Assignment 3 - Chat Server

You are tasked with creating a chat server that will serve multiple clients and hold multiple channels.

The server will receive messages sent from clients and relay the messages to other clients. It needs to keep an active record of all communication and provide context to anyone who has joined a channel.

Server

Your server needs to acquire information for itself. It needs to know what port it is listening to, which database it is reading from and manages a large number of concurrent connections. The server should try to utilise non-blocking IO and/or processes to allow for asynchronous communication. Consider using the select module and/or multiple processes for the server.

Any implementation that can only serve one client at a time will not be accepted.

Your server will be lauched in the following way.

```
python chat_server.py <port> [configuration]
```

Protocol

You need to implement the following protocol on the server. A message protocol is a set of agreed procedures between parties. In this case, these protocols specify the different type of messages that will be communicated between the client and the server.

The client and server are aware of the types of messages they can send to and receive from each other. Any messages that lie outside of the specified protocols can safely be ignored.

Message Format

The message format is string in `UTF-8` encoding. This character set supports languages other than English, e.g. CJK characters, which can also be sent as valid messages and usernames. However, protocol commands such as CONNECT and REGISTER are required to exist in their specified form.

Messages

- LOGIN :USERNAME :PASSWORD

Your server will receive this message upon connection from a client. Your server will process the username and password by checking if the user exists and the password matches.

- REGISTER :USERNAME :PASSWORD

The client will be able to send the REGISTER message to register their own username and password. The server will hold the username and password in a database. The password should be hashed and not stored in plain text.

Since this is only version 0.1 of our protocol, we will only communicate using plain text sockets.

- JOIN :CHANNEL

The JOIN message is sent to the server to indicate a channel that a user would like to join. If the channel exists, the user is able to join the channel and be able to send messages to others in that channel. If the channel does not exist, the user will receive a reply indicating the error.

- CREATE :CHANNEL

A user is able to create a channel with a given name. If the channel already exists, the channel will not be created and the user will receive a reply indicating the error.

- SAY :CHANNEL :MESSAGE

This allows the user to send a message to a channel. Assume the user has joined the channel, the server will receive this message and send it to the other users in the same channel. If the user has not jointed the channel, then the user will receive a reply indicating the error.

Once a user sends a `SAY` message, the user should receive a `RECV` of that message.

- RECV :USER :CHANNEL :MESSAGE

The client will receive this message, this occurs when a user sends a message to the server via SAY, the server will send a RECV message detailing the user who sent it, channel it was sent to and the message that was sent.

- CHANNELS

This command returns all the channels the user is currently subscribed to (joined). This will also include any direct communication between users (as this is just another type of channel). (Not very clear by the last sentence.)

- RESULT :TYPE :MSG [:MORE...]

This is a generic response from the server to the client. The result can contain different types such as:

- JOIN
- CREATE
- LOGIN
- REGISTER
- CHANNELS

Each type will have a message or more to confirm if the message was successful or invalid. Given the different types, this will impact the message(s) that can be sent.

Result Types

As discussed in the previous section, the result message can contain different types and messages. We will need to ensure that any message we send, the client can receive confirmation from it.

- RESULT JOIN :CHANNEL :CONFIRMATION

The server will send the JOIN RESULT, alongside the channel and if the operation was successful. If the operation is successful, the :CONFIRMATION would be represented as "1". If the operation was unsuccessful, the :CONFIRMATION would be represented as "0".

- RESULT CREATE :CHANNEL :CONFIRMATION

Similar to the JOIN RESULT, the message will contain the channel name that was requested and confirmation string of either "1" or "0" to indiciate if the operation was successful or not. The :CONFIRMATION component will indicate "1" on success or "0" if the operation was unsuccessful.

- RESULT (LOGIN | REGISTER) :CONFIRMATION

Given the username and password to be sent to the server, the server will respond with confirmation if the login or registration was successful. Similar to JOIN and CREATE RESULT types, the confirmation part will be represented as either "1" or "0" for determining success.

- RESULT CHANNELS [CHANNEL-NAME,...]

Unlike the other result types, this result does not provide a confirmation since it should always work. The result will contain a list of channel names that the client can connect to.

Testing and Client

You are required to write your own test cases for this assignment. This includes writing your own client and ensuring that your program can output code coverage information.

Extension - Federation

If you have passed all compliance test cases for the chat server, you can implement the federation extension within your chat server application. This extension enables cross-server communication. That is, to allow multiple servers to cooperate to allow users of different servers to message each other.

