CS425 MP3

Lishen He
Chris Coronado

1. Introduction

   Git repo: git@gitlab.engr.illinois.edu:lhe10/mp3.git
   Commit Hash: c1e60be4d0531267db3e5d9b9d90229d7f686f1d
   Compile instruction: only Python3 with essential packages are needed.
   Running instruction:
   - python3 client.py[This should run on any node other than 2 to 7]
   - python3 coord.py[This should run on node 2]
   - python3 brach.py[This should run on node 3 to 7]

   Cluster number: G51
2. ACI(D), Concurrency and Deadlock Prevention Design from a High-Level Point of View

   2.1. Atomicity

   Atomicity requires that all steps or no steps in a transaction should be carried out. No partial transaction is allowed. We guarantee this using **2 Phase Commit(2PC).** With 2PC, each transaction is atomic. It either proceeds or gets rejected at its entirety.

   2.2. Consistency

   Consistency requires that no account balance is negative due to a transaction. This is checked by COMMIT command at the end of the transaction, in phase 1 of 2PC. An additional run-time consistency check is that a “NOT FOUND” message for a non-existing account regarding to “BALANCE” and “WITHDRAW” command should return to the client.

   2.3. Isolation

   Isolation requires serialization of transactions from different clients. In theory, our design allows that unlimited number of clients can use the system at the same time. To prevent deadlock, we do not use two phase locking and instead, a **Timestamp Ordering** with **Dependency Tracking** is implemented. Our design ensures that: if different clients access different objects on the same server, no delay occurs; if different clients access the same object on the same server, then the latter may have to wait until the former transaction commits or aborts.

   2.4. 2 Phase Commit (2PC)

   On the one hand, 2PC ensures atomicity of transaction. On the other hand, 2PC ensures consistency by checking against negative account balance in its second phase. In the phase 1, or preparation phase, 2PC asks for votes for DoCOMMIT. If all vote yes, then the second phase, or commit phase, can be executed. Otherwise, the whole transaction is aborted.

   The following snippet is the implementation for 2PC. In phase 1, the transaction asks each server if they have inconsistent account balance. If yes, then transaction is aborted. If no, it goes to phase 2 and make changes to the database by sending DoCOMMIT to each database and invoke their RPCs.

```
86          if (action == "COMMIT"):
87              # phase 1: info gathering
88              for b , s in BRANCHES.items():
89                  s.send( ("COMMIT" + " " + tx).encode() )
90                  rsp = s.recv(1024)
91
92                  # msg:
93                      # OK tx
94  Phase 1         # ABORT tx
95                  rsp = rsp.decode()
96                  print("COMMIT: receive from branch:" + rsp)
97                  rsp = rsp.split()
98                  if (rsp[0] == "ABORT"):
99                      abort(socket, tx)
100                     return
101                 else:
102                     print("on_new_client: COMMIT: this branch reports consistent transactions.")
103                     #continue
104             print("on_new_client: COMMIT: All branch's tx are consistent. Ready to make changes to database.")
105             # reaching here means no abortion happend
106
107             # phase 2: making changes
108             for b, s in BRANCHES.items():
109                 s.send( ("DoCOMMIT" + " " + tx).encode() )
110                 rsp = s.recv(1024)
111 Phase 2         rsp = rsp.decode()
112                 if ( rsp.split()[0] != "OK" ):
113                     print("on_new_client: DoCOMMIT: something is wrong")
114
115             print("on_new_client: COMMIT: Finished making changes to database.")
116
117             needBegin = True
118             dp = list()
119             socket.send( "COMMIT OK".encode() )
```

2.5. Timestamp Ordering(TO)

To avoid using 2 phase locking that leads to deadlocks, we utilize Timestamp Ordering to prevent deadlocks. TO requires that each data maintains a read and write invariant. This is updated when a committed transaction (each transaction's ID is its own timestamp) has accessed that data. BALANCE is a read operation; DEPOSIT is both read and write; WITHDRAW is both. COMMIT or ABORT do not access data. The difficulty of maintaining TO is that an uncommitted transaction may have updated information. Thus, newer transaction should not only look at the database but also the transaction on the fly. A transaction is immediately rejected if it violates TO's requirement.

