COMP 512: Project 3

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

1 General design and architecture:

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

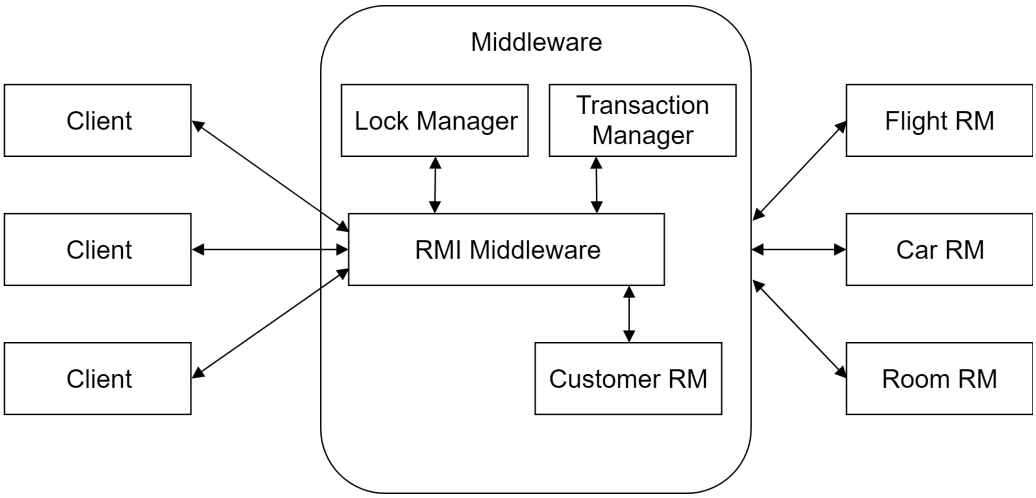

Figure 1: High-level Architecture of the System

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 (RM)

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 booking system mainly has four types of data: customers, flights, cars and rooms, which are managed by four RMs. Three RMs are physically in three remote machine separately and the Customer RM is in the Middleware. When a normal operation of a transaction, i.e. *add/delete/query/reserve*, comes in, Middleware records some information about this transaction, such as the corresponding RMs, in the Transaction manager. Then Middleware applies locks for this operation in the Lock manager. Next, it forwards the operation to the corresponding RMs. Finally, Middleware collects the results and returns them to the Client.

The system uses the Remote Method Invocation (RMI) in Java. The RM exposes several APIs that the Middleware can invocates them remotely and get the results from the resouce managers. So does that between Middleware and Client. The RMI method hides the details of network communication in the built-in RMI module and it's convenient to use.

Transactions and locking (2PL)

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 thrown. 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 thrown, the 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, the Lock manager will delete the READ lock and grant a WRTIE lock to this transaction.

Data shadowing

In case that some parts of the system may crash sometime, the data must be preserve into the disk, as well as be written into a file. However, there are a lot of I/O operations in saving data. It might be problematic if the system crash during this period. To guarantee the safety of data, we use data shadowing in the system.

When committing a transaction, both Middleware and RMs need to write some data to the disk. For every RM, there are two files shadowing the data, *RMname.A* and *RMname.B*. One is working file and the other is master records. There is a pointer indicating which file is the master records and which is the working file. Every time the RM executes commit transaction, it writes the transaction data to the working file. When finishing writing successfully, it flips the pointer from the original master records to the working file. And the original master records becomes the new working file. Besides, the pointer is preserved in another file, *RMname.master*. If the system crash when writing data into the working file, the pointer doesn't change. Therefore, when the RM restarts, it recovers the data from the original master records and loads them into the memory.

Logging and recovery (2-PC)

To guarantee the data consistency during committing, we implement the two phase commit procedure, which includes some logging and recovery procedures. For the 2PC procedure, the log is writen in a independent file, *name.log*.

When Client decides to commit a transaction, *twoPC()* in the Middleware will be called. In the Middleware, the Transaction manager is the coordinator of 2-PC. First, it logs the *"start-2PC"* information and calls the *prepare()* in every RM corresponding to this transaction concurrently by starting multiple new threads. Then it logs the voting results in a Hashmap *vote*. When the size of *vote* equals to the number of corresponding RMs, voting finishes and the it checks the voting results. If all RM vote YES, it logs *"COMMIT"* and calls the *commit()* in every corresponding RM. Otherwise, it logs *"ABORT"* and calls the *abort()* in every RM voting YES.

If the Middleware crashes and restarts, it will execute *restart()* automatically. In the recovery phase, the Middleware not only recovers the data from shadowing files, but also checks the 2PC log file. For every transaction logged in the file, the Middleware checks its status in 2PC. For the transactions with log *"START-2PC"* and *"ABORT"*, Middleware executes *"abort()"*. For the transactions with log *"COMMIT"*, Middleware executes *commit()* again.

In the RM side, when the *prepare()* is called, RM will check whether this transaction has

been aborted in its records. If the transaction has been aborted, RM returns NO. Otherwise, it logs *"YES"* and returns YES to the Middleware. When RM recieves the decision from the Middleware, as well as the *commit()* or *abort()* is called, RM will log *"COMMIT"* or *"ABORT"* and execute correspondingly.

If RM crashes and restarts, it will also execute *restart()* automatically. Besides recovering data from the shadowing file, RM also reads the information from the 2PC log file. For every transaction in the log file, if it has the log *"INIT"*, RM will aborts it. Otherwise, RM does nothing.

3 Special features:

Besides the main features above, we also implement some special auxiliary features to strengthen the robustness of the system.

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.

Time-to-Live mechanisim

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.

4 Problems:

For the crash mode 3 in RM, it requires that RM should crash after sending the vote to the coordinator while before recieving the decision. In our implementation, we use RMI as the communication method between Middleware and RM, thus after returning the vote to the Middleware, the thread for *prepare()* ends and there is no place to execute the crash operation.

To solve this problem, we create a new thread at the end of the *prepare()*. If the flag for creash mode 3 is up, this thread sleeps for 10 milliseconds to ensure the vote returns to the Middleware and then it executes *System.exit(1)* to crash the RM.

5 Testing:

Testing mainly includes concurrency control test and crash/recover test.

Concurrency control

To test the locking mechanism, we test serveral examples in different aspects, including testing simple commit, simple abort, Customer lock, lock conversion, deadlock, time-to-live check, bundle atomicity and multiple operations using local copy. Take the deadlock test as an example, we starts two Clients simultaneously. One queries on item A and adds on item B, while the other queries on item B and adds on item A. Then both Clients are caught into a deadlock. After waiting for timeout, the first transaction will be aborted and the other will succeeds.

Crash and recovery

We insert breakpoints for 8 crash modes in the Middleware and for 5 crash modes in RM, which covers all critical operations in the 2PC procedure. When some of the flags for these breakpoints are true, the Middleware or RM will crash when committing a transaction. For different crash modes, the log information or the status of other servers might be different when one machine crashes. So we can test the data shadowing and recovery mechanism in 2PC procedure by crashing in different modes and restarting manually. After restarts the Middleware or RM that crashed, the Middleware/RM should be able to load the data back to memory from the shadowing file. And the decision for every transaction should also be consistent in Middleware and all RMs (committed or aborted).