Distributed Systems: Project 5[1]
Client-Server Database With Replicas

David Gillman

November 27, 2015

1 Introduction

In Projects 0 and 1 you implemented the basic functionality of a client-server database:

- the user can perform reads and writes on the data; that is, read random fortunes and compose and store new ones;
- there is only one server machine managing the data, and many clients can connect to the server to issue read and write requests.

In Projects 2, 3, and 4, you built the core components of a distributed system:

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

The goal of this project is to improve the naïve database of Project 1 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[2] 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 propagated to all other copies (on the rest of the peers). As a result, any read operation issued by a client will be guaranteed to see the same data regardless of the peer chosen for reading.

In the scenario described above, each peer that owns a replica of the data is called a *replica manager*, an important concept for the current project. You can find all you need to know about replica managers in the lecture notes. In particular, make sure you understand the read-any/write-all protocol, which your replica managers will follow.

[1] Based on an project by Petru Eles, Distributed Systems, Linköping University.

[2] In this context the peers are the servers.

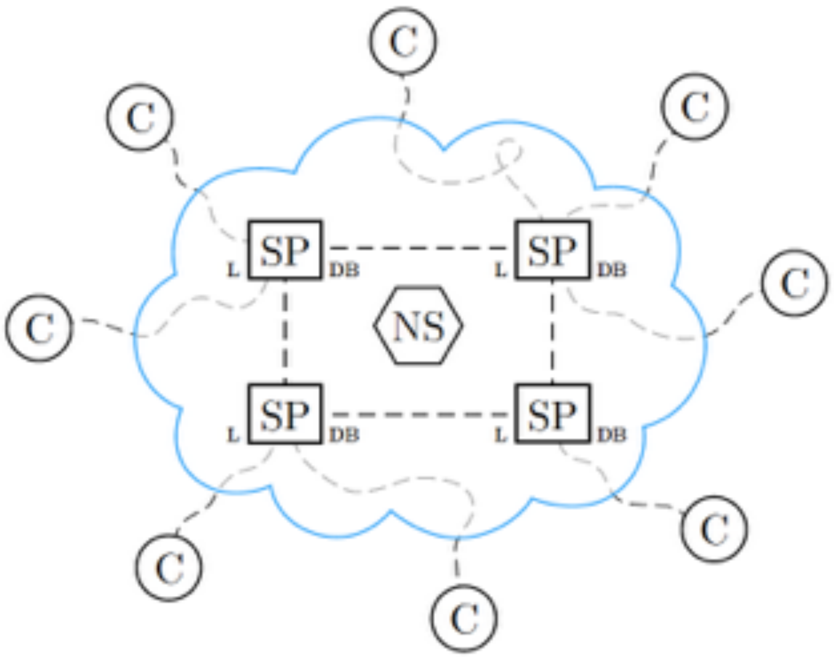

Figure 1: A distributed database with a number of servers/peers (SP) and a number of clients (C). Each server maintains a copy of the database (DB) protected by a distributed lock (L). The discovery process is facilitated by virtue of a name service (NS).

2 Your Task

2.1 Preparation

In the archive, you will find the following files:

- `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 Project 0);
- `modules/Common/nameServiceLocation.py` — the same as for Project 2 (no changes are needed);
- `modules/Common/objectType.py` — the same as for Project 2 (update according to your previous changes);
- `modules/Common/orb.py` — the same as for Project 2 (overwrite with your previous implementation);
- `modules/Common/wrap.sh` — the same as for Project 1;
- `modules/Server/database.py` — the same as for Project 1 (overwrite with your previous implementation);

- `modules/Server/peerList.py` — the same as for Project 3 (overwrite with your previous implementation);
- `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 Project 4 (overwrite with your previous implementation);
- `modules/Server/lock/readWriteLock.py` — the same as for Project 1 (no changes are needed).

Study the code of the two main applications given in the source files client.py and serverPeer.py and learn the underlying idea. 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 interaction:

```
$ ./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
```

The system does not as yet 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 (as in Projects 0 and 1).

A client chooses a server to connect to by virtue of the name service. You can see this by looking at the code. Specifically, in client.py, you'll find two new functions in the interface of the name service:

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

These functions were omitted in the description of the interface of the name service given in Project 2.

It is important to notice that there two types of concurrent accesses that can made against each peer. The first type, “local,” was described in Project 1: one server can serve several clients simultaneously in separate threads. The thread-safeness of local accesses should still be ensured by the ReadWriteLock class from Project 1.

The second type of access is called “global” or “distributed”: another peer can try to synchronize its data with the data of the current peer. This 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. You must write your code to resolve such situations. Your DistributedLock is necessary in this context.

2.3 Implementation

As you can see in the source code of this project, the system has already been assembled for you. Assuming that you have properly transferred your (correct!) implementations from the previous project 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 (explain!). 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 written DistributedReadWriteLock, you must use the lock to finish the implementation of serverPeer.py following the read-any/write-all policy. Specifically, you will have to write two functions with the (natural) names `read` and `write`.

3 Conclusion

Congratulations! You have successfully completed the final programming project of the course. In this project, 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!