MPCS 52040 Distributed Systems
Homework 3: ZMQ Chat Server
Due: 1/31 @ 10:00 pm

Multi-user chat server

In this homework assignment you will implement a multi-user chat server using ZeroMQ. The purpose of this assignment is to reinforce your understanding of different communication patterns. The assignment includes four parts: first you will create a client/server pair for publishing messages to a chat channel; you will then create a pub/sub mechanism for displaying the contents of the channel; next you will integrate these two models to create an interactive, multi-user chat client/server; finally, you will create a multi-channel client/server using topics to filter message.

Part 1: posting to the chat channel

Create a server that can receive messages from a client. Messages should include the user who posted the message, the time the message was posted, and the message body:

```
<username, time, message>
```

The server should run continuously (until you explicitly terminate it) and record all messages to an in-memory data structure. Start the server with the following command:

```
python post_server.py <ip> <port>
```

Create a client that can post messages to the server using the following command:

```
python post_client.py <ip> <port> <username> "message"
```

For example, if I were to post a message to a server deployed on localhost with port 7777 I would write:

```
python post_client.py 127.0.0.1 7777 kyle "hello"
```

Note: you do not need to deploy your client and server on different computers, you can simply open different terminals.

Part 2: publishing messages to other clients

Extend your server so that it publishes all messages to a ZeroMQ PUB channel. This will allow other clients to register (subscribe) and see the messages published to the channel. If a client subscribes after messages have already been sent you do not need to retrieve old messages. Note: the PUB channel will need to be on a different port than the post channel.

The server should now be started with the following command:

```
python pub_server.py <ip> <post_port> <pub_port>
```

Create a new pub_client that can listen to the channel and print messages in the following format:

```
username: message (time)
```

The client should stay connected to the channel and print messages as they are posted. You should use your post_client from part 1 to send messages to the pub_server via the post_port.

The client should be started with the following command:

```
python pub_client.py <ip> <pub_port>
```

Part 3: interactive chat client

Now you will combine the post_client and pub_client into a single client to develop an interactive chat client. The client should allow users to input messages and send them to the server when the user hits enter. The client should continually update by printing messages to stdout that are posted to the channel.

The client should be started with the following command:

```
python interactive_client.py <ip> <post_port> <pub_port> <name>
```

Test your server by starting multiple clients with different usernames (you can create a new client in a new terminal). Test the ability to post messages from different clients to check that it can receive messages from multiple clients and publish channel state to multiple connected clients.

Note: you will need to make the posting and publishing actions occur concurrently in the client such that one does not block on the other (i.e., via threads).

Part 4: Multi-channel client

In the final part of the project you will extend the interactive chat client so that it can support multiple channels. Here, users should be able to post and listen to different channels. First, update your client so that it can connect to a specific channel when it is started.

```
python mc_client.py <ip> <post_port> <pub_port> <channel> <name>
```

You will need to extend both your client and server to include the channel in the messaging format. You will then need to define a filter to subscribe only to messages relevant to your channel. Extend your client and server such that specifying the channel ALL will subscribe to all channels and therefore see message posted in all channels. Note, the "ALL" channel should be read only such that if you connect to ALL you will not be able to post messages.

Write up how you have tested your client and server to confirm that the multi-channel model works as expected. Write this description in a file called test.txt.

Submission

Please use the following link to create a repository for homework 3 in our GitHub classroom.

`https://classroom.github.com/a/EQ1rI8X8`

To assist the grading process please submit each part individually in one of four directories:

1. Part1: post_client.py and post_server.py
2. Part2: pub_client.py and pub_server.py
3. Part3: interactive_client.py and interactive_server.py
4. Part4: mc_client.py, mc_server.py, test.txt

Grading Guide

This HW will be graded out of 80 points, divided as follows. All points are for functionality/correctness.

(80 points total):

- Task part 1: correctly set up a zeromq server to listen for messages (5pts):
- Task part 1: correctly set up a zeromq client to send messages (5pts):
- Task part 1: message sent correctly (5pts):
- Task part 1: messages received by server and recorded (5pts):
- Task part 2: correctly set up server using PUB channel and publish server messages to it (10pts):
- Task part 2: pub_client can listen to messages on the channel (10 pts):
- Task part 3: sets up client and server sockets correctly (5 pts):
- Task part 3: client can send messages (5pts):
- Task part 3: server can handle connections from multiple clients concurrently (10pts):
- Task part 4: client can connect to a specific channel (5 pts):
- Task part 4: messages sent by client go to specific channel (5 pts):
- Task part 4: server parses message channel and routes messages properly (10 pts):