MP3: Distributed transactions

Due: Dec 8, 11:59pm

In this MP you will be implementing a distributed transaction system. You goal is to support transactions that read and write to distributed objects while ensuring full ACI(D) properties. (The D is in parentheses because you are not required to store the values in durable storage or implement crash recovery.)

Clients, Branches, and Accounts

You are (once again) implementing a system that represents a collection of accounts and their balances. The accounts are stored in five different branches (named A, B, C, D, E). An account is named with the identifier of the branch followed by an account name; e.g., `A.xyz` is account `xyz` stored at branch `A`. Account names will be comprised of lowercase english letters.

You will need to implement a server that represents a branch and keeps track of all accounts in that branch, and a client that executes transactions by communicating with all the necessary branches. You may optionally use a separate coordinator server for coordinating the transactions.

Unlike the previous MPs, you *do not have to handle failures* and can assume that all the servers remain up for the duration of the demo. Clients may exit but you do not have to deal with clients crashing in the middle of a transaction.

Configuration

Each server must take two arguments. The first argument identifies the branch that the server must handle. The second argument is a configuration file. E.g.: `./server C config.txt`

The configuration file has 5 lines, each containing the branch, hostname, and the port no. of a server. The configuration file provided to each server is the same. A sample configuration file for a cluster running on group g01 would look like this:

```
A fa21-cs425-g01-01.cs.illinois.edu 1234
B fa21-cs425-g01-02.cs.illinois.edu 1234
C fa21-cs425-g01-03.cs.illinois.edu 1234
D fa21-cs425-g01-04.cs.illinois.edu 1234
E fa21-cs425-g01-05.cs.illinois.edu 1234
```

A client takes two arguments – the first argument is a client id (unique for each client), and the second argument is the configuration file (same as above), which provides the required details for connecting to a server when processing a transaction. E.g., `./client asdf config.txt`

Client Interface

At start up, the client should automatically connect to all the necessary servers and start accepting commands typed in by the user. The user will execute the following commands:

- `BEGIN`: Open a new transaction, and reply with “OK”.
- `DEPOSIT server.account amount`: Deposit some amount into an account. Amount will be a positive integer. (You can assume that the value of any account will never exceed 1,000,000,000.) The account balance should increase by the given amount. If the account was previously unreferenced, it should be created with an initial balance of `amount`. The client should reply with `OK`

```
DEPOSIT A.foo 10
OK
DEPOSIT B.bar 30
OK
```

- `BALANCE server.account`: The client should display the current balance in the given account:

```
BALANCE A.foo
A.foo = 10
```

If a query is made to an account that has not previously received a deposit, the client should print `NOT FOUND, ABORTED` and abort the transaction.

- `WITHDRAW server.account amount`: Withdraw some amount from an account. The account balance should decrease by the withdrawn amount. The client should reply with `OK` if the operation is successful. If the account does not exist (i.e, has never received any deposits), the client should print `NOT FOUND, ABORTED` and abort the transaction.

```
BEGIN
WITHDRAW C.baz 5
NOT FOUND, ABORTED
```

- `COMMIT`: Commit the transaction, making its results visible to other transactions. The client should reply either with `COMMIT OK` or `ABORTED`, in the case that the transaction had to be aborted during the commit process.
- `ABORT`: Abort the transaction. All updates made during the transaction must be rolled back. The client should reply with `ABORTED` to confirm that the transaction was aborted.

Notes:

- You should ignore any commands occuring outside a transaction (other than `BEGIN`).
- You can assume that all commands are using valid format. E.g., you will not see `DEPOSIT A.foo -232` or `WITHDRAW B.$#@% 0`.
- A transaction should see its own tentative updates; e.g., if I `DEPOSIT A.foo 10` and then call `BALANCE` on `A.foo` in the same transaction, I should see the deposited amounts. Whether updates from other transactions are seen depends on whether those transactions are committed and isolation properties, discussed below.
- If an operation requires a lock, you should not reply until the lock has been acquired. However, the user may type `ABORT` before getting a response and you must abort the transaction immediately.
- Suppose a transaction creates an account (by issuing a deposit on it for the first time), and the transaction gets aborted, the account will be considered non-existent by a future transaction. Here are a few related scenarios:
  - Suppose a transaction T1 creates an account (say by calling `DEPOSIT A.foo 10`), and a concurrent transaction T2 also issues a deposit on the same account (say by calling `DEPOSIT A.foo 30`). Also suppose that T1 is before T2 in the serial equivalence order. If T1 gets aborted and T2 is committed, the account `A.foo` would then effectively be created by T2 with a balance of 30. Moreover, if T2 issues any `WITHDRAW` or `BALANCE` commands on `A.foo` after its deposit, these commands should return successfully irrespective of whether T1 commits or aborts. Note that `A.foo`'s balance would be different depending on whether T1 commits or aborts, and the success of T2's commit would be contingent on meeting the consistency requirement specified below.
  - Suppose a transaction T1 creates an account (say by calling `DEPOSIT A.foo 10`), and a concurrent transaction T2 calls `WITHDRAW A.foo 5` (without making any prior deposits to the account). If T1 is before T2 in the serial equivalence order and T1 commits, then T2's `WITHDRAW` command should return successfully. On the other hand, if T1 is before T2 in the serial equivalence order but T1 aborts, then T2's `WITHDRAW` command should return `NOT FOUND, ABORTED`. Likewise, if T2 is before T1 in the serial equivalence order, then T2's `WITHDRAW` command should return `NOT FOUND, ABORTED`. A similar logic would hold if T2 issues `BALANCE A.foo` instead of `WITHDRAW`.
- You may, as needed, spontaneously abort a transaction while waiting for the next command from the user. The user will need to open a new transaction using `BEGIN`. E.g.:

```
BEGIN
DEPOSIT A.foo 10
OK
DEPOSIT B.bar 30
OK
ABORTED
BEGIN // next transaction begins
OK
```

Atomicity

Transactions should execute atomically. In particular, any changes made by a transaction should be rolled back in case of an abort (initiated either by the user or the server) and all account values should be restored to their state before the transaction.

