Pranav Gaka - pgaka2
Rohan Sreenivasan - rohanjs3
MP3 Report

List of our VMs / Cluster Numbers:
sp24-cs425-1401.cs.illinois.edu
sp24-cs425-1402.cs.illinois.edu
sp24-cs425-1403.cs.illinois.edu
sp24-cs425-1404.cs.illinois.edu
sp24-cs425-1405.cs.illinois.edu
sp24-cs425-1406.cs.illinois.edu
sp24-cs425-1407.cs.illinois.edu
sp24-cs425-1408.cs.illinois.edu
sp24-cs425-1409.cs.illinois.edu
sp24-cs425-1410.cs.illinois.edu

GIT COMMIT ID: 47ecf96ed15de9bd2ed3934480a7f0e1c2008635

Instructions to run:

1. Install go on all VMs if not installed
2. Clone the repo:
3. cd to ~/CS425/MP3
4. Run these commands on all nodes:

go build server.go
go build client.go

Run ./server and ./client as needed

Design Document

We used timestamp ordering to ensure serial transactions in our code. Our client randomly chooses a coordinator to send a command to that is timestamped (of the BEGIN command). Subsequent commands of this transaction all have the same timestamp. The coordinator stores the transaction with the correct timestamp id and commands involved and passes the command to the proper branch. The branch then performs the necessary action. Writes are stored in the Tentative writes list and reads are stored in the reads list. Reads are also stored under the appropriate transaction if they need it to be committed first. When transactions are committed we use a two phase commit approach that sends a prepare, commit and have committed messages to ensure that all affected branches have applied the commit. In the event that a transaction is aborted we propagate the abort to all nodes and remove the transaction from the tentative write list. We also add the transaction from an aborted transaction list. We compute balances by going through the tentative write list every time so getting rid of

anything from the aborted transaction from the tentative write list makes sure no partial results from the transaction are used in future computations. The timestamp ordering method we used does not deadlock at all so we did not need to deal with this. We have mutex locks per account for modifications to the read and write lists.

Please see our whiteboard implementation for any other details on structs, data structures or logic:

Client:

- Connect to all servers from config.txt
- Take input from stdin
- Random choose a coordinator
- take input, and send txns to coordinator using command struct, new coordinator for each txn.
- for each transaction create a new transaction struct and add to map
- Listen for server responses and update transactions appropriately

Recieve messages:

- OK, from begin command, dep and withdraw
- Aborted from aborted commands
- Commit OK from committed commands
- Balance: BALANCE A.foo
  A.foo = 10
- NOT FOUND ABORTED

Client Struct:

Txn map: dict mapping txn id to txn struct

Command Struct:

TXN id: timestamp of the transaction
Sender: client id of origin
Coordinator: coordinator ID (A,B,C,D,E)
Transaction: String of the actual sent transaction (EX: deposit A.foo 3)

Transaction Struct:

TXN id: timestamp (probably)
Sender: client id of origin
Coordinator: coordinator ID (A,B,C,D,E)
Commands: list of commands for a given transaction (EX: deposit 3, withdraw 2….)

Committed Flag: bool for has committed

Readers List: list of TXNs that want to read this value, we will want to send their coordinator a message with the value that can be read.

Server:

- Connect to all servers from config.txt and listen for incoming requests from any clients
- Take input from stdin
- Server initialized

General Server Logic (assume message received has isClient: False and the transaction is intended for the correct branch):

- Servers should not receive a BEGIN, unless its a coordinator
- if not begin or commit,
    - Add to transaction MAP. Run appropriate read/write logic as shown below
    - If we have deposit, initialize account add to accounts list, update transactions map map and entry in TW list with command in affected accounts. Send OK
    - If withdraw, if account is made, update transactions map and entry in TW list with command in affected accounts. Send OK. If account not found, indicate this and ROLL BACK TRANSACTION sent to coordinator. Send NOT FOUND, ABORTED
    - If Balance, run read logic and add to read list. If account does not exit (balance = -1), ROLL BACK. Else: Send Balance

If commit:

iterate TW list, for this txn id, Compile net balance from commands list, if the net balance being applied is valid for all the accounts involved in this transaction then send, COMMIT OK to coordinator. ELSE if any account will be invalid, ROLL BACK TRANSACTION. Remove from affected TW lists regardless of reply. Update transactions Map. If this transaction created an account for the first time, then set balance to -1

Coordinator Logic (incoming message has isClient: True):

-If BEGIN received from client, make transaction struct from Command struct in message, add to coordinator map. Store that this server is the coordinator for this TXN id.

