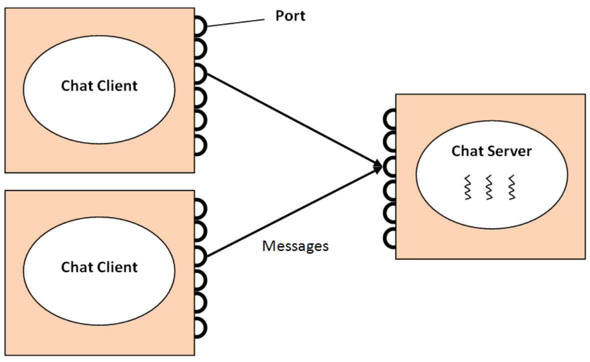
Distributed Systems
COMP90015 2015 SM2
Project 1 - Chat System

Synopsis

The assignment is to create a "chat" application, using the client/server architectural model. The chat system consists of two main distributed components: *chat server* and *chat client*, which may run on different hosts in the network. Chat clients are Java programs which can connect to a chat server.

The chat server is a Java application which can accept multiple incoming *TCP connections*. The chat server maintains a list of current *chat rooms* and chat clients can move between *chat rooms*. Messages sent by a chat client are broadcast to all clients currently connected to the same chat room.



Chat Server

In a nutshell, the chat server is primarily responsible for managing all the chat clients currently connected to the chat server and for distributing chat messages.

The chat server listens for requests from clients for making new connections. When a client connects to the server:

- the server generates a unique id for the client which is **guest** followed by *the smallest integer greater than 0 that is currently not in use by any other connected client*, e.g. **guest5**
- the server tells the client its id using an **NewIdentity** message
- the server adds the new client to the **MainHall** chat room
    - ♦ this involves sending more messages which is explained later

The protocol that the chat server must follow is specified exactly later in this project specification. All of the message types are specified exactly as well.

Client Command Line Interface

The client must accept input from *standard input*. Each line of input (terminated by a newline) 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.

The client must write all output to *standard output*. Since chat is asynchronous, messages may arrive at any time and will be written to standard output when they arrive.

This project will specify exactly what commands are possible, how they should be interpreted and what should be output to standard output.

Example Client Session

Here is a quick example of how the client session may look:

```
$java -jar chatclient.jar localhost
Connected to localhost as guest5.
MainHall: 6 guests
comp90015: 14 guests
FridayNight: 8 guests
guest5 moves to MainHall
MainHall contains guest1 adel chao* guest34 guest2 guest5
[MainHall] guest5> #join comp90015
guest5 moves from MainHall to comp90015
comp90015 contains ...
[comp90015] guest5>
guest8: we have a newcomer to the group!
Hi everyone
guest5: Hi everyone
[comp90015] guest5> bye
guest5: bye
[comp90015] guest5> #quit
guest5 leaves comp90015
Disconnected from localhost
```

Note that the example above was manually constructed and as such there may be slight differences with the actual program output. It's just an example.

JSON Format

All protocol messages, i.e. sent between client and server, will be in the form of newline terminated JSON objects.

A JSON library must be used to marshal JSON output and to unmarshal JSON objects. Do not implement JSON (un)marshalling yourself.

All data written to the TCP connection must be UTF8 encoded.

Chat Protocol

The Chat Protocol specifies exactly the communication between client and server. Your project must implement the protocol exactly as stated. Deviation from the protocol will loose marks. Incomplete implementation will loose marks.

The high level steps of the protocol are:

- Client initiates connection to server.
- Server responds with **NewIdentity** message
- Server joins client to **MainHall** (which involves other messages)
- Asynchronously, client sends commands/messages to server and server sends responses and messages to client.
- Client sends **Quit** message to server to disconnect and server responds.

New Identity

The server sends a **NewIdentity** message whenever the client requests an identity change using a **IdentityChange**, or when the client first connects to the server. The **identity** field gives the identity of the client. It may or may not be the same as what was requested by the client.

```
{
  "type":"newidentity",
  "former":"",
  "identity":"guest3"
}
```

Note that the **type** field tells the type of the protocol message. When that is known, then the other fields of the message will also be known.

