

Design Document for CS425/ECE428 MP3

Zhuo Li¹, Ruiqi Zhong²

1. Dept. of Computer Science, UIUC

2. Dept. of Electrical and Computer Engineering, UIUC

Email: {zhuol2, rzhong5@illinois.edu}

1 Overview

In this MP, we implemented a distributed transaction system which satisfies atomicity, consistency and isolation. We also ensure that the distributed transaction system is able to detect deadlock and abort minimal number of transactions to let other transactions proceed.

2 Algorithm Designs

In this section, we will take a deep look at the algorithms we choose and the reason why we choose them. Pseudocodes will be provided for implementation details and the end of each sub-section.

2.1 Basic Functionality of Distributed Transactions

We have 1 coordinator, 5 servers and 3 clients in total. For basic functionality, each client has its own transaction id. Clients can enter commands. If we find that the transaction by the client requires locks held by others, we use 2P Locking to arrange the order of transactions.

2.1.1 Algorithm

We use RMI to help client call the remote functions in the server interface.

Each client has a tentative storage to store all its uncommitted SET commands. In tentative storage, the key is the server name, the value is the key and value pair of a SET command, like (X, 4). Once a transaction is to be committed, all the SET commands will be executed. Once a transaction is to be aborted, all the commands in the tentative storage will be cleared.

Each server has a local storage to record the occupying status of its serverObjects. For example, for X, it has its own value in the local storage of server, its status marking if it is read-locked or write-locked, as well as the owners of its read-locks and write-locks.

We use 2P Locking to determine the wait-for relationship and promotion of locks, which realizes the atomicity and isolation of transactions.

2.1.2 Implementation Details

```
"SET server.key value"
    if ABORT or not BEGIN yet or invalid inputs:
        print error message
    else:
        localStorage.put(server, key, value)
        canPut = tryPut(server, key) // check if locked
        while !canPut:
            wait(500ms)
            canPut = tryPut(server, key)
        if canPut == 'ABORT':
            abort()
        else:
            put(server, key, value) // do put key-value

"GET server.key"
    if ABORT or not BEGIN yet or invalid inputs:
        print error message
    else:
        value = localStorage.get(server, key)
        if value == null:
            canGet = tryGet(server, key)
            while !canGet:
                wait(500ms)
                canGet = tryGet(server, key)
            if canGet == 'ABORT':
                abort()
            else:
                value = get(server, key)
```

2.2 Deadlock Detection

For deadlock detection, we use graph library in java to help detect cycles. Once a command fails due to the occupancy of locks by others, it will add the wait-for relation into a graph which is stored in the coordinator class if the wait-for relation does not exist in that graph. We use TimerTask in java to periodically detect cycles in the graph.

2.2.1 Algorithm

Once a command fails due to the occupancy of locks, it will continually and periodically call the `tryGet` and `tryPut` function to check if the locks occupied are released. Every time before the function returns FAIL, it will add wait-for relation to the graph for deadlock detection if the wait-for relation does not exist.

We use `TimerTask` named `DeadlockDetector` to periodically monitor cycles in the graph. Once a cycle is found, one `transactionId` with the largest timestamp in the cycle is chosen to be aborted and stored it in the coordinator class.

During the invocation of method `tryGet` and `tryPut` for the pending commands, the transaction will check whether itself is in the set for transactions to be aborted. If so, it will release all locks occupied by itself, clear its local storage for uncommitted commands, remove itself from the map held by the coordinator to store the pending transactions along with its timestamps marked by `BEGIN` command, also remove its `Id` from the map to store the transactions to be aborted and all of its relations in the graph stored in coordinator.

Our implementation of deadlock detection realizes the consistency of the whole system.

