Fourth assignment DHT (Chord)

Send the finished task by e-mail to your teaching assistant. Deadlines for the submission are on the web page of the course: http://d3s.mff.cuni.cz/teaching/middleware/.

The assignment describes an extension to a simple chat application, where users send private messages, with the possibility to join so-called channels, which distribute the messages to all participants that join the channel (similarly to e.g. IRC). Understanding the assignment requires no special knowledge.

Prerequisites

The chosen technology for sending messages is a peer-to-peer overlay providing Key-Based Routing. The particular implementation is python-chord (python implementation of the Chord overlay). The following knowledge is needed for the implementation:

- The core idea of key-based routing: addressing messages to peers obtained by hashing; working with keys in the Chord implementation (functions: `Address.__hash__()`, `Node.id()`, `hash_string()`, `Local.is_ours()`).
- Chord overlay initialization, joining the overlay network, bootstrap from a known peer (functions: `Local.__init__()`).
- The methods for sending messages: message format (string message type without whitespace, string payload without newlines), registration of asynchronous upcall functions for receiving messages (`Local.register_command()`), and processing of the predecessor change event (`Local.set_notify_handler()`), message sending (`Remote.user_command()`, `Local.user_command()`).
- The methods for finding other network nodes (`Local.find_successor()`, `Local.successor()`, `Local.predecessor()`)

An example simple chat implementation will be provided; you may implement the assignment by extending this example.

Assignment details

`http://d3s.mff.cuni.cz/teaching/middleware/files/as4.zip`

Implement a simple application for chatting over network without a central server using the Chord peer-to-peer overlay Python implementation.

Your tasks are:

1. When starting the application, the user will provide a nickname to identify himself in the chat and his node's address to identify his node. (The node's address=IP+port is used to identify a node in the Chord network.) The user may also specify an address and port of a machine to bootstrap the overlay from. After its startup, the application will initialize Chord using given node's address (and eventually the bootstrap information) and waits for the user's input.
2. The user can send a private message to a different user in the same network using a command (`msg <other node's id> <message>`).
   The message will be addressed to the proper client given by the id. The receiving client will verify if the id equals its own key, and eventually show the received message to the user.

   Note. *Items 1 and 2 are already implemented in the provided example.*

   Note. *Every node outputs its id on startup so that you know it.*

3. The user can join a chosen channel using a command (`join <channel>`). The application will send a join request by hashing the channel name – Chord will deliver the request to the client whose own key is the closest to the hashed channel key. This client becomes responsible for the channel – it maintains a list of joined participants and delivers channel messages to them (see item 4). For this purpose, the join request should contain an appropriate identification of the joining client.

4. The user can leave a channel using a command (`leave <channel>`). Subsequent channel messages will not be resent to him.

5. The user can send a message to a channel using a command (`send <channel> <message>`). The channel is addressed the same way as in the previous item. The channel maintainer will resend this message to all clients in the channel. The client that sends the message does not have to join the channel – if it does, the message is sent back to the sender as well. (Obviously, the channel maintainer does not have to join the channel it maintains.)

   Note. *The first part of assignment assumes, that no clients join or leave the overlay network after all the channels are established. However, when a new client joins the network, subsequent messages addressed to some channel might be delivered to him instead of the current maintainer. Similarly, when a client maintaining a channel leaves the network, the messages will be delivered to other clients, who may not know the lists of channel participants. The goal of the subsequent tasks is to at least partially alleviate these shortcomings.*

6. When a new client joins the overlay network, his successor should (in the upcall registered by `Local.set_notify_handler()`) check all the channels he has whether they still belong to him (`Local.is_ours()`). All the channels that do not belong to him anymore, it should migrate to his new predecessor. The old maintainer will perform the migration by sending the list of all participants to the new maintainer. If any clients join the channel via the new maintainer before the migration is completed, the migration will join the lists of old and new participants to a union. Manipulation with the channels and the migration process shall be thoroughly documented, at least by text messages.

7. The user can terminate the application using a command (`exit`). The application should first leave all channels that the user has joined (for this, it has to remember them locally). Then it migrates all channels it maintains to its successor (its address can be obtained by the `Local.successor()` function) and exits.

8. The `msg` command changes to `msg <peer's name> <message>`. For this to happen, every node after its startup has to send a “set name map” message (by hashing his name) with his address. The receiver of this message will remember mapping from the user's name to it's node. For this purpose, the “set name map” request message should contain an appropriate identification of the sending client. When the `msg` command is invoked, then the client has to convert the name to an identification of a client by sending a “get name map” message to the appropriate node. Whent it has the node's identification, it can send him the message. In order for this mechanism to work when nodes are joining and leaving, the nodes keeping these mapping must migrate them when they are leaving or they get a new predecessor the same way as they migrate maintained channels. On exit, the node shall delete the mapping from its name to its id in the network.

Consider situations, where even this solution may fail. How could such cases be remedied?