MP3 - Distributed Transactions

Names:

Ali Husain (alijh2), Satej Sukthankar (satejrs2)

Cluster Numbers:

4201 - 4210

Git URL:

https://gitlab.engr.illinois.edu/cs4251732105/mp3-distributed-transactions

Git Revision Number:

388a0df0bc89b431dc7d4402566a1798fb294e9a

Getting started

1. `cd mp3`

Starting servers

1. `./server {server name} config.txt`

Starting client

1. `./client {client ID} config.txt`

Concurrency Approach

We followed the timestamp concurrency control strategy described in the lecture. We used timestamped ordering in order to implement our concurrency control. We maintained a map of the read timestamps with a key of the account name and timestamp of when the transaction was generated, designated as the transaction ID. We also used a map of the latest committed write timestamp with a key of the account name and the value of transaction ID. For tentative writes, we utilized a map with key account name to the list of tentative writes that we would update for that account. The

tentative write data structure was maintained with the transaction ID, the balance of the account at that timestamp as an int, whether or not the tentative write has been committed through a boolean, and whether or not the tentative write has been aborted also through a boolean. Timestamped ordering ensures concurrent control by applying the read and write rules described in the lecture to the execution of our program. Upon committing or aborting, we remove the tentative write stamps from the structure. By using timestamps in order to maintain concurrency control, we can ensure that commands from concurrent transactions run serially and only after other commands of concurrent transactions affecting the same accounts are either aborted or committed.

Rolling Back Transactions

Before we decide to commit a transaction, we check whether any of the balances from the current transaction is negative by iterating through the tentativeWrite structure. Upon finding one tentativeWrite that would result in a negative balance, we initiate an abort. In order to roll back aborted transactions, we iterate through our list of tentative writes for each account and check if the current transaction ID matches that of the account in the tentative write timestamps list for that account. If it does, we perform the opposite action of that command (i.e. if the transaction's original message type was WITHDRAW, we would add to the balance of the account when the tentative write timestamp ID matches that of the current transaction ID). We also set the state of IsAborted of the tentative write of that account when the current transaction ID matches to true then remove any tentative write timestamps for that account where IsAborted is true. Finally, we delete any accounts where the latest committed write timestamp is equal to 0. This gets rid of any accounts that haven't already been committed. We divided DEPOSITS and WITHDRAW into read/write sections, and BALANCE into just a read section. We encapsulated our write section within that of the read section to ensure that the read rule was being reapplied and to ensure that writing to a balance occurs following a read. This prevents the usage of partial results from aborted transactions since any writes to an account are constantly being looked at by the read rule.

Deadlock Prevention

In order to prevent deadlocks, we implemented timestamped ordering; however, we still could not find a way to prevent deadlocks. We utilized the latest committed values of each account, their read timestamps, a tentative write list, and timestamped transaction IDs, but we still experienced some issues when performing actions on accounts that were being affected by concurrent transactions.