COMP512 - Distributed Systems
Deliverable II

Vincent Foley-Bourgon <vincent.foley-bourgon@mail.mcgill.ca>
Carol Morneau <carol.morneau@mail.mcgill.ca>

November 2013

1 Usage

All components of the project are launched by using the *make* command with the appropriate task name. Note that the *make* commands for the RMIs and middleware also start rmiregistry instances in the background. Be sure to execute *killall -9 rmiregistry* before you restart the servers.

```
$ make runcar    # Start the car RMI
$ make runflight # Start the flight RMI
$ make runhotel  # Start the hotel RMI
$ make runserver # Start the middleware server

$ make runclient # Interactive client
$ make populate  # Automatically load 1000 items in the RMIs
$ make automatedclient # Send random commands to the RMs
```

2 Architecture

2.1 Lock Manager

We modified the code in *LockManager.java* to accomodate lock conversion (from read-only lock to read-write lock). The two methods we had to modify were *Lock* and *LockConflict*. In *Lock*, if the *bConvert* bit is set, we remove the *TrxnObj* and *DataObj* from the manager and put in objects with the same properties, except the *LockType* has been changed from read to write.

In *LockConflict*, if the transaction wants a write lock on a data object and that this transaction already has a read lock, we set the *bConvert* bit to true. If the transaction already has a write lock on the data object, we throw a RedundantLockRequest exception.

Every resource manager has a LockManager instance to control access to its resources.

2.2 Transaction manager

The transaction manager is a singleton object inside the middleware server; it maintains a hash table mapping transaction IDs to a vector of resource managers. This allows the middleware to know what kinds of resources (cars, flights, hotels) a transaction is interacting with.

The transaction manager also has an *isValidTransaction* method that is called before every RMI call; this is to make sure that all operations in a valid transaction context.

2.3 Working set

The *WorkingSet* class allows a transaction to perform different actions without writing directly into the RMs' hash tables. A *WorkingSet* maintains 3 hash maps:

- A table mapping transaction IDs to a vector of commands
- A table mapping transaction IDs to locations
- A table mapping locations to the current state of an object

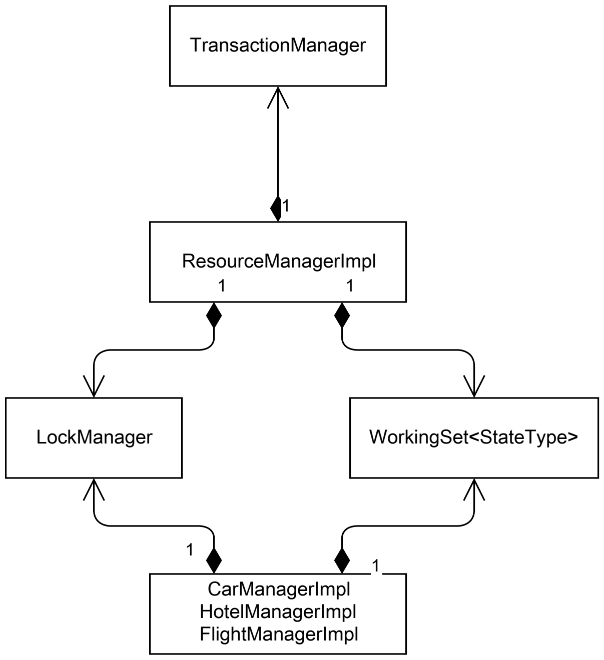
A symbolic representation of the commands sent by the clients is stored in the first table (no effects occur in the RM at this time). Once the client commits his transaction, the commands are executed sequentially, this modifying the state of the RM. When a transaction is aborted, the entries for the transaction ID are simply dropped from the tables.

