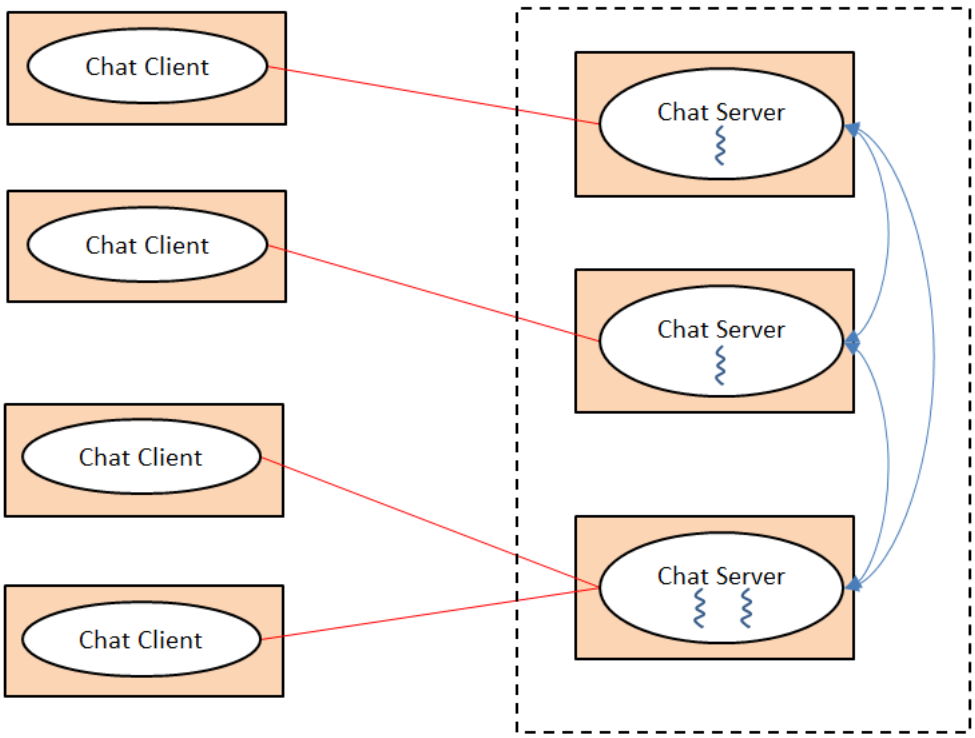
Distributed Systems
COMP90015 2016 SM2
Project 1 - Multi-Server Chat System

1. Synopsis

The assignment is about creating a "chat" application. The chat system consists of two main distributed components: *chat servers* and *chat clients*, which may run on different hosts in the network. The architecture of the system is presented in the following diagram.



Chat clients are Java programs that can connect to a single chat server; this server can be any of the servers available in the system. Chat clients can be used to send requests for creating, deleting, joining and quitting a chat room. They can also be used to send requests to see the list of available chat rooms in the system and the list of client identities currently connected to a given chat room. Finally, they can be used to send chat messages to other chat clients connected to the same chat room.

Chat servers are Java programs accepting multiple incoming *TCP connections* from chat clients. There are multiple servers working together to serve chat clients. The number of servers is fixed and does not change once the system is active. Each server is responsible for managing a subset of the system's chat rooms. In particular, a server manages only those chat rooms that were created locally after receiving a request from a client. In order to join a particular chat room, clients must be connected to the server managing that chat room. As a result, clients are redirected between servers when a client wants to join a chat room managed by a different server. Chat servers are also responsible for broadcasting messages received from clients to all other clients connected to the same chat room.

In this assignment, the chat client program is provided to you and you need to design and implement the chat server program.

2. Chat Client

The chat client executable jar file is provided to you, there is no need to implement your own. A chat client is executed as follows:

```
java -jar client.jar -h server_address [-p server_port] -i identity [-d]
```

Square brackets ([]) are used to indicate optional arguments.

- *server_address* corresponds to the ip address or hostname of the server to which the client is connecting to.
- *server_port* corresponds to the port in which the server is listening for incoming client connections. The default *server_port* is *4444*.
- *identity* corresponds to the client's identity (i.e. username) which must be unique in the entire system. If the identity already exists, the server will send a message to the client indicating the error and will then close the connection. If the identity doesn't exist, a connection is established and the server places the client on its *MainHall* chat room. More details on this are given later.
- The *-d* option can be used to start the client in debug mode. This means that all the received and sent messages will be printed on the standard output.

