Università della Svizzera italiana

Faculty of Informatics

Distributed Systems **Autumn 2024/25**

Code Assignment 3 Chat Server Part 3: Direct Communication

The deadline to uploading your source code to iCorsi is on the 7th of November, at 23:59. Late submission policy will apply; see the Intro slides for details. Upload a single Python project, with separate main files for each exercise. You can use the provided `template-*.py` files as a starting point. If you are using a language other than Python, include a Dockerfile that builds and runs your project.
Please take extra care to only submit source code (and not build artifacts).

Exercise 1 - (3 points)

Implement client-to-client messages, that is, a client should be able to send a message to another client via the server.

To test the message passing, run the server and two or more clients and send messages between the clients. The format for user input on the client should be "[id] [msg]", e.g., "3 Hi there!" sends the message "Hi there!" to client 3. To make things easier you can have the client id be a running argument, similar to the port number on the Server's starting point code. If a message is sent to a nonexistent client the message should be dropped by the server, without an exception being thrown.

Exercise 2 - (3 points)

Let's improve our system to better handle the case in which the receiver is not present when a message is sent. Update your implementation so that messages are stored in the server until the receiver connects to it, at which point the recently connected client received all the buffered messages.
As the clients now need a way to identify themselves with a specific ID, extend the protocol to allow clients to inform the server of the ID they want to be known as. The server should then assign the requested ID to the client, if it is not already in use; otherwise, the server should reject the request and continue with the already-assigned ID.
Tip: to allow the client to identify itself, take the desired ID as an argument when starting the client.

Exercise 3 - (4 points)

Now it is time to switch from a client-server architecture to a peer-to-peer architecture, in which clients and servers are subsumed by peers. Model your system such that messages can be broadcast among different peers until they reach the destination. Some implementation tips are listed below.

Tip 1: start by combining the code from the server and the client, while making the necessary changes.
Tip 2: to keep things simple you can make the server code only send messages and the client code only receive them.
Tip 3: since the server and the client are in the same process now, they need to be executed as threads.
Tip 4: to be able to type the messages without problem, you can remove the server operator from your code,
Tip 5: if you wish to ensure that messages are not buffered forever, you can add a confirmation receipt message that propagates throughout the network,

Tip 6: to accept one of multiple kinds of messages, you can use the oneof msg { ... } construct in the protocol buffer file.

As there is no central server to assign IDs to the peers, we will be using a simplified variant of an ID generation scheme known as "Snowflake". In this scheme, the ID is composed of a timestamp, an assigner ID, and a sequence number; a python implementation of the simplified scheme is given in the ancillary file `snowflake.py`. We will be using the following scheme:

$$[0^{1}||\texttt{timestamp}^{38}_{10ms}||\texttt{assigner ID hash}^{16}||\texttt{seq}^{9}]^{64}$$

Where `timestamp` is the number of 10-millisecond intervals since a given origin time, `assigner ID hash` is the SHA-256 hash of the assigner ID (`xor`-folded to 16 bits), and `seq` is a sequence number that is incremented for each ID generated in the same $10ms$ interval. Every peer should use the above scheme to generate IDs for connecting peers, which also should be able to use a `--desired-id` option to specify their preferred ID (which might be rejected), similar to the previous exercise. Note that the ID specification is used here to simplify testing. In a real application, clients would only use only a robust ID generation scheme.

To run an example containing three peers, with IDs (a) 77252666718255104, (b) 77252666718255105, and (c) 77252666718255106 having connections between (a) and (b), and (b) and (c), you can use the commands listed below on your terminal; the general form is shown at the end, to test different topologies. Note that a message from (a) to (c) needs to be forwarded by (b), since there is not direct connection between (a) and (c). It should also be noted that you can specify any number as IDs, the ones above represent sample generated IDs from the given scheme.

```
python e3.py 127.0.0.1:8083 --desired-id 77252666718255106
python e3.py 127.0.0.1:8082 --desired-id 77252666718255105 127.0.0.1:8083
python e3.py 127.0.0.1:8081 --desired-id 77252666718255104 127.0.0.1:8082
```

The general form is:

```
python e3.py [my_ip]:[my_port] --desired-id [my_id] [peer1_ip]:[peer1_port]...
python e3.py [my_ip]:[my_port] [peer1_ip]:[peer1_port]...
```