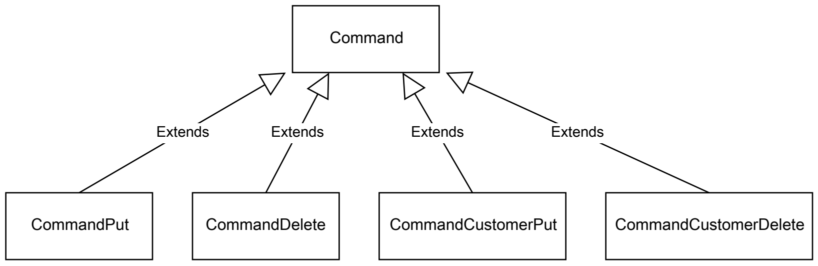
The two types of commands we store are:

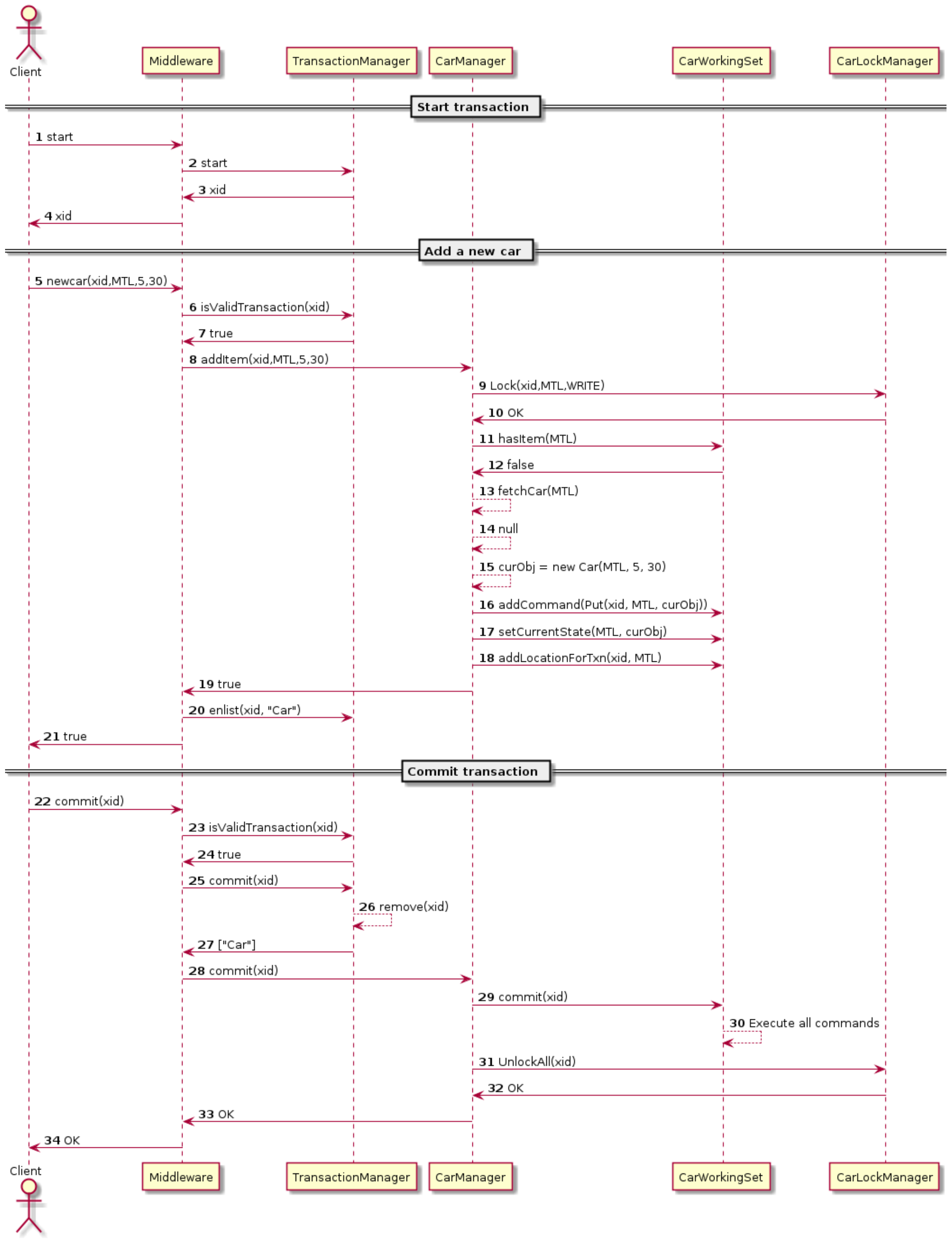
- *CommandPut* describing a write/modify action in a RM's hash table
- *CommandDelete* describing a delete action in a RM's hash table.

