Comp512 Milestone 2 Report

Architecture and Design for Distributed Transactions

Group 3
Nicole Ho
Younes Boubekeur

Introduction:

In the second part of the travel reservation system, our implementation is done using TCP from part 1 since it contains more advanced functionalities, such as the ability to perform certain operations asynchronously, making it easier to write non-blocking code.

Locking strategy:

Due to how Milestone 1 was implemented, we found it simplest to implement lock managers at each site. We extended the existing logic in the supplied `lock()` method by adding lock conversion (upgrade privilege from read to write), using the static *`LockTable`* field. If there are no conflicts in any table, first we remove the read locks for the transaction and the data object. Then the new locks are added, one for the transaction and another for the data object. At each time the transaction identifier `xid` is specified. Because this logic runs in a `synchronized (LockTable)` codeblock, there is no risk of race conditions.

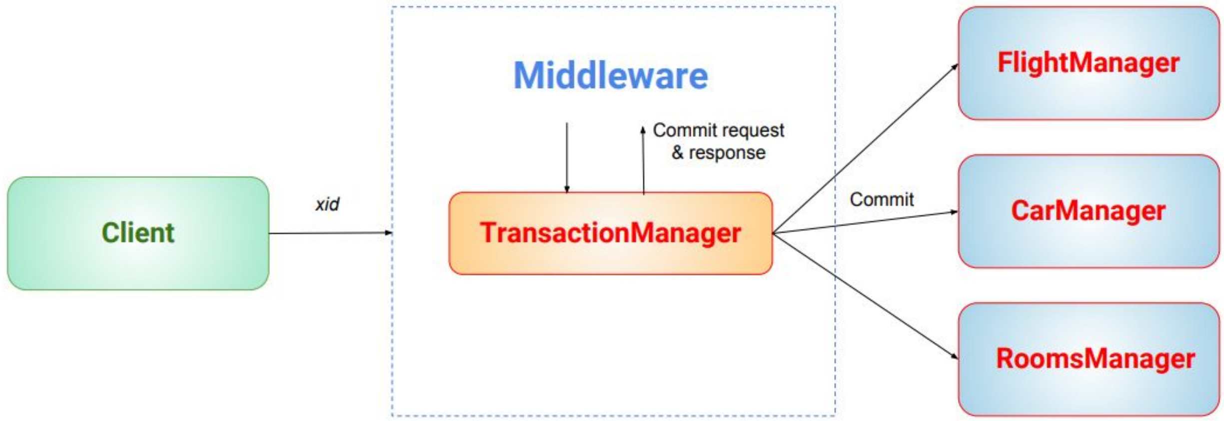
Transaction and TransactionManager architecture:

The `TransactionManager` contains fields mapping transaction ID's to individual `Transaction`s and `Timer`s, as well as a reference to the middleware. A `Transaction` has a list of `ResourceManager`s and a Status (active, committing, committed, aborting, aborted, timed out, invalid).

Transaction management:

Start: A new `Transaction` is created with the next available `xid`, which is returned to the caller (the Middleware).

Operations: The commit sequence can be described by the following diagram:



The client submits to the middleware a transaction request with an identifier *xid*. The middleware forwards this request asynchronously to the `TransactionManager` (using `CompletableFuture` as described in the first report). The `TransactionManager` runs its `commit()` method as described below, which leads to the transaction being committed at the `ResourceManagers`. If they successfully commit, they will return `true` to the middleware, which forwards this to the client.

Commit: The `commit()` method verifies that the transaction *xid* is in the active state, then sets its state to committing. It then iterates over the `ResourceManagers` and calls their `commit()` method with *xid* as a parameter, via the middleware. After that it sets *xid* to committed. If this process fails, an `InvalidTransactionException` is thrown.

Abort: Similar to `commit()`, the `abort()` method verifies that the transaction *xid* is in the active state, then sets its state to aborting. It then iterates over the `ResourceManagers` and attempts to tell them that *xid* should abort. After that it sets *xid* to aborted. If this process fails, an `InvalidTransactionException` is thrown.

Timeout handling:

As mentioned previously, each `Transaction` *xid* is associated with a `Timer` object, which upon timeout, tries to abort xid and set its status to timed out. The timeout period is set to be 30 seconds.