MP3: Distributed transactions

Paul Pok Hym Ng (ppng2)
Bruno Seo (sbseo2)

May 6th, 2020

1 Cluster number / Repository

Our cluster number is 15.
Below is the URL of our git repository.
URL: https://gitlab.engr.illinois.edu/ppng2/ece428/tree/master
(Revision number: f1b537f20fda4103da4ac490f279a8ba54000469) (Generated using git rev-parse HEAD)

2 Instructions for running code

Please follow the instructions written as below.

Dependencies

This code uses the Python library Pyro4 please install it via pip

```
python3 -m pip install Pyro4
```

Configuration

1. Please load `config.py` and update the following
2. Please setup the ip address where your coordinator and servers will be running.

```
COORDINATOR_URI = "PYRO:coordinator@" + vm1 + ":8080"
SERV_A_URI = "PYRO:servA@" + vm2 + ":9090"
SERV_B_URI = "PYRO:servB@" + vm3 + ":9091"
SERV_C_URI = "PYRO:servC@" + vm4 + ":9092"
SERV_D_URI = "PYRO:servD@" + vm5 + ":9093"
SERV_E_URI = "PYRO:servE@" + vm6 + ":9094"
```

In the given code above, vm1, vm2, ... , and vm6 is the string(ip) where you should use your ip address(eg: 192.168.1.1). Please also set the global variable `SETTING = 'ONLINE'`.
3. If you are running servers locally, feel free to use 'localhost' (`SETTING = "sbseo2"`) or '127.0.1.1' (`SETTING = "ppng2"`) in place of vmN.

Default Values

By default the coordinator is set to run on `sp20-cs425-g15-01.cs.illinois.edu`. Servers are set to run on

- `sp20-cs425-g15-02.cs.illinois.edu`
- `sp20-cs425-g15-03.cs.illinois.edu`
- `sp20-cs425-g15-04.cs.illinois.edu`
- `sp20-cs425-g15-05.cs.illinois.edu`
- `sp20-cs425-g15-06.cs.illinois.edu`

Clients can be run on any VM however we suggest clients be run on VMs 07-10.

Usage

Below are the parameters required for each file to run the program.

coordinator.py

```
$ python3 coordinator.py <NUM_SERVERS>
```

NUM_SERVERS is a number > 0 and ≤ 5 representing the number of servers running

server.py

```
$ python3 server.py <SERV_NAME>
```

<SERV_NAME> is A, B, C, D, or E. Please run one server per VM.

client.py

```
$python3 client.py <INSTACE_ID> <NUM_SERVERS>
```

We assume that $1 \leq$NUM_SERVERS ≤ 5.

- If you are using one server please spawn server A
- If you are using two servers please spawn server A and B
- If you are using three servers please spawn server A, B, and C
- If you are using four servers please spawn server A, B, C, and D
- If you are using five servers please spawn server A, B, C, D, and E

INSTANCE_ID should be set to 0 unless you are running more than one client on a VM. For example if you are running 3 clients on the same VM please use

- `python3 client.py 0 <NUM_SERVERS>`
- `python3 client.py 1 <NUM_SERVERS>`
- `python3 client.py 2 <NUM_SERVERS>`

3 Lifecycle of a Transaction

To understand how our system works we will now detail the simple transaction below. We use the following example

T0
BEGIN
DEPOSIT A.foo 10
COMMIT

In this example we have one client running transaction 0 ($T0$).

We use RMI (remote method invocation) and the Pyro framework to allow our clients to access methods and object a remote node has. We run a Pyro daemon on all of our clients, servers, and coordinator.

3.1 Client

Each client keeps track of its own IP, instance, transaction ID, URI (unique resource identifier), a `inTransaction` flag marking whether it is in the middle of a transasction, and a transaction ID counter. On the client side, the local transaction ID differs from the actual transaction identifier kept at the coordinator and server. This local transaction ID is only used to create a unique identifier for the server and the coordinator. In addition to this, each `(ip, instance)` pair allows us to keep track of more than one client per VM.

Let us now look at how the example detailed above. The client first initiates the transaction by entering a `BEGIN` command. With this, the client will call the function `server.Begin(ip, instance, localTid, uri)`. This will allow the server to initialize certain data structures and notify the coordinator of a new transaction. This function will return nothing.

