COMP90015
Distributed Systems
Project 2

Group Name: The Walking Head

1. Introduction

The aim of this project is to build a multi-server system which not only has the required functionalities such as client registration / login or activity-objects broadcasting but also achieves high stability, availability, consistency and flexibility. This document covers several important aspects of the system, including system assumptions, failure models, concurrency issues and scalability details.

2. System Assumptions

In this model, there are 4 assumptions, which are listed as followed:

a) The system is based on the "full-mesh" communicating model. Any server will be connected to every one of the other servers.

b) If the entire system has been perfectly divided into two same-sized parts due to some unexpected reasons, human interference is needed to reestablish the network construction.

c) There is a global clock in this system can provide absolute global time for every server.

d) If a server has lost all connections to other servers, it will be considered dead and all other servers will remove the information of it, even if it is still running.

3. Server Failure Models

In the last model in project 1, the servers were assumed to be forever on. And if a server ever fails (quits or crashes), there was little to do to keep the entire system running. As for the new model, a number of measurements have been implemented to provide the system with the capability of running properly even if some servers were down. The measurements are shown below:

a) Every server is equipped with a message buffer to store data temporarily, the capacity of which is approximately the size of data to be transmitted between servers in 15 seconds. In detail, if server A loses server B, then a confirmation needs to be done to checking whether the problem is caused by server B or the connection between server A and server B. And During the confirmation time, all the data from server A to server B will be temporarily stored in the message buffer. If the problem is confirmed to be from the connection, then a new link between server A and server B will be formed, and all the messages stored in the buffer will be sent to server B in the right sequence. If the problem is confirmed to be from server B, then all the messages stored in the buffer will be discarded.

b) Every server is equipped with a timer component. If server A has not heard from server B for at least 10 seconds, then a CONNECTION LOSS (B) message will be sent from server A to all other servers to have them check their connection states with server B, which might lead to two results. The first result is that if server C's connection with server B is still on, then server C will send a CONNECTION SOUND (B) message to server A, and the reconnect mechanism on server A will form a new connection to server B. The second result is that if none of the other servers is able to communicate with server B, then there will be a lot of CONNECTION LOSS (B) messages. And if the number of the loss messages reach a certain limit, server B will be considered to be down, and all other servers will cut off their connections with server B and remove the related information of server B from their server lists.

4. Availability

As for the availability analysis of this system, it could be divided into three aspects: messages sent from clients to servers, messages sent from servers to clients, and messages multicast between servers. And the above three aspects are explained in detail as followed:

a) Messages sent from clients to servers
As the system only allows one client to establish connection with one server, this aspect can be simplified as how to keep the messages available between one server and a number of clients. To achieve this goal, it is important to solve the concurrency issues that may appear. In particular, when two or more than two clients register at the same server at the same time, the server should adopt some mechanism to avoid conflicts. In this system, the servers are using a kind of thread lock called synchronized in Java to deal with the concurrent requests, which allows the server to handle the requests in sequence. In this way, the system can ensure that all clients can send messages at any time.

b) Messages sent from servers to clients
Suppose that a server has successfully received messages from other servers and wants to broadcast the information to its clients. If the server runs smoothly, it just needs to traverse all the connections between its clients and itself and send the messages directly. However, if the server crashes unexpectedly suddenly, the messages will not be sent, and the clients will not receive the information eventually.

c) Messages multicast between servers
There is no doubt that the message multicast between servers is a crucial and complicated step to guarantee high availability of this system. First of all, as this system adopts full mesh connection model, every server has a list to record all other connected servers. Consequently, if all connections are stable, this server

can spread the messages to other servers one by one. However, for instance, if one connection from server A to another server B interrupts due to network outage over 10 seconds, the server A which aims to send messages will send a CONNECTION_LOSS command to other servers to detect whether they still connect with server B. If those connections exist, server A will reconnect to server B and send the messages which server A has cached for 15s to it. Otherwise, the system will give server B extra 10 seconds to be online again.

For example, as Figure 4-1 shows, when server 1 has lost the connection with server 3 over 10s, it then sends a 'CONNECTION_LOSE' command to server 2. Server 2 finds the connection between itself and server 3 then returns 'CONNECTION_FOUND' command. After that, server 1 reconnects to server 3 and sends the messages cached during former 15s to it.

For example, as Figure 4-1 shows, when server 1 has lost the connection with server 3 over 10s, it then sends a 'CONNECTION_LOSE' command to server 2. Server 2 finds the connection between itself and server 3 then returns 'CONNECTION_FOUND' command. After that, server 1 reconnects to server 3 and sends the messages cached during former 15s to it.