Also note, in the above case the **former** field is blank, to indicate that the user did not previously have an identity; in other words this is the case when the user has just connected.

Identity Change

The client may at any time send an **IdentityChange** message to the server, to request the identity to be changed to something else. The requested identity must be an alphanumeric string starting with an upper or lower case character, i.e. upper and lower case characters only and digits. The identity must be at least 3 characters and no more than 16 characters.

```
{
  "type":"identitychange",
  "identity":"aaron"
}
```

E.g.:

```
[MainHall] guest5>#identitychange aaron
```

will generate this message.

Identity Change Protocol

The server must follow exactly this protocol:

- If an invalid or currently in use identity is given then no change to identity will result
- If the identity does not change then the server will respond with a **NewIdentity** message only to the client that requested the identity change.
- If the identity does change then the server will send a **NewIdentity** message to all currently connected clients.

The **NewIdentity** message will always contain **former** and **identity** fields. If there is no change to the identity then the fields will contain the same information. If there is a change to the identity, then **former** will contain the former identity and **identity** will contain the new identity.

```
{
  "type":"newidentity",
  "former":"guest5",
  "identity":"aaron"
}
```

Identity Change Protocol

The client must follow exactly this protocol when receiving a **NewIdentity** field:

- If the change in identity is with regard to the client's current identity:
    - If **former=identity** then the client outputs "Requested identity invalid or in use"
    - Otherwise the client outputs e.g. "guest5 is now aaron"
- If the change in identity is with regard to another client's identity then the client outputs e.g. "guest4 is now Adel"

E.g.:

```
[MainHall] guest5>#identitychange aaron
guest5 is now aaron
```

Join Room Protocol

The client may at any time send a **Join** message to the server to indicate that the client wishes to change their current room to the indicated room.

```
{
  "type":"join",
  "roomid":"comp90015"
}
```

E.g.:

```
[MainHall] guest5>#join comp90015
```

Join Room Protocol

When receiving a **Join** message the server must follow exactly this protocol:

- If the **roomid** is invalid or non existent then client's current room will not change.
- Otherwise the client's current room will change to the requested room.
- If the room did not change then the server will send a **RoomChange** message only to the client that requested the room change.
- If the room did change, then server will send a **RoomChange** message to all clients currently in the requesting client's current room and the requesting client's requested room.
- If client is changing to the **MainHall** then the server will also send a **RoomContents** message to the client (for the **MainHall**) and a **RoomList** message after the **RoomChange** message.

E.g.:

```
{
  "type":"roomchange",
  "identity":"aaron",
  "former":"MainHall",
  "roomid":"comp90015"
}
```

Note that when a client connects to the server; after sending a **NewIdentity** message to the client, the server generates a **RoomChange** message as if the client had sent a **Join** message for **MainHall**. In other words the client is effectively moving from nowhere to the **MainHall**.

Join Room Protocol

Similarly to changing identity the client will determine from the message whether the request was successful or not and will output relevant information:

- If the request was not successful: "The requested room is invalid or non existent."
- If the request was successful or if the request was for another identity, then e.g.: "aaron moved from MainHall to comp90015"

And on the command line:

```
[MainHall] guest5>#identitychange aaron
[MainHall] guest5>
guest5 is now aaron
[MainHall] aaron>#join comp90015
[MainHall] aaron>
aaron moved from MainHall to comp90015
[comp90015] aaron>
```

Room Contents Message

The **RoomContents** message lists all client identities currently in the room:

```
{
  "type":"roomcontents",
  "roomid":"comp90015",
  "identities":["aaron","adel","chao","guest1"],
  "owner":"chao"
}
```

The owner shows who created the room. The owner of **MainHall** is an empty string.

The client may at any time send a **Who** message to request a **RoomContents** message from the server for a given room:

```
{
  "type":"who",
  "roomid":"comp90015"
}
```

e.g.

```
[MainHall] guest5>#who comp90015
```

Note that obviously the server knows the identity of every client connection, so the identity of the sender is not required.

