CS5001 Object-Oriented Modelling, Design and Programming
Practical 3 – Chat Server

School of Computer Science
University of St Andrews

Due Friday week 9, weighting 20%
MMS is the definitive source for deadlines and weightings.

In this practical, you are required to write a Java chat server which implements a simplified version of the IRC protocol, allowing connected users to join channels and send each other messages.

Setup

For this practical, you may develop your code in any IDE or editor of your choice. However, please ensure that you create a suitable assignment directory `CS5001-p3-chatserver` on the computer you're using. All your source code should be in a `src` directory inside your assignment directory, `CS5001-p3-chatserver/src`, and your main method should be in an `IrcServerMain` class in `IrcServerMain.java`.

Internet Relay Chat (IRC)

IRC is a protocol for text-based chat that developed gradually from the late 1980s into the early 2000s. It was one of the first widely used chat protocols, and still forms the backend of a wide variety of chat services. There is no universally agreed specification, and different implementations include different features, but the core functionality is the same across most active servers.

An IRC server listens on an advertised port for connections from IRC clients. These clients send short strings over a stable TCP connection, to register, join channels, and send messages. The server then sends these messages onto whichever connected clients are supposed to receive them, along with any appropriate metadata.

In this practical, you will implement a chat server that should follow the simplified version of the IRC protocol described below. It should accept connections from clients using Java's `Socket` and `ServerSocket` objects, keeping multiple connections alive at once and sending the appropriate updates to each one in real time.

You only need to write the server for this practical; you don't need to create a client. However, you will want to test your program, and you can do this using an existing program called

`telnet`, which can send and receive simple strings over TCP. See the Test and Debug section below.

The main output of your program will be sent over the `Socket` objects you create. However, you might want to write some debugging information to standard output with `System.out`.

The Protocol

Your server should accept only certain formats of strings from connected clients, and in some cases it should send a *reply* to the client. Each supported client message has the form

```
<command> <arguments>
```

where `<command>` is equal to `NICK`, `USER`, `QUIT`, `JOIN`, `PART`, `NAMES`, `LIST`, `PRIVMSG`, `TIME`, `INFO`, or `PING`. The arguments should then appear after a space, and the arguments that are expected will depend on what the command was.

If the server needs to send a reply, it will have the form

```
:<server_name> <reply_code> <nick> <text>
```

where

- `<server_name>` is whatever name was specified for the server on the command line when it was started (see Compiling and Running);
- `<reply_code>` is a 3-digit number that identifies what sort of reply it is (errors use `400`);
- `<nick>` is the nickname of the client the reply is being sent to, or `*` if no nickname has been set; and
- `<text>` is some more text, which will usually start with a colon, and will have a format depending on what sort of reply it is. The colon is there to indicate that the text might contain spaces.

The different types of command and reply are explained below.

Commands

Your server should support the following commands from clients. The most important ones are listed first, but some of the later ones like `TIME`, `INFO` and `PING` might be the easiest ones to implement, so you could start with these if you want.

NICK

This message is sent by the client in order to declare what nickname the user wants to be known by. The whole message should have the following format:

```
NICK <nickname>
```

where `<nickname>` is the chosen nickname. A valid nickname has 1–9 characters, and contains only letters, numbers and underscores. It cannot start with a number.

If the nickname is valid, the server should remember that this is the nickname for this client, and send no reply. If the nickname is invalid, the server should reject it by sending the reply

```
:<server_name> 400 * :Invalid nickname
```

USER

This message should be sent by the client *after* they have send a `NICK` message (see NICK). The `USER` message allows the client to specify their *username* and their *real name*. It has the following form:

```
USER <username> 0 * :<real_name>
```

where

- `<username>` is the username, which should not include spaces (the username is stored internally but not used for anything else);
- `0` and `*` are meaningless (they are used for permissions in full IRC, but you can ignore them for this practical); and
- `<real_name>` is the user's real full name, which may include spaces.

If the message is valid, and a nickname has already been set, then the server should store all these details and regard the client as *registered*. A registered client can send and receive private messages, and join and leave channels. A reply should be sent of the form

```
:<server_name> 001 <nick> :Welcome to the IRC network, <nick>
```

If the message is invalid, one of the following replies should be sent:

```
:<server_name> 400 * :Not enough arguments
:<server_name> 400 * :Invalid arguments to USER command
:<server_name> 400 * :You are already registered
```

as appropriate.

QUIT

This message indicates that the user wants to end their connection to the server. It has no arguments, so it should just be the message:

```
QUIT
```

If the client is *registered* (see USER) then the server should send the message

```
:<nick> QUIT
```

to *all* connected clients (where `<nick>` is the quitting client's nickname). The quitting user should also be removed from any channels they may be in (see JOIN).

Finally, the connection to the quitting client should be closed.