The client user interface is command-line based and reads input from the *standard input*. Each line of input is *terminated by a new line* and is interpreted by the client as either a *command* or a *message*. If the line of input starts with a hash character "#" then it is interpreted as a command, otherwise it is interpreted as a message that should be broadcasted by the chat server to other clients in the same chat room. The list of *commands* supported by chat clients is as follows:

```
#list
#who
#createroom roomid
#join roomid
#deleteroom roomid
#quit
```

Pressing Ctrl-C terminate chat clients and works similar to **#quit**.

Here is an example of how a client session may look:

```
E:\>java -jar client.jar -h localhost -p 4444 -i Adel
Adel moves to MainHall-s1
[MainHall-s1] Adel> #createroom comp90015
[MainHall-s1] Adel> Room comp90015 is created.
[MainHall-s1] Adel> Adel moves from MainHall-s1 to comp90015
[comp90015] Adel> Maria moves from MainHall-s1 to comp90015
[comp90015] Adel> We have a new comer to the room, Hi Maria!
[comp90015] Adel> Maria: Hi everybody!
[comp90015] Adel> Maria: Bye
[comp90015] Adel> Maria moves from comp90015 to MainHall-s1
[comp90015] Adel> #deleteroom comp90015
[comp90015] Adel> Room comp90015 is deleted.
[comp90015] Adel> Adel moves from comp90015 to MainHall-s1
[MainHall-s1] Adel> Maria moves from MainHall-s1 to jokes
[MainHall-s1] Adel> #quit
```

```
[MainHall-s1] Adel> Adel quits
```

A client may receive messages at anytime, and these will be written to the standard output as soon as they are received, even if the client is in the middle of typing a message.

3. Chat Server

A chat server listens for incoming connection requests from clients. It is capable of accepting and managing multiple client connections simultaneously.

A chat server must store locally (there is no need to use persistent storage, data may be stored in memory):

- A list of all the existing chat rooms in the system, including those managed by other servers.
- A list of all the client identities currently connected to that server.

Communications in a chat server can be categorized into two main groups: *Communication with chat clients* and *Communication with other chat servers*.

3.1. Communication with chat clients

A chat server is primarily responsible for managing all the clients currently connected to it and for broadcasting chat messages. To achieve this, TCP connections between clients and servers are established and remain active throughout the duration of the client-server interaction. That is, the connection is only closed when:

- The identity of the client already exists,
- The client quits the system,
- The client program is abruptly terminated (e.g. using Ctrl C),
- The client is redirected to a different server.

In any of above cases, it is the responsibility of the server to remove the identity of the client, the client's chat room (if any), associated thread(s), the TCP connection with client (if any), after sending the appropriate protocol message to the client (if required).

3.2. Communications with other chat servers

To maintain consistency among servers' data, such as ensuring the global uniqueness of client identities and chat rooms, servers need to communicate with each other to reach consensus. To achieve this, servers use a protocol that consists of two phases:

- The lock (voting) phase, in which a coordinating server sends a lock request, **lockidentity** or **lockroomid**, to all other servers which in turn reply with their vote to either allow or deny the creation of the new identity/chat room, based on their local data (local list of client identities and list of chat rooms).
- The release (commit) phase, in which the coordinating server decides whether to allow (if all servers voted to allow) or to deny (if at least one server voted to deny) the creation of the new identity/chat room. Regardless of the result, the coordinating server notifies the result to all the other servers by sending them a **releaseidentity** or **releaseroomid** message.

Servers communicate with each other using TCP connections. There is no need for these connections to remain active throughout the lifetime of the system, and instead, they should be established and destroyed as required.

3.3. Chat Rooms

A *chat room* is a place in which clients can exchange chat messages. Each server initially has a chat room called MainHall-*serverid*. When a client initially connects to a server, it is placed in the MainHall chat room of that server.

