Distributed Systems: Lab 5
Client-Server Database with Replicas

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

December 20, 2015

1 Introduction

Let us summarize what has been achieved so far. In the first two labs, counting from zero, you implemented the basic functionality of a database following the client-server model. Specifically,

- the user can perform the read and write operations on the data, that is, read random fortunes and compose new ones (Lab 0 [1]);
- there is only one server machine responsible for managing the data, and there can be many clients that can connect to the server in order to issue the two possible commands (Lab 1 [2]).

In the last three labs, from two to four, you built the core components of a distributed system. These components are as follows:

- an object request broker, which, together with the name service, provides a handy middleware layer for your system (Lab 2 [3]);
- a smart peer list, which adequately keeps track of the peers that are currently present in the network (Lab 3 [4]);
- a distributed lock, which guards shared resources from being left in corrupted states due to concurrent operations (Lab 4 [5]).

The goal of this lab is to improve the naïve database obtained at the end of Lab 1 [2] by fusing it with the three components of a distributed system listed above. The target configuration of the system is displayed in Figure 1, in which each peer[1] maintains a copy of the data (that is, a list of fortunes) in such a way that this copy is kept consistent with the copies of other peers. In order to achieve this consistency, all writing operations performed on one copy of the data (owned by one of the peers) are to be properly propagated to all other copies (located at the rest of the peers). Having done so, any reading operation issued by a client will be guaranteed to see the same data regardless of the peer chosen for reading.

[1]The words *peer* and *server* are used interchangeably in our context. Similarly, the words *user* and *client* have the same meaning here.

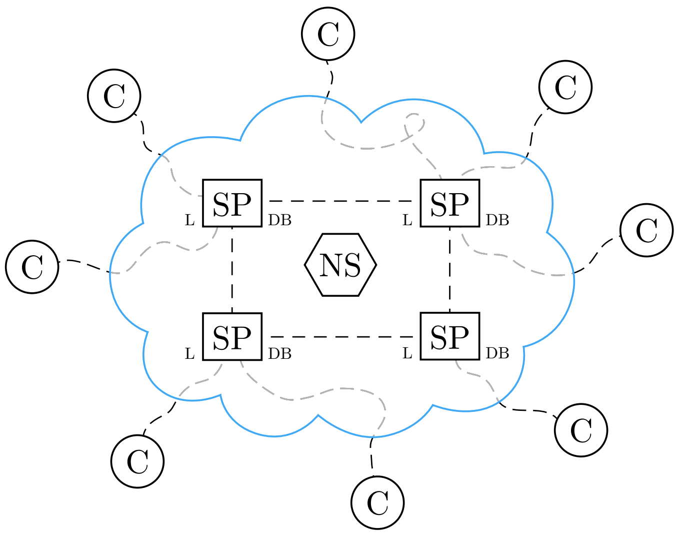

Figure 1: A distributed database with a number of servers/peers (SP) and a number of clients (C). Each server maintains a copy of the database (DB) protected by a distributed lock (L). The discovery process is facilitated by virtue of a name service (NS).

In the scenario described above, the peers duplicating the data are typically referred to as replica managers. This concept is of high importance for the current lab, and, therefore, it might be a good idea to get yourself familiar with it before diving deeper into the lab. All the needed information on the subject can be found in the corresponding lecture notes [6]. In particular, make sure you understand the read-any/write-all protocol, which our replica managers will be assumed to follow.

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.

- `lab5/client.py` — the client application representing a user of the database (no changes are needed);
- `lab5/serverPeer.py` — the server application representing a replica manager (should be modified);
- `lab5/test.sh` — a shell script that you can use for testing;
- `lab5/dbs/fortune.db` — a text file with a list of fortunes (the format is described in Lab 0 [1]);

