Distributed Systems: Assignment 5
Client-Server Database with Replicas

Petru Eles, Sergiu Rafiliu, and Ivan Ukhov
{firstname.surname}@liu.se

January 14, 2013

1 Introduction

Over the last three assignments, you have implemented all the needed components for the completion of the distributed database of fortunes started in Assignment 0 and Assignment 1. Specifically, now you have

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

The goal of the last assignment is to improve the naïve distributed database obtained at the end of Assignment 1 by fusing this database with the components listed above. We are aimed at the final configuration of the system wherein each peer maintains a copy of the data (our fortunes) in such a way that this copy is kept consistent with the copies of other peers. This scenario is displayed in Figure 1. To achieve the desired behavior, all writing operations on one copy of the data are to be properly propagated to the rest of the copies. In this case, any reading operation is guaranteed to operate on the same data no matter which peer is chosen for reading. The peers in such a system are typically called replica managers [1]. For simplicity, we assume that the protocol that a replica manager should follow is the read-any/write-all protocol [1].

2 Your Task

2.1 Preparation

Download [2] and extract the source code for the assignment. In the archive, you will find the following files:

- `lab5/serverPeer.py` — the server/peer application (each instance of this application represents a replica manager);

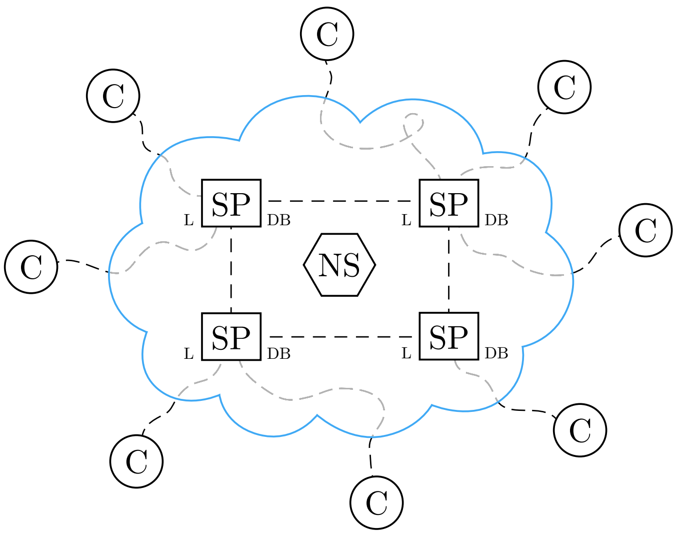

Figure 1: A distributed database with a number of servers/peers (SP) and a number of clients (C). Each server maintains a copy of the database (DB) protected by a distributed lock (L). The discovery process is facilitated by a name service (NS).

- `lab5/client.py` — the client application (each instance of this application represents a user of the database);
- `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 Assignment 0);
- `modules/Server/lock/distributedReadWriteLock.py` — the class containing a distributed version of the read/write lock `ReadWriteLocky` using the distributed lock `DistributedLock` (further explained in Section 2.2);
- `modules/Server/lock/distributedLock.py` — the same as for Assignment 4;
- `modules/Server/peerList.py` — the same as for Assignment 3;
- `modules/Common/nameServiceLocation.py` — the same as for Assignment 2;
- `modules/Common/objectType.py` — the same as for Assignment 2;
- `modules/Common/orb.py` — the same as for Assignment 2;
- `modules/Server/lock/readWriteLock.py` — the same as for Assignment 1;
- `modules/Common/wrap.sh` — the same as for Assignment 1.

Update the files with what you have already implemented for the previous assignments. Read and understand the two main applications, `serverPeer.py` and `client.py`; also, recall the purpose of `readWriteLock.py`.

In order to test the code, you need at least one instance of each of the main programs, *i.e.*, `serverPeer.py` and `client.py`. Here is an example:

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

As you might have already guessed, the system does not work: the replica manager, running in one terminal, returned `None` to the user, running in another terminal, despite the fact that the database is not empty.

2.2 Implementation

In Assignment 1, in order to handle concurrent attempts of interaction with the database, the server was using `ReadWriteLock` defined in `readWriteLock.py`. In the current scenario, this locking mechanism cannot be utilized directly as it is not sufficient for ensuring the consistency of the data across all the copies. Fortunately, you have written in Assignment 4 a distributed lock, `distributedLock`. The problem with the distributed lock, however, is that it is not capable of distinguishing read and write operations as `ReadWriteLock` is. This leads to inefficiency, which you should be able to notice and 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 `DistributedReadWriteLock` completed, you are asked to utilize the lock in order to complete the implementation of `serverPeer.py` following the read-any/write-all policy [1]. Specifically, you will have to write two functions with rather suggestive names: `read` and `write`.

3 Conclusion

Congratulations! You have successfully completed the final programming assignment of the course! In this assignment, you have put together many of the ideas you learned previously and have created a robust, distributed database that is able to serve many clients in parallel balancing the workload among several replica managers. We hope that you enjoyed this programming part of Distributed Systems. Good luck!

References

[1] http://www.ida.liu.se/~TDDD25/lecture-notes/lect8.frm.pdf.

[2] http://www.ida.liu.se/~TDDD25/labs/assignment5.zip.