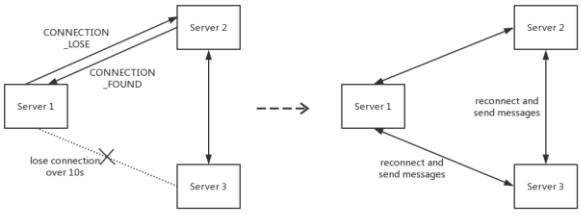


Figure 4-1

On other hand, as Figure 4-2 shows, if both server 1 and server 2 lose connections with server 3, the system will wait for another 10s. If server 3 can successfully get back during this period, it will reconnect to both server 1 and server 2 along with receiving related messages.

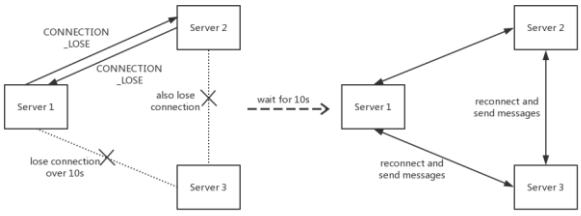


Figure 4-2

5. Consistency

The consistency of the whole system is also of great importance which should be achieved as high as possible. This following part discusses issues on how to balance the load of clients among all servers, how to preserve the delivery order of messages, and how to keep the system consistent when a new server joins in the network.

a) Balancing the load of clients

To balance the load of clients among the servers, the system adopts the mechanism that makes the servers multicast a server announcement every 5 seconds. Namely, the announcement contains each server's load information. Moreover, if a server receives the announcement, it will record the id and hostname of that server. Consequently, when a new client connects to one server, the server will check the load status of each of the other servers. If any of the other servers has a load of 2 clients less than other servers, the new client will be redirected to that server. As a result, the load of the clients is balanced between all servers.

b) Preservation of the message delivery order

Before discussing the approach to preserve the delivery order of messages, it is necessary to assume that the system can only ensure the order of messages that one server receives from its clients. The reason is that the project cannot change the skeleton of clients, hence it is impossible to be entirely sure that the messages one server receives are in the same order as the clients send them. Besides, as this system adopts TCP protocol to build connections, the messages won't miss, and the delivery order can also be guaranteed when connections still exist. Therefore, the only issue needs to be considered here is how to ensure the consistency of messages after reconnecting to the network. As is presented in Figure 5-1, server 2 just comes back after network issues, and server 1 passes the cache data (15s), user list and server list to it. Afterward, server 2 records all information received from server 1, and it tries to connect with server 3. However, during this period, a new client registered on server 3. Thus after server 2 successfully builds connection with server 3, server 3 will also passes its cache data (15s) which contains the newest information to it; and when server 2 receiving these data, it compares the newest data with its local data by the delivery order, then removes duplicated messages and adds new messages.

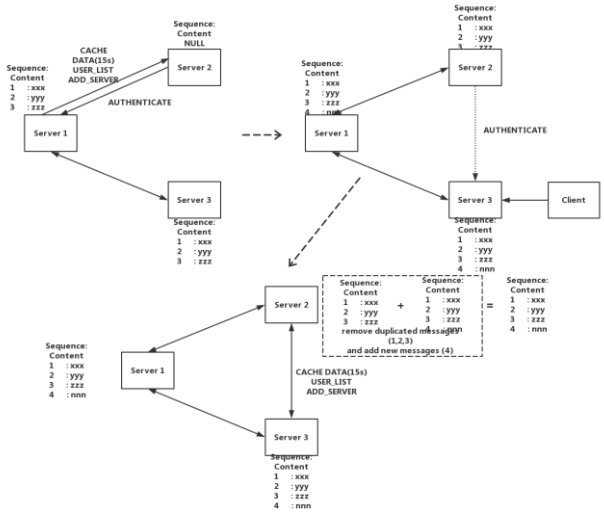


Figure 5-1

c) Keeping the system consistent when a new server joins

in the network

The system should have the ability to allow new servers to join in the network at any time. In particular, when a new server joins the network, it must firstly establish a connection to a targeted server which has been one of the members of the running servers. After building the connection, the targeted server will send a package which contains the information of cached data (15s), user list and server list to the new one, which allows the new server to know about the IP addresses and port numbers of all other servers in the network. After that, the new server can establish the connections with other servers. During the process, if the new server gets any piece of new information from other servers, it will ignore the duplicated information it has already known and update its server list and user list the with new information. The update will stop until the new server finishes establishing connections with all other servers in the network.

Appendix A

This appendix will present all JSON command used in the system:

1. AUTHENTICATE:

Sent from one server to another always and only as the first message when connecting.

Example:

```
{
 "command": "AUTHENTICATE",
 "senderID": "……",
 "already-connect": n/m,
 "secret": "……"
}
```

The m is the number of servers existing in the network, n is the number of connections that has built.

The senderID is the ID of the server who sends this command.

Receiver replies with:

- **AUTHENTICATE_FAIL** if the secret is incorrect
- **AUTHENTICATE_SUCCESS** if the secret is correct
- **INVALID_MESSAGE** if anything is incorrect about the message, or if the server had already

If anything other than authentication succeeded, then the connection is closed immediately after sending the response.

