COMP 512 - Project Phase 2 Report

Jérémie Poisson (260627104), Théo Szymkowiak (260619310)

November 13, 2017

1 Introduction

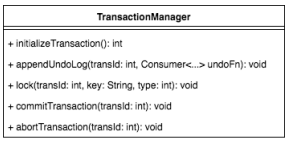
In this phase of the project, our main task was to implement the concept of transactions into our travel reservation system. Users can use a transaction to bundle a bunch of operations to be executed atomically. In the end, 3 operations were added to the middleware interface:

1. **start() : int** which allows the user to start a transaction and returns a unique transaction identifier.
2. **commit(transactionId : int) : void** which commits a transaction to the database given an identifier.
3. **abort(transactionId: int): void** which aborts a transaction being executed and rollback any changes made to the database during the transaction.

2 Resource Manager transaction implementation

Our approach was to first implement transaction to the resource managers. To do so, we first had to integrate a LockManager (provided to us) that provides a 2-phase-locking in order to lock the appropriate objects in the database. The LockManager provides 2 type of locks (read/shared lock and write/exclusive lock) and ensures the serializability of the schedules at the ResourceManager level —ie. concurrent accesses to data in the resource managers are re-arranged such that they are serializable. The LockManager is added to the transaction management module which is explained in further details below. This is because we have to lock database object on a per-transaction basis.

For transaction management, we added a *TransactionManager* class that keep track of opened and aborted transactions. Because we were tasked to implement a 1-phase commit system, this manager also contains a *undoLog*, which is simply a *Stack<Consumer<ResourceManagerDatabase>>* (stack of lambda functions taking as parameter the database instance to be called). Everytime an operation occurs in the transaction, we push a undo lambda function on top of the stack which, when applied in the future, will have the net effect of undoing the operation properly. This way, if we ever need to rollback the effect of a transaction due to an abortion, we simply have to pop every element in the stack and apply every lambda functions with the resource manager database instance.



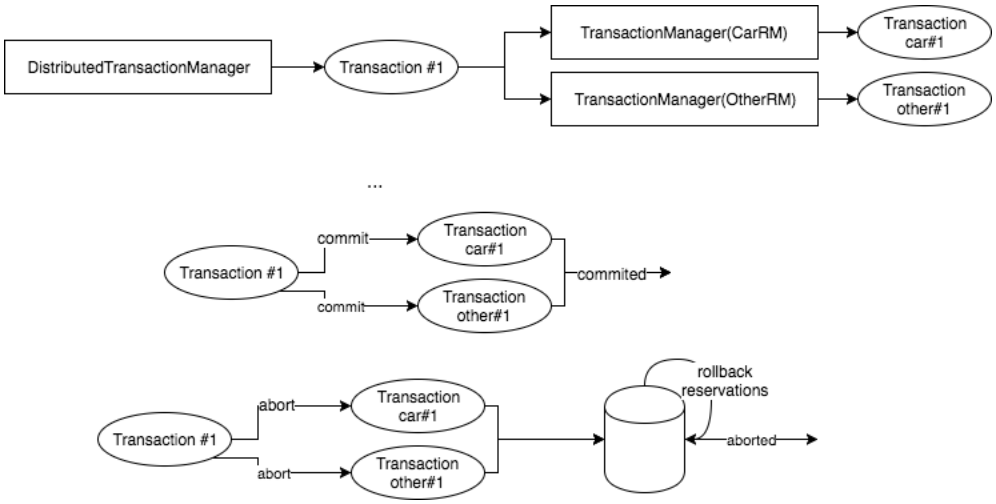
3 Distributed transaction implementation

Our second task was to implement a transaction management system at the middleware layer using the same distributed architecture of phase 1. Transactions were to be managed at the middleware layer using

the distributed 2PL protocol with a 1-phase commit.