Consistency

As described above, a transaction should not reference any accounts that have not yet received any deposits in a `WITHDRAW` or `BALANCE` command. An additional consistency constraint is that, *at the end of a transaction* no acccount balance should be negative. IF, when a user specifies `COMMIT` any balances are negative, the transaction should be aborted.

```
BEGIN
DEPOSIT B.bar 20
OK
WITHDRAW B.bar 30
OK
COMMIT
ABORTED
```

However, it is OK for accounts to have negative balances *during* the transaction, assuming those are eventually resolved:

```
BEGIN
DEPOSIT B.bar 20
OK
WITHDRAW B.bar 30
OK
DEPOSIT B.bar 15
OK
COMMIT
COMMIT OK
```

Isolation

You should support up to 10 simultaneous clients that execute transactions concurrently. You should guarantee the serializability of the executed transactions. This means that the results should be equivalent to a serial execution of all committed transactions. (Aborted transactions should have no impact on other transactions.) You may want to use two-phase locking to achieve this, though this is not a strict requirement. (E.g., you can implement timestamped concurrency instead.)

You *must* support concurrency between transactions that do not interfere with each other. E.g., if T1 on client 1 executes `DEPOSIT A.x, BALANCE B.y` and then T2 on client 2 executes `DEPOSIT A.w, BALANCE B.z`, the transactions should both proceed without waiting for each other. In particular, using a single global lock (or one lock per server) will not satisfy the concurrency requirements of this MP. You should support read sharing as well, so `BALANCE A.x` executed by two transactions should not be considered interfering.

On the other hand, if T1 executes `DEPOSIT A.x` and T2 executes `BALANCE A.x`, you may delay the execution of one of the transactions while waiting for the other to complete; e.g., `BALANCE A.x` in T2 may wait to return a response until T1 is committed or aborted.

Optional: Deadlock Resolution

For extra credit, you may implement a deadlock resolution strategy. One option is deadlock detection, where the system detects a deadlock and aborts one of the transactions. As discussed earlier, a client can spontaneously display `ABORTED` to the user at any point in time to indicate that the transaction has been aborted. Remember that deadlocks may span multiple servers and clients.

You should not use timeouts as your deadlock detection strategy because transactions will be executed interactively and this will therefore result in too many false positives. Likewise, you should not use lock ordering or early locking since the client interface does not allow you to specify the entire set of locks to be acquired.

On the other hand, you can use timestamped concurrency, or other strategies, that avoid deadlocks altogether, and you will receive extra credit for this part, assuming that the strategy is implemented correctly and successfully avoids deadlocks.

Submission Instructions

Unlike previous MPs, we are not asking you to perform experiments in this MP. You do need a design document that describes the following details about your implementation.

1. A walk-through of a simple transation that clarifies the roles that the clients, servers, and coordinator (if any) play; i.e., what messages are sent, what state is maintained by which of the nodes, etc.

2. An detailed explanation of your concurrency control approach. Explain how and where locks are maintained, when they are acquired, and when they are released. If you are using a lock-free strategy, explain the other data structures (timestamps, dependency lists) used in your implementation.

   If your algorithm implements a strategy that does not directly follow a concurrency strategy described in the lecture or the literature, you will also need to include an argument for why your strategy ensures serial equivalence of transactions.

3. A description of how transactions are aborted and their actions are rolled back. Be sure to mention how you ensure that other transactions do not use partial results from aborted transactions.

4. If you are implementing the extra credit, describe how you detect or prevent deadlocks.

You also need to either ensure that there are executables call `server` and `client` in your submission directory, or that you have a `Makefile` that, when running `make`, creates the two executables.

Code guidelines

The client should only print the responses to the commands or the `ABORTED` message (as described above). It should not print any additional messages.

For testing purposes, the client should support commands entered interactively via `stdin` (as exemplified above), as well as reading transaction commands one line at a time from a file redirected to `stdin`. For example, an input test file, `in.txt`, may contain the following lines:

```
BEGIN
DEPOSIT A.foo 20
COMMIT
BEGIN
DEPOSIT A.foo 30
WITHDRAW A.foo 10
COMMIT
BEGIN
BALANCE B.bar
DEPOSIT C.zee 5
COMMIT
BEGIN
BALANCE A.foo
COMMIT
```

Running `./client config.txt < in.txt` should print the following output:

```
OK
OK
COMMIT OK
OK
OK
OK
COMMIT OK
OK
NOT FOUND, ABORTED
OK
A.foo = 40
COMMIT OK
```

Any command entered after a transaction has been aborted, and before the next `BEGIN` command, must be ignored. In this case, the 10th and 11th lines of the input file (`DEPOSIT C.zee` and `COMMIT`) get ignored by the client, as they occur “outside a transaction”.

Every time a server commits any updates to its objects, it should print the balance of all accounts with non-zero values. Other than the above, the servers should not print any additional messages.

Graphs

You do not need to perform any experiment, or plot any graphs for this MP.

High-level Rubric

- Correct submission format and build instructions: 10 points
- Design document: 15 points
  - Design grades granted for implemented functionality only
- Functionality testing: 75 points
  - Atomicity: 20 points
  - Consistency: 20 points
  - Isolation: 35 points
- Extra credit:
  - Design: 5 points (granted only if functionality is correct)
  - Functionality: 10 points