2. AUTHENTICATE_FAIL:

Sent by the server when either a server fails to authenticate or when a client fails to login (see later).

Example:

```
{
 "command": "AUTHENTICATION_FAIL",
 "info": "the supplied secret is incorrect: ……"
}
```

After sending an AUTHENTICATION_FAIL the server will close the connection.

A client or server that receives AUTHENTICATION_FAIL must similarly close the connection.

3. INVALID_MESSAGE:

A general message used as a reply if there is anything incorrect about the message that was received. This can be used by

both clients and servers.

Examples:

{

“command”: “INVALID_MESSAGE”,

“info”: “the received message did not contain a command”

}

{

“command”: “INVALID_MESSAGE”,

“info”: “JSON parse error while parsing message”

}

The exact content of the info field is not specified, but should be something that indicates clearly why the message was deemed invalid.

After sending an INVALID_MESSAGE the sender will close the connection. Receivers of such a message must therefore close the connection as well.

4. AUTHENTICATE_SUCCESS:

{

“command”: “AUTHENTICATE_SUCCESS”,

“senderID”: “……”

}

The senderID is the ID of the server who sends this command.

No reply if the authentication succeeded.

5. ADD_SERVER:

{

“command”: “ADD_SERVER”,

“id”: “……”,

“load”: “……”,

“hostname”: “……”,

“port”: “……”,

“senderID”: “……”,

“sequenceNumber”: “……”,

}

The id is the identification of the server in this command.

The load is related load number of that server in this command.

The hostname is the IP address of targeted server, port is the port of targeted server.

The senderID is the ID of the server who sends this command.

The sequenceNumber is the sequence number of this command is the local server.

Server received the command will try to connect with the server through its hostname and port.

6. LOGIN:

Sent from a client to a server.

Example:

{

"command": "LOGIN",

"username": "……",

"secret": "……"

}

A username must be present. The value of username may be anonymous in which case no secret field should be given (it should be ignored). Otherwise a secret must be given.

The server replies with:

- **LOGIN_SUCCESS** only if the server recognizes the combination of username and secret in its local storage.
- **LOGIN_FAILED** in cases where the secret does not match that in the local storage, for the supplied username, or when the username is not found (not registered, see later).
- **INVALID_MESSAGE** in any case where the message is incorrect

7. LOGIN_SUCCESS:

Sent from server to client to indicate that the client successfully logged in.

Example:

{

"command": "LOGIN_SUCCESS",

"info": "logged in as user xxx"

}

The server will follow up a LOGIN_SUCCESS message with a REDIRECT message if the server knows of any other server with a load at least 2 clients less than its own.

8. REDIRECT:

Sent from a server to a client, to request the client to reconnect to a new server.

Example:

{

“command”: “REDIRECT”,

“hostname”: “123.456.78.9”,

“port”: 1234

}

After sending a REDIRECT message the server will close the connection.

After the client receives a REDIRECT message it must close the connection and make a new connection to the system, presumably to the server as given in the message. The client starts the protocol afresh.

9. LOGIN_FAILED:

Sent from server to client to indicate that the client did not successfully log in.

Example:

{

“command”: “LOGIN_FAILED”,

“info”: “attempt to login with wrong secret”

}

After sending a LOGIN_FAILED the server will close the connection.

10. LOGOUT:

Sent from client to server to indicate that the client is closing the connection.

Example:

{

“command”: “LOGOUT”

}

The client will close the connection after sending. The server will close the connection after receiving.

11. ACTIVITY_MESSAGE:

Sent from client to server when publishing an activity object.

Example:

{

“command”: “ACTIVITY_MESSAGE”,

“username”: “…...”,

“secret”: “……”,

“activity”: {.....}

}

The actual activity object is not shown above.

The server will respond with:

- AUTHENTICATION_FAIL if the username is not anonymous or if the username and secret do not match the logged in the user, or if the user has not logged in yet. After sending such a message the connection is closed.
- INVALID_MESSAGE if the message is incorrect in any way, and close the connection.

If the request succeeds, then the server will broadcast the activity object to all servers and they in turn will broadcast to all connected clients. The activity will be processed (see later) before being broadcast.

12. SERVER_ANNOUNCE

Broadcast from every server to every other server (between servers only) on a regular basis (once every 5 seconds by default).

{
"command": "SERVER_ANNOUNCE",
"senderID": "…",
"sequenceNumber": "……"
"load": 5,
"hostname": "128.250.13.46",
"port":3570,
"timestamp": "{"date": {"year": "2018", "month": "05", "day": "26"}, "time": {"hour": "10", "minute": "49", "second":
"41", "nano": "6350000005"}}"
}

Servers respond with an INVALID_MESSAGE if the message is incorrect or if the message is received from an unauthenticated server (i.e. on a connection that has not yet authenticated with the server secret). Then the server will close connection.

The senderID is the ID of the server who sends the command.

The SN is the sequence number of the message received in the server.

