COMP 512: Project 3

Group 14: Jin Dong, 260860634; Shiquan Zhang, 260850447

1 General design and architecture:

The booking system mainly has three parts: Client, Middleware and three Remote Managers. The architecture of the system is shown in Figure 1.

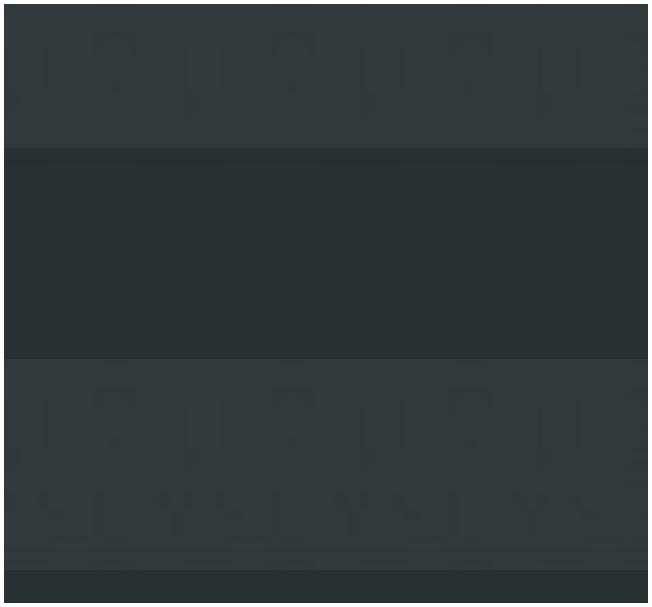

Figure 1: High-level Architecture of the System

Client

Client is directly communicate with clients. It deploys on the clients' machine and provides some APIs for clients, including 'AddCustomer', 'ReserveFlight', etc. It recieves commands from clients and proceeds them to the server (Middleware). After server finishing to process the commands, it recieves the execution results and prints out on the screen.

Middleware

Middleware is mainly consisted by four parts: RMIMiddleware, Transaction Manager, Lock Manager and Customer Remote Manager.

1. RMIMiddleware is the only part of the server which directly communicates with Client. It exposes all APIs that Client needs and coordinates all of these commands in the server side.

2. Transaction Manager mainly takes charge in the concurrency control of all transactions. It holds the transaction ID list and the corresponding RMs for every transaction. It also acts like the coordinator in the 2 Phase Commit.

3. Lock Manager ensures the data integrity among concurrent transactions. The system uses 2 Phase Locking algorithm and all locks of data are managed in the Lock Manager.

4. Customer RM takes charge in managing the data of the customers, including customer ID and reservations of every customer.

Remote Managers

There are three distributed databases deployed in three different remote machines, which are managed by three Resource Managers separately. They are Flight RM, Car RM and Room RM, which take charge in reading, writing and committing/aborting the data of flights, cars and rooms.

2 Individual features:

The distributed booking system has several features that insure it runs well when handling several transactions from different clients concurrently. Basically there are four main features: distribution and communication, transactions and locking (2PL), data shadowing and logging and recovery (2PC).

Distribution and communication

The system uses

Transactions and locking

We implement a centralized lock manager, letting middleware manage locks of all items. For each method a transaction calls, the middleware will first call its *lockSomething* method. The transaction manager in the middleware will first check if the transaction exists. If not, a *InvalidTransactionException* will be threw. Then the lock manager will try to grant a corresponding lock on that item to the transaction. If either the lock granting failed or a DeadlockException is threw, middleware will abort this transaction and throw a *TransactionAbortedException*. Only the *lockSomething* succeeds can the subsequent operations be executed. For example, in a *reserveCar* operation, the middleware will try to grant a WRITE lock on the car and a WRITE lock on the customer to this transaction. If both succeed, the real *reserveCar* operation will be executed.

When a transaction request a lock on an item on which the transaction already has a lock, there are three situations that may happen: (1) request: read, have: read (2) request: write, have: write; (3) request: write, have: read. For (1) and (2), Lock Manager will raise RedundantLockRequestException and return true; for (3), Lock Manager will first check if other transactions are granted locks on this item. If not, Lock Manager will delete the read lock and grant a write lock to this transaction.