This means there will be an extension to the protocol itself. This is to ensure that we do not encounter identifier clashing of users and channels. The extension should not break any existing compatibility with your compliant chat server.

How does federation work?

Federation will allow your users to participate in discussions and subscribe to channels on other servers. However your server will need to have a configuration mode that will specify what servers it will cooperate with.

- Federating with other servers

One server is able to federate with other servers when given a file that contains servers it can contact to federate with.

The file will be in the following format:

```
<IP Address 1>:<Port>
<IP Address 2>:<Port>
```

Example:

```
192.168.1.1:4400
168.172.3.23:2344
```

The server will attempt to send the following message to other servers:

- FEDERATE-OUT

The server that receives this message will add the connecting server to its own internal list, and then send back a confirmation with the message: FEDERATE-CONFIRM

The following special commands need to be implemented.

- FEDCHANNELS [CHANNEL-NAME,...]

Upon confirmation of federation of a server, servers will need to provide a channel list to others. This is to ensure that other servers can list channels that are not hosted by them but are available to their users.

- FEDNEW :CHANNEL-NAME

When a new channel is created, the server will need to send this message to all connected federation servers to inform that a new channel has been created and to update their current channel list.

- FEDJOIN :USER :CHANNEL

The server that the user belongs to, will request to join the channel on another server, this message originates from the user's connected federated server.

- FEDSAY :USER :CHANNEL :MESSAGE

This message originates from the user's server and is directed to the channel on a federated server. The federated server will receive this message and send the message to all users subscribed to that channel.

- FEDRECV :TOUSER :FROMUSER :CHANNEL :MESSAGE

When updates are made to a federated channel, subscribers to that channel may exist on a different federated server. The message is directed at the :TOUSER server.

- FEDRESULT :USER :TYPE :MSG [:MORE...]

This is a response message that will inform the user requesting a previously mentioned federation message if their request succeeded or failed. This result is just the federation version of RESULT.

End To End Interaction

From the client application perspective, the client's will only implement utilise the standard chat client protocol and will not implement the federation protocol. The federation protocol is between servers, not clients.

Here is an example of a user from SERVER1 joining a channel on SERVER2. Lets assume SERVER1 and SERVER2 have federated. Assuming the channel exists, the interaction is as followed.

```
client sends LOGIN bob password to server1.com
client sends JOIN helloworld:server2.com to server1.com
server1 receives JOIN helloworld:server2.com from client
server1 sends FEDJOIN bob@server1.com helloworld to server2.com
server2 receives FEDJOIN bob@server1.com helloworld from server1.com
server2 adds bob@server1.com to helloworld channel
server2 sends FEDRESULT bob JOIN helloworld:server2.com 1 to server1.com
server1 sends RESULT JOIN helloworld:server2.com 1 to client
client receives RESULT message from server1
```

After successful confirmation, the client is able to send messages to users on the helloworld channel on server2.

Distinguishing users and rooms

Federated users and rooms need to be distinguised from their local server counterparts. The best way to do this, is to ensure that any user or channel is identifiable by their local-server identifier and their host identifier that the server knows it by.

Federated users will appear with hostname after their identifier.

Example:

```
john.smith@myhost.com
```

Channels will be identifable similarly, however, instead of using the `@` symbol, they will be distinguished by the colon symbol `:`

```
secretchat:myhost.com
```

Note that a valid hostname or IP address should not contain `@` and `:` as defined.

Restrictions - What would warrant serious deduction

You will need to ensure that you do not break any of the following restrictions when writing your solution. If one or more of the following restrictions are broken, you will receive 0 for your assessment mark.

- Code that explicitly targets a test case

This is known as 'hard-coding' and it subverts assessment itself by explicitly writing code to handle a particular test case.

Marking Criteria

Your program must be submitted using git to Ed. We will only mark your final submission that has been committed to Ed. Your program will need to satisfy the following criteria to receive marks for each area.

- Automatic Test Cases - Compliance (16%)

Your program will be ran against automatic test cases to check if your server is compliant with the outlined protocol and behaves as expected. These will include public, private and hidden test cases.

- Your Test Cases (4%)

You will need to supply test cases that shows that you have tested your solution completely and present it in a easily digestable form. Aim to have multiple tests for each feature of your application.

Your test cases resolution must be presented in JSON format, otherwise your test cases will not be marked. Utilising netcat or write your own client to test your server.

- Extension (6%) - (6 bonus marks available)

You will need to implement this extension completely to achieve all marks. You are only eligible for the extension only if you have passed all public compliance test cases. You will need to implement your own set of test cases to show that your extension works.