An example is as follows. We have two clients modifying the same object. The first client has aborted the transaction. As a result, the second transaction is aborted at commit time. The order of transaction items is marked.

```
BEGIN
OK
DEPOSIT A.x 5     1
OK
ABORT             3
ABORTED
```

Client 1

```
BEGIN
OK
DEPOSIT A.x 5     2
OK
COMMIT            4
ABORTED
```

Client 2

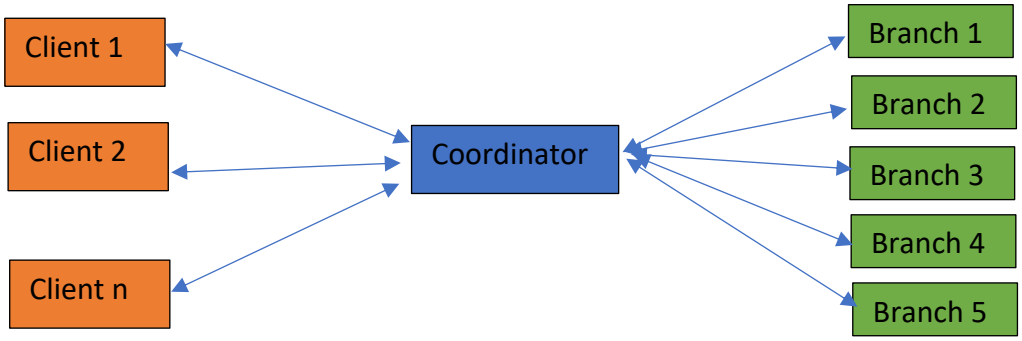
If a transaction only read database and ignores uncommitted data, isolation property is violated. More details about implementation will be discussed later.

2.6. Dependency Tracking

Timestamp Ordering brings a new problem not existing in lock-based approach. The problem is that when a transaction is aborted in the commit stage, other transactions that depend on this one should also get aborted. In this case, each transaction should have its own dependency list. Two global variables should be maintained: successful transactions and aborted transactions. A transaction item brings one dependency if has only read operation and two dependencies if it's both read and write.

3. System and Message Design

3.1. System Scheme Illustration



We utilize a coordinator to interact between client interface and database server. The main role for client is simply taking input from client and displaying return message from coordinator. The coordinator's role is the most complicated one. It generates a unique channel with a different client through *on_new_client*. In this function, the lifecycle of transactions is maintained. Also in this function, dependency list is maintained. The dependency list is enriched during RPCs from branches. The list is cleared in ABORT or COMMIT which leads to a new transaction.

3.2. From client to coordinator

This is the same as user's input, namely: BEGIN; DEPOSIT A.xyz 5; BALANCE A.xyz; WITHDRAW A.xyz 5; COMMIT; ABORT.

3.3. From coordinator to client

OK; ABORTED; NOT FOUND.

3.4. From coordinator to branch

The coordinator appends transaction ID to each message. Also, it needs a "DoCOMMIT" commit as the phase 2 in 2PC protocol.

```
# The information coming into branch.py is
# Pattern: Command + Account + transaction timestamp
# e.g. DEPOSIT xyz 5 2777.64
#       BALANCE xyz 2777.64
#       WITHDRAW xyz 5 2777.64
#       COMMIT 2777.64
#       DoCOMMIT 2777.64
#       ABORT 2777.64
```

3.5. From branch to coordinator

These messages should start with the result of operation, either “OK” or “ABORT”. Then, each operation need to optionally append result and dependency list.

```
# Message going back to coordinator
# Pattern: status +[value] + [dependency]; dependency = None or 2777.64 2778.43
# e.g. 1. ABORT TX
# e.g. 2. OK TX val
# e.g. 3. OK TX val tx_1 tx_2 <- dependency(ies)
```

4. Coding Details and Data Structures

4.1. Client Interface - client.py

Very simple. The goal is to build a socket to connect coordinator, take input and display returns.