JOIN

This message is sent by a client in order to *join* a channel. A channel is like a chat room that users can join, and any messages sent to a channel will be seen by all users that are in the channel (see PRIVMSG).

This message should have the form:

```
JOIN <channel_name>
```

where `<channel_name>` is the name of the channel to join. A channel name must be a single `#` symbol followed by any number of letters, numbers and underscores.

If the channel name is valid, this user should be added to that channel. If no channel with this name exists on the server, it should be created. All users in the channel (including the joining user) should be sent the message

```
:<nick> JOIN <channel_name>
```

to indicate that a new user has joined. That user should then receive all chat messages sent to that channel until they leave the channel or quit the server.

If the channel name is invalid or the user is not registered, one of the following error messages should be sent:

```
:<server_name> 400 * :Invalid channel name
:<server_name> 400 * :You need to register first
```

as appropriate.

PART

This message is sent by a client when that user wishes to leave a channel they are in. It has the form:

```
PART <channel_name>
```

where `<channel_name>` is the name of the channel they want to leave. If successful, the message

```
:<nick> PART <channel_name>
```

should be sent to all users in the channel, and the user should be removed from the channel. If the channel is now empty, it should be deleted from the server. If the channel exists, but the user is not in it, the server should do nothing.

If unsuccessful, one of the following error replies might be necessary:

```
:<server_name> 400 * :You need to register first
:<server_name> 400 * :No channel exists with that name
```

PRIVMSG

This is perhaps the most important command, because it allows registered users to send chat messages to each other! It has the form:

```
PRIVMSG <target> :<message>
```

where `<target>` is either a channel name or a user's nickname, and `<message>` is the full chat message they want to send, which may include spaces.

If the message is valid, then a message of the form

```
:<sender_nick> PRIVMSG <target> :<message>
```

should be sent to all appropriate users, where `<sender_nick>` is the nickname of the user that sent the message. If `<target>` is a channel, this should be sent to all users in that channel; if it is a user's nickname, it should be sent to that user only.

If the message is invalid, one of the following error messages may be sent:

```
:<server_name> 400 * :No channel exists with that name
:<server_name> 400 * :No user exists with that name
:<server_name> 400 * :Invalid arguments to PRIVMSG command
:<server_name> 400 * :You need to register first
```

NAMES

This message is sent by a registered client to request the nicknames of all users in a given channel. It has the form:

```
NAMES <channel_name>
```

If a channel with the given name exists, the server should send a reply of the following form:

```
:<server_name> 353 <nick> = <channel_name> :<nicks>
```

where

- `<server_name>` is the server name;
- `<nick>` is the nickname of the user that sent the request;
- `<channel_name>` is the channel being queried;

- `<nicks>` is a space-separated list of the nicknames of all users in the channel, for example `moeen zak jos ben jofra jonny`.

The server might need to reply with one of the following error replies:

```
:<server_name> 400 * :You need to register first
:<server_name> 400 * :No channel exists with that name
```

LIST

This message allows a registered client to request the names of all channels on the server. It has no arguments, so the whole message is just:

```
LIST
```

and the server should reply with one line for each channel, of the form:

```
:<server_name> 322 <nick> <channel_name>
```

followed by one final line of the form:

```
:<server_name> 323 <nick> :End of LIST
```

where `<nick>` is the nickname of the user who sent the `LIST` command.

If the user is not registered, they should receive the same error reply as they would for `NAMES`.

TIME

Clients can send the simple message:

```
TIME
```

to ask the server to respond with the current date and time. The server should send a reply of the form:

```
:<server_name> 391 * :<time>
```

where `<time>` is the server's local time, in the standard ISO 8601 format, something like

```
2022-10-13T14:23:34.443757
```

This is fairly easy to achieve if you use Java's `java.time.LocalDateTime` package. Look it up in the Java API!

INFO

The user can request some basic information about the server by sending the message:

```
INFO
```

The server should send a reply of the form:

```
:<server_name> 371 * :<message>
```

where `<message>` is a short string saying what the server is and who wrote it. The exact content of this is up to you, but it should fit on one line, and should not include your real name.

PING

Finally, any client can send a message of the form

```
PING <text>
```

where `<text>` is any string of characters. The server should respond with

```
PONG <text>
```

where `<text>` is the exact same string sent back. This can be useful for clients to make sure their connection is still active.

Compiling and Running

For your program to be compatible with the automated checker:

- It should be possible to compile all your source code from within the `src` directory using the simple command:

  ```
  javac *.java
  ```

  If you have several packages inside the `src` directory, the automated checker might be able to pick these up, but you should check this carefully.