-Else if not begin or commit:

Get this txn from the coordinator map, add this incoming command to the commands list. Send the incoming command to the intended branch sever. This will be specified in the command. If the coordinator IS the branch, run appropriate write/read logic for the account. When the server sends a response, it must read as OK. Else Abort the transaction, notify the client and DO NOT ACCEPT any more of this transactionID, We may need to store aborted txn ids in another map. If a txn is notified as aborted on ANY server, we need to notify all other branches that have been interacted with from this transaction. Send to server with isClient False

-If we see a commit:

Get this txn from the coordinator map, add this incoming command to the commands list. Send the incoming command to the intended branch severs (looked by txnid), there can be multiple, we will need to parse the commands list to know which servers to send a commit to. When the server sends a response, it must read as OK. Else Abort the transaction, notify the client and DO NOT ACCEPT any more of this transactionID, We may need to store aborted txn ids in another map. All commits must pass. If a txn is notified as aborted on ANY server, we need to notify all other branches that have been interacted with from this transaction. IF and only if all commits read as OK, then we will can send COMMIT OK to client. If the coordinator IS the branch, run appropriate commit logic mentioned in server logic. Send to server with isClient False

Update the coordinator map and handle responses from servers by sending to the appropriate clients.

Take messages from clients and send them to appropriate servers

The coordinator may also be the branch that has the account, so in this case run normal server logic

Structs:

Server Struct:

Coordinator map : dict mapping txn id to (client id), used to know which txns this server is the coordinator for

Accounts: array for all the accounts, array of account structs

Transactions map: dict from txn id to transaction struct

Account Struct:

Mutex

Balance: int, current balance, balance is -1 if the account creation was rolled back

Creator: the transaction that created this account, we may need to set roll back its creation if the transaction gets rolled back

TXN id: timestamp of the last committed transaction

Read Timestamps: List of transaction ids (timestamps) that have read the committed value.

Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps). This will be a sorted list of transaction structs, the list will be sorted by timestamp ascending order.

Transaction Struct:

TXN id: timestamp (probably)

Sender: client id of origin
Coordinator: coordinator ID (A,B,C,D,E)
Commands: list of commands for a given transaction (EX: deposit 3, withdraw 2….)
Committed Flag: bool for has committed
Readers List: list of TXNs that want to read this value, we will want to send their coordinator a message with the value that can be read.
AffectedBranch: list of servers that have been interacted with this transaction. If any fail then we must abort the TXN
AbortedFlag: if this has been aborted mark true

Server Logic:

WRITE LOGIC:

Transaction Tc requests a deposit/withdraw sequence operation on Account D

if (Tc ≥ max. read timestamp from read list on D && Tc > write timestamp on committed version of D)

Perform a tentative write on D:

If Tc already has an entry in the TW list for D, update it.

Else, add Tc and its write value to the TW list.

else abort transaction Tc

//too late; a transaction with later timestamp has already read or written the object.

READ LOGIC:

Transaction Tc requests a BALANCE operation on Account D

if (Tc > write timestamp on committed version of D) {

Ds = version of D with the maximum write timestamp that is ≤ Tc

//search across the committed timestamp and the TW list for object D.

if (Ds is committed):

read Ds and add Tc to RTS list (if not already added)

Else

if Ds was written by Tc: simply read Ds

else wait until the transaction that wrote Ds is committed or aborted, and reapply the read rule. // if the transaction is committed, Tc will read its value after the wait. // if the transaction is aborted, Tc will read the value from an older transaction.

} else abort transaction Tc //too late; a transaction with later timestamp has already written the object.