During the transaction, a copy of the object is kept in the working set and modified according to the different commands. This allows query operations to see the changed state of the data even though the updates haven't been committed.

2.4 Diagrams


TransactionManager
1
ResourceManagerImpl
1
1
LockManager
WorkingSet<StateType>
1
1
CarManagerImpl
HotelManagerImpl
FlightManagerImpl


Command
Extends
Extends
Extends
Extends
CommandPut
CommandDelete
CommandCustomerPut
CommandCustomerDelete


Client
Middleware
TransactionManager
CarManager
CarWorkingSet
CarLockManager
Start transaction
1 start
2 start
3 xid
4 xid
Add a new car
5 newcar(xid,MTL,5,30)
6 isValidTransaction(xid)
7 true
8 addItem(xid,MTL,5,30)
9 Lock(xid,MTL,WRITE)
10 OK
11 hasItem(MTL)
12 false
13 fetchCar(MTL)
14 null
15 curObj = new Car(MTL, 5, 30)
16 addCommand(Put(xid, MTL, curObj))
17 setCurrentState(MTL, curObj)
18 addLocationForTxn(xid, MTL)
19 true
20 enlist(xid, "Car")
21 true
Commit transaction
22 commit(xid)
23 isValidTransaction(xid)
24 true
25 commit(xid)
26 remove(xid)
27 ["Car"]
28 commit(xid)
29 commit(xid)
30 Execute all commands
31 UnlockAll(xid)
32 OK
33 OK
34 OK
Client
Middleware
TransactionManager
CarManager
CarWorkingSet
CarLockManager

3 Performance evaluation

3.1 Single client

We evaluated the marginal performance of the system by executing read, write and read/write operations on a single RM and on all three RMs. We also ended transactions with both commits and aborts. Because the times to execute a single command were so low, we executed 50 commands in total in each transaction. The average latencies of 1000 measures are given below in milliseconds.

	1 RM	3 RMs
Read + Commit	5.189131	6.955100
Read + Abort	5.434654	7.042538
Write + Commit	12.987475	10.064491
Write + Abort	12.985847	10.060301
Read/Write + Commit	7.466607	9.073547
Read/Write + Abort	7.465912	9.209547

A few surprising facts emerge from these figures:

- When only writes are concerned, accessing the 3 RMs is faster than accessing only one.
- Though they have more work to perform (i.e. executing the buffered commands), the commit transactions are usually a little faster than the abort transactions.

A fact that can only be observed by looking at the raw data is how the system gets faster as more commands are sent to the system; the first few transactions execute in 30ms before reaching numbers usually below 10ms. We attribute this to the Java JIT that warms up and starts optimizing the RM methods after they've been ran a sufficient number of times.

3.2 Multiple clients

For our performance evaluation, we used the following parameters:

- A fixed number of clients was selected (5, 10, 25, 50, 100, 200, 500, and 1000)
- A sleep interval (in milliseconds) between commands was selected (1000, 500, 250, 100, 50, 10)

For every combination of parameters, the following process was executed:

1. The servers were started with empty hash tables.
2. The RMs were populated with 1000 cities (for cars and hotels), 1000 flights and 1000 customers.
3. Random commands were executed and committed by the clients for 60 seconds.

