MP3: Distributed transactions

Group member: Xu Xiong(xuxiong2), Hanqi Wang(hanqiw3)

Cluster name: g05

Github url: https://github.com/xiongxug/ece428_mp3

Instructions for running

Dependency

We use Python lib **Pyro4** for RPC, please install it via pip3 on each vm.

```
pip3 install Pyro4
```

Config

The config file looks like below:

```
COORDINATOR fa21-cs425-g05-01.cs.illinois.edu 1234
A fa21-cs425-g05-02.cs.illinois.edu 1234
B fa21-cs425-g05-03.cs.illinois.edu 1234
C fa21-cs425-g05-04.cs.illinois.edu 1234
D fa21-cs425-g05-05.cs.illinois.edu 1234
E fa21-cs425-g05-06.cs.illinois.edu 1234
```

Line 1 sets the COORDINATOR, and line 2-6 set the servers. The first column is the `name` and second column is the `hostname`, the third column is the `port`. Note that the config file for server, coordinator and the client are the same.

Usage

1. Run `coordinator.py` on vm1

   ```
   python3 coordinator.py <config_file_name>
   ```

For this mp the `config_file_name` should be `config.txt`

2. Run `server.py` on vm2-vm6

   ```
   python3 server.py <server_name> <config_file_name>
   ```

   We assume that the number of servers is 5, then you should run following commands on vm2-vm6 respectively.

```
python3 server.py A config.txt
python3 server.py B config.txt
python3 server.py C config.txt
python3 server.py D config.txt
python3 server.py E config.txt
```

3. Run `client.py` on vm7

   ```
   python3 client.py <instance_id> <config_file_name>
   ```

   For this mp we can run as follows:

   ```
   python3 client.py 0 config.txt
   ```

Code Design

Overview

We use Pyro lib to allow clients to call methods on a remote node. Pyro daemon will be running on all of the clients, servers and coordinator to route requests. We use timestamped concurrency to guarantee the serializability of the executed transactions and avoid deadlock.

Client

Each client has its own **instance_id**, together with vm's ip, which allows us to run more than one instance on a single vm. A inTransaction flag will marking whether it is in a transaction. Also we keep track the local transaction_id to create a unique identifier for the servers and coordinator.

The transaction will be initiated by a `BEGIN` command from stdin. Then the client will call the server.Begin(ip, instance, transaction_id, uri) function, which will notify the coordinator of a new transaction.

When enter **DEPOSIT A.foo** 10 into the terminal of client. This will then once again be sent to the server. This will return either an **OK** or an **ABORTED** depending on conditions set by our concurrency control. If a transaction has been aborted in the deposit stage we do not need to commit as the client side transaction has been terminated by the server.

The server is able to abort a transaction on the client side because we exposed a function called **client.Abort()**.

When enter **COMMIT**, if the server is able to, then the client will receive a **COMMIT OK**, otherwise it will receive a **COMMIT ABORT**.

SERVER

When the client calls `server.Begin()`, the server will initialize a new transaction by inserting this transaction into tentativeDicts, and add the client uri into clients list. Then call coordinator.Begin() to notify the coordinator.

When the client executes a DEPOSIT. We first assert that the transaction exists in tentativeDicts, then we notify the coordinator we need to read from account. If the account doesn't exist, we assign the value to 0. After this we notify the coordinator that we need to writing to this account on the server specified.

When the client executes a COMMIT, the server will check if the current transaction being aborted. If not, then we check if the account balances are negative. If they are, then we abort the transaction. If not, we call coordinator.Commit(server_name, ip, instance_id, localTransactionID). If the coordinator approve, we return **COMMIT OK** to the client.

When the client executes a BALANCE, we call coordinator.Read.

Coordinator

We use coordinator to determine if transactions are allowed to proceed or to abort. And we use timestamp concurrency to avoid the lock and deadlocks.

When server call coordinator.Begin(ip, instance_id, transaction_id), we append the transaction to TranIdList and append the transaction accounts to TranIdAccList.

When server executes a DEPOSIT, the server will first call coordinator.Read(), on the coordinator side, we first check the transaction and account exist, if not, we append it. Then we obtain a timestamp of the transaction, we check if the write timestamp is greater than the timestamp of the transaction. If it is we are working on an older transaction and need to abort any newer transactions also operating on the same account. Otherwise we update our read time stamp for the account. Secondly, the server will call coordinator.Write(), and it's similar to the read. Finally, if the server calls coordinator.Commit(), on the coordinator side, we check the transaction is active, and if there are any other older uncommitted transactions using the same accounts. If there are we abort. Else there are no conflicts and we clear the the current transaction.

Timestamp Concurrency

The rules are as follows:

- If a transaction T performs a write on X:
  - If $TS_{read}(X) > TS(T)$, $TS_{write}(X) > TS(T)$, we abort T
  - Else we execute the write and update timestamp
- If a transaction T performs a read on X:
  - If $TS_{write}(X) > TS(T)$ we abort T
  - Else we execute the read and update the timestamp

Extra on Deadlock

We use timestamp concurrency, this strategy would not lead to any deadlocks. Thus the system wouldn't have any deadlock issues.