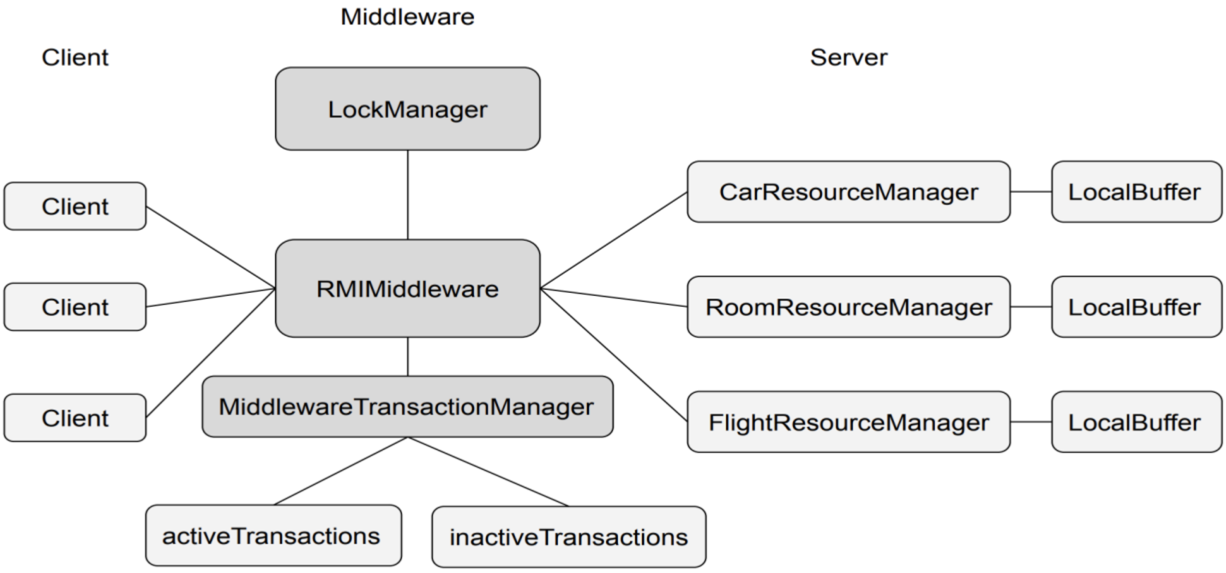
Project Part 2: Transaction and Concurrency Control
COMP512 Distributed System

Group 8: Zhang Yong, Eric Shen, Winnie Tan

Architecture

As the Figure below shows, we continued using RMI implementation as we did in Milestone 1. In addition to all the designs we have done in M1, the transaction manager and `LockManager` both reside at the middleware. Meanwhile, all the unflushed data are stored in a `LocalBuffer` before commitment or abortion.



Lock Manager & Locking

In our design, we have a centralized locking manager at middleware to guarantee two-phase locking on the individual data items. That is to say when a client wants to write an item in the database, `LockManager` would check whether there are other transactions that hold the LOCK_READ or LOCK_WRITE of this particular item. If not, then the lock could be safely granted. When a client tries to read an item, `LockManager` would check whether there are other transactions that hold the LOCK_WRITE. If not, then the LOCK_ READ could be granted to the transaction. All of these locks would only be released until the transactions have done the commit or abort.

Lock conversion is handled by the `Lock` function when receiving the X-lock request from the client. During runtime, there exists a situation when the client is holding the share-lock of an object while no other transactions are currently holding any kind of locks on the object (`bConvert` is true). In this case, we are able to do the lock conversion: firstly we have to remove the share-lock of object X from `lockTable`, then add the new instance of `TransactionLockObject` and `DataLockObject` (X-lock of the new object) back to `lockTable`.

In the `LockConflict` function, we are trying to check if the lock request of a certain object conflicts with the existing locks. However, it is possible that the transaction is currently holding some kinds of locks of this object and the client still wants to get the X-lock of this object. Therefore, there are 2 cases to be considered:

First, the transaction is currently holding the X-lock of this object, then this request is redundant, and we will throw the `RedundantLockRequestException`.