4.2. Coordinator - coord.py

4.2.1. Data structures

- CLIENT
  A dictionary to find client socket. Key equals client’s IP; value equals client socket().
- BRANCHES
  A dictionary to find branch’s socket. Key equals branch’s name, “A”, “B”,…”E”. Value equals branch’s socket.
- TX_OK
  A list of IDs of successfully committed transactions.
- TX_ABT
  A list of IDs of aborted transactions.

4.2.2. Main Functions

- on_new_client(socket, addr)
  Each on_new_client is a thread dedicated for one client. The while loop in this function successively maintains lifecycles of transactions. Within the while loop, input from client is parsed and proper messages are sent to branches or back to the client. A dependency list dp() is reset for each new while loop. The list is appended based on the responses from branches.
- Abort(client, tx)
  Abort function send abort messages to branches and client. Also, it puts transaction ID in the TX_ABT list.

4.3. Branch Server - branch.py

4.3.1. Data Structures

- Class account
  Data structure for account. “name” is the account name string. “value” is the account balance. “rts” is the write timestamp. “wts” is the write timestamp.
- ACCOUNTs

The collection of accounts hold by the branch server. It is a dictionary with key equal to account name and value equal to account instance.

- VIEWs

  This is the most distinctive and innovative data structure by our group for this MP. View is a collection of tentative updates. This is dictionary of dictionary. Top level is the "view", or the tentative update, for each transaction, mapped from transaction ID to a dictionary of its view. The bottom level represents the view of a certain transaction. It is a dictionary of account name mapped to account instances.

4.3.2. Main Functions

4.3.2.1. isNewerOrSame(obj1, obj2, flag)

Checks if obj1 is newer than obj2 in terms of their "wts" and "rts". The flag can be either "read" or "write" and leads to different criteria. This is a utility function for getObj for the search.

4.3.2.2. getObj

The goal of getObj is to find the object that current transaction item depends on. This item can be in the ACCOUNTs or in the VIEWs. If it's the former, there is no dependency. If it's the latter, the must be dependency. This function search in the order of most recent to oldest transaction in the VIEWs; if not found, it searches the ACCOUNTs. This guarantees the account balanced used by the current account is the most recently updated one. All dependencies are included because if Tn depends on Tn-1, then it also depends on Tn-1's dependencies.

4.3.2.3. deposit, balance, withdraw, commit, doCommit, abort

They implement operations by adding dependencies, modifying VIEWs and ACCOUNTs, performing proper operations and sending messages back to coordinator.

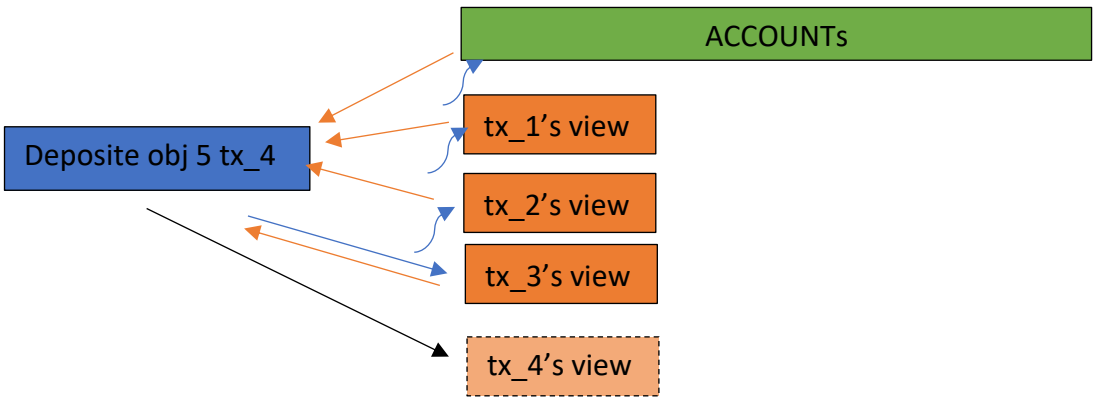
4.3.2.4. msg_handler

Based on incoming messages from coordinator, dispatch proper function above mentioned to handle. Slightly process these incoming messages.

