. Thus, once reconnection is done, whole network will return a normal status.

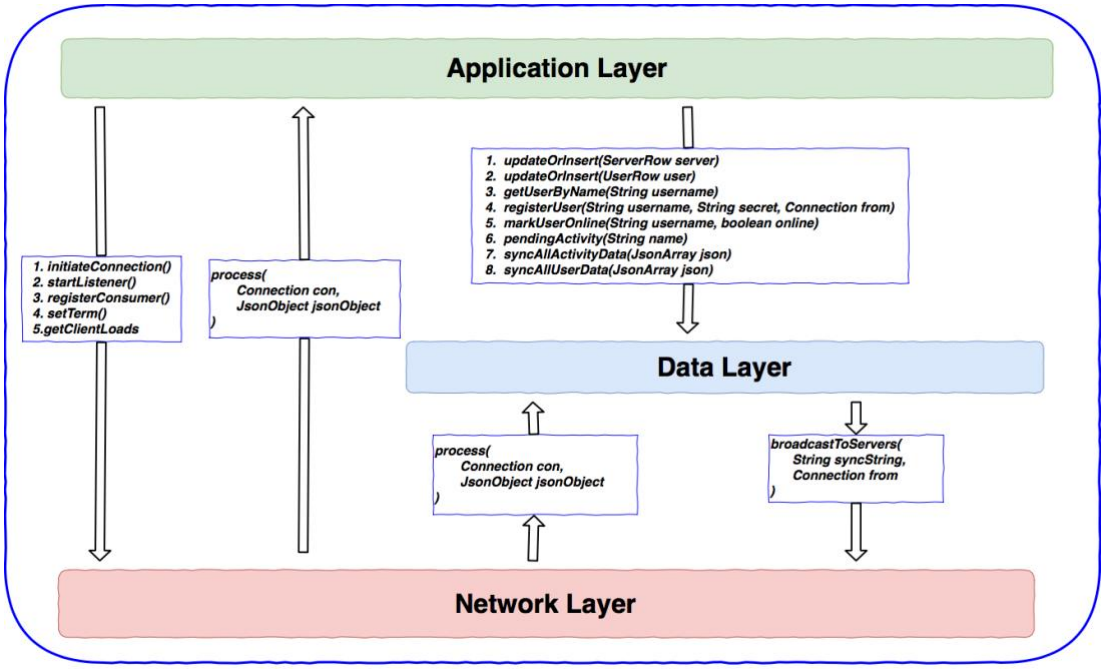


Figure 2 Communication between layers

Per specification requirement, client part should be the same as project 1. If a server crashed, all connected clients have to be redirected to another server in backup list manually.

Crash scenario diagram is as *Figure 3 Crash scenario.*

- Full data copy of new join server

New join server will get a full data copy with existing servers. The server authentication process has been improved by adding a new message - ***AUTHENTICATE_SUCC***. When a new join server sends authenticate message to an existing server, the existing server will reply with authenticate success message if no error occurs in authenticate process. Authentication success message includes all information in data layer (*ServerTable, UserTable and ActivityTable*). Therefore, whenever a new server joins the existing system, it will have the same data as the existing system by syncing a full copy of data from the server it connects to.

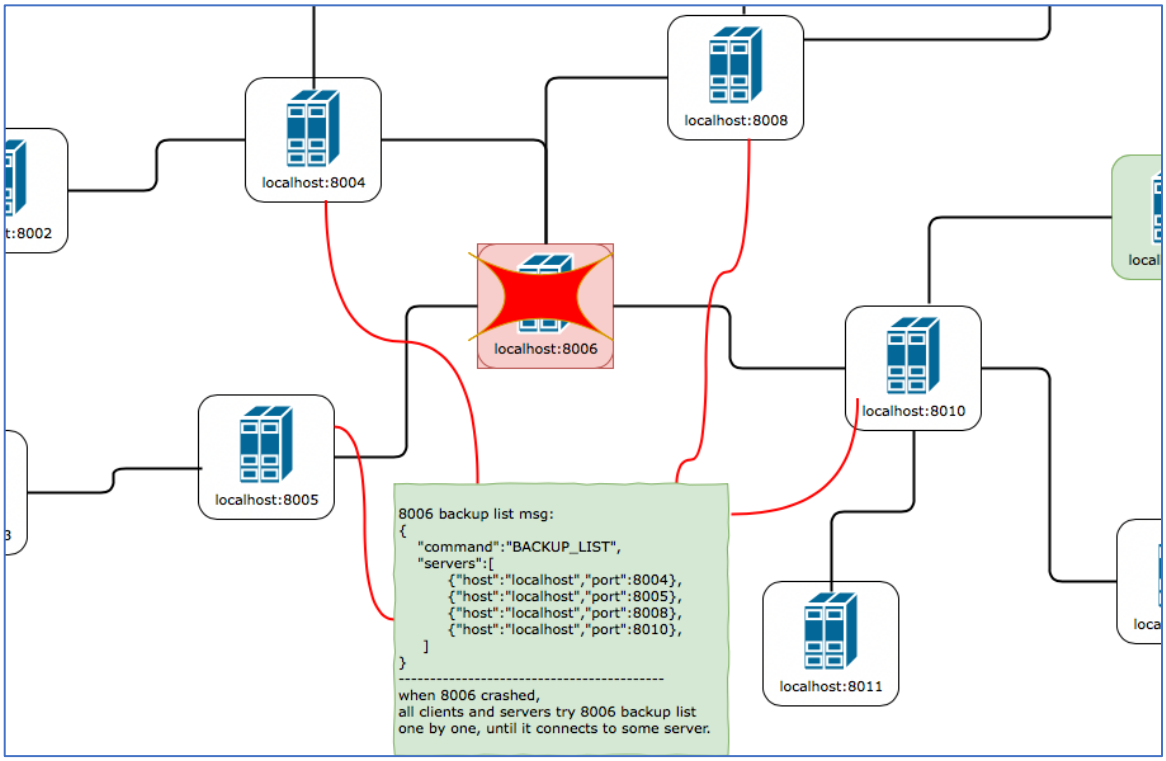


Figure 3 Crash scenario

5. Eventual Consistency

Two processes are developed in this system to achieve eventual consistency. Both of them are data syncing protocols in data layer:

5.1. Data transaction notification:

This is an incremental synchronization. If there is any change in data layer of a server, that change will be broadcasted to the whole system and other servers will update their local data accordingly. For example, if a user's online status is changed in server A, this user's information will be broadcasted to whole system with updated *updateTime* to indicate the changing time.

5.2. Periodical synchronization:

This is a global synchronization. Each server broadcasts all information in its data layer to the whole network in a particular period of time (5 seconds by default), and servers receive this message will compare it with their local data and update accordingly *(insert or update occur only when remote updateTime > local updateTime).*

These 2 synchronization mechanisms can also guarantee message delivery and message delivery order.

Before explaining how our system ensure delivery and delivery order, another feature must be introduced: implementation of our system for delivering activity to client is synchronous, which means a client may need to wait a period of time (1s by default) before he/she can actually receive an activity sent by other users. Let us name this period of time as ***activity_interval***.