We figured we could that we could simply re-use the centralized approach we developed in section 2 at each resource managers. As for coordinating the middleware transactions, we simply implemented a distributed transaction managers that keeps track of every enlisted resource managers for a given transactions. Every time we enlist a resource manager for the first time, we start a transaction unique to that resource manager and keep track of the identifier for subsequent operations. Whenever it is time for the active middleware transaction to be committed or aborted, we simply commit/abort the transactions at every enlisted resource managers before committing the one at the middleware level.

The only problem with this approach is to manage and rollback the customer reservations in the unlikely case of a transaction abortion as they are stored in the middleware (see part 1). For example, if a transaction is aborted while deleting a flight, we need a rollback mechanism to add back the user's reservations in the middleware database to cancel the net effect of the transaction. To solve this problem, we used a similar approach to the undo log discussed previously, but in the distributed transaction manager. Therefore, every distributed transaction comes with a *reservationUndoLogs* stack that contains lambda functions of undo function to rollback changes done to the reservation database. In the event we abort a distributed transaction, we simply pop every undo function and apply them to the middleware database instance, similar to what we did at the resource managers.



4 Testing

To test the system, we extended our test suites library we developed in phase 1. Because a lot of effort has been deployed to handle abortion transaction gracefully, most of the tests carried out in this phase were developed to test the atomicity of transactions and their effect on customers reservations. It is possible to view the test cases in the *test.rmi.** package. Our testing strategy is fairly simple: for each reservable items (cars, flights, rooms), we test the following scenarios:

1. Basic operations not involving customer reservations (creation, queries) on both committed and aborted transactions.
2. Reservation operations making sure the reservations are valid after both committed and aborted transactions (for example reservations are given back after an aborted transaction, etc.)
3. Deletion operations making sure the reservations are also valid after both committed and aborted transactions.

We also simulated artificial load using a single and multiple clients on the system (see section 5). Those also tested transaction support and concurrency of the system under large amounts of requests.

5 Performance Analysis

In order to perform performance analysis, we created two clients, one using a single thread and another using n-threads (to simulate multiple clients). For each type of clients, we put the system under different amount of stress and measured the response, failed requests and early aborts (a request that did not initiate due to the previous request being in flight).

Only interesting test runs are reported here, as to not overflow the report.

5.1 Single client testing

In this section, we will test our distributed system using a single client. Each request involves getting a write lock on a resource manager and querying objects from each resource manager.

Request parameters are randomized over a big enough set as to no produce deadlock or resource conflicts. It is worth noting that during those tests, all servers were running a single computer, removing the network latency parameter from the observations.

Also, two types of transactions were tested. One operating on a single resource hence putting some load on a single resource manager. The other is operating on multiple resource therefore putting some load on multiple resource managers. The number of operations in each type is equal to a comparable overall load.

The first test runs with 10 requests per second. From the following report, we can conclude that the distributed system handled the load well.

```
---- REPORT #0 ----
Requests per seconds: 10.0/s
Requests: 1200
Success: 1200 (100.0%)
Early abort: 0 (0.0%)
Errored requests: 0 (0.0%)
Average response time: 21.946667ms
Highest reponse time: 82ms
Lowest reponse time: 14ms
```

The second test involves 70 requests per seconds, the first time we started seeing some request overflow and early aborting. We can see a small percentage (1.88%) of early aborted requests (due to the previous request still being in flight). This essentially means that some request took too long to respond, hence the load is making the system increasingly unresponsive.

```
---- REPORT #0 ----
Requests per seconds: 70.0/s
Requests: 21000
Success: 20607 (98.13%)
Early abort: 393 (1.88%)
Errored requests: 0 (0.0%)
Average response time: 4.973762ms
Highest reponse time: 834ms
Lowest reponse time: 2ms
```

