CS 425/ECE 428
Distributed System

MP3 Distributed Transactions

Design Documents

Group Member:

Name: Rui Xia
NetID: ruixia2
UIN:660935771

Name: Youjie Li
NetID: li238
UIN: 668129373

Spring 2017
May 1, 2017

Distributed Transaction System

In this MP, a distributed transaction system is designed and implemented to support read and write to distributed objects while ensuring full ACI properties. The atomicity is guaranteed by the two phase commit and the isolation of concurrent transactions is realized by the non-strict two phase locking (read/write locking). During the 2P locking, global deadlocks are common, so deadlock detection is essential to our design.

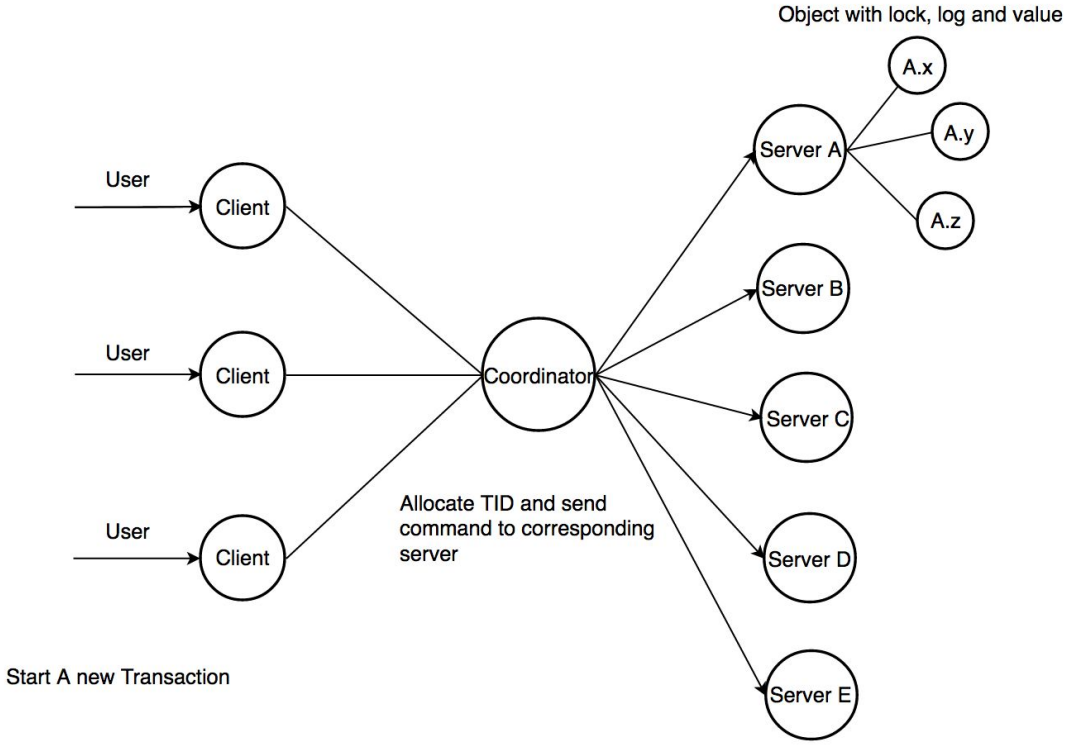

Figure 1 Design Structure

Category	Function
Client	Accept input from users; Test validity of input command; Send command to coordinator
Coordinator	Leader for 2 Phase Commit and Deadlock Detection; Transmit command to server
Server	Supervise lock for each object; Execute command

Table 1: Function for different categories of node

Concurrency and Isolation Control

In order to guarantee the serializability of the concurrent transactions, we have used the Read and Write Lock for each object. The lock compatibility is shown below:

Lock Already Set	Read Lock Request	Write Lock Request
None	Grant	Grant
ReadLock	Share	Block
WriteLock	Block	Block

Table 2: The two-phase lock

To be specific, the 2PL algorithm is designed as follows. (Assumes on server nodes.)