Once a transaction has been initialized we are then able to send commands. In the case of $T0$, we will 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

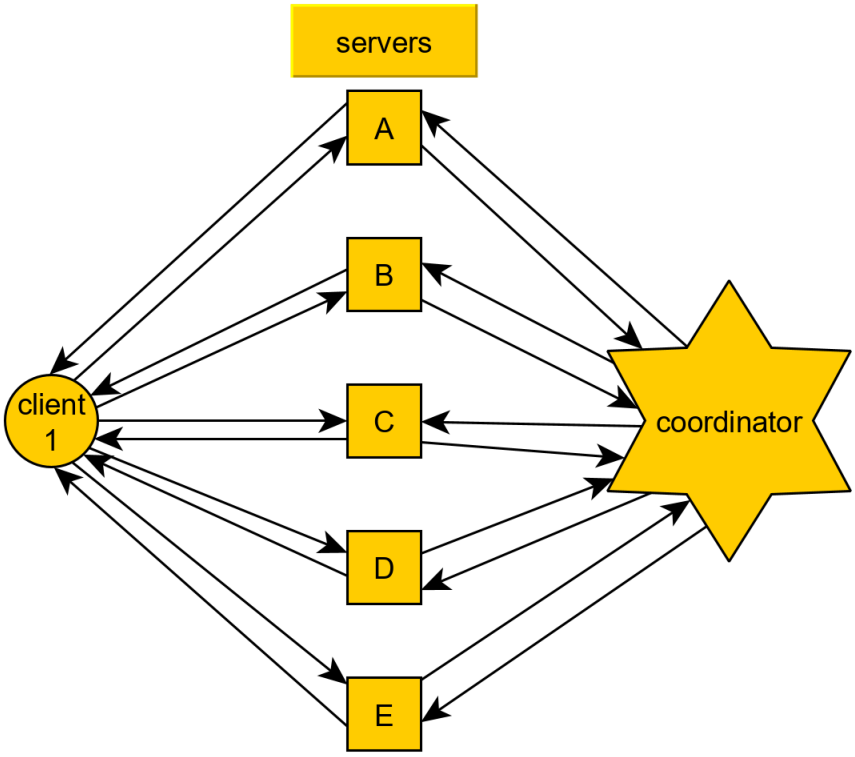

Figure 1: The connections between clients, servers, and the coordinator

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()`. This will set the `inTransaction` flag (on the client) to false, increment the local transaction ID counter and print `ABORTED` for the client. However if we do not receive an abort message we are able to send a commit command by entering `COMMIT` into the client terminal. This invokes a function on the server `server.Commit(ip, inst, localTid)`. The function will then check if we are able to finalize and appy changes to the final balances of accounts. If the server is able to the client will receive a `COMMIT OK` message. Otherwise it will see a `COMMIT ABORTED` if the transaction leads to a negative balance or cannot be committed due to conflicts with other transactions in flight from other clients.

3.2 Server

The server acts as the branch of our banks. In this assignment we have 5 servers named `A, B, C, D, E`.

First let us discuss the data structures used in the server.

1. `finalDict`: This is a dictionary containing the balances of accounts that are the result of committed transactions. The key used is the account name and the value is an amount. For example

   `{'foo':10,'baz':50}`

   contains two accounts foo and baz each with a respective account balance of 10 and 50. Each account has a maximum balance of 1,000,000,000. If the balance is higher than that value post deposit it will be rounded down to that 1,000,000,000. This occurs even if it is in the middle of the transaction. For example `DEPOSIT A.foo 1234567890` will result in the tentative balance of `A.foo` to be `1000000000`.

2. `finalDictLock`: This is a lock used to update our dictionary to ensure that we do not have errors when accessing the final balance dictionary. This gives us an **atomic** update.

3. `tentativeDicts`: This is a dictionary which stores all tentative transactions that are currently in flight. The key used is a `(ip, instance, localTransactionID)` tuple. The value is a dictionary of accounts and balances. For example

   `{('1.1.1.1','0', 0):{'foo':10},('2.2.2.2','3',5):{'baz':50}}`

   contains two in flight transactions at the server. One is from client at IP `1.1.1.1` and instance `0` with tentative account value for foo at 10. Another is from client at IP `2.2.2.2` and instance `3` with tentative account value for baz at 50.

