

---

# COMP 512: Project 2

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

**1. Lock conversion:** 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.

**2. 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.

**3. Transaction management:** We also implement the transaction manager in the middleware, but add some other data structures in the resource managers.

## Data Structures in TransactionManager

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 Transaction-Manager 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 WRITE 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.

**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.