CESC 327
Assignment 4: Distributed File System

Professor Oscar Morales

Due date November 21

1 Distributed File System

The objective of this assignment is to implement a distributed file system that is tolerant to failures. The core of the system is the implementation of an atomic commit protocol and a request-reply protocol.

There will be two components. The Distributed File System Service (DFSS) and The Distributed File System Client (DFSC). DFSS will be the implementation of the Chord that supports file replication and transaction. However, this time the user cannot write the commands directly. Instead, it implements the server side of a request-reply protocol with the following commands:

```
write(filename): upload filename to the chord (with replication)
write(directory): upload all the files in  directory to the
     DFSS (with replication)
read(filename): download filename from DFSS in the user space.
delete(filename): remove filename from the DFSS.
```

The DFSC is a simple program that implements the client side of the request-reply protocol. You decide your own user interface.

2 Chord with replication

Each file will be stored in three different peers. Given a filename *name*, we create three guids by applying the hash function to *name* concatenated with

i where $i \in [1, 2, 3]$. Since we use only a few peers, the guids will be module K. In other words:
$guid1 = md5(name + 1) \bmod K$,
$guid2 = md5(name + 2) \bmod K$ and
$guid3 = md5(name + 3) \bmod K$.

The replica i will be stored in the successor peer of $guid1$, $guid2$ and $guid3$, respectively. If the ring is enough large, the peers will be different with high probability. However, with 6 peers, it is a good chance that the peers are the same.

To guarantee that the files will be the same, we will use the atomic commit protocol described in Section 3. In an Atomic-Commit Protocol, the transaction is either all or nothing. Observe that two transaction can be executed concurrently by two peers which may result in having an inconsistent file. Also a peer can join or leave while transactions are being performed.

Therefore, when there exists two transaction t_1 and t_2 modifying the same file such that t_1 happened before t_2, $(t_1 \rightarrow t_2)$, the peer has to emit a negative vote to t_2. During the protocol, the peers that are storing the file are the participants and the coordinator is the peer that is trying to store the files.

To read a file *name* from the distributed file system, it first try to get the file $guid1 = md5(name + 1)$. If does not exist, then $guid2 = md5(name + 2)$. Finally, if it does not exists then $guid3 = md5(name + 3)$. We consider the deletion of a file as another transaction.

The transaction is given by

```
class Transaction
{
  public enum Operation { WRITE, DELETE}
  public enum Vote { YES, NO}
  Integer TransactionId;
  Integer guid;
  public enum Operation { WRITE, DELETE};
  Vote vote;
  FileStream fileStream;
  public Transaction(Operation op)
  {
   //id = md5(date + ip+port);
   this.op = op;
  }
```

```
  ...
}
```

3 Atomic Commit Programming Interface

- $canCommit?(trans) \rightarrow Yes/No$: Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.
- $doCommit(trans)$: Call from coordinator to participant to tell participant to commit its part of a transaction.
- $doAbort(trans)$: Call from coordinator to participant to tell participant to abort its part of a transaction.
- $haveCommitted(trans, participant)$: Call from participant to coordinator to confirm that it has committed the transaction.
- $getDecision(trans) \rightarrow Yes/No$: Call from participant to coordinator to ask for the decision on a transaction when it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.

3.1 Protocol

4 Grading

Criteria	Weight
Documentation of your program	15%
Source code (good modularization, coding style, comments)	15%
Execution output	
Write	10%
Read	10%
Writing a directory	5%
Atomic Commit	25%

- Phase 1 (voting phase):

    1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
    2. When a participant receives a *canCommit*? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving the transaction in permanent storage (use a temporal directory, e.i., $./i/temp/$). If the vote is No, the participant aborts immediately.

- Phase 2 (completion according to outcome of vote):

    1. The coordinator collects the votes (including its own). (a)If there are no failures and all the votes are Yes, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
    (b)Otherwise, the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted Yes.
    2. Participants that voted Yes are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

It is strongly recommended that students that have completed the third assignment implement these functionalities in the same project. Those students that did not complete the assignment can use the skeleton that implements writes and read of the previous assignment.

Students that want a more challenging project, instead of implementing the atomic commit previously described, can implement the Paxos protocol described in either [1] or [2]. The assignment with paxos will count 12% points of the final grade.

References

[1] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[2] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42, 2015.