Finally, we tested with 80 requests per second and 90 requests per second. This higher stress clearly created a higher percentage of early aborts. Thus we can draw the conjecture that the number of early aborts is exponentially related to the number of requests per second.

```
---- REPORT #0 ----
Requests per seconds: 80.0/s
Requests: 24000
```

```
Success: 17404 (72.52%)
Early abort: 6596 (27.49%)
Errored requests: 0 (0.0%)
Average response time: 7.2929583ms
Highest reponse time: 1204ms
Lowest reponse time: 2ms
---- REPORT #0 ----
Requests per seconds: 90.0/s
Requests: 27000
Success: 23000 (85.19%)
Early abort: 4000 (14.82%)
Errored requests: 0 (0.0%)
Average response time: 5.274667ms
Highest reponse time: 46ms
Lowest reponse time: 2ms
```

This performance analysis also provides good insight on what is the bottleneck in our architecture. Clearly, communication cannot be a bottleneck issue as we are experiencing on a local network. Also, the transaction types operate over a big enough unique set of identifier so that locking at the database level cannot be an issue. The bottleneck appears to be at the middleware level and resource manager level.

5.2 Multiple client testing

In this section, we run multiple clients alongside each other while querying the server at a steady pre-defined rate.

The same requests described in section 5.1 are involved here.

The first test is conducted on 2 servers sending 2 requests per second. Which comes down to 0.5 requests per second for each server.

```
---- REPORT #0 ----
Requests per seconds: 0.5/s
Requests: 150
Success: 150 (100.0%)
Early abort: 0 (0.0%)
Errored requests: 0 (0.0%)
Average response time: 5.3933334ms
Highest reponse time: 13ms
Lowest reponse time: 3ms
---- REPORT #1 ----
Requests per seconds: 0.5/s
Requests: 150
Success: 150 (100.0%)
Early abort: 0 (0.0%)
Errored requests: 0 (0.0%)
Average response time: 5.693333ms
Highest reponse time: 17ms
Lowest reponse time: 3ms
```

As in the first test of section 5.1, all transactions went through without any problem.

The second test involved a higher request per second rate of 50 requests per second with 2 clients.

```
---- REPORT #0 ----
Requests per seconds: 25.0/s
Requests: 7500
```

```
Success: 7428 (99.04%)
Early abort: 72 (0.97%)
Errored requests: 0 (0.0%)
Average response time: 11.140133ms
Highest reponse time: 2967ms
Lowest reponse time: 2ms
---- REPORT #1 ----
Requests per seconds: 25.0/s
Requests: 7500
Success: 7438 (99.18%)
Early abort: 62 (0.83%)
Errored requests: 0 (0.0%)
Average response time: 11.080133ms
Highest reponse time: 2521ms
Lowest reponse time: 2ms
```

We can start noticing a few early aborts. The last test uses 150 requests per second on 3 clients.

```
    ---- REPORT #0 ----
Requests per seconds: 50.0/s
Requests: 15000
Success: 13247 (88.32%)
Early abort: 1753 (11.69%)
Errored requests: 0 (0.0%)
Average response time: 8.164ms
Highest reponse time: 10024ms
Lowest reponse time: 2ms
---- REPORT #1 ----
Requests per seconds: 50.0/s
Requests: 15000
Success: 13249 (88.33%)
Early abort: 1751 (11.68%)
Errored requests: 0 (0.0%)
Average response time: 8.155933ms
Highest reponse time: 10023ms
Lowest reponse time: 2ms
---- REPORT #2 ----
Requests per seconds: 50.0/s
Requests: 15000
Success: 13250 (88.34%)
Early abort: 1750 (11.67%)
Errored requests: 0 (0.0%)
Average response time: 8.1328ms
Highest reponse time: 10024ms
Lowest reponse time: 2ms
```

We can clearly see the higher percentage of early aborts ($\sim$11%) for each client. From further testing not included in this report, we can conclude that after 50-60 requests per second, the distributed system starts to hit a saturation point that degrades the service.

This also falls in line with what we observed in section 5.1: the more concurrent client we have, the less becomes is the request per second load possible per client. Therefore, it is now clear that the bottleneck lies at the middleware and resource manager layers. One way to increase the load our system may support would be to increase the number of resource manager servers in the back end or even split the load with multiple middleware servers.

6 Appendix

Our distributed architecture (from phase 1):