2.2.2 Implementation Details

```
Client input SET
    If we cannot set value due to occupied locks
        Query server to tryPut until tryPut do not return
        FAIL
        If tryPut returns ABORT
            Execute abortTransaction()

Client input GET
    If we cannot get value from the client's local storage
        Query server to tryGet until tryGet do not return
        FAIL
        If tryGet returns ABORT
            Execute abortTransaction()

In tryGet and tryPut
    If transaction id of itself equals the transaction id to
    be aborted
        return ABORT
    Use 2P Locking to check if the required locks are
    available
```

```
Add edge to the coordinator's graph for deadlock
detection
```

```
In abortTransaction
```

```
Release all writeLocks and all readLocks for this
transaction
```

```
Clear tentative local storage
```

```
Remove the transactionId of itself from the
coordinator's storage and reshape the
graph for deadlock detection of coordinator
```

```
In coordinator
```

```
remote method addEdge can be called by the tryGet
function once wait-for relationship is detected
```

```
Execute DeadlockDetector TimerTask periodically to
detect cycles and select the transaction to be aborted
according to their timestamps
```

3 System Evaluation

3.1 Description of Testing Approach

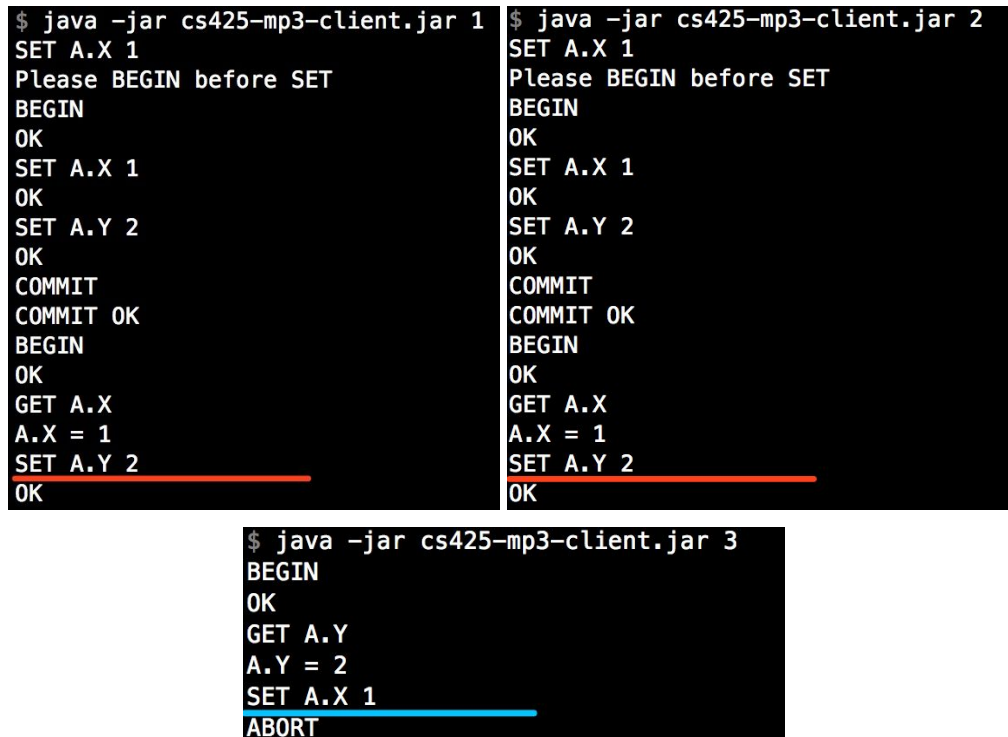
We first test cases without deadlock. This is trivial. Just BEGIN a transaction and do several SET/GET operations. For a specific client, it will find what they SET, when they GET locally before COMMIT, while other clients can not GET the object until COMMIT. We also test concurrent cases when we SET or GET simultaneously from different clients.

Then we test for deadlock detection. This is the interesting part. We first test deadlock between two clients. For example, client 1 wants to GET but waiting for client 2's SET, and client 2 waits the opposite. Then we test for a "wait-for" among client 1, 2, 3.

Then we also test for two simultaneous deadlock. The scenario can be: there are A.x and A.y in server A. We first let client 1,2 GET A.x at the same time. Then client 3 GET A.y followed by SET A.x 1. At this time, client 3 should wait for client 1 and client 2. Then client 1 and 2 SET A.y 2 at the same time. Then two deadlocks exists (i.e. one between client 1 and 3, one between client 2 and 3). We give our solution in screenshot Figure 1 and Figure 2. Our

approach is to abort the transaction with largest timestamp. If we BEGIN in client 1 and 2 first, we will ABORT client 3 (Figure 1). If we BEGIN in client 3 first, we will ABORT both client 1 and client 2 (Figure 2).