Another case is that the transaction is holding the share-lock of this object and we may consider lock conversion. Again, We can divide this situation into 2 cases. If other transactions are currently holding the locks on the object, we should return True and put this request into `waitTable`. If this request does not conflict with any other transactions, then we can return True and set `BitSet` to True.

Transaction Management

Start:

In our design, every time a client sent the start command, `RMIMiddleware` would deliver it to

MiddlewareTransactionManager. Then, the manager would generate a global and unique identifier (simply by incrementing a counter every time) to denote the transaction ID. After that, MiddlewareTransactionManager would create a Transaction object to store related information about this transaction (e.g. which resource managers are involved). Finally, the transaction would be put in the activeTransaction hashmap of MiddlewareTransactionManager.

Operations:

Every time the client executes a normal command, RMIMiddleware would check whether this command is from an active transaction at first. If not, an InvalidTransactionException would be thrown. Then, the middleware would determine what type of locks on a specific object this command needs (e.g. X-lock of customer 10 and room “Montreal” for *reserveRoom(1, 10, “Montreal”)*) and try to get the locks from LockManager. The LockManager would throw a DeadLockException if there is a deadlock happening. If the locks are successfully acquired, then all the data this transaction has modified would be put in a LocalBuffer of a resource manager. These temporary data would not be flushed or deleted until the commit or abort time. Every time a data object is accessed or modified, the related resource manager would be put in the list of managers of the transaction. For instance, *{addRoom,1, “Montreal”, 10, 10}* would put the RoomResourceManager to the list of managers of transaction 1.

Commit:

Same as the operations part, RMIMiddleware would check whether this command is from an active transaction at first. Then RMIMiddleware would tell other related resource managers to commit the transaction. Upon receiving the commit request from RMIMiddleware, the resource manager would write the temporary data into their dataset and clean the local buffer by deleting the write set of this particular transaction. After all the resource managers have done the commits on their local database, RMIMiddleware would do the same thing for the Customer data type. Then it would move this transaction from the activeTransactions hashMap to inactiveTransactions hashMap and label it as *InactiveStatus.COMMITTED*. At the last, LockManager would unlock all the transaction’s locks.

Abort:

Again, transaction availability would be checked at first, then RMIMiddleware would inform other related resource managers to clean their local buffer of the transaction denoted by xid. Once they are all done, RMIMiddleware would clean the customer data in the local buffer. Then it would move this transaction from the activeTransactions hashMap to inactiveTransactions hashMap and label it as *InactiveStatus.ABORTED*. At the last, LockManager would unlock all the transaction’s locks.

Others:

Meanwhile, at the middleware side, there is an extra thread continuously checking whether the active transactions have reached the time-to-live every 3 seconds with the total time-to-live equals 50 seconds.

Example:

To give a tiny example with a transaction (xid = 1), these commands are executed: *{start} {addFlight,1,10,10,10} {commit, 1}*.

At first, upon receiving the start request, RMIMiddleware would ask MiddlewareTransactionManager for a unique ID and return it back to the client. Transaction 1 would be set as active and added to the activeTransactions hashmap in MiddlewareTransactionManager.

Upon receiving the *addFilght* command, RMIMiddleware would try to get the write lock of the flight object with flight number 10. After checking the lock table, LockManager would grant the lock to this transaction to perform writing. Then RMIMiddleware would add FlightResourceManager to the involved manager list of transaction 1 and invoke *addFlight* method on FlightResourceManager to do the operation. FlightResourceManager would write the data in a local buffer but not directly in the database.

Finally, when receiving the commit, RMIMiddleware would get the transaction object at first, and read its involved resource manager list. Then, RMIMiddleware would call commit on FlightResourceManager in this case. Then, this manager would get the local copy of the data in his LocalBuffer and write it to the database. Once all the commits are done, RMIMiddleware will move transaction 1 from activeTransactions hashmap to inactiveTransactions hashmap and label it as *COMMITTED*. In the end, all the locks would be released, and the feedback is given to the client.