The following rules regarding chat rooms must be enforced by a chat server:

- Each client is only allowed to be a member of one chat room at a time.
- It is the chat server's job to broadcast messages to all the chat clients in a room.
- Clients who are members of a chat room must be connected to the server managing that chat room.
- If a client creates a chat room, then that chat room will be managed by the server that the client is connected to.
- A chat client is able to owns one chat room at a time.
- Each server must maintain a list of globally available chat rooms and a list of its locally managed chat rooms.
- If a client joins a room managed by the server it is connected to, then the server simply places the client in the corresponding room. If the room is managed by another server, then the server sends a message to the client redirecting it to the server managing that chat room.
- Only the owner of a chat room (i.e., the client that created the chat room) can delete a chat room.
- If a client deletes a chat room, the members of the room are moved to the MainHall of the server. The server will send a message to all the corresponding clients informing them they have changed rooms.
- A chat room is deleted if the owner client quits or disconnects abruptly. The remaining members of the chat room are moved into the MainHall of the server.

The protocol (including all the messages and their format) that the chat server must follow is specified later in the project specification.

3.4. JSON Format

All communication in the system takes place by exchanging messages encapsulated in *JSON format* passing through *TCP connections*.

- All messages will be sent as JSON objects with a *newline* ("\n") to delimit objects.
- A JSON library must be used to marshal and unmarshal JSON objects. Do not implement JSON (un)marshaling yourself.
- All data written to the TCP connection must be UTF8 encoded.

3.5. Executing a chat server

Chat servers should be executable *exactly* as follows:

```
java -jar server.jar -n serverid -l servers_conf
```

- *serverid* is the name of the server,
- *servers_conf* is the path to a text file containing the configuration of servers.

Each line of the configuration file contains the following "tab" delimited data:

```
serverid    server_address  clients_port    coordination_port
```

- *serverid* is the name of the server,
- *server_address* is the ip address or hostname of the server,
- *clients_port* is the port used to communicate with clients,
- *coordination_port* is the port used by the server to communicate with other servers.

A sample configuration file looks like:

```
s1  localhost   4444    5555
s2  localhost   4445    5556
s3  192.168.1.2 4444    5000
```

- Each server will find its own confguiration in the configuration file using *serverid*.
- Use command line argument parsing (e.g. using the args4j library or your choice).
- Clients can connect to servers only after all the servers have been started.

4. Chat Protocol

The chat protocol specifies exactly how the client-server and server-server communication happen. Your solution must implement the protocol exactly as stated. Incomplete implementation and deviation from the protocol will result in dedcution of marks.

4.1. New Identity

The client sends a **newidentity** request with its **identity** immediately after connecting to the server.

Example:

```
{"type" : "newidentity", "identity" : "Adel"}
```

The identity must be an alphanumeric string starting with an upper or lower case character. It must be at least 3 characters and no more than 16 characters long.

If the identity is not in use by any of local clients of the server or is not locked by any other server, then server sends a **lockidentity** message to all the other servers.

Example:

```
{"type" : "lockidentity", "serverid" : "s1", "identity" : "Adel"}
```

serverid is the id of the server acquiring the lock and **identity** is the identity that should be locked.

The receiving server replies with a **lockidenity** message which contains an indication of whether the lock was allowed (the server does not have a client with that identity connected to it or the identity is not locked by other servers) or denied (the server has a client with the same identity connected to it or a lock for that identity).

Example of an allowed lock:

```
{"type" : "lockidenity", "serverid" : "s2", "identity" : "Adel", "locked" : "true"}
```

Example of a denied lock:

```
{"type" : "lockidenity", "serverid" : "s2", "identity" : "Adel", "locked" : "false"}
```

serverid is the id of a server sending the reponse. Note that the value of the **locked** field is a string as it is enclosed in quotes.

Once s1 (the coordinating server) has received all of the **lockidenity** replies, then it can either:

1) Allow the new identity to be created (if all the servers voted true) and place "Adel" into its MainHall and will reply to the client with the following message:

```
{"type" : "newidentity", "approved" : "true"}
```

2) Deny the creation of the new identity (if at least one server voted false) and will reply to the client with the following message:

```
{"type" : "newidentity", "approved" : "false"}
```

In either case (allow or deny) s1 sends a message releasing the lock to all of the other servers:

```
{"type" : "releaseidentity", "serverid" : "s1", "identity" : "Adel"}
```

When a server receives the **releaseidentity** message, it should release the identity, but only if the **serverid** of the lock request issuer (s1) is the same as the **serverid** of the release request issuer.

