

For deliverable 2, we have chosen to implement distributed transaction ontop of our TCP system.

**The Centralized LockManager**

We have implemented the LockManager at the MW level as a static field. All lock requests have to go through the middleware in order to lock and unlock items.

**The Centralized TransactionManager**

The TransactionManager is implemented as an instance associated with each MWRunnable. As clients connect to the MW TCPServer, a MWRunnable is created to accept this client connection, and this MWRunnable will spawn a TransactionManager instance that is specific to this connection. When client submits non transaction requests, the MWRunnable will call its implementation of the RM interface methods. When the client issues start(), the TM of that connection will start the TTL coundown, generate a system wide unique transaction ID and return that to to the client. Until this transaction is aborted or commited, all requests will go through the TM's implementation of the RM interface methods instead of the MWRunnable's. The TM's implementation of the RM interface methods simply attemps to acquire all necessary locks, it then builds an undo command to undo the current request in case of abort and save this undo into the undoStack of the current transaction. It then goes on to call the MWRunnable implementation of this command which actually performs the appropriate reads and writes of data. The TM then saves the involved RM's into the global transaction table that is shared by all TM's. If Client issues the commit() command, the undoStack and all other state related to the current transaction is wiped from the TM so that it can be ready for the next transaction. If instead the abort() command was triggered, then TM will execute each command in the undoStack to revert all the effects in the current transaction. After an abort or commit, the list of involved RM's for this transaction is taken off the global transaction table. Each TM will allow 1 active transaction at a time for its client. After start(), entering an request with an XID other than the one returned to the client will result in that particular request being ignored.

**Time To Live**

After a start() comand is received by the TM, it will start a countdown thread which sleeps for TTL amount of time. Upon wakening, this countdown thread will call abort(). Following the start of the countdown thread, each request received by the TM from the client will reset this countdown thread, hence renewing its time to live.

**System Shutdown**

Once the MWrunnable receives a shutdown() command from the client, it will first consult the global transaction table to see if there are any active transactions still going on. If so it will ignore the shutdown command, otherwise it will first issue shutdown commands to the RM TCPServers then it will shut itself down.

**Difficulties**

The most difficult part of this deliverable was figuring out the undo process once a transaction is aborted. We opted for an undoStack that execute each undo command one after the other in stack order to revert any changes upon abort.   
  
Another difficult part would be lock request for deleteCustomer() and newCustomer(). DeleteCustomer requres locking of reserved items as well so we need a specific method just for that. NewCustomer() method was especially difficult because between the time it generates a unique customerId and the time it creates a customer with that ID, another transaction might use newCustomerID() using the same customerId that the system has generated for newCustomer(). Therefore the solution was to to lock on the generated customerId before newCustomer() can create the actual customer using it.

**Performance Evaluation (could be beyond the 2 page limit)**