The timestamp is the local time of the server.

The load is the number of clients currently connected to the server. The hostname and port are the address of the server

13. ACTIVITY_BROADCAST (For Clients):

Message broadcast from each server to its clients, that contains a processed activity object.

{
"command": "ACTIVITY_BROADCAST",

“activity”: {……}

}

The senderID is the ID of the server who sends the command.

The SN is the sequence number of the message received in the server.

The timestamp is the local time of the server.

Servers respond with an INVALID_MESSAGE if the message is incorrect or if the message is received from an unauthenticated server (i.e. on a connection that has not yet authenticated with the server secret). Then the server will close connection.

14. ACTIVITY_BROADCAST (For Servers):

Message broadcast between servers, that contains a processed activity object.

{

“command”: “ACTIVITY_BROADCAST”,

“senderID”: “……”,

“sequenceNumber”: “……”,

“timestamp”: “{“date”: {“year”: “2018”, “month”: “05”, “day”: “26”}, “time”: {“hour”: “10”, “minute”: “49”, “second”: “41”, “nano”: “6350000005”}}”,

“activity”: {……}

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The timestamp is the local time of the server.

Servers respond with an INVALID_MESSAGE if the message is incorrect or if the message is received from an unauthenticated server (i.e. on a connection that has not yet authenticated with the server secret). Then the server will close connection.

15. REGISTER:

Sent from client to server when the client wishes to register a new username.

Example:

{

“command”: “REGISTER”,

“username”: “……”,

“secret”: “……”

}

The client selects the secret that it wishes to register with.

The server replies with:

- REGISTER_FAILED if the username is already known (registered) by any server in the system. Connection is closed.
- REGISTER_SUCCESS if the username was not already known by any server in the system, and now the user can login using the username and secret.
- INVALID_MESSAGE if anything is incorrect about the message, or if receiving a REGISTER message from a client that has already logged in on this connection. Connection is closed by server.

16. REGISTER_SUCCESS:

Sent by server to client to indicate that an attempt to register a username and client has succeeded.

Example:

{

"command": "REGISTER_SUCCESS",

"info": "register success for xxx"

}

17. REGISTER_FAILED:

Sent by server to client to indicate that an attempt to register a username and client has failed.

Example:

{

"command": "REGISTER_FAILED",

"info": "xxx is already registered with the system"

}

The connection is closed by the server after sending this message.

18. LOCK_REQUEST:

Broadcast from a server to all other servers (between servers only) to indicate that a client is trying to register a username with a given secret.

Example:

{

"command": "LOCK_REQUEST",

"username": "xxx",

"secret": "……",

"senderID": "……",

“sequenceNumber”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

A server that receives this message will:

- Broadcast a LOCK_DENIED to all other servers (between servers only) if the username is already known to the server with a different secret.
- Broadcast a LOCK_ALLOWED to all other servers (between servers only) if the username is not already known to the server. The server will record this username and secret pair in its local storage.
- Send an INVALID_MESSAGE if anything is incorrect about the message or if it receives a LOCK_REQUEST from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

Using the above mechanism, all servers currently in the system will learn about register attempts.

19. LOCK_DENIED:

Broadcast from a server to all other servers (between servers only), if the server received a LOCK_REQUEST and to indicate that a username is already known, and the username and secret pair should not be registered.

Example:

{

“command”: “LOCK_DENIED”,

“username”: “…….”,

“secret”: “…….”,

“senderID”: “……”,

“sequenceNumber”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

When a server receives this message, it will remove the username from its local storage only if the secret matches the associated secret in its local storage.

The server will:

- Send an INVALID_MESSAGE if anything is incorrect about the message or if it receives a LOCK_DENIED from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

20. LOCK_ALLOW:

Broadcast from a server to all other servers if the server received a LOCK_REQUEST and the username was not already

known to the server in its local storage.

Example:

{

“command”: “LOCK_ALLOWED”,

“username”: “……”,

“secret”: “……”,

“senderID”: “……”,

“sequenceNumber”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The server will:

- Send an INVALID_MESSAGE if anything is incorrect about the message or if it receives a LOCK_ALLOWED from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

21. CANCEL_LOCK:

Broadcast from a server to all other servers and the servers that received this command need to delete the user information in its local storage.

{

“command”: “CANCEL_LOCK”,

“username”: “……”,

“secret: “……”,

“senderID”: “……”,

“sequenceNumber”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The server will delete related user information after receiving.

22. CONNECT_LOSS:

Broadcast from a server to all other servers when the server loses one connection.

{

“command”: “CONNECT_LOSS”,

“senderID”: “……”,

“sequenceNumber”: “……”,

“lossID”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The lossID is the ID of the server who has lost connection.

Servers who receives this command will try to detect whether the connection between itself and the server with lossID still exists. Besides, the servers will also record the lossID and senderID of the server who has been disconnected, and when all senders found that server has been disconnected, this server will eliminate the information of that server and broadcast SERVER_ELIMINATE command.