5.3. Message Ensure

Let us explain how this system ensure delivery. Consider a series of situations as below:

1. Activity 1 was sent from user A at time T, when user B was online at that time. This activity will be inserted into both user A's and user B's activity list in server 8002. And this ***transaction*** will be broadcasted over the system which

means server 8003 will have this data later.

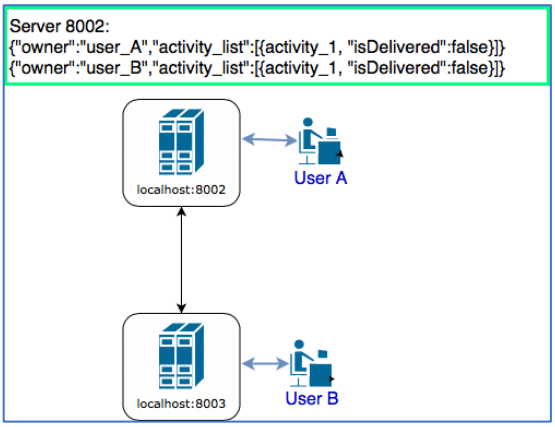


Figure 4 Delivery Ensure - 1

2. If User B logout before he/she receives this data, then user B's activity list will be stored in every server of this system and once user B login again at any server of this system, user B will eventually receive this message.

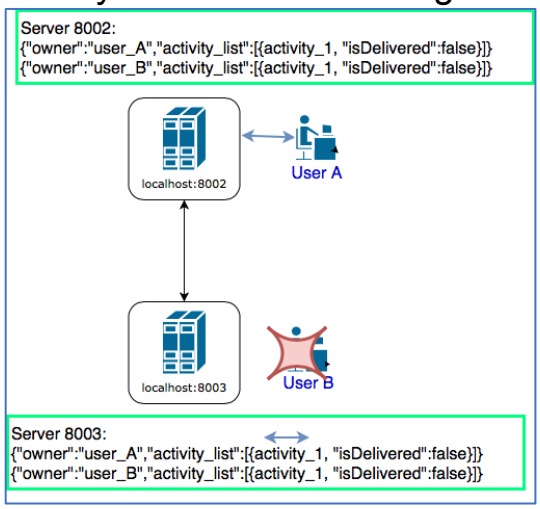


Figure 5 Delivery Ensure - 2

3. In order to avoid infinity delivery, once a message is delivered to its owner, that activity will be marked as "delivered" over the system (all servers will mark this message as "delivered").

5.4. Order Ensure

Example for order guarantee is similar. A simple case is described as below:

1. At time 1, 3 servers and 2 clients were working normally and activity 1 was sent from User A. Server 8001 crashed before it could send activity 1 to other servers, which means server 8003 will not receive that from 8001.

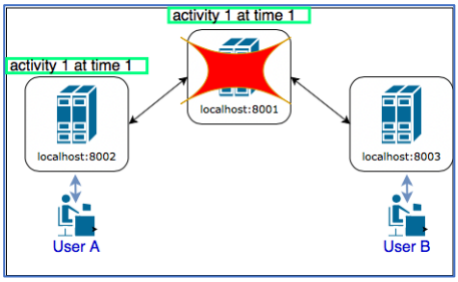


Figure 6 Ensure Delivery - 1

2. After high available strategy applied, 8002 and 8003 were connected and activity 2 was sent from user A before a sync message was sent from 8002, which means 8003 only received activity 2.

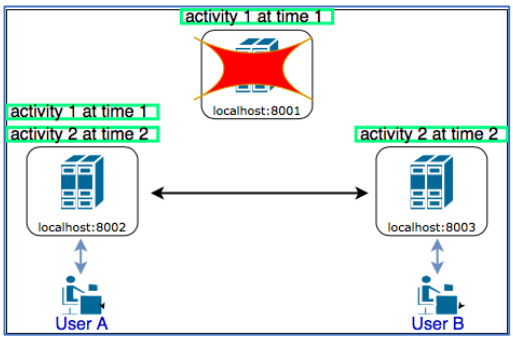


Figure 7 Ensure Delivery - 2

3. If a sync message from server 8002 was broadcasted within ***activity_interval***, which means 8003 had not yet sent activity 2 to client and activity 1 was inserted by sync process. Then 8003 can order these 2 messages by their sending time and then send both of them in right order.

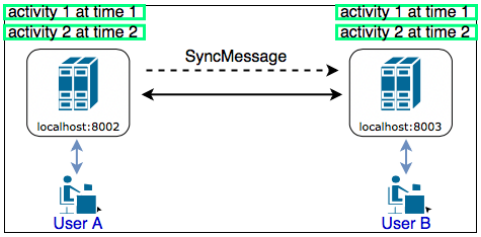


Figure 8 Ensure Delivery - 3

An issue for this design is that if global synchronization period is longer than ***activity_interval***, the order may not be ensured.

Examples for delivery guarantee are similar. A simple case is that user B is online when an activity was sent from user A at time T, but user B gets offline before it can receive this activity

6. 6. New Protocols

6.1. SERVER_ANNOUNCE

In order to make sure the redirection process never redirect a client to a crashed server, another attribute "action" should be added into announcement protocol to let the whole network know if a server is crashed. When server A crash, servers directly connected with A will detect that failure and an announcement will be broadcasted from these servers to delete server A from online server list. In this scenario, "action" in message is "delete", and only "action" and "serverId" are mandatory (other fields can be ignored). In normal scenarios, "action" is "update".

```
{
  "command":"SERVER_ANNOUNCE",
  "action":"update/delete",
  "serverId":"serverId",
  "load":"load",
  "ip":"ip",
  "port":"port"
}
```

6.2. USER_UPDATE

This protocol is used for incremental synchronization. Compare updateTime first and decide whether it needs to be updated. If updateTime in this message is newer, then compare a particular user information with local data layer and create/update accordingly.

```
{
  "command":"USER_UPDATE",
  "username":"username",
  "secret":"secret",
  "online":"true/false",
  "udpateTime":"updateTime"
}
```

6.3. USER_SYNC

This is used for global synchronization. Compare updateTime first and decide if it needs to update, if so, then compare all users' information with local data layer and update accordingly.

```
{
  "command":"USER_SYNC",
  "user_list":[
   {"username":"username",
    "secret":"secret",
    "online":"true/false",
    "udpateTime":"updateTime"},
   {"username":"username",
    "secret":"secret",
    "online":"true/false",
    "udpateTime":"updateTime"}
   ...
  ]
}
```

6.4. ACTIVITY_UPDATE

This protocol is used for incremental synchronization. Compare updateTime first and decide if it needs to update, if so, then compare a particular activity information with local data layer and create/update accordingly.

```
{
  "command":"ACTIVITY_UPDATE",
  "owner":"username",
  "activity_list":
  {
    "udpateTime":"updateTime",
    "sendTime":"sendTime",
    "isDelivered":"false/ture",
    "activity":{
      "authenticated_user":"authenticated_user",
      "other":"other"}
  }
}
```

6.5. ACTIVITY_SYNC

This is used for global synchronization. Compare updateTime first and decide if it needs to update, if so, then compare activity lists for all registered users with local data layer and update accordingly.

```
{
  "command":"ACTIVITY_SYNC",
  "activity_entity":[
    {USER1_ACTIVITY_UPDATE_JSON},
    {USER2_ACTIVITY_UPDATE_JSON},
    ...
  ]
}
```