Finally, if the new identity was approved, s1 must broadcast a **roomchange** message to all the members in the MainHall-s1 room including the connecting client:

```
{"type" : "roomchange", "identity" : "Adel", "former" : "", "roomid" : "MainHall-s1"}
```

The **former** field contains the former chatroom identity showing that the client is moving from nowhere to the MainHall-s1.which is null in this case.

4.2. List

The client can ask for the list of chat rooms in the system with the command "#list". The command is sent to server as a JSON message:

```
{"type" : "list"}
```

The server replies with a list of all the chat rooms in the system:

```
{
"type" : "roomlist",
"rooms" : ["MainHall-s1", "MainHall-s2", "jokes"]
}
```

4.3. Who

The client can ask for the list of clients in the current chat room using the command "#who". Its JSON format is:

```
{"type" : "who"}
```

The server replies with the list of clients in the chat room:

```
{
"type" : "roomcontents",
"roomid" : "jokes",
"identities" : ["Adel","Chenhao","Maria"],
"owner" : "Adel"
}
```

4.4. Create Room

A connected client can create a chat room by using the command "#createroom *roomid*", if and only if the client is not the owner of another chat room. When a client successfully creates a room, it should automatically join the room.

The **roomid** must be an alphanumeric string starting with an upper or lower case character. It must be at least 3 characters and no more than 16 characters long.

If creation of the chat room fails, then the server sends the following message to the client:

```
{"type" : "createroom", "roomid" : "jokes", "approved" : "false"}
```

The client who creates the chat room becomes the owner of the chat room. The owner of the MainHall in each server is "" (empty string).

When the client with identity "Adel" passes the command "#createroom *jokes*", the client sends a JSON request to the server as:

```
{"type" : "createroom", "roomid" : "jokes"}
```

The server (s1) then sends a **lockroomid** message to all the other servers to lock the roomid:

```
{"type" : "lockroomid", "serverid" : "s1", "roomid" : "jokes"}
```

serverid corresponds to the id of the server acquiring the lock and **roomid** is the name of the chat room that is going to be created.

The receiving servers reply with a **lockroomid** indicating their vote. They reply with the following message if they approve the lock:

```
{"type" : "lockroomid", "serverid" : "s2", "roomid" : "jokes", "locked" : "true"}
```

or with the following message if they deny the lock:

```
{"type" : "lockroomid", "serverid" : "s2", "roomid" : "jokes", "locked" : "false"}
```

Once the coordinating server (s1) has received all of the **lockroomid** replies, it aggregates the votes and decides whether to create "jokes" or not.

It then sends a **releaseroomid** request to other servers:

```
{"type" : "releaseroomid", "serverid" : "s1", "roomid" : "jokes", "approved":"true"}
```

when all the servers allowed the lock, or,

```
"type" : "releaseroomid", "serverid" : "s1", "roomid" : "jokes", "approved":"false"}
```

when at least one server denied the lock.

Servers receiving a **releaseroomid** message with **"approved" : "true"** must then release the lock and record it as a new chat room with id "jokesâ that was created in server s1.

Finally, s1 replies to the client with the following message if the room was successfully created:

```
{"type" : "createroom", "roomid" : "jokes", "approved" : "true"}
```

s1 will also broadcast a **roomchange** message to the client and all the clients that are members of the chat room "Adel" was previously in:

```
{"type" : "roomchange", "identity" : "Adel", "former" : "former_room", "roomid" : "jokes"}
```

If the room is not successfully created, then s1 replies to the client with:

```
{"type" : "createroom", "roomid" : "jokes", "approved" : "false"}
```

4.5. Join Room

The client can join other rooms if he/she is not the owner of the current chat room by using the command "#join *roomid*", which in JSON format is represented as:

```
{"type" : "join", "roomid" : "jokes"}
```

Here, **roomid** is the name of destination room.

If the join is not successful (e.g. when joining non-existent chat room, owning a chat room, etc.) then the server will reply with a **roomchange** message with the same **former** and **roomid** values like:

```
{"type" : "roomchange", "identity" : "Maria", "former" : "jokes", "roomid" : "jokes"}
```

Here, **former** and **roomid** are the source and the destination chat rooms, repectively.