The server will broadcast:

- CONNECTION_LOSS: if the server cannot find the connection.
- CONNECTION_FOUND: if the server finds the connection.

23. CONNECT_FOUND:

Broadcast from a server to all other servers when the server finds one connection.

{

“command”: “CONNECT_FOUND”,

“senderID”: “……”,

“sequenceNumber”: “……”,

“lossID”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The lossID is the ID of the server who has lost connection.

The servers who receive this command will try to detect whether the connection between itself and the server with lossID still exists. If the connection exists, the receiving server will ignore this command. Otherwise, it will try to build connection with the server with lossID again. And after building the connection, it will pass user list and server list to that server.

24. SERVER_ELIMINATE:

Broadcast from a server to all other servers to announce that they need to eliminate the server information declared in this message.

{

“command”: “SERVER_FLIMINATE”,

“senderID”: “……”,

“sequenceNumber”: “……”,

“lossID”: “……”

}

The senderID is the ID of the server who sends the command.

The sequenceNumber is the sequence number of the message received in the server.

The lossID is the ID of the server who has lost connection.

The server will eliminate related server information declared in this message after receiving.

25. USER_LIST:

Sent from a server to another server to pass the user list to it.

{

“command”: “USER_LIST”,

“senderID”: “……”,

“sequenceNumber”: “……”,

“userlist”: {“xxx”: “……”, “yyy”: “……”}

}

The senderID is the ID of the server who sends this command.

The sequenceNumber is the sequence number of this command is the local server.

The userlist contains all usernames and related secrets which registered in the local server.

The information is encrypted by MD5 to keep secure.

The targeted server will send this message to the new server after building the connection.

Appendix B

This appendix will demonstrate how the system runs:

1. Create a new server

The CMD can use this command: -lh (local host); -lp (local port); -rh (remote host); -rp (remote port); -a (activity interval in milliseconds); -s (secret).

Example:

```
G:\ds_project2\code\bin>java -Djava.ext.dirs=../lib activitystreamer.Server -lh localhost -lp 3780 -s xxxxxx
reading command line options
starting server
Initialising Connection
listening for new connections on 3780
using activity interval of 5000 milliseconds
doing activity
doing activity
```

This server's local host number is localhost; local port is 3780; secret is xxxxxx.

2. Create two new servers

Build the network with two new servers, each server broadcasts server announcement every 5s.

Example:

Server 1's local host number is localhost; local port is 3780; secret is xxxxxx.

Server 2's local host number is localhost; local port is 3781; secret is xxxxxx; connected to Server 1.

3. Register a new client on Server 1

Register a new client on Server 1.

Example:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -u test1 -rh localhost -rp 3780
20:51:02.620 [main] INFO  activitystreamer.Client reading command line options
20:51:02.630 [main] INFO  activitystreamer.Client starting client
20:51:02.660 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
20:51:02.850 [main] INFO  activitystreamer.client.ClientSkeleton This is your password: o3m2rvjeikn42ih3ogkb9m7d7v
20:51:02.870 [main] INFO  activitystreamer.client.ClientSkeleton register success for test1
20:51:02.880 [main] INFO  activitystreamer.client.ClientSkeleton logged in as user test1
```

The username of this client is test1, its secret is 3pdcp9phoibvp087kn5q69nm3c.

4. Login this client on Server 2

Use the same username and secret to login this system on Server 2.

Example:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -u test1 -rh localhost -rp 3781 -s o3m2rvjeikn42ih
3ogkb9m7d7v
20:51:22.096 [main] INFO  activitystreamer.Client reading command line options
20:51:22.106 [main] INFO  activitystreamer.Client starting client
20:51:22.136 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
20:51:22.164 [main] INFO  activitystreamer.client.ClientSkeleton logged in as user test1
```

5. Add a new server

Under the condition that Server 1 and Server 2 have been in the network and User test1 has registered, adding a new server Server 3.

Example:

```
G:\ds_project2\code2.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Server -lh localhost -lp 3782 -rh localhost -rp 3780 -s xxxxxx
reading command line options
starting server
Initialising Connection
listening for new connections on 3782
outgoing connection: localhost/127.0.0.1:3780
using activity interval of 9000 milliseconds
{"command":"AUTHENTICATION_SUCCESS","senderId":"9v92if4nleeruru2ba98hf15vb"}
{"command":"USERLIST","senderId":"9v92if4nleeruru2ba98hf15vb","sequenceNumber":7,"users":[{"test1":"o3m2rvjeikn42ih3ogkb9m7d7v"}]}
{"command":"ADD_SERVER","senderId":"9v92if4nleeruru2ba98hf15vb","sequenceNumber":8,"id":"idv5vcn5thna9u8s9bsv5t689p","hostname":"127.0.0.1","port":3781,"load":0}
outgoing connection: /127.0.0.1:3781
{"command":"SERVER_ANNOUNCE","senderId":"9v92if4nleeruru2ba98hf15vb","sequenceNumber":9,"hostname":"localhost","port":3780,"load":0}
{"command":"AUTHENTICATION_SUCCESS","senderId":"idv5vcn5thna9u8s9bsv5t689p"}
{"command":"SERVER_ANNOUNCE","senderId":"idv5vcn5thna9u8s9bsv5t689p","sequenceNumber":6,"hostname":"localhost","port":3781,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"9v92if4nleeruru2ba98hf15vb","sequenceNumber":10,"hostname":"localhost","port":3780,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"idv5vcn5thna9u8s9bsv5t689p","sequenceNumber":7,"hostname":"localhost","port":3781,"load":0}
doing activity
```

