Replicated Reliable Banking System

You may work in the same teams as you did the in-class presentations. The project must be done in C/C++ in a Linux or Unix environment. Neither Java nor Windows are permitted. You may use threads (Pthreads) instead of separate processes for each client request. Messages must be in the form of RPCs or UDP sockets using C. I think RPCs over UDP is probably the most efficient communication once you understand the interface.

The goal is to implement a reliable banking system. Keywords for this assignment are fault-tolerance, resynchronization using logs, centralized locking, reliable.

Create an online banking system with three replicated servers and a coordinator. The purpose of the coordinator is to act as an interface to the bank servers. Each client sends its request to the coordinator which in turn forwards the request to the all three servers. Additionally, the coordinator fetches any response from any of the active servers and sends it back to the client.

1. **Replicated Database Servers (3)**
   * ***Replication:***  
     All accounts of the bank are replicated on three servers. For the assignment assume that all accounts are already created and the client operations are for **deposit, withdraw and balance check**. Further, information of accounts is to be stored and updated in a file (and not in memory) as we are going to test for fault--tolerance (where servers fail and wakeup and need to restore state of previous operations.), i.e., if system started with account# 101 having $100, and when server I crashed it had $150, on wakeup server I should know that account# 101 balance is $150, unless server 2 or 3 updated it (which is dealt with using logs at both servers).
   * ***Logging:***  
     Each server maintains a log for each operation performed, which can be of the form
   * Operation# Account# Balance
   * ------------------------------
   * 1 101 2000
   * 2 102 5000
   * 3 105 500
   * .
   * .
   * .

The log must be stored/updated in a file for persistence. Each server uses a different log file. They do not share a single log file.  
Additionally, all account information is stored in files and hence all update operations also modify records in the account information file.

* + ***Resynchronization and Wakeup:***   
    When a server fails and wakes-up it needs to resynchronize with the other replicas in-order to restore a consistent state for each account.   
    On wakeup, the server sends an **ALIVE** message to the coordinator to signal its wakeup.   
    Next, the server sends a **resynchronize** command to the other servers, along with the last operation number in its log. The servers that had not failed then look up their logs and send all [account #, balance] tuples which the failed server has not seen. The operation number sent from the failed server is used to get the starting point for the log replay. The servers send all log tuples and also results of operations being performed when the resynch command arrived.  
    The assignment assumes only one server can fail at a time and hence at least two servers always service requests and keep the log up-to date.   
    Once the resynchronization is complete, the server can flush(delete) its log as all three servers are consistent. Also, the server that had failed now being in a consistent state sends a **RESYNCH-DONE** message to the coordinator.

1. **Coordinator**   
   Each client request is forwarded to the coordinator, which in-turn forwards the request to **all three** servers or the servers that have not failed.
   * ***Account--level locking:***  
     For each request that reaches the coordinator, it locks the account the operation is going to update (withdraw/deposit operations). So, if two operations for the same account simultaneously arrive at the coordinator, one of them will be queued till the first one finishes and the lock released. Locking will have to be done through message passing since the servers and coordinator should be on different hosts.   
     Each server signals an end-of-operation with an **OP-DONE** message, which can of be of the form [DONE, Account#]. On receiving OP-DONE messages from all awake servers, the lock on the account is released.
   * ***HeartBeat Messages:***  
     The coordinator uses periodic heartbeat messages to determine the state of each server. Based on the number of servers (2 or 3 in this case), it waits for the corresponding OP-DONE messages to release the lock. Additionally, the periodic heartbeat messages will indicate whether a server failed after getting the lock. In this case, the coordinator only waits for OP-DONE messages from the alive servers.
   * ***Resynchronization:*** When resnchronization is in progress, no requests are forwarded by the coordinator. The resynchronization starts with an **ALIVE** message from a server and ends with a **RESYNCH-DONE**
   * ***Client Response:***  
     The coordinator forwards the result of the operation after the locks are released and operation completed to the client.

**Some things to keep in mind:**

* Assume each server has the same file with all records and account information at startup. The records in the file get updated on each update operation.
* To keep the replication simple, assume clients do not create new accounts and only query or update existing accounts.
* The system should be fault tolerant, meaning no operation should fail or make the system inconsistent due to failures.

**3 Evaluation and Measurement**

**Correctness**   
Demonstrate that your system works correctly according to requirements stated in the description and functionality of the system. In particular:

1. Show that the bank database is **distributed and replicated**, operations are sent to all three servers or two servers (based on failure).
2. Demonstrate that the account-level locking mechanism works correctly. i.e: show (via output snapshots) that simultaneous updates to an account are handled correctly by the locking mechanism.
3. Demonstrate the logging mechanism, how logs are updated on operations and maintained.
4. Demonstrate the proper functioning of the resynchronization step.   
   - between the 3 servers   
   - no requests being sent by coordinator
5. Demonstrate the working of heartbeat messages, by showing that a coordinator initially waits for 3 OP-DONE messsages, but a heartbeat discovers a server failure and waits for only 2 OP-DONE messages.

**Evaluation**  
Additionally, experiment with your system to measure its performance in different scenarios and test conditions.   
Design and present results of your own experiments to demonstrate the characteristics of the system. A few examples are:

1. Measure the average time of requests, when all servers are on and no queuing at the coordinator.
2. Measure the average time of requests, when all servers are on and there is queuing at the coordinator (i.e., queued due to account-level locking).
3. Keeping request rate of each client constant, vary the number of clients and measure the latency of each request.
4. Keep number of clients constant, but vary request rate to measure latency of each request.
5. Measure how much time resychronization requires by measuring the time between ALIVE and RESYNCH-DONE messages at the coordinator.   
   How does this change with log size?

**It is important that you describe the results of your experiment and not just describe what the experiment did. Please state what the experiment demonstrates or what you expected and what was seen etc.   
These are guidelines only, so be creative in what can be evaluated and measured as part of your experiments to test the system.**

**4 What you will submit**

When you have finished implementing the complete assignment as described above, put all the code in a separate directory in your account on one of the linux machines in Maes 208.   
  
You are required to submit electronically your solution in the form of **printouts** (please only attach relevant outputs that demonstrate your points and demonstrate functionality. (**DO NOT** printout entire output logs and source code). Each team will be required to submit a hardcopy of their source code.

Each program must work correctly and be **documented**. You should hand in:

1. Outputs generated by running your program. (in Maes 108).
2. Outputs to demonstrate correct working of the system. (in a printout from a Maes 108 account) .  
   This is important, as it will show that your system works according to the requirements.
3. A separate Users’Guide document of approximately two pages describing the overall program design, a description of "how it works", and design tradeoffs considered and made. Describe clearly how each system is designed and implemented. Also describe possible improvements and extensions to your program (and sketch how they might be made). Submit this electronically.
4. Prepare a list of design considerations you made while designing your system and describe each briefly. (in Edlab and Printout)
5. A program listing containing in-line documentation. (in Maes 108 account)
6. Instructions to compile and run the code.
7. A separate description of the tests you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly. Submit electronically.
8. Performance results to test scalability and performance parameters. Submit electronically. These results should be available in the Maes 108 account for the project.

If any of the above turn out to be large, just save the relevant information in a file, leave it on your Maes 108 account and submit the name of the file.

**5 Grading policy for all programming assignments**

**Grading:**

* Program Listing
  + works correctly ------------- 50%
  + in-line documentation -------- 15%
* Design Document
  + quality of design ------------ 15%
  + understandability of doc ------- 10%
* Thoroughness evaluation ---------- 10%