If the room is in the same server to which the client is connected, the server simply places the client in the new room and broadcasts a **roomchange** message to the members of the former chat room, to the members of the new chat room, and to the client joining the room.

```
{"type" : "roomchange", "identity" : "Maria", "former" : "MainHall-s1", "roomid" : "jokes"}
```

If the chat room is managed by a different server, then the server first replies to the client with a **route**

message redirecting it to another server. The server also removes the client from its list and broadcast a **roomchange** message to all the members of the former chat room.

```
{"type" : "route", "roomid" : "jokes", "host" : "122.134.2.4", "port" : "4445"}
```

Here, **host** and **port** values are the address and listening port of the server containing the **roomid** jokes.

Upon receiving the **route** message, the client initiates a **movejoin** request to the new server (e.g., s2).

```
{"type" : "movejoin", "former" : "MainHall-s1", "roomid" : "jokes", "identity" : "Maria"}
```

It is possible that the target room is deleted in the middle of a route change. If the **roomid** does not exist at the moment of receiving the **movejoin** message, then the server places the client in its MainHall chat room.

s2 accepts the requests and adds "Maria" to its clients list and places her in the jokes chat room. It also broadcasts a **roomchange** message to all the members of jokes. In addition, it replies to "Maria" with:

```
{"type" : "serverchange", "approved" : "true", "serverid" : "s2"}
```

After receiving the **serverchange** message, the client closes its connection with server s1.

4.6. Delete Room

If the client is the owner of the chat room, he/she can delete the room by using the command "#deleteroom *roomid*". In JSON format, it is sent by the client as:

```
{"type" : "deleteroom", "roomid" : "jokes"}
```

If not successfully deleted, s1 replies to the client with:

```
{"type" : "deleteroom", "roomid" : "jokes", "approved" : "false"}
```

If successfully deleted, s1 informs other servers by sending the message:

```
{"type" : "deleteroom", "serverid" : "s1", "roomid" : "jokes"}
```

The server will treat this as if all users are joining the MainHall. It broadcasts **roomchange** messages to all members of the deleted room showing that each member is moving from the deleted room to the MainHall. The server will also send **roomhange** messages per clint of the deleted room to all clients currently in the MainHall.

Finally, s1 replies to the client who is deleting the room with:

```
{"type" : "deleteroom", "roomid" : "jokes", "approved" : "true"}
```

4.7. Message

Chat messages are encapsulated as:

```
{"type" : "message", "content" : "Hi there!"}
```

The message is broadcasted to all the members in the chat room by the server using the following message:

```
{"type" : "message", "identity" : "Adel", "content" : "Hi there!"}
```

4.8. Quit

The client can send a **quit** message at any time by using the command "#quit":

```
{"type":"quit"}
```

When a server receives a **quit** message from a client, the client should be removed from the client list.

If the client is the owner of a chat room, **quit** should be considered as deleting that chat room by following the **delete** protocol.

The **roomid** value of the **roomchange** message will be an empty string to indicate that the client is disconnecting.

When the server sends the **roomchange** message to the disconnecting client, then the server closes the connection.

When the client that is disconnecting receives the **roomchange** message with **roomid** value of an empty string , then it closes the connection.

4.9. Abrupt disconnections by clients

If a client gets disconnected (e.g. if the TCP connection is broken), then the server treats this as if the client had sent a **quit** message. The server should send appropriate messages to other clients in the system and deletes the client's identity and his/her chat room.

5. Assumptions

In this project, we need not to consider failure handling for servers. We assume that chat servers do not fail or crash.

6. Technical Aspects

Use Java 1.8 or later.

Package your server into a **runnable jar file**.

Your jar file should be executable exactly as stated in Section 3.5.

7. Your Report

Use 10pt font, double column, 1 inch margin all around. Put your name, login name, and student number at the top of your report.

Write **300 words** at most for each of the following challenges of distributed systems as discussed in the lectures, how does the system address the challenges, or not. Give examples from the project to support your

claims.

- Heterogeneity
- Scalability
- Failure Handling
- Concurrency
- Transparency

Use critical thinking.

8. Submission

You need to submit the following via LMS:

- Your report in PDF format only.
- Your server.jar file.
- Your source files in a .ZIP or .TAR archive only.

Submissions will be due on Sunday 18th of September, midnight. Submissions will be via LMS and more details will be given closer to the due date.