Distributed Systems: Lab 4
Middleware: Distributed Locks

Petru Eles and Adrian Horga
petru.eles@liu.se and adrian.horga@liu.se

December 6, 2018

1 Introduction

The database of fortunes that you are developing in this programming project is an example of a resource which is distributed across a network. Eventually, each peer will have a local copy of the data, and this copy will be kept consisted with all other copies. Having achieved this goal, any client/user connected to the database of any peer will see the same data, and any operation performed on these data will be automatically propagated to all other peers and, thus, to all other clients. As this point, it might be helpful to have a look at the first illustration given in the documentation for Lab 0 [1]. It is apparent that several peers might try to access shared resources simultaneously. Thus, such resources have to be protected in order to eliminate corrupted states of the data.

The goal for this lab is to implement an algorithm for distributed mutual exclusion, which will address the aforementioned concern. This locking-enhanced configuration of the system is depicted in Figure 1. More specifically, we shall focus on the second Ricart–Agrawala algorithm [2] (the one based on tokens). In a nutshell, using this algorithm, peers will be able to uniquely determine which of them is allowed to enter its critical section, i.e., to operate on the shared resource. This will be achieved by virtue of a so-called token, which will be passed between the peers according to a certain strategy.

Your foremost task is to get yourself familiar with the Ricart–Agrawala algorithm by reading the corresponding lecture notes. Apart from the material given in [2], you might also want to refresh your knowledge on logical clocks described in [3] as they are an essential part of the algorithm.

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.

- `lab4/mutexPeer.py` — the main application (no changes are needed);
- `lab4/test.sh` — a shell script that you can use for testing;

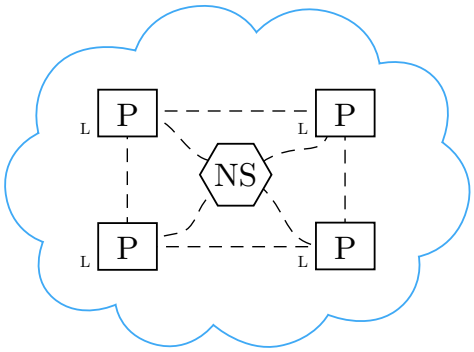

Figure 1: A name service (NS) and several peers (P) enhanced by a distributed locking mechanism (L).

- `modules/Common/nameServiceLocation.py` — the same as for Lab 2 [4] (no changes are needed);
- `modules/Common/objectType.py` — the same as for Lab 2 [4] (should contain your previous changes);
- `modules/Common/orb.py` — the same as for Lab 2 [4] (should contain your previous changes);
- `modules/Common/wrap.sh` — the same as for Lab 1 [5];
- `modules/Server/peerList.py` — the same as for Lab 3 [6] (should contain your previous changes);
- `modules/Server/Lock/distributedLock.py` — the distributed lock (should be modified).

As usual, you should reuse your implementations from the previous labs and overwrite the corresponding files listed above.

Carefully go through the code in `mutexPeer.py` and `distributedLock.py`. The former is the main executable file of this lab. It is a simple test application serving the only purpose of acquiring and releasing the distributed lock in `distributedLock.py` by choosing the corresponding commands from the menu of the application (see below). Run several instances of `mutexPeer.py` in parallel and try to acquire the lock in all of them simultaneously:

```
$ ./mutexPeer.py -t ivan
Choose one of the following commands:
    l  ::  list peers,
    s  ::  display status,
    a  ::  acquire the lock,
    r  ::  release the lock,
    h  ::  print this menu,
    q  ::  exit.
ivan(0):RELEASED> a
Trying to acquire the lock...
```

```
ivan(0):LOCKED>
...
ivan(1):LOCKED>
...
ivan(2):LOCKED>
...
```

As you can see, all the peers have successfully entered their critical sections at the same time; in other words, our distributed lock does not work.

2.2 Implementation

Your task is to implement the second Ricart–Agrawala algorithm for distributed mutual exclusion. Read the description of the algorithm given in [2] and complete the following functions of the `DistributedLock` class:

- `initialize` — the function initializes the state of the lock for the current peer based on a populated peer list; make sure that the token is initially granted to only one peer.
- `destroy` — called when the current peer leaves the system; if the peer has the token, it passes the token to somebody else.
- `register_peer` — called when some other peer (not the current one) joins the system; the newcomer should be properly taken into account.
- `unregister_peer` — called when some other peer (not the current one) leaves the system; the quitter should be excluded from consideration.
- `acquire` — called when the current peer tries to acquire the lock; if the token is not present, the peer should notify the rest about its desire and suspend its execution until the token is passed to the peer.
- `release` — called when the current peer releases the lock; if there are peers waiting for the token, the current peer should pass the token to one of them (carefully think through to whom in order to ensure fairness).
- `request_token` — called when some other peer requests the token from the current peer (should the current one have the token or not).
- `obtain_token` — called when some other peer gives the token to the current peer; if the current peer is waiting for the token (and, thus, its execution is suspended in `acquire`), it should resume its execution.

Make sure that the algorithm properly handles situations when peers dynamically join and leave the system. For simplicity, you may assume that the peer holding the token never disappears (e.g., due to a system crash) without passing the token to someone else.

3 Conclusion

In this lab, you have implemented one of the algorithms for distributed mutual exclusion. As motivated in the introduction, such an algorithm is a crucial

component of a distributed system as it makes it possible to maintain shared resources in consistent states. The final step is to integrate all the components that you have implemented so far into a robust distributed database, in which an arbitrary number of peers will be maintaining the content of the database, and an arbitrary number of clients/users will be able to seamlessly connect to this database and to safely perform all the needed operations.

References

[1] https://gitlab.ida.liu.se/TDDD25/labs/raw/master/doc/lab0.pdf.

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

[3] https://www.ida.liu.se/~TDDD25/lectures/lect5.frm.pdf.

[4] https://gitlab.ida.liu.se/TDDD25/labs/raw/master/doc/lab2.pdf.

[5] https://gitlab.ida.liu.se/TDDD25/labs/raw/master/doc/lab1.pdf.

[6] https://gitlab.ida.liu.se/TDDD25/labs/raw/master/doc/lab3.pdf.