Room List Message

The **RoomList** message lists all room ids and the count of identities in each room:

```
{
  "type":"roomlist",
  "rooms":[{"roomid":"MainHall","count":5},
           {"roomid":"comp90015","count":7},
           {"roomid":"FridayNight","count":4}]
}
```

The client may at any time send a **list** message to request a **RoomList** message from the server:

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

Room Information at the Client

The client displays information regarding rooms and room lists whenever they arrive. Examples have been given on previous slides.

Note that, as shown in previous examples, the owner of a room is shown with a "*" character after their identity.

Create Room

Anyone can create a room using a **CreateRoom** message, if the requested name of the room is valid and does not already exist. The room name must contain alphanumeric characters only, start with an upper or lower case letter, have at least 3 characters and at most 32 characters.

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

The server replies with a **RoomList** message only to the client that was creating the room. If the room was created, then it will appear in the list.

The client outputs either e.g. "Room jokes created." or "Room jokes is invalid or already in use."

Room Ownership

- Every room has an assigned owner; the user that created the room. The server of course maintains this relationship.
- If the user changes their identity then rooms owned by the user should also change ownership to the changed identity.
- If a user disconnects from the server, any rooms owned by the user are set to have an empty owner (empty string). Such rooms can never be owned again by anyone.
- If any room other than **MainHall** has an empty owner and becomes empty (i.e. has no contents) then the room is deleted immediately.

Kick User

The owner of a room can at any time send a **Kick** message to the server:

```
{
  "type":"kick",
  "roomid":"jokes",
  "time":3600,
  "identity":"guest1"
}
```

The server will treat this as if it had received a **RoomChange** from, in this example, **guest1** to change to **MainHall**. All appropriate messages will be sent. It will only do so if the client is the owner of the room.

A user kicked from a room must not be allowed to move back to the room for up to **time** seconds. The server must ensure this. Note that **time** is not a string, but a positive integer.

Delete Room

The owner of a room can at any time send a **Delete** message to the server:

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

The server will first treat this as if all users of the room had sent a **RoomChange** message to the **MainHall**. Then the server will delete the room. The server replies with a **RoomList** message only to the client that was deleting the room. If the room was deleted, then it will not appear in the list.

Messages

The client can at any time send a **Message**:

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

A message received by the server is relayed to all clients that are within the same room as the client who sent the message. The id of the sender is appended to the message:

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

Clients will never receive messages for rooms that they are not currently in.

Quit

The client can at any time send a **Quit** message:

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

The server will remove the user from their current room, sending an appropriate **RoomChange** message to all clients in that room. The **roomid** of the **RoomChange** message will be an empty string, to indicate the user is disconnecting. Rooms owned by the user who is disconnecting are set to have an empty owner.

When the server sends the **RoomChange** event to the disconnecting client, then it can close the connection.

When the client that is disconnecting receives the **RoomChange** message, then it can close the connection.

Abrupt Disconnections

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 set room ownership appropriately.

Technical aspects

- Use Java 1.7 or later.
- All message formats should be JSON encoded. Use a JSON library for this.
- Your program should be cleanly finished by terminating all running threads.
- Package everything into a single runnable jar file, one for server, one for client.
- Your server and client should be executable *exactly* as follows:

```
java -jar chatserver.jar [-p port]

java -jar chatclient.jar hostname [-p port]
```

Square brackets ([]) indicate that what is inside the bracket is optional.

- Use command line option parsing (e.g. using the args4j library or your choice).
- The default server port should be **4444**. A command line option can override this.
- Pressing Ctrl-C should terminate either client or server.

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 2000 words at most. For a selection of the challenges of distributed systems as discussed in the first lectures, how does the system address the challenges, or not. Use critical thinking. Describe how a multi-server architecture could be implemented. Describe the messages that would be used between the servers. How would the data be distributed? How would the client change?

Submission

You need to submit the following via LMS:

- Your report in PDF format only.
- Your chatserver.jar and chatclient.jar files.
- Your source files in a .ZIP or .TAR archive only.

Submissions will be due on Friday, Week 8, midnight. Submissions will be via LMS and more details will be given closer to the due date.