4. `tentativeDictsLock`: This is a lock used for updating our tentative dictionaries. We need this as we may be deleting keys from this dictionary and if we do not lock we may run into errors in Python.

5. `coordinator`: This is the URI used for the coordinator.

6. `clients`: This is a list of URIs for each client which is connected to the server. This allows the server to call the `client.Abort()` function to abort a transaction on the client side.

Now let us discuss the example transaction.

When the client calls and executes `server.Begin(ip, instance, localTid, uri)` several things are done.

1. We initialize our `tentativeDicts` with a new transaction by inserting

   `{(ip, instance, localTransactionID) : {}}`

2. We add the client URI into our `clients` list

3. We call `coordinator.Begin(ip, instance, localTransactionID)` to notify the coordinator that we have began a new transaction.

With this, we are now set up to receive commands from the client.

When a client executes a `DEPOSIT A.foo 10` command the server needs to do the following.

1. First we assert that the transaction exists in our `tentativeDicts`

2. Secondly we need notify the coordinator that we are reading from account 'foo' on server 'A'. Then we attempt to read the balance of the account 'foo' from our `tentativeDicts`. This is done because we need to use the most updated version of the balance for the current transaction. If the client has not modified this account before we will attempt to read an account balance for 'foo' from the `finalDicts` if a value does not exist then we assign our `readValue` to 0.

3. Thirdly, we need to notify the coordinator that we are writing to account 'foo' on server 'A'. As we now know the `readValue` (account balance of 'foo' prior to deposit) we simply update our `tentativeDicts` as follows.

   `tentativeDicts[(ip, instance, localTransactionID)]['foo'] = readValue + 10.`

Finally when the client executes `COMMIT` the server will do the following.

1. The server will check if the current transaction being aborted exists by checking the membership of key `(ip, instance, localTransactionID)` in `tentativeDicts`

2. If the transaction exists we first check if any of the tentative account balances are negative. If they are we abort the transaction and delete the tentative transaciton from `tentativeDicts`.

3. If the transaction exists and no negative balances occur we then are able to commit the transaction. This is done by first checking with the coordinator if we are able to commit by calling `self.coordinator.Commit(SERV_NAME, ip, instance, localTransactionID)`. If the coordinator approves we update `finalDict` with the final balances of affected accounts and delete the transaction from `tentativeDicts`. Subsequently, we return `COMMIT OK` to the client. If the coordinator does not approve the commit we delete the transaction from `tentativeDicts` and return `COMMIT ABORTED` to the client.

Other commands such as `BALANCE` will only need to perform a `coordinator.Read` as it does not need to write any value. `ABORT` from a client will delete the transaction from `tentativeDicts` and notify the coordinator about the aborted transaction.

3.3 Coordinator

The coordinator is used to determine if transactions are allowed to proceed or are required to abort. Let us first discuss the structures used in the coordinator. Instead of locks on accounts, we use timestamp concurrency to remove the usage of locks and deadlocks.

1. `tsDict`: This contains a dictionary of timestamps for each pending transactions per server. The key is the server name (A,B,C,D,E) and the value is another dictionary with key account name, and a timestamp tuple `(readTS, writeTS)`.

2. `tsDictLock`: This is a lock used to update the `tsDict`. This is required as there may be multiple in-flight transactions which want to access the timestamp tuple for an account which want to update these values.

3. `txIdList`: This contains a list of transactions which the coordinator is keeping track off. The values in the list are `(ip, instance, localTid)`. The **timestamp** at which the transaction arrives at the coordinator is the **index** of the tuple in the list.

4. `txIdListLock`: This is a lock to ensure we don't have weird behaviors when it comes to appending the list.

5. `txAccList`: This is the complement to the `txIdList`. This stores the accounts that each transactions accesses. Once again the index into this list represents the transaction's timestamp. Each element is a set with tuples of `(serv, account)`.

6. `txAccListLock`: For the same reasons as `txIdListLock` we must lock this as well.

7. `deadTxId`: This is used to store the transaction identifiers of each commited or aborted transaction. This is a set which contains `(ip, instance, localTransactionID)` tuples.

8. `deadTxIdLock`: As mutliple transactions may attempt to access and modify this set at the same time we must lock this due to the set membership we do.

