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 reconnect 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/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:

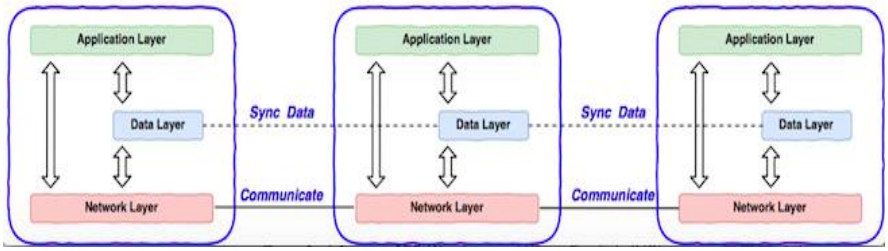

Figure 1 Layered Structure

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.

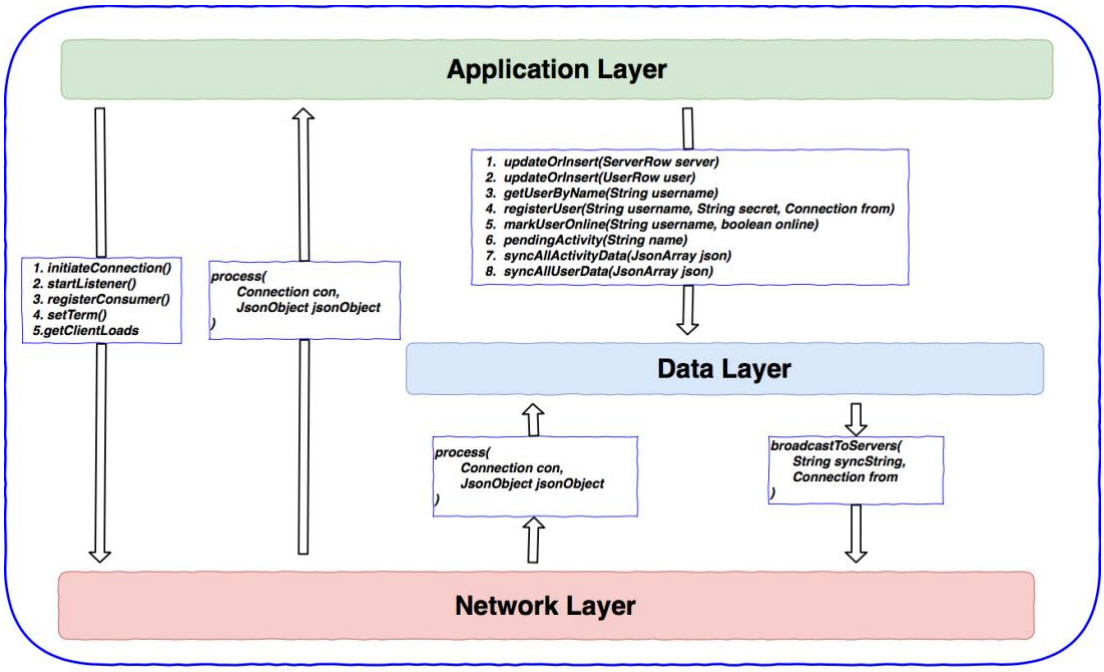

Figure 2 Communication between layers

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. Server authenticat-ion 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.