3.2 Screenshot

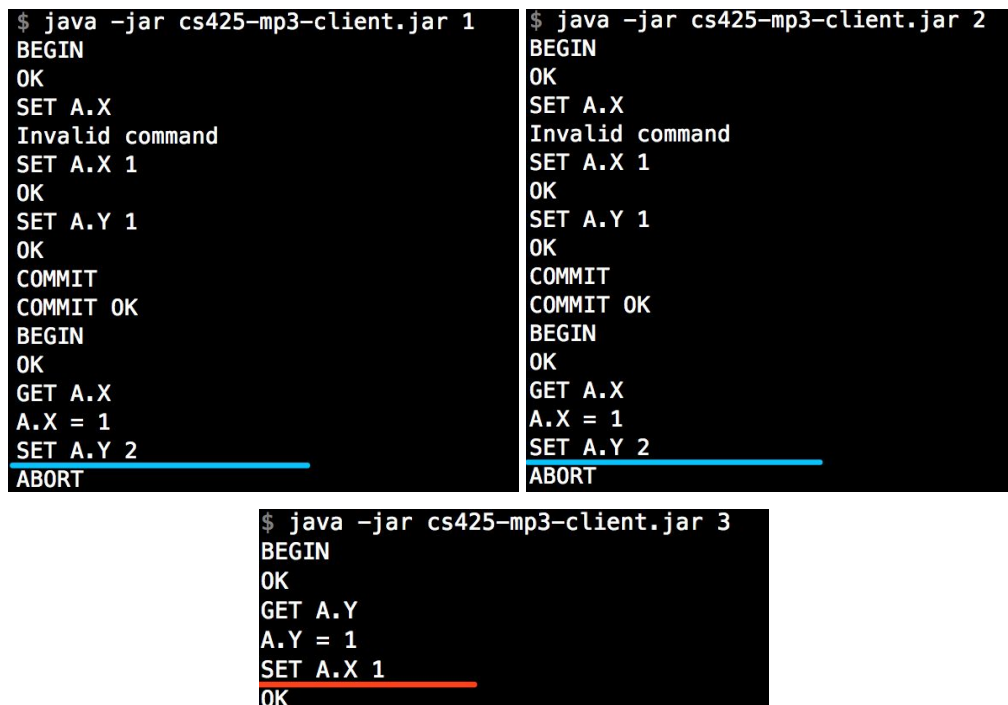


```
$ java -jar cs425-mp3-client.jar 1
SET A.X 1
Please BEGIN before SET
BEGIN
OK
SET A.X 1
OK
SET A.Y 2
OK
COMMIT
COMMIT OK
BEGIN
OK
GET A.X
A.X = 1
SET A.Y 2
OK
```

```
$ java -jar cs425-mp3-client.jar 2
SET A.X 1
Please BEGIN before SET
BEGIN
OK
SET A.X 1
OK
SET A.Y 2
OK
COMMIT
COMMIT OK
BEGIN
OK
GET A.X
A.X = 1
SET A.Y 2
OK
```

```
$ java -jar cs425-mp3-client.jar 3
BEGIN
OK
GET A.Y
A.Y = 2
SET A.X 1
ABORT
```

Figure 1



```
$ java -jar cs425-mp3-client.jar 1
BEGIN
OK
SET A.X
Invalid command
SET A.X 1
OK
SET A.Y 1
OK
COMMIT
COMMIT OK
BEGIN
OK
GET A.X
A.X = 1
SET A.Y 2
ABORT
```

```
$ java -jar cs425-mp3-client.jar 2
BEGIN
OK
SET A.X
Invalid command
SET A.X 1
OK
SET A.Y 1
OK
COMMIT
COMMIT OK
BEGIN
OK
GET A.X
A.X = 1
SET A.Y 2
ABORT
```

```
$ java -jar cs425-mp3-client.jar 3
BEGIN
OK
GET A.Y
A.Y = 1
SET A.X 1
OK
```

Figure 2

4 Summary

In this MP, we build a distributed transaction system using 2P locking to determine the sequence of transactions, which also enable deadlock detection by finding cycles in wait-for relationship graph. Every time there is a wait-for relationship presented, the server will send the edge to the graph in the coordinator. The coordinator periodically check for cycles in the graph and tell the server to abort the transaction with the latest timestamp. The system survives in different scenarios quite well.