6.6. AUTHENTICATE

Add “serverid”, “host” and “port”, when a new server is authenticating to an existing server, new server needs to tell its host and port so that backup list of the existing server can be formed.

```
{
  "command":"AUTHENTICATE",
  "serverid":"serverid",
  "secret":"system_secret",
  "host":"ip to provide services(set from command line arg)",
  "port":"port to provide services(set from command line arg)"
}
```

6.7. AUTHENTICATE_SUCCESS

When a new join server authenticated successfully, it will receive AUTHENTICATE_SUCCESS, which contains full information of data layer. Therefore, this new join server has same data cope with other existing servers and can provide service to clients.

```
{
  "command":"AUTHETICSTE_SUCC",
  "serverid":"serverid",
  “server_list":SERVER_SYNC_JSON
  "user_list":USER_SYNC_JSON,
  "activity_entity":ACTIVITY_SYNC_JSON
}
```

6.8. BACKUP_LIST

Each server has a backup list in its network layer. Backup list contains host and port information of neighbor servers. Backup list of a server is broadcasted to its neighbor servers so that neighbor servers know how to reconnect next server once this server fails.

```
{
  "command":"BACKUP_LIST",
  "servers":[
    {"host":"host_ip","port":"prot_num"},
    {"host":"host_ip2","port":"port_num2"}
  ]
}
```

Appendix A Document Link

- Github

Public repository link is as below:
https://github.com/DistributeSystem2018
All detailed documents in development process have been uploaded in this repository, large diagrams can be found in this repository.

Appendix B Meeting Minutes

Date: 25 May 2018

Duration: 1 hours

Attentance: Yirun Pan, Ning Kang, Nannan Gu, Wenyi Zhao

0. Report finalise
1. Usage & Test cases finalise

Date: 21 May 2018

Duration: 2 hours

Attentance: Yirun Pan, Ning Kang, Nannan Gu, Wenyi Zhao

0. Testing results discussion
1. Discuss the final report structure
2. Documentation tasks assignment

Date: 17 May 2018

Duration: 2 hours

Attentance: Yirun Pan, Ning Kang, Nannan Gu, Wenyi Zhao

0. Task process tracing
1. Simple overall testing of current project by task owners
2. Test cases designing
3. Test cases tasks assignments
4. Useless code clearance

Date: 10 May 2018

Duration: 2 hours

Attentance: Yirun Pan, Ning Kang, Nannan Gu, Wenyi Zhao

0. Finalise the refactoring of the old system
1. Design some new protocals to achieve the aims
2. Assign tasks

Date: 03 May 2018

Duration: 2 hours

Attentance: Yirun Pan, Ning Kang, Nannan Gu, Wenyi Zhao

0. Discuss overview requirements of this project
1. Discuss how to realize the aims and the possible new protocals
2. Discuss how to ensure high availability and eventual consistency
3. Design a new architecture of the whole system to meet the new requirements
4. Assign system refactoring tasks

Project 2 - Usage & Test Cases

Introduction

This is an advanced version of project 1 which provides:

- High Available
- Eventually Consistancy

NOTE: Our implementation for delivering activity to client is synchronous, so that you may need to wait a period of time before you can actually receive an activity (default period is 1 second)

System User Guide

System Set Up

There are two JAR files in source code package, `ActivityStreamerClient.jar` and `ActivityStreamerServer.jar`.

Jar file usage:

Server startup

```
usage: ActivityStreamer.Server [-a <arg>] [-activity_check_interval <arg>]
       [-lh <arg>] [-lp <arg>] [-rh <arg>] [-rp <arg>] [-s <arg>]
       [-sync_interval <arg>] [-time_before_reconnect <arg>]
An ActivityStream Server for Unimelb COMP90015

 -a <arg>                          announce interval in milliseconds

 -lh <arg>                         local hostname
```

```
 -lp <arg>                          local port number
 -rh <arg>                          remote hostname
 -rp <arg>                          remote port number
 -s <arg>                           secret for the server to use
 -sync_interval <arg>               Provide the interval (in milliseconds,
                                    5000 by default) to sync data amoung
                                    servers.
 -time_before_reconnect <arg>       Provide the time (in milliseconds, 0 by
                                    default) to wait before reconnect if a
                                    server crashes, mainly for testing
                                    eventually consistancy
 -activity_check_interval <arg>     Provide the interval (in milliseconds,
                                    1000 by default) to check whether there
                                    is new activity coming in.
```

Client startup

```
usage: ActivityStreamer.Client [-rh <arg>] [-rp <arg>] [-s
       <arg>] [-u <arg>]
An ActivityStream Client for Unimelb COMP90015
 -rh <arg>   remote hostname
 -rp <arg>   remote port number
 -s  <arg>   secret for username, if not provided, run "register" process
 -u  <arg>   username, if not provided, login as "anonymous".
```

Test Scenario

Per projectspecification, this system is supposed to achieve following functions:

- High Availability: system can reconnect automaticallyafter network partition
- Clients can join (register/login) and leave (logout)the network at any time, Servers can join the network at any time
- Unique Register: a given username can only beregistered once over the server network
- Message ensure: a message sent by a client can reachall clients that are connected to the network at the time
- Message order: all activity messages sent by a clientare delivered in the same order at each receiving client
- Load balancing: clients are evenly distributed overthe servers

Our implementationfor delivering activity to clients is synchronous, so you may need to wait aperiod of time before you can actual receive an activity, default period is 1second.

Six scenarios havebeen designed for test case.

NOTE: All test screenshot shown below are using our new version client which is more useful for debug. If you want to test the last version client(client of project1) , change the command to user `ActivityStreamerClient-old.jar` as the client jar package. The old version also passed all these test cases.

High Available

Test Case

1. Start 4 servers

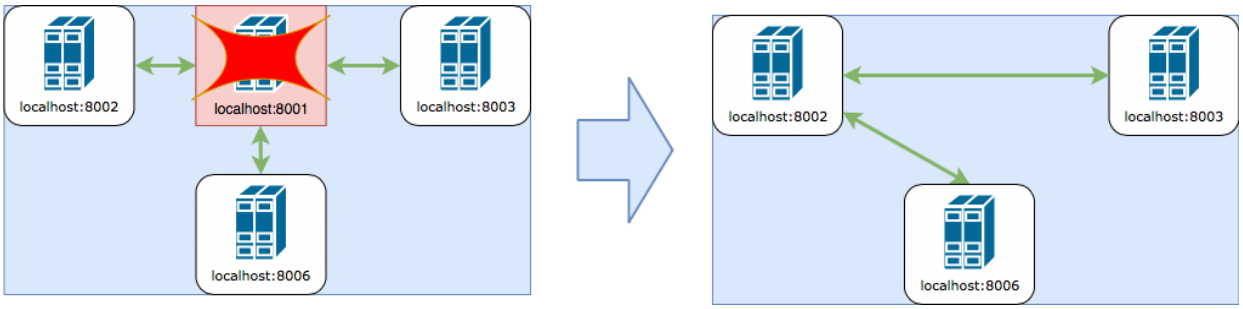
```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
java -jar ActivityStreamerServer.jar -lh localhost -lp 8003 -s abc -rh
localhost -rp 8001
java -jar ActivityStreamerServer.jar -lh localhost -lp 8006 -s abc -rh
localhost -rp 8001
```