- It must be possible to specify the server name and port on the command line using the `String[] args` parameter. For example, executing the following command from within the `src` directory should start your server with the server name `hello`, on port 12345:

  ```
  java IrcServerMain hello 12345
  ```

- If your server is started without supplying the command-line arguments, or with an invalid port number, it should simply print the usage message indicated below and exit:

```
Usage: java IrcServerMain <server_name> <port>
```

Running the Automated Checker

As for previous practicals, we have provided some basic tests. In order to run the automated checker on your program, make sure your code is on a lab machine, **change directory** to your `CS5001-p3-chatserver` directory, and execute the following command:

```
stacscheck /cs/studres/CS5001/Coursework/p3-chatserver/Tests
```

If you use a Linux or Mac machine, you can also install stacscheck on your own computer. However, you should check that your code works on the School infrastructure as well.

When you are ready to submit, make sure that you also run the checker on the archive you are preparing to submit, by calling, for example:

```
stacscheck --archive p3.zip /cs/studres/CS5001/Coursework/p3-chatserver/Tests
```

The numbered tests check basic operation of your server. The first few tests check whether the correct usage message is displayed when the command-line arguments are invalid. The remaining tests check how your server responds to a range of possible messages from clients, some valid and some invalid. If any of these fails, then you are not sending back the expected message. In case of failures, the tests should indicate what your program has sent to the client over the socket, along with what we expected you to send.

If any tests are failing, you should look at the files themselves on StudRes so you can see what they're doing. Most tests contain the following three files:

- `progRun.sh`: a script that starts a server running, opens a client connection to it, and checks its output;
- `sendMessages.sh`: the script that acts as the client, sending some messages in the protocol and waiting at some points using the `sleep` command;
- `expected.out`: the text that should be received by the client.

A few tests also have a `sendMessages2.sh` script, which simulates a second client that connects to the server at the same time. This tests your server's ability to handle multiple connections and facilitate two clients sending chat messages to each other.

Note that `Test09` and `Test10` check only part of your output, by searching for a substring using `grep`. This is because the `TIME` command produces different output each time you call it, and because the `INFO` command might have output specific to you, perhaps including your matriculation number. In each case, `expected.out` should be the *start* of the output that your server sends.

The final test, `TestQ`, runs the style checker over your source code. Pay attention to its output, because it can be really helpful for improving your code quality.

Test and Debug

In order to test your server manually, you should be able to use simple existing client programs to send messages to your server and print out the reply. You can use `telnet` from the command line on the lab computers, and this is available on most operating systems if you're working on your own machine. While running your server on port 12345 in one terminal window, you can open another terminal window on the same machine and type

```
telnet localhost 12345
```

If your server program accepts the connection, you can then type a message directly into telnet (such as `TIME`) and the response from the server will be printed immediately below it in the terminal.

In your own code, take care to flush sockets and close them when appropriate. Otherwise you may not see the responses made by the server on the client and you may have issues running your server program due to sockets being left open.

You might also consider performing more rigorous automated testing. If you decide to adopt this approach, it is probably a good idea to start by looking at the existing tests on StudRes and follow the same approach. You can create new tests in a local sub-directory in your assignment directory and pass the directory of your own tests to stacscheck when you run it from your assignment directory.

You can learn more about the automated checker at https://studres.cs.st-andrews.ac.uk/Library/stacscheck/.

Deliverables

Along with your source code, you should include a short README file if you have attempted any additional features beyond what this document specified. You should hand in a zip archive of your assignment directory (containing all your source code, any sub-directories, and README), via MMS as usual.

You do not need to include a README if you have not attempted any additional requirements. However, you should include one if there is anything else in your solution that you want to draw our attention to.

Marking

Grades will be awarded according to the General Mark Descriptors in the Feedback section of the School Student Handbook. The following table shows some example descriptions for projects that would fall into each band.

Grade	Examples
1–3	Little or no working code, and little or no understanding shown.
4–6	Acceptable code for part of the problem, but with serious issues such as not compiling or running. Some understanding of the issues shown.

Grade	Examples
7	A running server that accepts connections from clients, and uses sockets to send and receive strings, that implements commands similar to those in the specification and goes some way towards allowing users to chat. May have bugs, poor style, and many missing features.
8–10	A reasonably stable server that handles multiple clients at once, and implements several of the listed commands correctly, allowing connected users to chat to each other. Possibly lacks some of the more complex features, has problems with error-handling and poor style. Some documentation.
11–13	Implements most of the commands in the specification, with reasonable error-handling and acceptable program design and code quality. Some object-oriented design, and few bugs. Reasonable documentation.
14–16	Implements almost all of the commands in the specification correctly, with almost all the specified error messages, clear code, a well-structured object-oriented design, and very little code repetition. Good documentation.
17–18	Implements all the commands in the specification correctly, with full error-handling including sensible decisions for errors not mentioned in the specification. Clean code in an excellent object-oriented design, with almost no repetition. Excellent documentation.
19–20	Outstanding implementations of all features as above, and possibly some extra features not mentioned in the specification, with exceptional clarity of design and implementation.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8-hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/