Server 3's local host number is localhost; local port is 3782; secret is xxxxxx; connected to Server 1.

6. Login this client on Server 3

Login the User test1 on Server 3 to test whether the user list has been passed to the new server.

Example:

Client:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -u test1 -rh localhost -rp 3782 -s o3m2rvjeikn42ih
3ogkb9m7d7v
20:51:41.642 [main] INFO  activitystreamer.Client reading command line options
20:51:41.652 [main] INFO  activitystreamer.Client starting client
20:51:41.682 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
20:51:41.702 [main] INFO  activitystreamer.client.ClientSkeleton logged in as user test1
```

Server 3:

```
incomming connection: /127.0.0.1:11217
{"command":"SERVER_ANNOUNCE","senderId":"9v92if4nleeruru2ba98hf15vb","sequenceNumber":11,"hostname":"localhost","port":3780,"load":0}
{"command":"LOGIN","username":"test1","secret":"o3m2rvjeikn42ih3ogkb9m7d7v"}
logged in as user test1
{"command":"SERVER_ANNOUNCE","senderId":"idv5vcn5thna9u8s9bsv5t689p","sequenceNumber":8,"hostname":"localhost","port":3781,"load":0}
{"command":"LOGOUT"}
connection closed to /127.0.0.1:11217
```

As the example shows, the Client test1 can successfully login on Server 3.

7. Network Outage and Repair of one server

Firstly, building a network of three servers (Server 1, Server 2 and Server 3).

Example:

```
G:\ds_project2\code2.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Server -lh localhost -lp 3780 -s xxxxxx
reading command line options
starting server
Initialising Connection
listening for new connections on 3780
using activity interval of 9000 milliseconds
incomming connection: /127.0.0.1:11667
{"command":"AUTHENTICATE","senderId":"truqddjqtpma728pnm8oqajrbs","connect":0,"secret":"xxxxxx"}
{"command":"SERVER_ANNOUNCE","senderId":"truqddjqtpma728pnm8oqajrbs","sequenceNumber":0,"hostname":"localhost","port":3781,"load":0}
doing activity
incomming connection: /127.0.0.1:11668
{"command":"AUTHENTICATE","senderId":"hql0gl6c0osmg2pkn34ijodk77","connect":0,"secret":"xxxxxx"}
{"command":"SERVER_ANNOUNCE","senderId":"hql0gl6c0osmg2pkn34ijodk77","sequenceNumber":0,"hostname":"localhost","port":3782,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"truqddjqtpma728pnm8oqajrbs","sequenceNumber":2,"hostname":"localhost","port":3781,"load":0}
```

Server 1's port number is 3780; Server 2's port number is 3781; Server 3's port number is 3782.

Secondly, shut down the connection between Server 1 and Server 2 by this command:

TELNET localhost 780 ——> {"command":"CUT","n":1}.

Now, the connection is closed:

```
closing connection /127.0.0.1:11667
```

Thirdly, the system will reconnect Server 1 with Server 2 because Server 3 is still connecting with Server 2, and Server 1 will pass its user list, server list and cached data to Server 2.

Example:

```
outgoing connection: /127.0.0.1:3781
{"command":"AUTHENTICATION_SUCCESS","senderId":"truqddjqtpma728pnm8oqajrbs"}
{"command":"SERVER_ANNOUNCE","senderId":"truqddjqtpma728pnm8oqajrbs","sequenceNumber":3,"hostname":"localhost","port":3781,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"truqddjqtpma728pnm8oqajrbs","sequenceNumber":4,"hostname":"localhost","port":3781,"load":0}
```

As this example shows, this system allows servers drop at any time and supports them to come back.

8. Network Outage and Repair of one server (User List Testing)

The main steps are similar as 7. However, while the connection between Server 1 and Server 2 is partitioned, creating a new client on Server 1:

Client:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -u test1 -rh localhost -rp 3780
21:47:33.195 [main] INFO  activitystreamer.Client reading command line options
21:47:33.205 [main] INFO  activitystreamer.Client starting client
21:47:33.245 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
21:47:33.465 [main] INFO  activitystreamer.client.ClientSkeleton This is your password: gb2qkl3vqcjclt1dbokvrpceas
```