2. Force quit server 8001

Click ***Close*** icon in *server UI* or press ***CTRL+C*** in *command line*

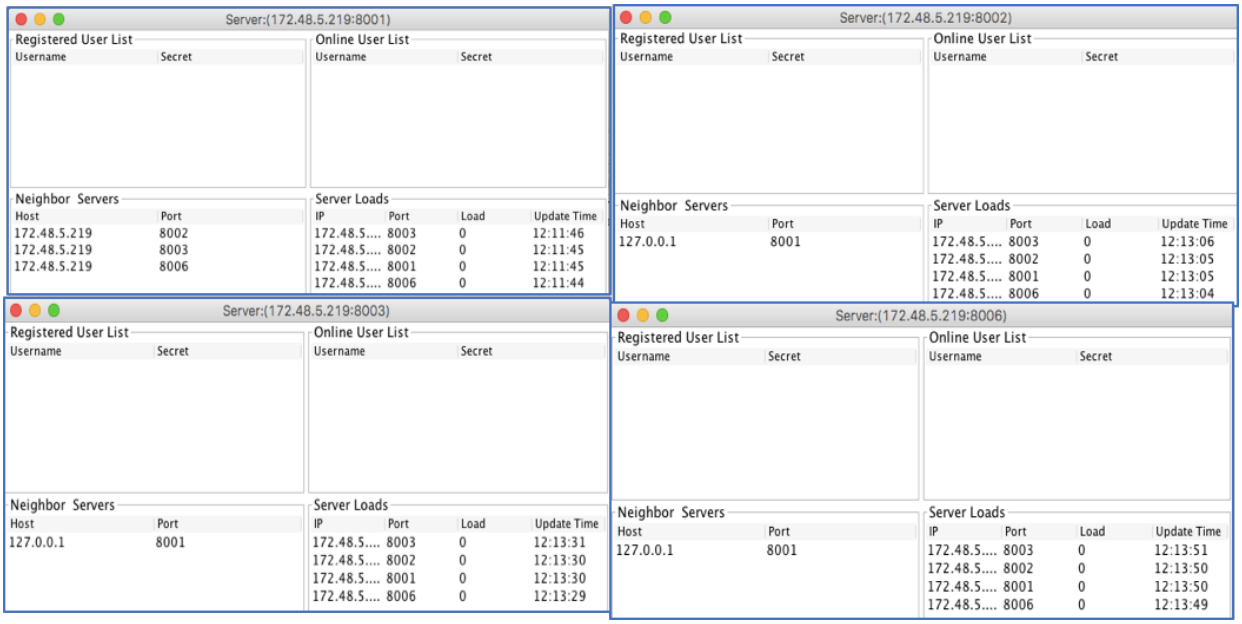
Expected Result

After that you will see server 8002, 8003, 8006 will automatically connected. The picture shows a successful situation (the one, 8002, that takes 8001's place may vary).

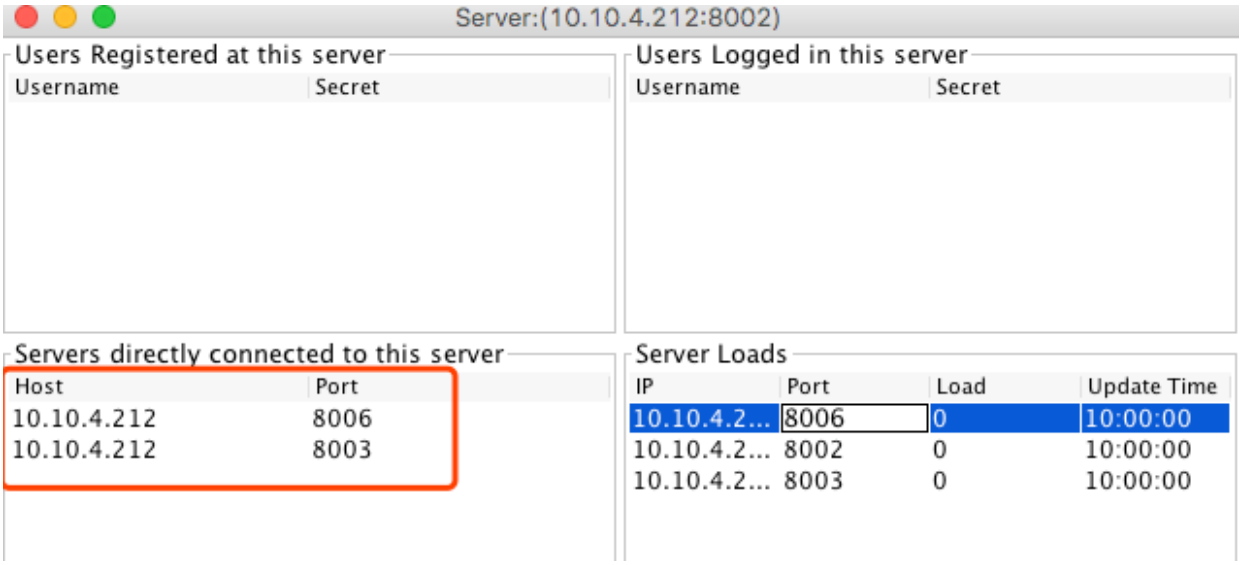


Screentshot:

After 4 servers were started:



After force quit server 8001:



Testing Result:

As expected.

Client can join and leave any time

Test case

1. Start the very first server

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
```

2. Register a user at 8001 and ***remember*** its secret

```
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
```

3. Quit client of last step (close GUI or press CTRL+C in terminal)
4. Start a new server and connect it to 8001

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
```

5. Login user1 at new server 8002 (replace `$secret` by actual secret)

```
java -jar ActivityStreamerClient.jar -u user1 -s $secret -rp 8002 -rh
localhost
```

Expected Result

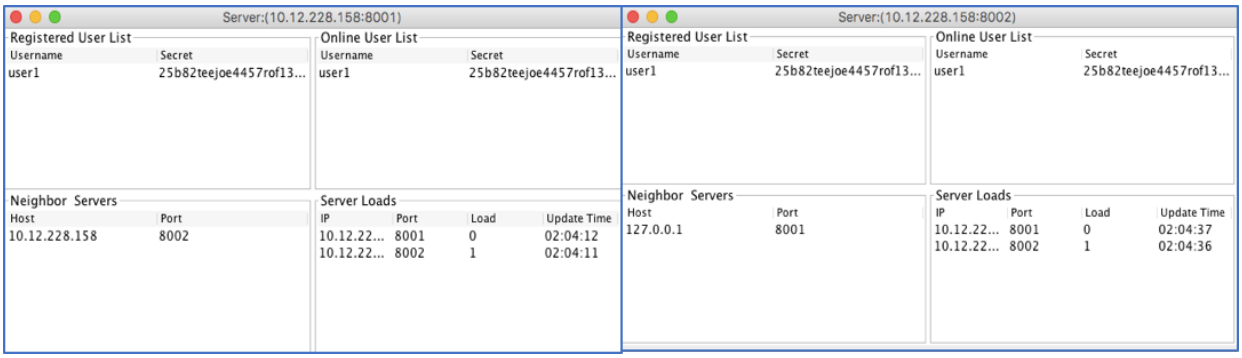
User1 should login on new server 8002 successfully, and all data of 8002 should be consistent with 8001.

