Distributed Systems: Lab 3
Middleware: Peer-to-Peer Communications

Petru Eles and Ivan Ukhov
petru.eles@liu.se and ivan.ukhov@liu.se

January 24, 2016

1 Introduction

The goal of the current lab is to make the code written for Lab 2 [1] properly handle situations when peers arbitrary join and leave your distributed system. This functionality will be delegated to a separate class called `PeerList`, which will take advantage of the object request broker implemented as a part of the previous lab. In addition, you will introduce interactions between peers themselves, i.e., peers will be remotely invoking methods [2] of each other (as you remember, so far peers were communicating only with the name service). In order to put all these pieces together and ensure that they work properly, you will implement a test application for exchanging text messages between peers. Such a system of messengers is depicted in Figure 1.

2 Your Task

2.1 Preparation

Continue working with the same code base that you have been working with so far, including all the changes that you have made. The files relevant to this lab are listed below. You should read and understand them.

- `lab3/chatPeer.py` — the chat application (no changes are needed);
- `lab3/test.sh` — a shell script that you can use for testing;
- `modules/Common/nameServiceLocation.py` — the same as for Lab 2 [1] (no changes are needed);
- `modules/Common/objectType.py` — the same as for Lab 2 [1] (should contain your previous changes);
- `modules/Common/orb.py` — the same as for Lab 2 [1] (should contain your previous changes);
- `modules/Common/wrap.sh` — the same as for Lab 1 [3];
- `modules/Server/peerList.py` — the module that maintains a list of peers of a given object type (should be modified).

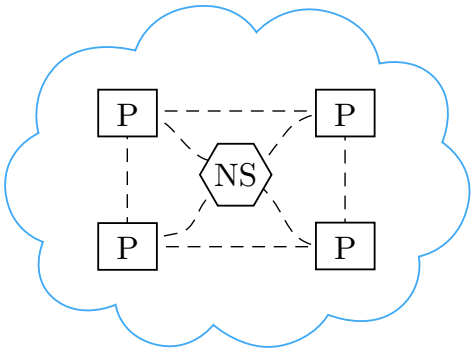

Figure 1: A name service (NS) and a number of chat-peers (P). First, the peers discover each other by means of the name service, and then they start exchanging messages directly.

The main functionality of this lab is concentrated in `peerList.py`, which contains the `PeerList` class. As the name suggests, this class represents a list of peers, and it is utilized as follows. Each peer has an instance of `PeerList`, and this instance maintains local images, that is, `Stubs`, of all the peers that are currently present in the network (restricted to your name space specified by your object type[1]). Therefore, whenever a peer wants to invoke a method of some other peer, it just needs to find the addressee in the peer list and make a call to the corresponding image; the rest will be done automatically by the object-request-broker functionality that you implemented earlier. Carefully read the code in `peerList.py` and understand what it does. In particular, ensure that you understand the purpose of the locks utilized throughout the module.

Run `chatPeer.py` in several terminal windows. If you have overwritten `orb.py` with your correct implementation from Lab 2 [1], the application should start without any errors and display its menu as shown below:[2]

```
$ ./chatPeer.py -t ivan
Choose one of the following commands:
    l                       ::  display the peer list,
    <PEER_ID> : <MESSAGE>   ::  send <MESSAGE> to <PEER_ID>,
    h                       ::  print this menu,
    q                       ::  exit.
ivan(1)> l
List of peers of type 'ivan':
ivan(1)> 2 : Can anybody hear me?
Cannot send messages to 2. Make sure it is in the list of peers.
ivan(1)>
```

Unfortunately, this instance of the application cannot see the other instances, which are running in parallel, and, therefore, cannot send messages to them.

[1] Object types were discussed in Lab 2 [1].

[2] As you have probably discovered in the code, you can specify your object type with the `-t` command argument instead of editing `objectType.py`.

2.2 Implementation

The `PeerList` class has two missing parts, which you are asked to implement. More specifically, the two parts are in the `initialize` and `destroy` functions, and you should complete them as follows:

- `initialize` — the function populates the peer list of a newly started peer and notifies the other peers about this event. To this end, the function should (a) request the peer list of your name space from the name service and (b) register the current peer in each peer from the obtained list.
- `destroy` — the function is called whenever the peer that owns this instance of `PeerList` leaves the system. The function should unregister the peer by calling an appropriate method of each peer in the peer list.

As usual, there is the following requirement to your implementation:

- Your peers should be resistant to potential exceptions that can happen during remote method invocation; ensure that your peers stay alive.

After the implementation has been completed, run the system once again and make sure that now you can exchange messages without any problems. Use the test script to start several clients at once.

3 Conclusion

At this point, you have a rather solid ground for your future distributed database of fortunes. Now you can perform remote method invocations in a straightforward manner, thanks to your object request broker and peer list. However, we shall postpone the integration of this code with the database-related code from Lab 1 [3] until the final lab. The reason is that there is one crucial component which is still missing. This component is a distributed lock [4], which you will closely look at in Lab 4 [5].

References

[1] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab2.pdf.

[2] https://www.ida.liu.se/~TDDD25/lectures/lect3.frm.pdf.

[3] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab1.pdf.

[4] https://www.ida.liu.se/~TDDD25/lectures/lect6-7.frm.pdf.

[5] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab4.pdf.