Data shadowing

Logging and recovery (2-PC)

3 Special features:

We also implement the transaction manager in the middleware, but add some other data structures in the resource managers.

Map from xid to related RMs in TransactionManager: When Middleware reconnects to RMs after whoever crashes, we need to update the whole map if we store RM as a reference here. Instead, we only store the RM's name and add a small Hashtable which maps the name to the reference. So when reconnection happens, TransactionManager only need to update each reference one time at most.

Crash detection: Middleware need to detect the crash of RMs so that it can reconnect to the RM on time. We create a new Thread in which Middleware will ping the three RMs (send an empty message). Whenever this thread catches a RemoteException it knows that this RM crashed, so it will try to reconnect the RM and update the reference in TM.

Data Structures in TransactionManager

4 Problems:

5 Testing:

1. Hashtable¡Integer, Vector¡IResourceManager¿ ¿ xid_rm: store active transactions and corresponding ResourceManagers on which the transaction do some operations.

2. Hashtable¡Integer, Long¿ xid_time: store active transactions and the start time of its last operation.

Data Structures in ResourceManager

1. HashMap¡Integer, RMHashMap¿ origin_data: store transactions that modified data of this RM and the list of original data that the transaction modified.

Ways of Finishing Related Operations

Start: When a client call Middleware's *start* method, Middleware forwards it to TransactionManager TM, TM has a static variable, *num_transaction*, recording the number of transaction and return the next number as xid, which will be returned to the client. Meanwhile, TM will also add xid to its *xid_rm* and *xid_time*.

Operations: For each operation, Middleware will first apply a lock for this transaction by calling *lockSomething*. In a ***Query*** operation, Middleware will apply a READ lock on this item; in a ***Add*** or ***Delete*** operation, Middleware will apply a WRITE lock on this item; in a ***ReserveItem*** operation, Middleware will apply a WIRTE lock on both the item and the customer. In a ***DeleteCustomer*** and a ***Bundle*** operation, Middleware will apply WRITE locks on the customer and all related items. Meanwhile, in all ***Add, Delete, Bundle, DeleteCustomer*** operations, we need to store the original data to **origin_data** in case abort happened. Considering all modify-related operations are finished by calling ***writeData*** and ***removeData*** on corresponding RMs, we add the original data to ***origin_data.get(xid)***. If the operation creates new item, we add the original data as (key, null), in which null indicates that this is a new added item and thus need to be deleted when abort happens.

Abort: Abort will happen in three situation: (1) Client call Abort; (2) DeadLock happens; (3) The transaction timed out and aborted by the transaction manager in the middleware. The only difference among them is the way that client is notified about the abort. In (1), it's no need to notify the client; in (2), client is notified by catching a TransactionAbortedException threw by the middleware; in (3), client will not be notified until it calls the next operation on this transaction and catch a InvalidTransactionException. In all three situation, the middleware will call tm.abort(xid) first, in which TM will call abort of RMs related to this transaction. Each RM will recover the data stored in ***origin_data.get(xid)***, and reset ***origin_data.get(xid)***. Then the middleware will also recover its own data, as a Customer-RM. At last, Lock Manager LM will release all locks of this transaction by calling *unlockAll*.

Commit: When a client calls *commit*, the Middleware forwards it to TM, and TM will commit all related RMs and delete this transaction from activeTransaction. Each RM only need to reset ***origin_data.get(xid)***. Then the middleware will also reset its ***origin_data.get(xid)***, as a Customer-RM. At last, Lock Manager LM will release all locks of this transaction by calling *unlockAll*.

Shutdown: When a client calls *shutdown* of the Middleware, Middleware will first call *shutdown* of three RMs, and then *exit()* itself. In each RM, RM will exit itself, too.

6 4. Time-to-live Mechanism:

We implement this function by creating a new Thread in Middleware, which for-loop all active transactions and checks if System.currentTimeMillis() - xid_time.get(xid) ¿ MAX_EXIST_TIME. If so, Middleware will abort this transaction. Client will know this abortion when it calls next operation on this transaction and catch a UnvalidTransactionException. We update xid_time.get(xid) in the middleware's *lockSomething* function because each operation will call Lock first.