Screentshot

Register success and auto login with given secret

```
2018-05-24 13:57:20 [main] INFO  clientLogger - send register to server with user=user1 secret=25b82teejoe4457rof13au20fb
2018-05-24 13:57:20 [Thread-1] DEBUG clientLogger - Receive data {"command":"REGISTER_SUCCESS","info":"register success for
user1"}
2018-05-24 13:57:20 [Thread-1] INFO  clientLogger - Register successfully to server localhost:8001
2018-05-24 13:57:20 [Thread-1] INFO  clientLogger - Close client and login with parameters:
2018-05-24 13:57:20 [Thread-1] INFO  clientLogger - -u user1 -s 25b82teejoe4457rof13au20fb -rh localhost -rp 8001
2018-05-24 13:57:20 [Thread-1] INFO  clientLogger - Login automatically after register success, accroding to head tutor's co
mment
2018-05-24 13:57:20 [Thread-1] INFO  clientLogger - send login to server with user=user1 secret=25b82teejoe4457rof13au20fb
2018-05-24 13:57:20 [Thread-1] DEBUG clientLogger - Receive data {"command":"LOGIN_SUCCESS","info":"login successfully as us
er [user1]"}
```

User1 relogin on 8002 (user1 login successfully, 8001 and 8002 is consistent)



Testing Result

Result as expected.

Server can join at any time

Test case

1. start the very first server

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
```

2. register a user at this server and remember its secret.

```
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
```

3. Quit client of step 2
4. start a new server connecting to server 8001

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
```

5. Login user1 at the new server (8002) by replace `$secret` of below script

```
java -jar ActivityStreamerClient.jar -u user1 -s vl3et80v8mmn3ho6dm93hjqgg0
-rp 8002 -rh localhost
```

Expected Result

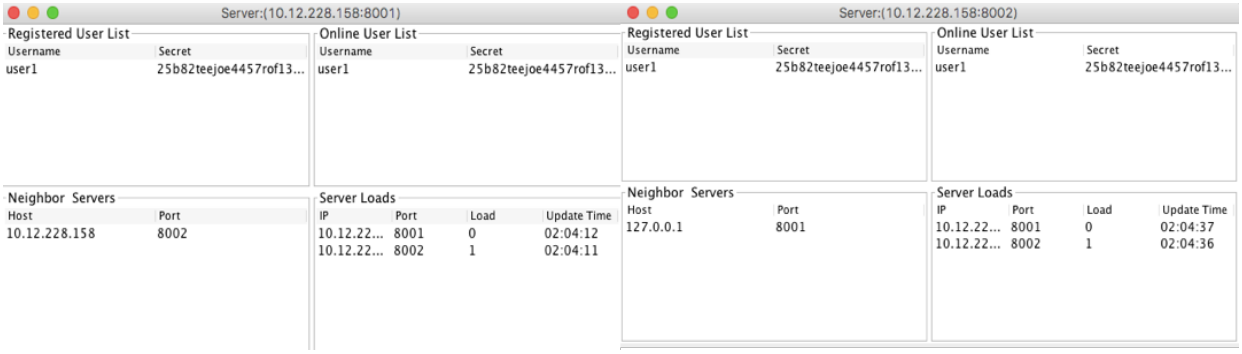
- user1 should login successfully at new server (8002) and all data of 8002 should be synced with 8001
- From test case Message ensure we can also see that:

> user A is online at the time T, when a activity is sent by some other user B and A loses its connection it can receive this message.
>
> When user A reconnects to any server of this system, it can also receive this lost message.

Screentshot:

Snapshot of register success and auto login with given secret

Snapshot of user1 relogin on 8002 (user1 login successfully, 8001 and 8002 is consistent)



Testing Result

Result as expected.

Unique Register

Test case

1. start several servers, say 3

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
```

1. register user1 at server 8001

```
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
```

1. try to register user1 at another server, say 8002

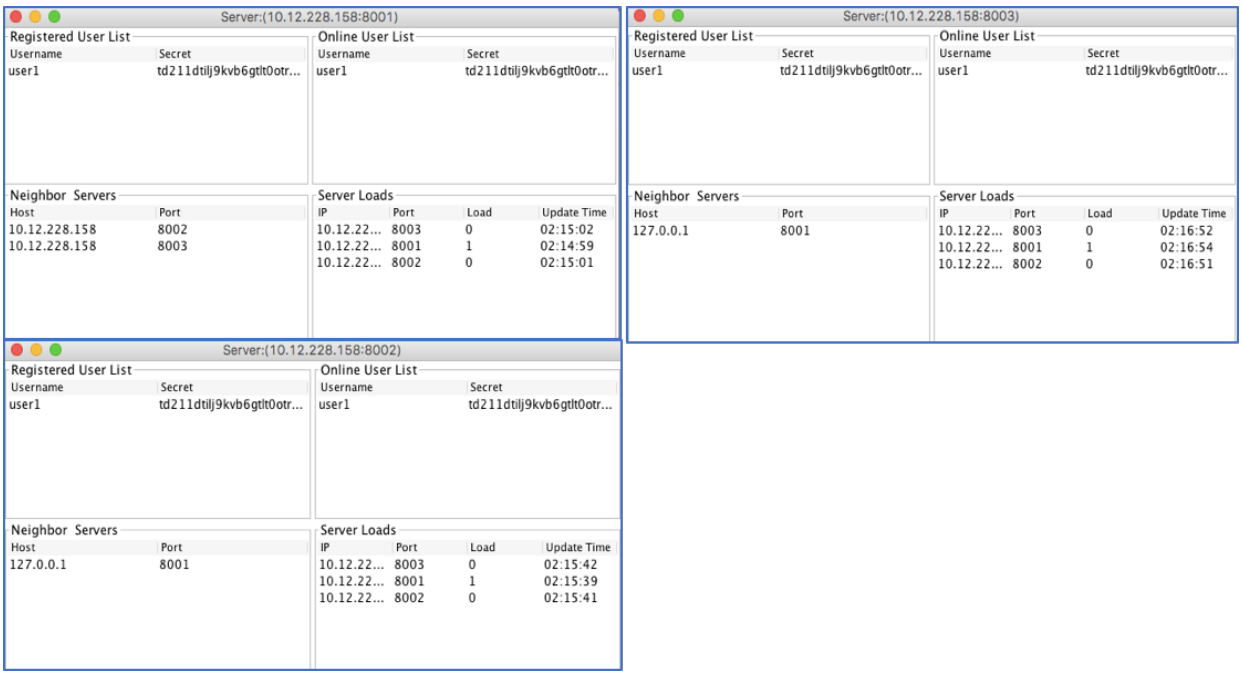
```
java -jar ActivityStreamerClient.jar -u user1 -rp 8002 -rh localhost
```

Expected Result

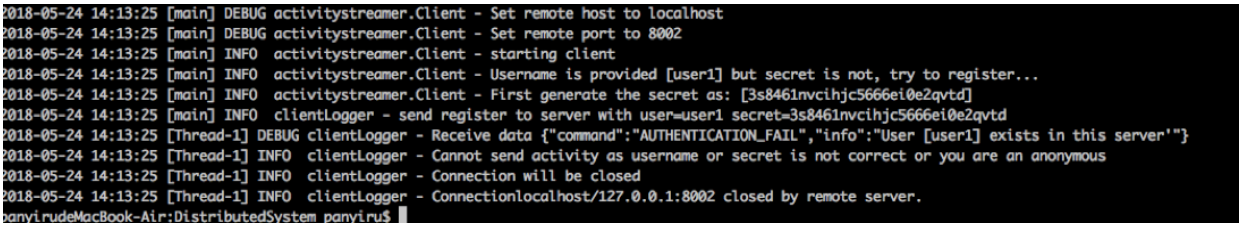
- the registration of step 3 (server 8002) will fail with error like "user already exists".

Screentshot

Snapshot of 3 servers' GUI



Snapshot of error message (user1 already exists in server)



Testing Result

Result as excepted.

Message ensure

Test case

In order to simulate message loss case, let us start servers with a parameter to ***delay*** the reconnection function.

1. Start 4 servers with `time_before_reconnect=10000 (10 seconds)`