5. Other Considerations

5.1. Tentative Updates Implementation

As above mentioned, tentative update is saved into VIEWs[tx][name]. Once tentative change is committed, we simply loops over VIEWs[tx] and update ACCOUNTs. In this way, we ensure that no transaction can change database before committed.



5.2. Concurrency Control

Transactions can be run in parallel because: 1) clients are handled in dedicated threads; 2) transaction work with their VIEWs and do not interfere with each other unless timestamp violation happens; 3) threads are spawn for each transaction item in branch.py.

5.3. Transaction Abortion, Rolling Back and Avoidance from Using Partial Results

If a transaction is aborted, either by user or timestamp violation, it rolls back by not updating the database and put its ID into TX_ABT. Further transactions which depend on this will also get aborted because they check TX_ABT during commit stage. Partial results will not be used because 2PC guarantees atomicity.

5.4. Multiple Client Connection

Our design ensures unlimited number of clients to connect to coordinator as long as they use different IP. Each client is using a dedicated thread so they can run concurrently. Current socket design between client and branch assume that the frequency is low so they share the same channel. If the frequency is high, clients may interfere with each other. To remedy this is simple: we need to create a global variable that buffers and differentiates incoming data for each client. As scaling to extremely large number of clients is not required in this MP, we simple mention this without implementaion.

6. A Walk Through of using a Simple Example

We deposit A.x with 5 dollars. The getObj will not find anything in ACCOUNTs or VIEWs. Thus, it returns an object with empty “wts” and “rts”. This will force deposit to create a new view for transaction whose ID is 1588757475.3679802, VIEWs[1588757475.3679802] = dict() and put tentative update in VIEWs[1588757475.3679802]. Upon committing, since there is no dependency, the item in VIEWs[1588757475.3679802] is used to update ACCOUNTs[“x”]. The following screenshots are from client.py, coordinator.py and branch.py’s debugging messages.

Client:

Coordinator:

```
on_new_client: ready to take a new tx item.
im here
on_new_client: received: BEGIN
on_new_client: ready to take a new tx item.
im here
on_new_client: received: DEPOSIT A.x 5
on_new_client: receiving msg: OK 1588757475.3679802 0
on_new_client: finished tx item:DEPOSIT A.x 5
on_new_client: ready to take a new tx item.
im here
on_new_client: received: COMMIT
COMMIT: checking dependencies.
COMMIT: receive from branch:OK 1588757475.3679802
on_new_client: COMMIT: this branch reports consistent transactions.
on_new_client: COMMIT: All branch's tx are consistent. Ready to make changes to
database.
on_new_client: COMMIT: Finished making changes to database.
on_new_client: finished tx item:COMMIT
on_new_client: ready to take a new tx item.
im here
on_new_client: received: BEGIN
on_new_client: ready to take a new tx item.
im here
on_new_client: received: BALANCE A.x
on_new_client: receiving msg: OK 1588757483.8549433 5 1588757475.3679802
on_new_client: adding depency: 1588757475.3679802
on_new_client: finished tx item:BALANCE A.x
on_new_client: ready to take a new tx item.
```

Branch:

```
connected to coordinator
msg_handler: msg to process is:DEPOSIT A.x 5 1588757475.3679802
Deposite x of 5 by 1588757475.3679802
getObj: account not found in ACCOUNTs or VIEWs.
Deposit: obj val = 0
Deposit: Timestemp Ordering fine
Deposit: Val =   0
Deposit: Adding tx's view.
Deposit: put obj in View: (tx,name,val)=1588757475.3679802 x 5
Finished handeling transaction: 1588757475.3679802
msg_handler: msg to process is:COMMIT 1588757475.3679802
Finished handeling transaction: 1588757475.3679802
msg_handler: msg to process is:DoCOMMIT 1588757475.3679802
DoCOMMIT: have made changes to transaction: 1588757475.3679802
Finished handeling transaction: 1588757475.3679802
msg_handler: msg to process is:BALANCE A.x 1588757483.8549433
Balance of x for 1588757483.8549433
getObj: found account in ACCOUNTs.
Balance: Timestemp Ordering fine
Balanced: finished.
Finished handeling transaction: 1588757483.8549433
```