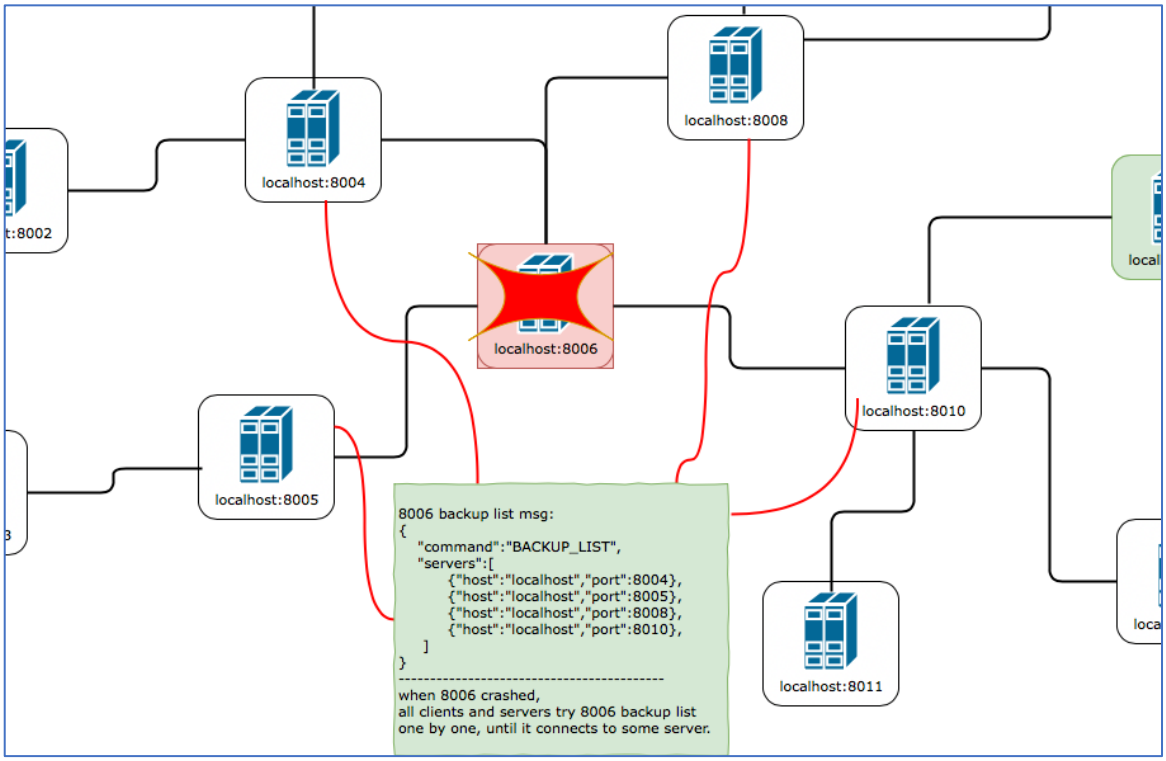

Figure 3 Crash scenario

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 two 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.

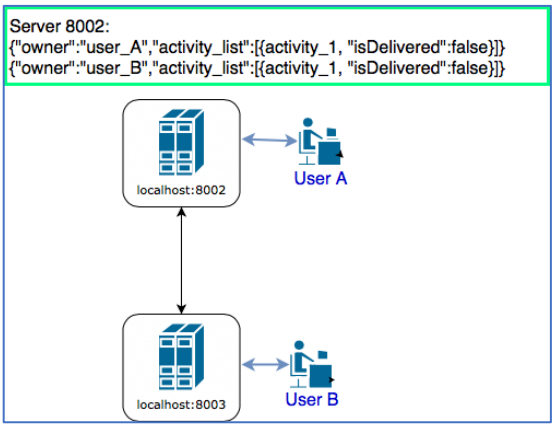

Figure 4 Delivery Ensure - 1

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.

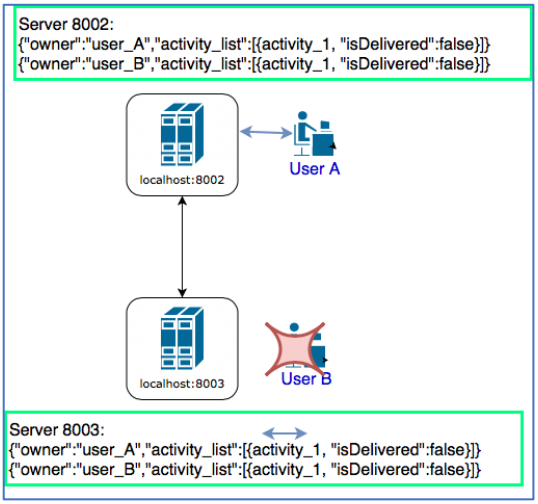

Figure 5 Delivery Ensure - 2

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.

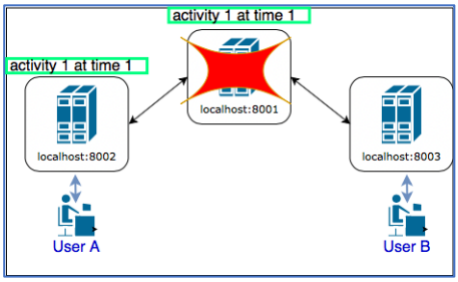

Figure 6 Ensure Delivery - 1

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.

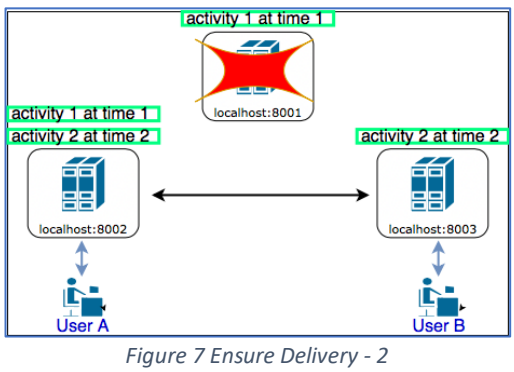

Figure 7 Ensure Delivery - 2

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.

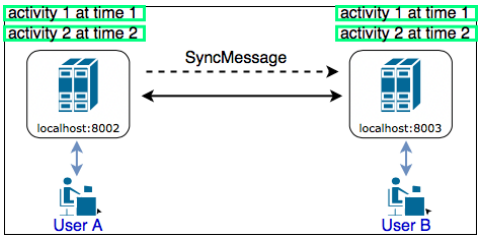

Figure 8 Ensure Delivery - 3

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

- 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.