Server 1:

```
incomming connection: /127.0.0.1:11812
{"command":"REGISTER","username":"test1","secret":"gb2qkl3vqcjclt1dbokvrpceas"}
{"command":"LOCK_ALLOWED","senderId":"9enlerrcdneo7lcihj7iko5819","sequenceNumber":4,"username":"test1","secret":"gb2qkl3vqcjclt1dbokvrpceas"}
{"command":"LOCK_ALLOWED","senderId":"dima037ulh8um9qr66vikm38kd","sequenceNumber":5,"username":"test1","secret":"gb2qkl3vqcjclt1dbokvrpceas"}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"9enlerrcdneo7lcihj7iko5819","sequenceNumber":5,"hostname":"localhost","port":3782,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"dima037ulh8um9qr66vikm38kd","sequenceNumber":6,"hostname":"localhost","port":3781,"load":0}
connection /127.0.0.1:11812 closed with exception: java.net.SocketException: Connection reset
```

Then, after Server 2 reconnecting with Server 1, login this client on Server 2:

Client:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -u test1 -rh localhost -rp 3781 -s gb2qkl3vqcjclt1
dbokvrpceas
21:48:31.774 [main] INFO  activitystreamer.Client reading command line options
21:48:31.783 [main] INFO  activitystreamer.Client starting client
21:48:31.813 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
```

Server 2:

```
connection closed to localhost/127.0.0.1:3780
{"command":"CONNECT_FOUND","senderId":"9enlerrcdneo71cihj7iko5819","sequenceNumber":9,"lossId":"19aehojgg7e01tr761kebbcnot"}
outgoing connection: /127.0.0.1:3780
{"command":"AUTHENTICATION_SUCCESS","senderId":"19aehojgg7e01tr761kebbcnot"}
{"command":"SERVER_ANNOUNCE","senderId":"19aehojgg7e01tr761kebbcnot","sequenceNumber":11,"hostname":"localhost","port":3780,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"19aehojgg7e01tr761kebbcnot","sequenceNumber":12,"hostname":"localhost","port":3780,"load":0}
received exception, shutting down
{"command":"SERVER_ANNOUNCE","senderId":"19aehojgg7e01tr761kebbcnot","sequenceNumber":14,"hostname":"localhost","port":3780,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"9enlerrcdneo71cihj7iko5819","sequenceNumber":10,"hostname":"localhost","port":3782,"load":0}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"19aehojgg7e01tr761kebbcnot","sequenceNumber":15,"hostname":"localhost","port":3780,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"9enlerrcdneo71cihj7iko5819","sequenceNumber":12,"hostname":"localhost","port":3782,"load":0}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"19aehojgg7e01tr761kebbcnot","sequenceNumber":16,"hostname":"localhost","port":3780,"load":0}
{"command":"SERVER_ANNOUNCE","senderId":"9enlerrcdneo71cihj7iko5819","sequenceNumber":13,"hostname":"localhost","port":3782,"load":0}
doing activity
```

As is shown in this example, after Server 2 coming back, it can receive all messages by order that it missed when network outage.

9. Restrict Register Time

This system can maintain that a given username can only be registered once over the server network.

Firstly, creating a server Server 1 and registering a client test2 on it:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -rh localhost -rp 3780 -u test2
22:54:03.605 [main] INFO  activitystreamer.Client reading command line options
22:54:03.615 [main] INFO  activitystreamer.Client starting client
22:54:03.645 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
22:54:03.835 [main] INFO  activitystreamer.client.ClientSkeleton This is your password: 704rv8dhud6idqa09e8gg2j0f
22:54:03.855 [main] INFO  activitystreamer.client.ClientSkeleton register success for test2
22:54:03.855 [main] INFO  activitystreamer.client.ClientSkeleton logged in as user test2
22:54:06.355 [AWT-EventQueue-0] INFO  activitystreamer.client.ClientSkeleton closing connection localhost/127.0.0.1:3780
```

Secondly, adding a new server Server 2:

```
incomming connection: /127.0.0.1:12409
{"command":"AUTHENTICATE","senderId":"ottgge8od7shrfekif160ercv3","connect":0,"secret":"xxxxxx"}
{"command":"SERVER_ANNOUNCE","senderId":"ottgge8od7shrfekif160ercv3","sequenceNumber":0,"hostname":"localhost","port":3781,"load":0}
```