```
# start the very first server, which will be terminated
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
# start other servers
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001 -time_before_reconnect 10000
java -jar ActivityStreamerServer.jar -lh localhost -lp 8003 -s abc -rh
localhost -rp 8001 -time_before_reconnect 10000
java -jar ActivityStreamerServer.jar -lh localhost -lp 8006 -s abc -rh
localhost -rp 8001 -time_before_reconnect 10000
```

2. Connect 3 clients to 3 different servers

Note: Please record the secret of user1 for future use

```
# Keept the secret after register successfully
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
# you can just run below 2 clients and no need to record their secretrs
java -jar ActivityStreamerClient.jar -u user2 -rp 8002 -rh localhost
java -jar ActivityStreamerClient.jar -u user3 -rp 8003 -rh localhost
```

1. Terminate server 8001 and send a message from user2 within 10 seconds

- Click ***Close*** icon in *server UI* or press ***CTRL+C*** in *command line* (user 1 will lose connection)
- Send message `{"a":1}` from user2.
- Wait for reconnection happens (10 seconds)

4. Reconnect user1 to any working server, let's say 8006

Replace `$secret` of below script with the secret from step 2.

```
java -jar ActivityStreamerClient.jar -u user1 -s $secret -rp 8006 -rh
localhost
```

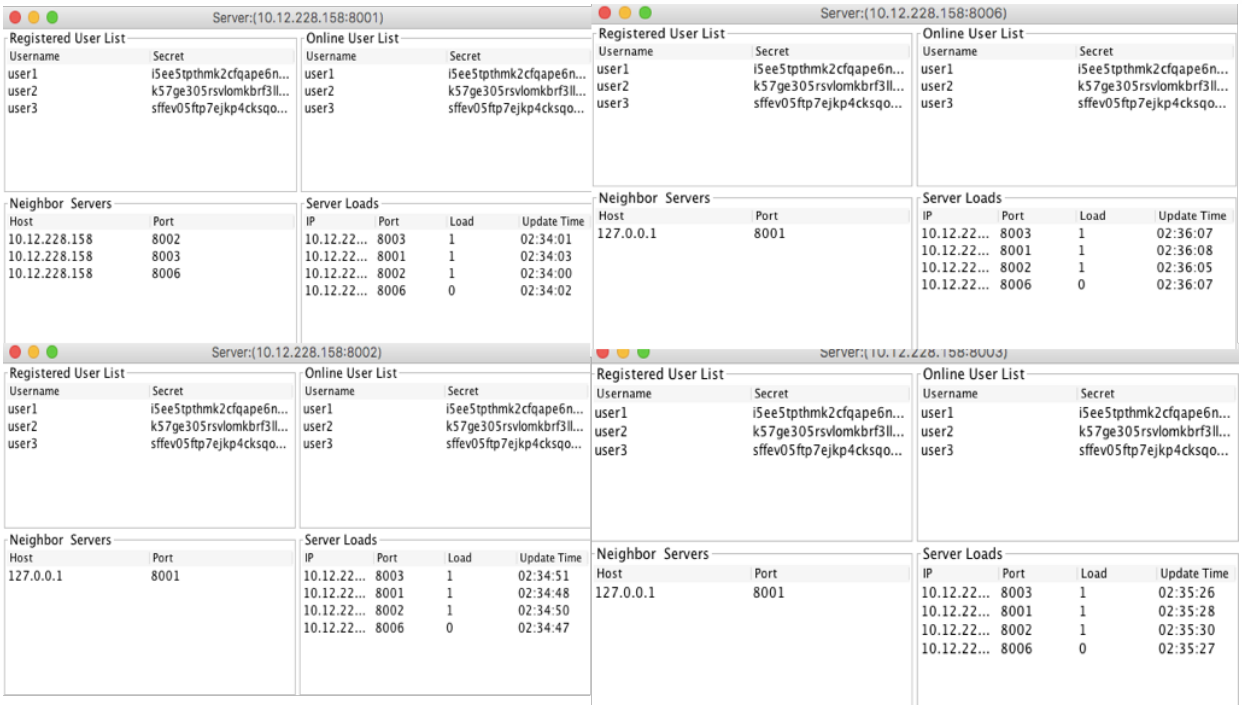
Expected Result

- user3 will receive the activity of user2 after reconection is done (about 10 seconds after disconnection)
- user1 will receive the activity of user2 after relogin to server 8006

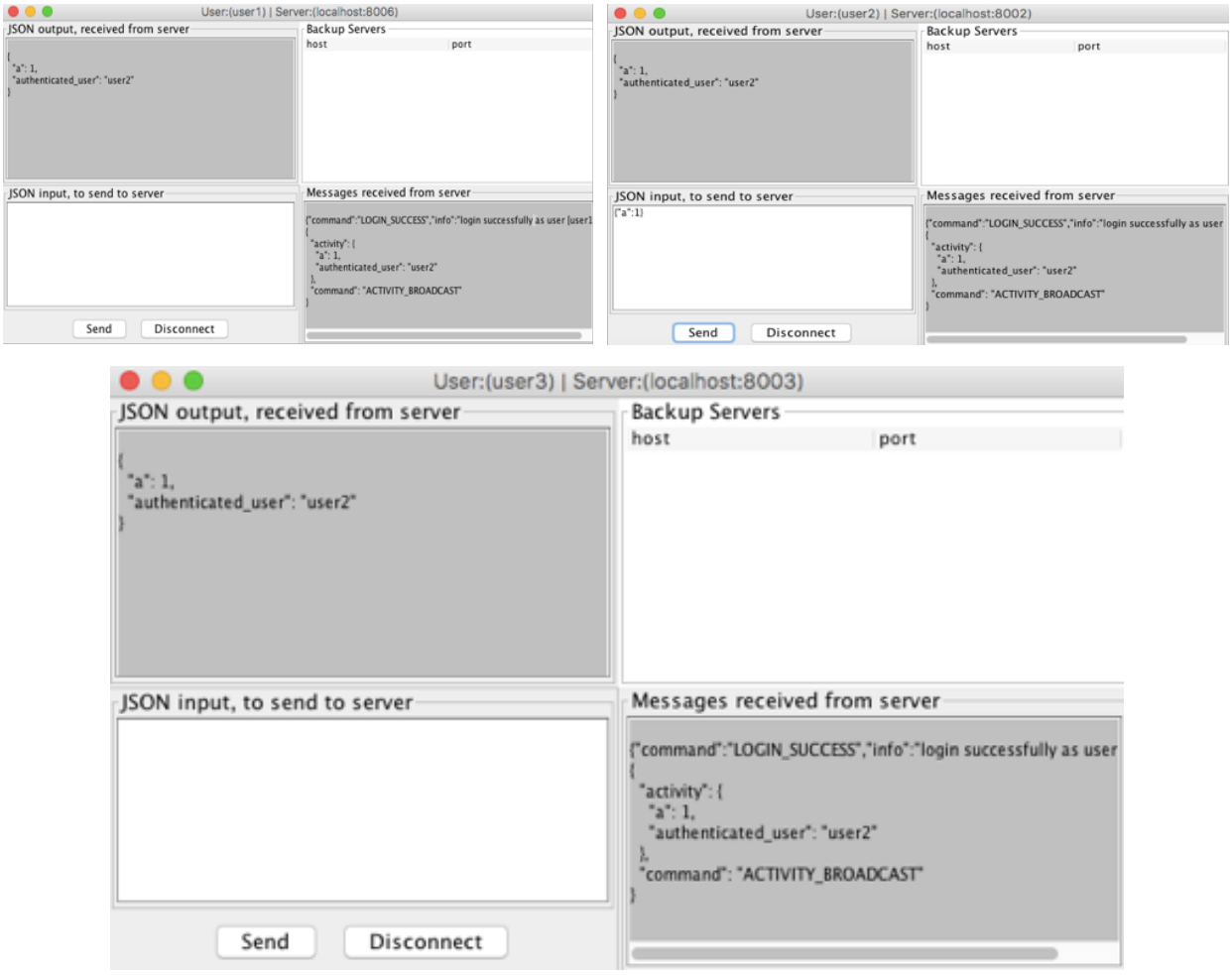
> user A is online at the time T, when a activity is sent by some other user B and A loses its connection it can receive this message.
>
> When user A reconnects to any server of this system, it can also receive this lost message.

Screentshot

clients login on 8001,8002, 8003 respectively



clients after reconnection(user1, user2, user3 all received activity from user2)



Testing Result

Result as excepted.

Message order

In order to simulate message disorder case, let us use a ***telnet session*** to simulate a ***server*** and make the order checking period a littler longer with `activity_check_interval=10000`. Fake messages will be broadcasted by the telnet server with a hooker ***"backTime"*** to set the send time of fake messages to be a time in the past.

> 'timeBack' field is a back door used for this kind of testing. If that field exists in an ActivityBroadcast message, then set the `sendTime` of this activity to `currentTimeInMillis() - timeBack`

Operations

1. Start 1 server with `activity_check_interval=10000 (10 seconds)`