- `modules/Common/nameServiceLocation.py` — the same as for Lab 2 [3] (no changes are needed);
- `modules/Common/objectType.py` — the same as for Lab 2 [3] (should contain your previous changes);
- `modules/Common/orb.py` — the same as for Lab 2 [3] (should contain your previous changes);
- `modules/Common/wrap.sh` — the same as for Lab 1 [2];
- `modules/Server/database.py` — the same as for Lab 1 [2] (should contain your previous changes);
- `modules/Server/peerList.py` — the same as for Lab 3 [4] (should contain your previous changes);
- `modules/Server/lock/distributedReadWriteLock.py` — a distributed version of the read/write lock given by `ReadWriteLock` using the distributed lock given by `DistributedLock` (should be modified);
- `modules/Server/lock/distributedLock.py` — the same as for Lab 4 [5] (should contain your previous changes);
- `modules/Server/lock/readWriteLock.py` — the same as for Lab 1 [2] (no changes are needed).

Study the code of the two main applications given in the source files `client.py` and `serverPeer.py` and try to grasp the main idea behind the scene. It might be helpful to run the code. To this end, you need to have at least one instance of each of the applications. Here is an example:

```
$ ./serverPeer.py -t ivan
Choose one of the following commands:
    l  ::  list peers,
    s  ::  display status,
    h  ::  print this menu,
    q  ::  exit.
ivan(1)>
...
$ ./client.py -t ivan
Connecting to server: (u'130.236.205.175', 45143)
None
```

It is apparent that the system does not work: the replica manager, running in one terminal window, returned `None` to the user, running in another terminal window, despite the fact that `fortune.db` is non-empty.

2.2 Understanding the Setup

An instance of `serverPeer.py` is a server/peer that maintains a local copy of the data and collaborates with other servers in order to keep this copy up-to-date. An instance of `client.py` is a client/user of the distributed database that connects to one of the peers and can issue the read/write operations via the corresponding text menu (similar to the first two labs). A client chooses a server to connect to by virtue of the name service, which you can easily tell by looking at the corresponding code. More specifically, in `client.py`, you can discover two new functions that the interface of the name service has:

- `require_any` — the function returns the address of a randomly chosen peer, that is, a randomly chosen replica manager;
- `require_object` — the function returns the address of a specific peer (in case you want to connect to a particular server for debugging purposes).

These functions were intentionally omitted in the description of the interface of the name service given in Lab 2 [3].

Another important aspect to realize is that there two types of concurrent accesses that might occur with respect to each peer. The first type can be labeled as "local," and it is due to the same reasons as the ones described in Lab 1 [2]: one server can serve several clients simultaneously in separate threads. Consequently, the thread-safeness should still be ensured, which was previously the job of the `ReadWriteLock` class. The second type can be labeled as "global" or "distributed," and it is due to the fact that another peer might try to synchronize its data with the data of the current peer. It happens when a client connected to a peer writes a new fortune into the local database of that peer, and this new fortune needs to be pushed into the local databases of all other peers. Consequently, such situations should be also properly resolved, and your `DistributedLock` is in high demand in this context.

2.3 Implementation

As you can see in the source code of this lab, the system has already been assembled for you. Assuming that you have properly transfered your (hopefully correct) implementations from the previous lab to the current one (see Section 2.1), there is only one problem to solve.

Neither `ReadWriteLock` nor `DistributedLock` is sufficient by itself. The former is not aware of the distributed nature of the system, and the latter is not capable of distinguishing between read and write operations. This leads to inefficiency, which you should be able to explain. Therefore, your task now is to merge the two locks to produce a distributed read/write lock. The class to look at is `DistributedReadWriteLock` in `distributedReadWriteLock.py`.

Having completed `DistributedReadWriteLock`, you are asked to utilize the lock in order to finish the implementation of `serverPeer.py` following the read-any/write-all policy [6]. Specifically, you will have to write two functions with rather suggestive names: `read` and `write`.

3 Conclusion

Congratulations! You have successfully completed the final lab of the course. In this lab, you have put together many of the ideas that you learned previously and have created a robust distributed database, which is able to serve many clients in parallel balancing the workload among several servers spread out across the network. We hope that you enjoyed this programming project. Good luck!

References

[1] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab0.pdf.

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

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

[4] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab3.pdf.

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

[6] http://www.ida.liu.se/~TDDD25/lectures/lect8.frm.pdf.