9. `servA, servB, servC, servD, servE`: These are the URIs of each server. These are used by the coordinator to force an abort on the server.

Now let us discuss the example transaction.

When the server calls `coordinator.Begin(ip, inst, tid)` the following occurs at the coordinator.

1. If the transaction is not present yet in our `txIdList` we append it

2. If the transaction accounts is not yet present yet in our `txIdAccList` we append an empty set.

This has now set up the coordinator to handle a new transaction.

When the server executes a `DEPOSIT` transaction recall that the server checks first whether it can read an account before it can write an account.

Let us first discuss a read. The server calls `coordinator.Read(SERV_NAME, ip, inst, tid, acc)`. This is what occurs on the coordinator end.

1. Check if the transaction exists in our `txIdList` if it does we proceed. If it does not we create entries by appending items to `txIdList` and `txIdAccList`.

2. From here on we obtain the timestamp of the transaction we are working on (index to `txIdList` and the read and write timestamps of the account we are working on by accessing `tsDict`.

3. 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. Finally we return true.

Secondly let us discuss a write. The server calls `coordinator.Write(SERV_NAME, ip, inst, tid, acc)`. This works quite similarly to the read.

1. Check if the transaction exists in our `txIdList` if it does we proceed. If it does not we create entries by appending items to `txIdList` and `txIdAccList`.

2. From here on we obtain the timestamp of the transaction we are working on (index to `txIdList` and the read and write timestamps of the account we are working on by accessing `tsDict`.

3. We check if the write timestamp is greater than the timestamp of the current transaction. or the read time stamp is greater than the timestamp of the current transaction. If it is we are working on an older transaction and need to abort any newer transaction also operating on the same account. Otherwise we update our write time stamp for the account. Finally we return true.

Finally when the server receives a `COMMIT` from the client it will check with the coordinator if it is able to commit the transaction. This is done when the server calls `coordinator.Commit(SERV_NAME, ip, inst, txId)`. This is what occurs on the coordinator end.

1. First we check if the transaction is currently active (in `txIdList`, aborted or committed previously (in `deadTxId`), or nonexistent.

2. If the transaction is currently active, we check if there are any other older uncommitted transactions using the same accounts. If there are we abort. Else if there are any future transactions that are using the same accounts we abort those future transactions by calling `server.coordinatorAbort(ip, instance, localTransactionID)`. Else there are no conflicts and we clear the the current transaction in `txIdAccList[ts]`, `txIdListList[ts]` by assigning `None`.

4 Concurrency Control

Because we use timestamp based concurrency control we do not need to maintain locks for our accounts. As mentioned before we maintain a `tsDict` in our coordinator which stores the read and write timestamps for each account on every server.

Moreover, we also keep track of each transaction's timestamp in `txIdList` and the accounts it accesses in `txAccList`. These two lists are not directly related to the actual computation of whether a transaction needs to be aborted but rather the cleanup of aborted transactions.

The rules that are used to determine if we abort are as follows.

If we are performing a write

1. If a transaction T performs a write on X...
   - If $TS_{read}(X) > TS(T) TS_{write}(X) > TS(T)$ we abort T.
   - Else execute the write and update the write timestamp of X to T's TS.
2. 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 read transaction of X to $max(TS(T), TS_read(X))$

5 Aborting Transactions

Let us first discuss the methods of aborting.

5.1 Client Aborts

The client can issue an abort by issuing an `ABORT` command. This does the following things.

1. This triggers `server.clientAbort(ip, instance, localTransactionID)`.
2. This will delete the transaction from `tentativeDicts` on the server.
3. Moreover the server will notify the coordinator which sets `txIdList, txIdAccList` to `None` for the respective transaction.

The client also has its own abort function exposed to the server via the `client.Abort()` command. This function does the following.

1. This sets the `inTransaction` flag to false
2. This increments the `localTransactionID` by 1.

5.2 Server Aborts

The server has a function `server.coordinatorAbort(ip, instance, loaclTransactionID)`. This is used by the coordinator to abort a transaction in the server if it detects a conflict. Moreover this function calls the client's abort function. This function on the server does the following things.