```
java -jar ActivityStreamerServer.jar -activity_check_interval 10000 -lh
localhost -lp 8001 -s abc
```

2. Start a normal client connecting to server 1

```
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
```

3. Start a terminate and using telnet to simulate a client in following steps

- start telent session

```
telnet localhost 8001
```

- paste below string to authenticate this "server" with server 8001

```
{"command":"AUTHENTICATE","serverId":"serverId01","secret":"abc","host":"lo
calhost","port":8002}
```

- Broadcast 2 "fake" activities (**within 10 seconds**) by pasting below 2 string **separately(one by one)** into telnet session to simulate disordered message.

You can ignore the message telnet session receives. All of them are used by real server to sync data.

Message 1: a "fake" message that was sent 0 second ago

```
{"id":0,"activity":
{"message_num":2,"authenticated_user":"user2"},"isDelivered":false,"command
":"ACTIVITY_BROADCAST","timeBack":0}
```

Message 2: a "fake" message that was sent 10 seconds ago, which is early than preious one.

```
{"id":0,"activity":
{"message_num":1,"authenticated_user":"user2"},"isDelivered":false,"command
":"ACTIVITY_BROADCAST","timeBack":10000}
```


Expected Result

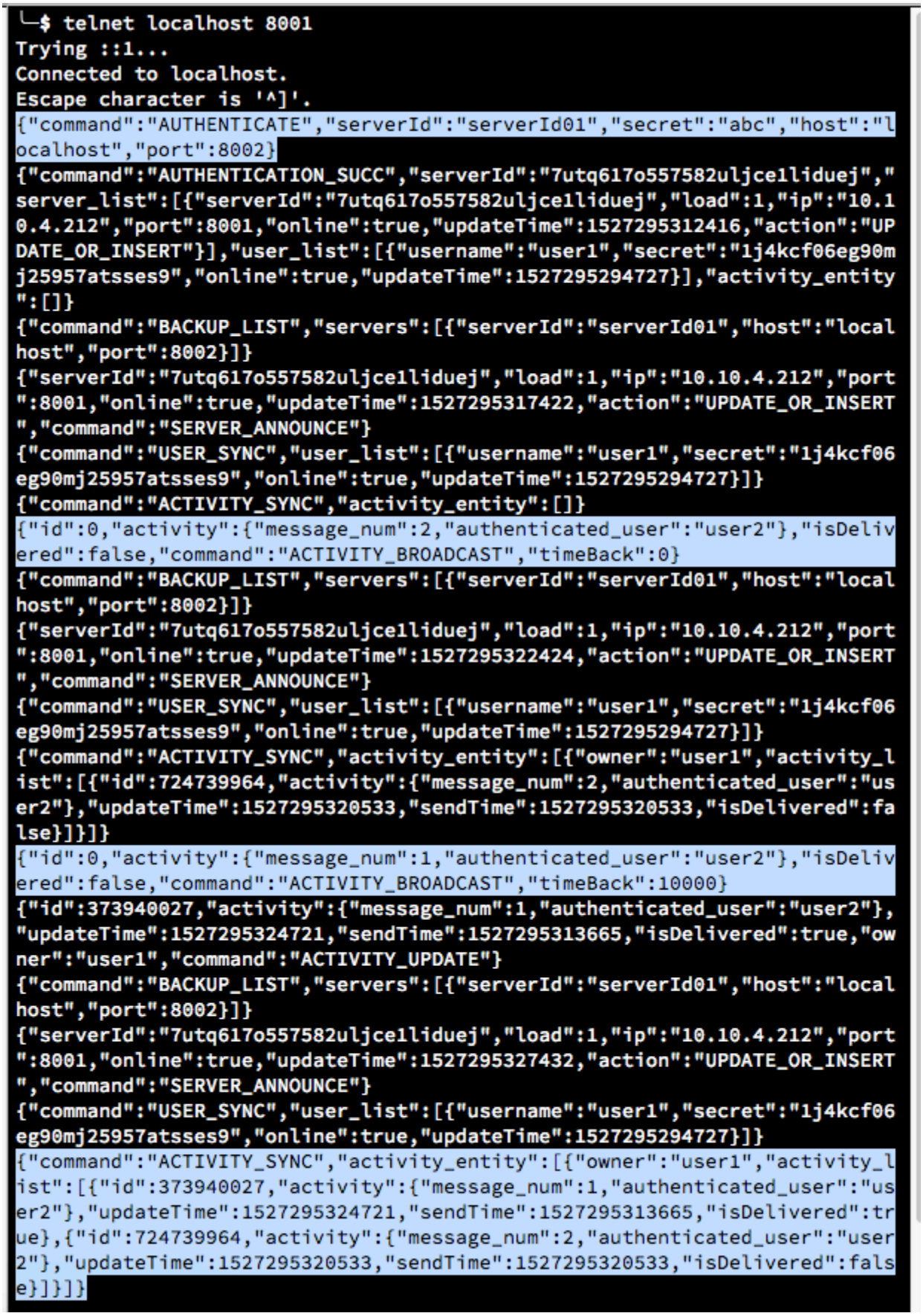