Thirdly, login test2 on Server 2:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -rh localhost -rp 3781 -u test2
22:54:19.363 [main] INFO  activitystreamer.Client reading command line options
22:54:19.373 [main] INFO  activitystreamer.Client starting client
22:54:19.403 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
22:54:19.603 [main] INFO  activitystreamer.client.ClientSkeleton This is your password: fn8aebqb5fjnj8ild40rjvt0i2
22:54:19.623 [main] ERROR activitystreamer.client.ClientSkeleton Exception:test2 is already registered with the system
22:54:19.623 [main] ERROR activitystreamer.client.ClientSkeleton connection closed to localhost/127.0.0.1:3781
```

As is shown in the example, the user can only be registered once over the network.

10. Activity Broadcast on time

Firstly, building a network of two servers (Server 1 and Server 2).

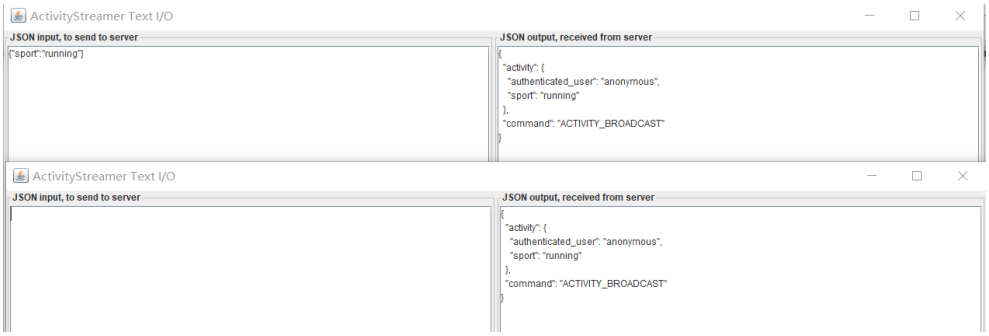
Example:

```
G:\ds_project2\code2.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Server -lh localhost -lp 3780 -s xxxxxx
reading command line options
starting server
Initialising Connection
listening for new connections on 3780
using activity interval of 9000 milliseconds
incomming connection: /127.0.0.1:12137
{"command":"AUTHENTICATE","senderId":"sttsv4rjpvjbdr5vklre8730o9","connect":0,"secret":"xxxxxx"}
{"command":"SERVER_ANNOUNCE","senderId":"sttsv4rjpvjbdr5vklre8730o9","sequenceNumber":0,"hostname":"localhost","port":3781,"load":0}
```

Secondly, login a client on Server 1 and send a message {"sport":"running"} at time T:

```
incomming connection: /127.0.0.1:12138
{"command":"LOGIN","username":"anonymous"}
logged in as user anonymous
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"sttsv4rjpvjbdr5vklre8730o9","sequenceNumber":2,"hostname":"localhost","port":3781,"load":1}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"sttsv4rjpvjbdr5vklre8730o9","sequenceNumber":3,"hostname":"localhost","port":3781,"load":1}
{"command":"ACTIVITY_MESSAGE","username":"anonymous","activity":{"sport":"running"}}
```

Meanwhile (maybe has little time difference, hard to ensure), login another client on Server 2 at time T:



As is presented in this example, the activity messages can be guaranteed to arrive at all clients on time.

11. Load Balance

Firstly, building a network of two servers (Server 1 and Server 2).

Example:

```
G:\ds_project2\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Server -lh localhost -lp 3780 -s xxxxxx
reading command line options
starting server
Initialising Connection
using activity interval of 5000 milliseconds
listening for new connections on 3780
incomming connection: /127.0.0.1:12388
{"command":"AUTHENTICATE","senderId":"il28osp4spcvd2nf2c4bua05g6","connect":0,"secret":"xxxxxx"}
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":0,"hostname":"localhost","port":3781,"load":0}
```

Secondly, login two clients on Server 1:

```
incomming connection: /127.0.0.1:12389
{"command":"LOGIN","username":"anonymous"}
logged in as user anonymous
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":2,"hostname":"localhost","port":3781,"load":0}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":3,"hostname":"localhost","port":3781,"load":0}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":4,"hostname":"localhost","port":3781,"load":0}
doing activity
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":5,"hostname":"localhost","port":3781,"load":0}
doing activity
incomming connection: /127.0.0.1:12390
{"command":"LOGIN","username":"anonymous"}
logged in as user anonymous
{"command":"SERVER_ANNOUNCE","senderId":"il28osp4spcvd2nf2c4bua05g6","sequenceNumber":6,"hostname":"localhost","port":3781,"load":0}
doing activity
```

Thirdly, login the third client on Server 1, the system lets this client to login on Server 2:

```
G:\DS\code3.0\bin>java -Djava.ext.dirs=../lib activitystreamer.Client -rh localhost -rp 3780
22:48:24.275 [main] INFO  activitystreamer.Client reading command line options
22:48:24.285 [main] INFO  activitystreamer.Client starting client
22:48:24.314 [main] INFO  activitystreamer.client.ClientSkeleton Successful connect to server
22:48:24.334 [main] INFO  activitystreamer.client.ClientSkeleton logged in as user anonymous
22:48:24.574 [Thread-1] INFO  activitystreamer.client.ClientSkeleton this server is overload, you can retry on server 12
7.0.0.1:3781
22:48:24.574 [Thread-1] DEBUG activitystreamer.client.ClientSkeleton connection closed to localhost/127.0.0.1:3780
```

As is shown above, the system automatically balances the loads over 2.