Comp512 Milestone 3 Report

2-Phase Commit

Group 3
Nicole Ho
Younes Boubekeur

Introduction:

In the final part of our distributed travel reservation system, we build on the previous milestones by fixing past bugs and implementing 2-Phase Commit with shadowing and full recovery at the middleware.

System design and architecture:

At a high level, the system as a whole can be described using the diagram from the Transaction management section (see below). Three servers play the role of resource managers (flights, cars, and rooms). A centralized middleware connects to these servers and accepts connections from clients, of which there can be multiple running simultaneously. Each of the aforementioned components can be run on a separate machine, leading to a truly distributed system.

The communication between the processes is done through TCP. This allows for the ability to perform certain operations asynchronously, making it easier to write non-blocking code. The data itself is transferred by message passing using `ObjectOutputStream` and `ObjectInputStream`. Both the middleware and the resource managers have a `onNewConnection(Socket)` method that gets invoked on incoming messages and responds accordingly.

Customers are replicated at the resource managers. This avoids the complexity of storing customer data on a separate server or at the middleware.

This system is largely based on transactions. Each action (such as adding or querying an item) is done within a context of a transaction. Each transaction must either commit or abort. If it commits, then its effect is persisted to storage, otherwise it has no effect. 2-Phase locking and 2-Phase commit are detailed in the following sections.

Implementation of individual features:

Build, deployment, and communication:

For convenience, a project Makefile which builds the entire project was created. It cleans the old build and compiles the java source files to their appropriate directories. The bash script that is used to run the components passes command line arguments to the Java Virtual Machine. The Java code binds each server with a port number. This allows for the server to accept remote connections and react to them. The client nodes connect to the servers by looking up the appropriate server name and port, which they obtain from command line arguments.

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.

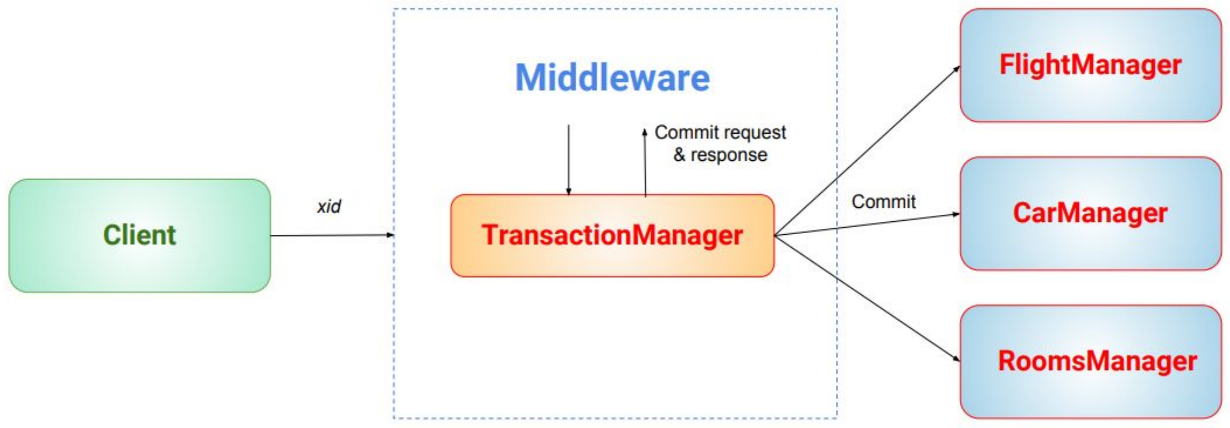
Transaction and TransactionManager architecture:

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.

Data shadowing:

The purpose of data shadowing is to allow persistence, which is useful for recovery from failures. Our implementation of shadowing is relatively straightforward. We created custom versions of `ShadowPage` and `MasterRecord`. `ShadowPage` is a parameterized type that can save data to a file periodically and load it back from a file. In practice, the data types stored are `RMHashMap`, corresponding to the program state, and `MasterRecord`. The files involved are a transactions file at the middleware level and three files at each resource manager: a before image, an after image, and a master record. After initialization, the files are written to as follows. The transactions file during the `prepare()` method: once at the beginning of the 2-Phase Commit protocol, and then at its end, at a commit or abort. The before and after images and master record are written to after writing the data to memory in the commit phase.

Logging and recovery (2-PC):

Logging for the Middleware and ResourceManagers is written to a single shared file with the properties `ManagerName, TransactionId, Message`. Logs are sent at every state change during the protocol. Recovery of the middleware is achieved by flushing all transactions to file each time the TransactionManager logs a state. Following a crash, the restarted TransactionManager restores transaction states and resumes it's transaction counter at the lowest consecutive ID number it does not have information about.

Special features:

We implemented many special features throughout the three milestones:

Implementing Concurrency using new Java 8 features
To simplify our codebase and make it more efficient, we used the following relatively recent Java features:

1. **Executors:** To manage a fixed size of concurrent threads automatically with minimal syntax.
2. **Lambda functions:** We used these throughout to express event handling code in a more expressive functional style. Concretely, this means we can specify an action (method) as a parameter to another method, so it can be run asynchronously. Other benefits of the Java lambda interface include better parallel performance and the ability to use `CompletableFuture`, detailed next.
3. **`CompletableFuture`:** This is the most powerful feature used in the first milestone, since it gives us a better API for programming reactively, without forcing us to write explicit synchronization code needed when using thread primitives. Put simply, a `CompletableFuture` is equivalent to a JavaScript `Promise`, a piece of code does something that could take time (such as contacting a remote server) and promises to return the result at a later time. In the meantime, the executing thread is free to do other things, such as handle other requests. Thanks to this asynchronous behavior, multiple requests can be serviced concurrently, thereby ensuring that the server is non-blocking. Another advantage of `CompletableFuture` is that they can be chained and composed together as illustrated by the following pseudocode:

```
var future = CompletableFuture.supplyAsync(firstAction)
                              .thenApplyAsync(transformation1)
                              .thenApplyAsync(transformation2)
                              .thenCompose(anotherCompletableFuture)
                              .completeExceptionally(exception); // handle errors
```

Advanced Performance Analysis:

For the second milestone, multiple extra dimensions of performance analysis were performed, notably:

- The logging was done in main memory at runtime, with the (I/O intensive) writing to disk happening only at the end, using shutdown hooks.
- Detailed data collection and presentation, through the use of 17 graphs that covered each node in the following scenarios: single client-single RM, single client-multiple RMs, and multiple client- multiple RMs.
- The calculation of the mean and stddev for every single graph, and explaining their significance.
- A brief network analysis, which concluded that the network is unlikely to be the bottleneck.

Problems encountered:

Some problems we encountered included the complexity of the build and deployment system. It took a long time to compile the code correctly on the Trottier machines, despite appearing to work in Eclipse. After some research, we discovered that Eclipse has its own intelligent build system that abstracts away a lot of complexity from the average programmer. For example, it can automagically detect the proper classpaths for packages in different folders so they can reference each other. On the Linux machines, classpaths and file paths need to be set manually in the build file at compile time and in the bash script at run time.

Another problem was encountered in milestone two, where the 2-phase locking needed to be corrected. Initially we would lock the entire RMHashMap whenever we needed to write data, which was a poor choice in the case of multiple clients. So we added granularity to the locks to allow for multiple concurrent transactions.

Testing for correctness:

Our system was tested for correctness by running a set of known commands that cover all the scenarios we could envisage.

If more time was available, it would have been worthwhile to automate the above process by reusing some of the automation capabilities added in the `PerformanceAnalyzer` class.