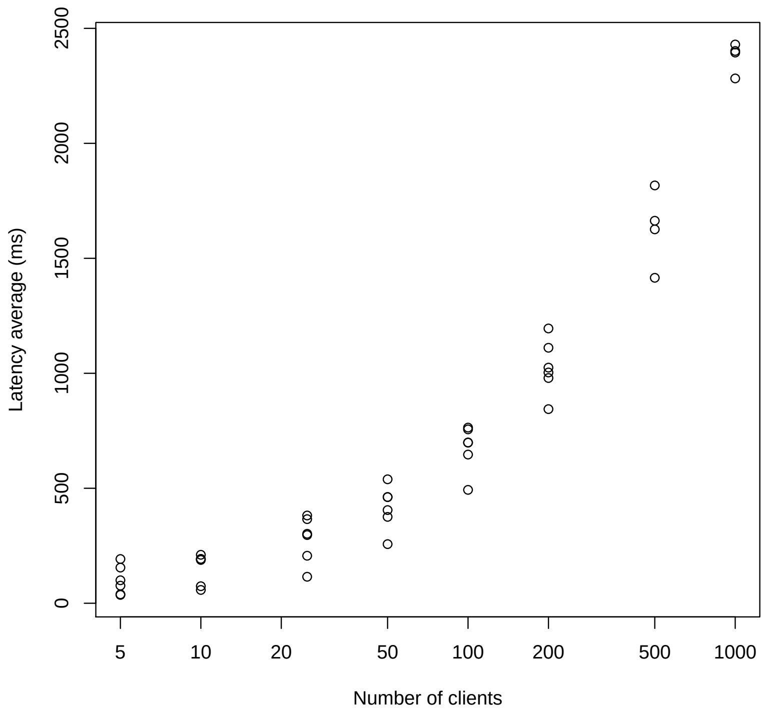
4. The latency times were recorded in a file.

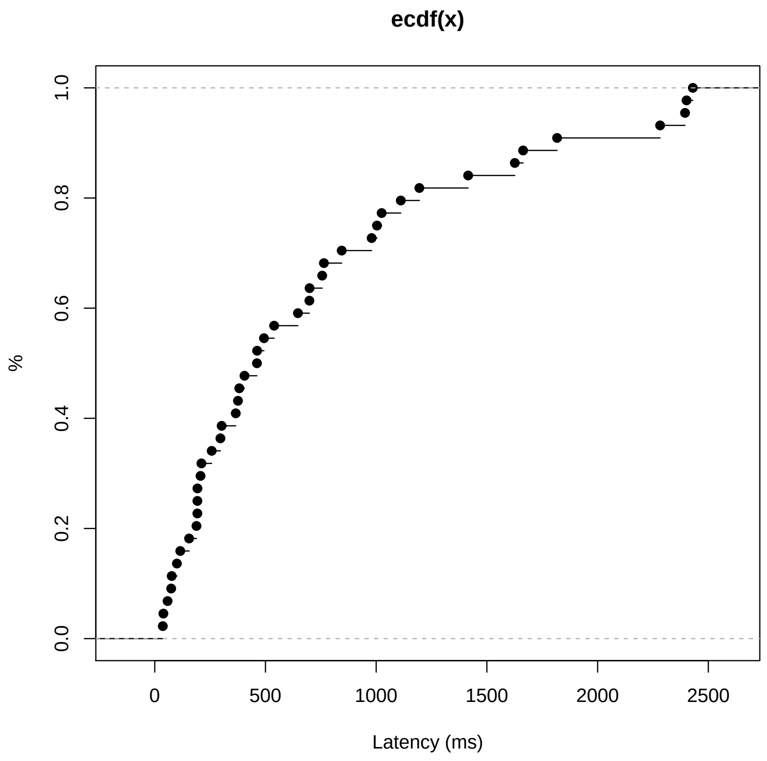
The tests were ran on a Intel i7-3770S 3.10GHz. To obtain times that reflected the performance of the application, the server, RMs and clients were all running on the same machine. We took the arithmetic average of the latencies and obtained the chart below. From this chart, we can make a few observations:

- The number of clients has a bigger impact on performance than the delay between commands
- The latency time does not grow linearly with the number of clients.

Also, not apparent in the chart, the number of deadlocks increased sharply as the number of clients increased, which is to be expected as more clients will randomly select the same resource to access.

Finally a chart showing the cumulative distribution function of the latency times is shown.




ecdf(x)
%
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500
Latency (ms)