MP3: Distributed transactions

Due: 11:59pm, **Wednesday Apr 26**

Notes

- You are welcome to switch groups. If you do change groups, please update us so we can update the cluster assignments.
- You are allowed to use any programming language; however, the TAs will only help with the four supported languages: C/C++, Go, Java, and Python.
- Your implementation will be tested on the CS425 VMs. It is your responsibility to make sure that the code runs on these VMs.

Overview

In this MP you will be implementing a distributed transaction system. Your 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, Servers, 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. *For each transaction*, a client randomly picks one of the five servers to communicate with. The chosen server acts as the coordinator for that transaction, and communicates with the client and all other necessary servers. The coordinator can therefore be different for different transactions. More details on transactions and their properties have been provided later in this specification.

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. You need to handle multiple clients, where each client issues a single transaction and exits once the transaction completes. However, 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 A 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 sp23-cs425-0101.cs.illinois.edu 1234
B sp23-cs425-0102.cs.illinois.edu 1234
C sp23-cs425-0103.cs.illinois.edu 1234
D sp23-cs425-0104.cs.illinois.edu 1234
E sp23-cs425-0105.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 abcd config.txt`

Client Interface and Transactions

At start up, the client starts accepting commands from `stdin` (typed in by the user). The user will execute the following commands:

- `BEGIN`: Open a new transaction. The client must connect to a randomly selected server which will coordinate the transaction, and reply "OK" to the user.

```
BEGIN
OK
```

- `DEPOSIT server.account amount`: Deposit some amount into an account. The amount will be a positive integer. (You can assume that the value of any account will never exceed 100,000,000.) The account balance should increase by the given amount. If the account does not exist (in other words, if it is the first time a deposit operation on the account has been issued), the account 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 server must abort the transaction and inform the client. The client should then print `NOT FOUND, ABORTED`.

```
BEGIN
OK
BALANCE C.baz 5
NOT FOUND, ABORTED
```

- `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 server must abort the transaction and inform the client. The client should then print `NOT FOUND, ABORTED`.

```
BEGIN
OK
WITHDRAW B.bar 5
OK
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.

The client will forward each command in a transaction to the coordinating server for that transaction (that is randomly selected, as mentioned earlier). The coordinating server communicates with the appropriate server (corresponding to the branch in which the account is maintained) by sending it the command and receiving the outcome, which it then forwards to the client. The coordinating server handles ABORT and COMMIT commands by appropriately communicating with all other servers. Each server must process each command it receives in a way that satisfies the atomicity, consistency, and isolation properties discussed later in this spec.

Each client tackles a single transaction. Once the transaction ends (i.e. it has either been committed or aborted and the client has accordingly replied back with `COMMIT OK` or `ABORTED`), the client should exit.

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 a 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 account balance should be negative. If any balances are negative when a user specifies `COMMIT`, the transaction should be aborted.

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

Isolation

Your system should be able to support concurrent transactions from *multiple clients* simultaneously (we will run tests with up to 5 simultaneous clients). Note that each individual client handles sequential commands, only one transaction at a time. Your system 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.) A natural choice is to use two-phase locking or timestamped concurrency to achieve this (though these are not strict requirements).

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.

Deadlock Resolution

Your system must implement a deadlock resolution strategy. One option is deadlock detection, where the system detects a deadlock and aborts one of the transactions. You can also use concurrency control strategies that avoid deadlocks altogether (e.g. via timestamped concurrency control).

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. Moreover, your deadlock resolution strategy cannot assume that the objects requiring a lock are known apriori (when the transaction begins), as the user will input commands interactively.

The server system may choose to spontaneously abort a transaction (e.g. to break a deadlock, or while implementing timestamped concurrency control). When this happens, it must inform the client, and the client must display `ABORTED` to the user to indicate that the transaction has been 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 call `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 on the isolation properties and serial equivalence order (as discussed later).
- 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 above.
- 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`.
- As discussed earlier, a server may need to spontaneously abort a transaction if required by the concurrency control mechanism. In such cases, the system must inform the client, and the client should display `ABORTED` to inform the user. In order to continue issuing commands, the user will need to open a new transaction with a new client using `BEGIN`. E.g.:

```
BEGIN
OK
DEPOSIT A.foo 10
OK
DEPOSIT B.bar 30
ABORTED
….
```

- Part of the learning experience for this MP is for you to thoroughly test that your system satisfies the atomicity, consistency, and isolation properties by designing your own test scenarios. You also need to think of ways to create deadlocks (if you are using lock-based concurrency control) to test your deadlock resolution strategy.

Submission Instructions

Your code needs to be in a git repository hosted by the Engineering GitLab. After creating your repository, please add the following NetIDs as members with *Reporter* access:

- gmk6
- jw22
- eashang2
- sm106
- radhikam
- ayxu2
- hanbog2
- srm14

You will also need to submit a report through Gradescope. In your report, you must include the URL of your repository and the full revision number (git commit id) that you want us to grade, as shown by `git rev-parse HEAD`. Make sure that revision is `push`ed to GitLab before you submit. Only one submission per group – it is set up as a "group submission" on Gradescope, so please make sure you add your partner to your Gradescope submission.

In addition to this, your report should include:

- The names and NetIDs of the group members
- The cluster number you are working on
- Clear instructions for building and running your code. Please include a `Makefile` if you're using a compiled language! If there are any libraries or packages that need to be installed, please list those, too. Like the other MPs, we will run your code ourselves on our VMs and check that it works as expected, so it is important that your instructions are sufficiently clear.
- Design document described below.

Design document

1. A 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 (e.g. timestamped ordering), explain the other data structures 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.

2. A description of how transactions are aborted and their actions are rolled back. Be sure to mention how your strategy ensures that other transactions do not use partial results from aborted transactions.

3. Describe how you detect or prevent deadlocks.

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
DEPOSIT A.foo 30
WITHDRAW A.foo 10
DEPOSIT C.zee 10
BALANCE A.foo
COMMIT
```

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

```
OK
OK
OK
OK
OK
A.foo = 40
COMMIT OK
```

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.

Sample Test

We have provided a sample script here that runs locally to test whether your code runs with the basic functionality. Place your folder with the code and the compiled executables inside the testing directory and rename it to `src`. You can run the test by executing `./run.sh`. You should see no output for the diff command (after printing `Difference between your output and expected output:`) if your code passes the test. Otherwise, double check the difference between your client's output and the expected output. The output of your clients will be saved as `output1/2.log` and the output of your servers will be saved as `server_A/B/C/D/E.log`. The testing script sets a timeout of 5s for each client (i.e. a client has up to 5s to finish executing its transaction and exit; it will be automatically killed after 5s) – ideally, your code should run much faster than that. The testing script used for grading will be similar to this sample test except that we will be running it in a distributed manner on the VMs and will be testing more complicated aspects of your code (e.g. how well you handle atomicity, consistency, isolation, and deadlocks). Passing this sample test does not automatically imply that you will pass all our test-cases. The sample test is provided only for you to sanity check whether the input/output format and the setup for your nodes complies with the specifications.

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 (5 points)
- Design Document (15 points)
- Functionality testing (60 points)
  - Atomicity (15 points)
  - Consistency (15 points)
  - Isolation (25 points)
  - Deadlock avoidance (5 points)

Acknowledgement

We would like to acknowledge Prof. Nikita Borisov for the initial design of this MP.
The source code of the website has been borrowed from Nikita Borisov