1. If the transaction is not in the `tentativeDicts` do nothing.
2. If the transaction is in the `tentativeDicts` we delete the transaction from tentativeDicts.
3. Finally we abort the client by calling `client.Abort()`

In addition to this server function we also call several abort functions which reside on the coordinator. These functions will be defined later in the coordinator section. In the `server.Balance(ip, instance, localTransactionID, account)` function we call `coordinator.abortInMiddle(SERV_NAME, ip, instance, localTransactionId)` if the read fails. As the name implies this aborts a transaction in the middle. In `server.Commit(ip, instance, localTransactionID)` we call a similar

5.3 Coordinator Aborts

On the coordinator we have several functions for aborting. Some of these functions are used by the server to notify the coordinator to abort and others are used by the coordinator itself to clear `txIdList, txIdAccList`.

Let us first discuss `abortInMiddle` and `abortCommit`. These two functions essentially work the same way and are called exclusively by the server.

1. Determine if the transaction exists in `txIdList` if it does clear obtain the index of the transaction in `txIdList` and `txIdAccList` and set these values in the list to `None`. Moreover, we add this transaction to `deadTxList`. This marks the transaction as completed.
2. Else if the transaction exists in the `deadTxList` this transaction has already been aborted by another server. This is possible if a client uses `ABORT`. Each server will call the function. Only one of the server can actually cause the coordinator to clear the lists. Afterwards it will be empty.
3. Else the transaction does not exist in the coordinator and we do nothing.

In coordinator's write and read functions it is possible for us to need to abort newer transactions. Therefore we define a function called `abortNewerTx(serv, ip, inst, tid, acc, rw)`. The function performs the following steps.

1. Given an older transaction we note down any newer transactions (larger index in `txIdList`) that share the same input account.

2. We abort all noted down transactions by calling `server.coordinatorAbort((ipToAbort, instanceToAbort, localTransactionIDToAbort)` for each server.

3. Afterwards we clear `txIdList` and `txIdAccList` for the aborted transactions.

4. Finally we need to update the read and write timestamps to the now newest (input) transaction's.

6 Test Cases

We investigated 10 combinations of the following commands in order to avoid the conflicts. They are deposit, balance, withdraw, and commit.

The 10 combinations are

1. (deposit, deposit)
2. (deposit, balance)
3. (deposit, withdraw)
4. (deposit, commit)
5. (balance, balance)
6. (balance, withdraw)
7. (balance, commit)
8. (withdraw, withdraw)
9. (withdraw, commit)
10. (commit, commit)

When these pairs are used within the same transaction, there is no conflict issue. However, when the commands in each pair is used in different transactions, there can a conflict issue. The example below shows where the conflict may occur.

T0	T1
write x	
	write x
read x	
	ABORT

In the table above, T0 and T1 is writing the same account which will result in conflict. Write here means that deposit, withdraw, or commit are used to edit the account x. Any combination of deposit, withdraw, and commit violates the concurrency. Therefore, in this

circumstance, we ABORT the transaction from T1 because T0 has begun the transaction earlier than T1.
ABORT is ensured by testing multiple test cases. Below are the test cases where we guarantee that no conflict occurs.

T0	T1
write x	
	write x
read x	
	ABORT

T0	T1
write y	
	write x
read x	
ABORT	

T0	T1
write x	
	write x
	commit
	ABORT

T0	T1
read y	
	write x
write x	
	ABORT

T0	T1
read x	
	write x
read x	
	ABORT

T0	T1
write y	
	write x
write x	
	ABORT

T0	T1	T2
write x		
	write x	
		write x
read x		
	ABORT	
		ABORT

T0	T1
write y	
	read x
	write x
	commit x
	ABORT

T0	T1
write y	
	read x
	write x
	commit x
	ABORT

T0	T1
write x	
	write x
commit x	
	ABORT

Once a transaction is aborted, the system needs to roll back its state. Rolling back was discussed in the previous sections. However in general, we need to clear the `tentativeDicts` in the server, `txIdList` and `txIdAccList` in the coordinator, and the `inTransaction` flag in the client.

7 Extra Credit: Deadlock Resolution

Since we use timestamp strategy, this inherently does not lead to any situation where we have any deadlocks. The timestamp strategy will prioritize older transactions and always abort newer ones. Therefore, the proposed system cannot have any kind of deadlock issue. Our implementation details and the condtions upon which aborts occur have been detailed in the previous sections.