7. More Tests
   7.1. Basic Functionalities
      7.1.1. DEPOSIT

      We test multiple deposit into account A.x. The first deposit creates an account for A with 5 as initial balance. Successive deposits all succeed.

```
[BEGIN
 OK
[DEPOSIT A.x 5
 OK
[BALANCE A.x
 5
[DEPOSIT A.x 5
 OK
[BALANCE A.x
 10
[DEPOSIT A.x 5
 OK
[BALANCE A.x
 15
```

7.1.2. BALANCE

If an account already exists, the “BALANCE” would work as shown above. If an account does not exist, the following will show that it would work.

```
[BEGIN
 OK
[BALANCE A.x
 NOT FOUND
```

7.1.3. WITHDRAW

WTHDRAW would work for both cases where an account exist or not, as shown below. As required, WITHDRAW will not check invalid of account balance. The check will be performed by COMMIT.

```
[BEGIN
 OK
[WITHDRAW A.x 5
 NOT FOUND
[DEPOSIT A.x 5
 OK
[WITHDRAW A.x 2
 OK
[BALANCE A.x
 3
[WITHDRAW A.x 100
 OK
[BALANCE A.x
 -97
```

7.1.4. COMMIT

Case 1: COMMIT OK

Any valid transaction shall be able to make be committed, illustrated using an example as follows.

```
[BEGIN
 OK
[DEPOSIT A.x 5
 OK
[COMMIT
 COMMIT OK
[BEGIN
 OK
[BALANCE A.x
 5
[COMMIT
 COMMIT OK
```

Case 2: ABORTED

Any invalid balance will be found during phase 1 stage in commit and hence, the transaction will be aborted.

```
BEGIN
OK
DEPOSIT A.x 5
OK
WITHDRAW A.x 10
OK
COMMIT
ABORTED
```

7.1.5. ABORT

Spontaneously abort a transaction and roll back to previous state. The accounts in tentative updates are not created if they have been aborted.

```
BEGIN
OK
DEPOSIT A.x 5
OK
ABORT
ABORTED
BEGIN
OK
BALANCE A.x
NOT FOUND
```

7.1.6. Multiple Clients and Multiple Branches(servers)

Multiple clients will be illustrated in the next section of Deadlock prevention. Here we illustrate case of dual servers.

```
BEGIN
OK
DEPOSIT A.x 5
OK
DEPOSIT B.x 5
OK
COMMIT
COMMIT OK
BEGIN
OK
BALANCE A.x
5
BALANCE B.x
5
COMMIT
COMMIT OK
```

7.2. Advanced Functionalities

7.2.1. Concurrency for Multiple Clients

Multiple clients can executed their transactions without influencing each other if their transactions don't access same objects.

Two clients with interleaved transaction in real time can executed their orders without waiting:

Client 1

```
[BEGIN
 OK
[DEPOSIT A.y 5
 OK
[DEPOSIT A.z 5
 OK
[COMMIT
 COMMIT OK
```

Client 2

```
[BEGIN
 OK
[DEPOSIT A.x 5
 OK
[DEPOSIT A.aaa 5
 OK
[COMMIT
 COMMIT OK
```

7.2.2. Deadlock Prevention

We need to have two clients to illustrate deadlock prevention. Below, we have two client interfaces with interleaved transaction commands. They are numbered in time order.

```
[BEGIN 1
 Ready to send message:BEGIN
 OK
[DEPOSIT A.y 5  4
 Ready to send message:DEPOSIT A.y 5
 OK
[DEPOSIT A.x 5  6
 Ready to send message:DEPOSIT A.x 5
 OK
```

Client 1

```
[BEGIN 2
 Ready to send message:BEGIN
 OK
[DEPOSIT A.x 5 3
 Ready to send message:DEPOSIT A.x 5
 OK
[DEPOSIT A.y 5  5
 Ready to send message:DEPOSIT A.y 5
 ABORTED
```

Client 2

Transaction 5 is aborted because in transaction 4, the data is modified by the other client. After 5 is aborted, 6 is able to proceed as because there is no dependency to client 2's aborted transaction.