- After waiting ***10-20*** seconds, user1 (normal client with GUI) will receive 2 activities in order (message_num=1 first and then message_num=2) separately.

In real server, this order checking period can be relately shorter, like 0.5 or 1 second.

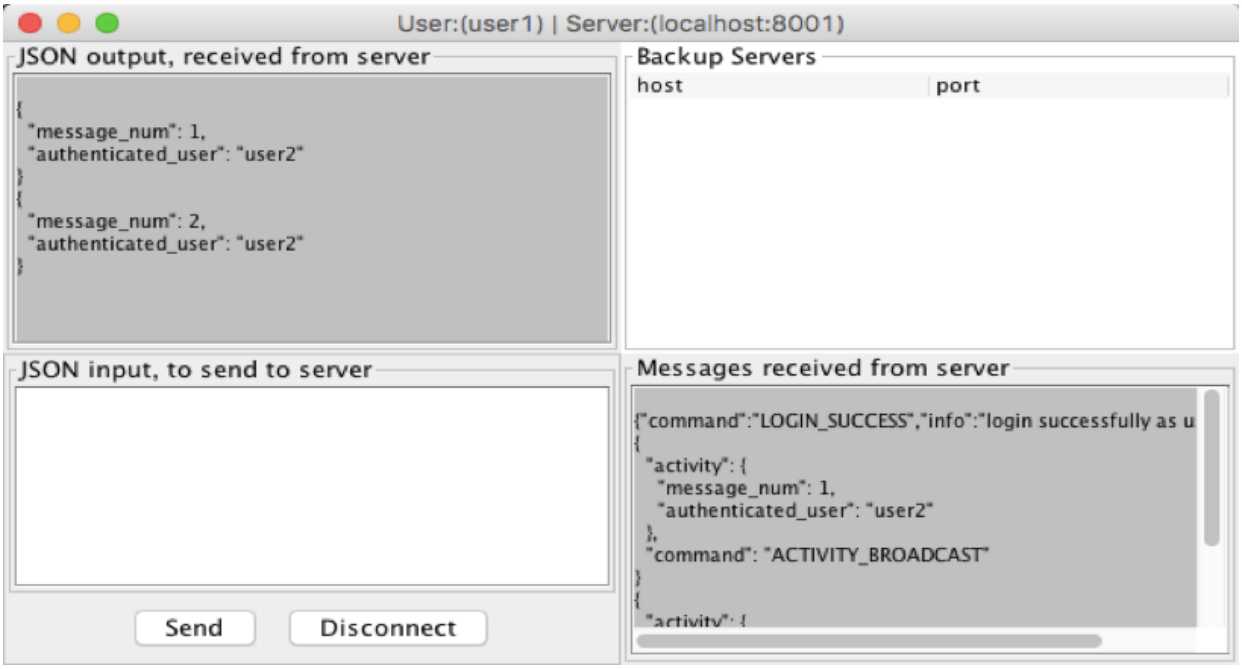
Screentshot

Telnet session input (in white, you can ignore other information, they are sync message from server)

The 3rd white block shows the ordered message list, in which the first is the one with message_num=1



Messages user1 received (message_num1 is before message_num 2)



Testing Result

Result as excepted.

Load balancing

Operations

1. start 2 servers

```
java -jar ActivityStreamerServer.jar -lh localhost -lp 8001 -s abc
java -jar ActivityStreamerServer.jar -lh localhost -lp 8002 -s abc -rh
localhost -rp 8001
```

2. Register and login 2 clients both to server 8001

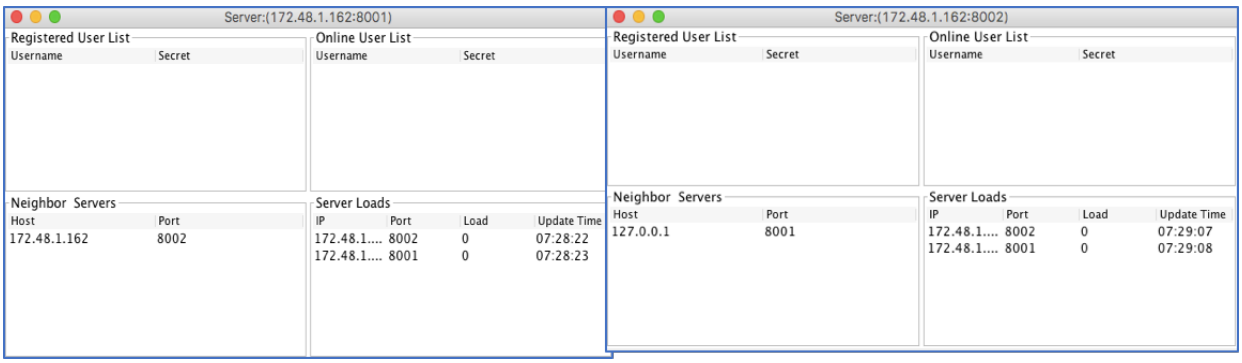
```
java -jar ActivityStreamerClient.jar -u user1 -rp 8001 -rh localhost
java -jar ActivityStreamerClient.jar -u user2 -rp 8001 -rh localhost
```

Expected Result

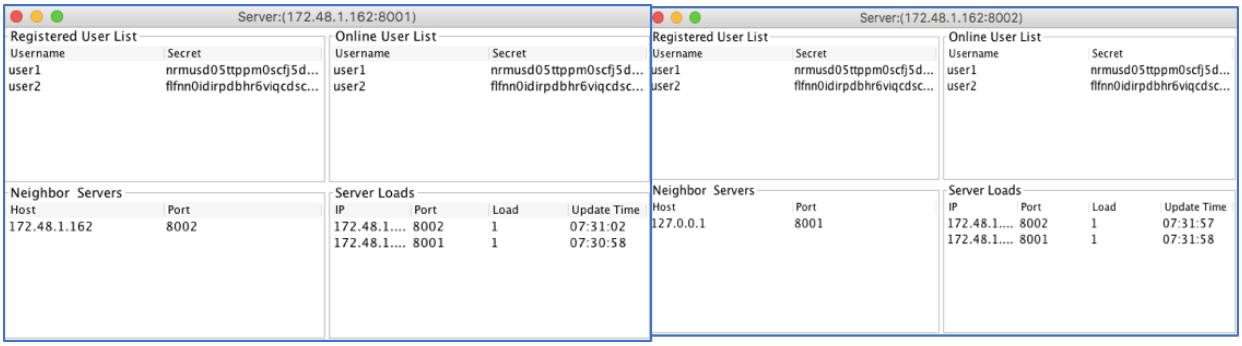
- user2 will be redirected to server 8002

Screentshot

Starting 2 servers



After two clients login(load of each server has been changed to 1)



Testing Result

Result as excepted.