Sheldon Benard – Group 7

Distributed Systems – Milestone 2

Architecture

For Milestone 2, I decided to continue with the RMI implementation from Milestone 1, and the layout of the architecture is depicted in Figure 1.

Clients connect to the middleware and the middleware coordinates with the Transaction Manager, Lock Manager, and the Resource Managers to deliver a client's transaction.

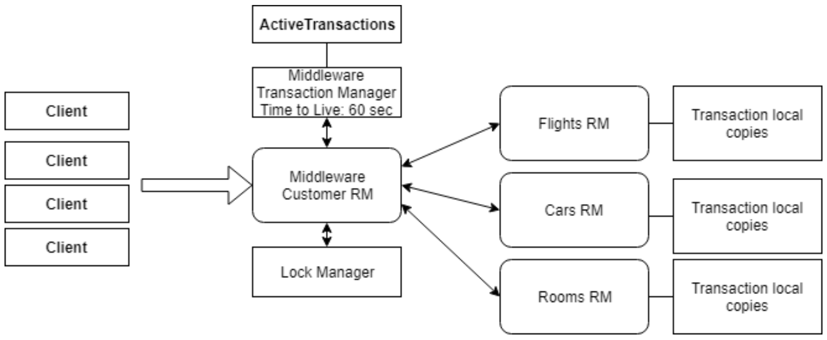

Figure 1 - Architecture

Locking

2-Phase Locking was implemented on the individual data items at the Middleware level (thus, I used a centralized locking manager at the Middleware, rather than locking managers at each site). When a client performs a query on *data_item*, the locking manager grants a READ_LOCK for *data_item* only if no other transactions (besides the current transaction) has a WRITE_LOCK on the *data_item.* When a client modifies the state of the data, the locking manager grants a WRITE_LOCK for *data_item* only if no other transactions (besides the current transaction) has a WRITE_LOCK or READ_LOCK on the *data_item.* Moreover, a transaction unlocks all their locks at the end of the transaction (after an abort or commit). Although restrictive, 2-Phase Locking prevents any data corruption and allows for the correct coordination of transactions in the distributed environment.

As part of the architecture, I had to extend the Lock Manager to include lock conversions. If a transaction requests a WRITE_LOCK, and the only other lock on the object is a READ_LOCK of the same transaction, then convert the READ_LOCK to a WRITE_LOCK. Thus, this entails:

1. Detecting that transaction t_i is requesting a WRITE_LOCK on a data item for which t_i already has a READ_LOCK and no other transaction has a lock on the item
2. Removing the *TransactionLockObject* and *DataLockObject* from the lock table (the READ_LOCK) and write a new WRITE_LOCK *TransactionLockObject* and *DataLockObject* to the lock table.

Transaction Management

To illustrate how transactions are handled, I'll go through an example scenario:

Start t_1*,*t_2 → *addFlight,1,1,1,1* → *queryFlight,2,1* → *commit,1* →*addFlight,2,1,1,1* → *abort,2*

Start

When a client executes the *Start* command, the middleware asks for a new transaction id (xid) from the Middleware Transaction Manager. The Middleware Transaction Manager simply increments a counter *i* and returns *i* back to the middleware. The middleware returns this to the client. In the background, the Transaction Manager creates a new *Transaction* object (xid=*i)* and adds this object to the HashMap of active transactions.

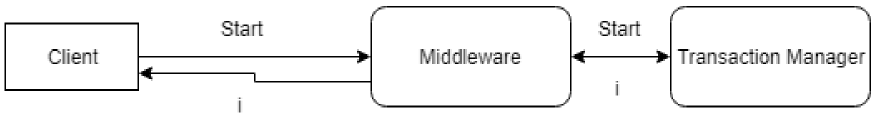

Figure 2 – Start transaction t_i

Moreover, at the bootup of the Middleware Transaction Manager, a thread to monitor the *Time to Live* of each of the active transaction manager is set up. Now that there are active transactions, the thread will periodically (every 5 seconds) check the state of each transaction. If the time of the latest transaction action plus the allotted time to live value (60 seconds) is less than the current time, the Transaction Manager assumes the connection was lost and the transaction is aborted.

addFlight,1,1,1,1

Now, t_1 wants to add a flight. The client calls the above command, and the middleware coordinates the delivery. At the middleware, the order of execution is as follows:

1. Check that the transaction still exists (i.e. part of the Middleware Transaction Manager's active transaction HashMap). If the xid doesn't exist or the transaction has already been committed or aborted, the client will be notified, and the command will fail. Else, continue
2. Next, acquire a WRITE_LOCK from the Lock Manager, since we are modifying (adding to) the data. This is granted since no other locks exist for *flight-1*.
3. Add to the transaction the fact that the *Flight* resource manager was used.
4. Forward the request to the *Flight* resource manager. The *Flight* resource manager creates a local RMHashMap for the transaction and creates *flight-1* in this local copy.

queryFlight,2,1

t_2 wants to query *flight-1*. At the Middleware, check for the existence of the active transaction and request a READ_LOCK on *flight-1*. This is **not** granted, since t_1 has a WRITE_LOCK, and t_2 is blocked. (NOTE: if this blocking lasts too long, a deadlock exception is thrown. In this case, t_2 would be aborted).

Commit,1

t_1 wants to commit. Using Middleware Transaction Manager, find the list of the resource managers used. For each of the resource managers used (in this case, only *Flight* was used. In general, the list could contain *Flight, Car, Room,* or *Customer*), commit t_1 at that manager. This consists of moving the local RMHashMap data associated with the transaction to the global RMHashMap. After the commit, t_1 unlocks its locks and the Transaction Manager moves t_1 from active to inactive(commit=True), and t_2 can proceed with the query (adding *Flight* to t_2's list of managers used, forwarding to the *Flight* manager, and copying *flight-1* from global to local RMHashMap and returning the value).

addFlight,2,1,1,1

t_2 first converts its *flight-1* READ_LOCK to a WRITE_LOCK in the Lock Manager. Next, the *Flight* manager modifies *t_2's* local RMHashMap with the additional seat for *flight-1*.

abort,2

t_2 wants to abort. For each of the resource managers used, abort the transaction at that manager. Since local copies are used, rather than modifying the global RMHashMap, abort is an inexpensive task: simply remove *t_2's* local copy of the RMHashMap. Finally, the Middleware Transaction Manager moves t_2 from active to inactive(commit=False) and t_2 unlocks its locks.