Upon receiving "GET object":

```
If the object is unlocked:
        Read its value and Set the object as "ReadLock"ed
Else if the object is "ReadLock"ed:
        If it is "ReadLock"ed by my TID:
                Read its value
        Else:
                Read its value and Share the "ReadLock"
Else if the object is "WriteLock"ed:
        If it is "WriteLock"ed by my TID:
                Read its value
        Else:
                Add this GET request to the waitlist of this object
```

Upon receiving "SET object":

```
If the object is unlocked:
        Set its value and Set the object as "WriteLock"ed
Else if the object is "ReadLock"ed:
        If  it is "ReadLock"ed ONLY by my TID:
                Set its value and Promote the Lock
        Else:
                Add this SET request to the waitlist of this object
Else if the object is "WriteLock"ed:
        If it is "WriteLock"ed by my TID:
                Set its value
        Else:
                Add this SET request to the waitlist of this object
```

The above algorithm is executed either when GET/SET requests are received at Server nodes or any objects are unlocked by COMMIT or ABORT. Also, ABORT TID request will delete the entries belong to this TID in the wait-list of every object.

Deadlock Detection Algorithm

Coordinator is designed to supervise the deadlock detection by in our algorithm. The detection is time triggered. Coordinator will heartbeating an msg “ask” for ‘wait for’ information periodically. Then the server will report the waiting information for each transaction to coordinator as soon as they received the ‘ask’ msg.

When receiving the wait information from all servers in the same term, the coordinator will construct the waiting graph and detect any circle in it. The wait for graph will be like Figure 2:

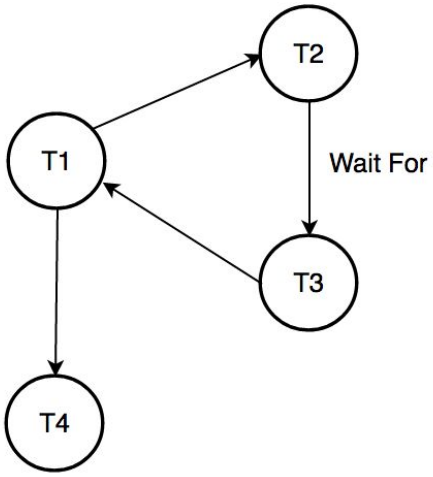

Figure 2 Wait Graph

In Figure 2, T1 are waiting for T2 and T4’s lock while T2 waits for T3 and T3 waits for T1. This circle will be detected as a deadlock. We implemented the wait graph using dictionary in python and detect the circle by a recursive Depth First Search algorithm. The detected entry of circle in graph will be decided to abort by coordinator. Then the ABORT command will be sent to server with an notification to client as well so that user will be noticed that the last transaction is aborted.

Parameter

The Period of heartbeating : T=2000ms

Modified Two Phase Commit

We have modified slightly the 2PC protocol to guarantee the atomicity, as shown below.

Coordinator:

Phase 1:

Upon receiving COMMIT request from client, multicast “canCommit?” with transaction TID to all participant servers and gather their votes.

Phase 2:

Only if Votes received from all participants, multicast “doCommit” with TID to all participants. Else wait for their votes.

Participant Server:

Phase 1:

Upon receiving “canCommit?” request from coordinator, check the waiting list of each object on this server to see whether this TID is still waiting.

Only if this TID has no waiting operations, reply vote to coordinator.

Phase 2:

If receive “doCommit”, commit this TID transaction.

Else if receive “Abort”, abort this TID transaction.

Evaluation

We have implemented our distributed transaction system using Python and Pypy. In order to test the scalability and performance of our system, one major metric is evaluated -- the throughput of transaction operations. The operations in this evaluation include (BEGIN, SET, GET, COMMIT, ABORT). The throughput can be calculated as the number of completed operations divided by total time elapsed. To measure the throughput, a python script is used to load operations to the client nodes and to measure the timing according. To be specific, 10000 operations per client node are evaluated where the most of SET and GET operations have key conflicts. All operations are executed successfully. The table below shows the experimental results.

Number of Clients	Average Throughput	95% Confidence Interval
1	12.47 operations/second	[12.43, 12.53]
2	24.91 operations/second	[24.58, 25.26]
3	37.04 operations/second	[36.41, 37.69]

As can be seen, the throughput of the implemented system scales efficiently from 1 client node to 3 nodes. The operations from each client nodes are executed concurrently while the performance of the system doesn’t drop. The experimental results verify the scalability and performance of our design.