Justin Domingue – 260588454

Daniel Pham – 260526252

Harry Simmonds – 260502863

COMP – 512 Distributed Systems, Fall 2015

Project Part III: 2PC

**System Architecture, Feature Implementation, & Special Features:**

*Part I*

The design of the first part of the project (web service) follows very closely the design that we were given. We left the code given in the *rm* package in that same package. We then created the middleware so it would have most of the same methods. Hence, the client would call the middleware web service the same way it would the resource managers in the given code. In other words, the logic for Car/Flight/Room *add/delete/query* was put in the resource manager, the middleware only acting as a dispatching layer by selecting the right resource manager to send the request to.

*ReserveItinerary* is handled by the middleware. It simply contacts the individual resource managers to reserve the items. It will try to reserve has many items as possible without halting if one item is not available.

Finally, no extra concurrency handling step has been taken. The code we were given was already using 'synchronized' statements to block the *Hashtable* (which is also synchronized).

*Part 2*

For extending the project for programming assignment 2, we added a centralized Transaction Manager (TM) residing as an object the middleware. The TM exposes three main methods: *start, commit, abort* as described in the requirement document.

In addition to the centralized TM, each resource manager maintains a list of active transactions in order to provide an undoing functionality. It exposes methods *start, commit* and *abort*. *On start(id),* it saves transaction with id *id* as active. The behavior of *commit* and *abort* depends on the approach for making update transaction atomic:

**Update-in-place**. Any request is executed immediately by the resource manager. Any modifying operation in the resource manager will be saved in an *Operation* object and added to the list of operations of the relevant transaction. Then on *abort*, the resource manager goes over the list of operations from most recent to oldest and undoes each of them.

**Deferred-update**. Any request is buffered until commit time. To do so, the resource manager keeps a list of active transactions and operations associated with it. It also keeps a write set for each active transaction so that any *read* will read the most recent value. On commit transaction X, the RM goes over the list of buffered operations of transaction X and performs all the write operations from oldest to newest.

In order to ensure safety, we have created an instance of the *LockManager* within the middleware, and whenever a read or write operation is being carried out within the middleware *(ex: addFlight, deleteFlight, queryFlightPrice),* we will lock the data item for each transaction. This is then unlocked within the commit/abort methods.

Also, within the *middleware*, we moved the customer logic from a separate server to the middleware instance itself. The customer hash table exists within the *middleware*, and the logic creating / removing customers also exists within the *middleware*.

The *TransactionManger*, stored within the middleware is given the proxies of the *ResourceMangers*, so it can forward the one phase commits/aborts to the appropriate RM. We have an *enlist* method, that when an operation (request) is called, it will add the appropriate relationship between the transactions and the RM’s involved with each transaction.

**Time-to-Live(TTL).** In order to create this feature, we maintain a table of transactions with their corresponding expire time. Whenever we start a new transaction we set the time appropriately, and reset the timer every time a new operation comes in. On a set period, we traverse through the table of transactions and abort all whose expire time is less than the currentTime within the ‘clock’.

*Part 3*

**Two-Phase Commit:**

For this project, we used the deferred update for managing our transaction, where intentions are buffered into a write set before commit.

Upon commit, the transaction manager (TM) will collect all the locks for the relevant objects for the transaction. Then, the TM contacts all the affected resource manager to ask them to prepare and if they can commit. An RM can commit if objects buffered in its read set is equal to what can be found in the committed hash table. It then responds yes or no to the TM. Once the TM has collected information from all the RMs it can decide to commit or abort the transaction and sends that decision forward. The RMs then either abort or commit accordingly.

**Data Storage/Shadowing:**

All the data is stored using java serializations that are then saved into designated files.

Each RM has 2 files and a pointer file to determine which one of them is the current master. When saving to disk, we try to save to the file not pointed to and once every thing has executed correctly we update the pointer to point to the other file.

**Data Recovery:**

Within the constructor of the Middleware (MW), Transaction Manager (TM), and Resource Manager (RM), we check for an existing data file within the appropriate file path. If it doesn’t exist, we simply create new instances of the data, and if it does, we will call a function that will attempt to update the transaction statuses, and update any incomplete operations. Analyzing the values of the stored data will give us the *TransactionStatus (Committed/Unknown/Active)* and once we have retrieved that information, we can then go on to update any incomplete operations within the data base.

**Site Failures:**

In distributed systems, servers can crash at any time and so we have to take extra care: servers could never respond back, throw exceptions or not be reachable. A good failure simulation engine is thus required during development to insure proper behavior or the distributed system. In this project, we decided to request user input at strategic places. For example, before the middleware sends the vote request, the program will ask the user ‘About to send vote requests – should I crash?’. The user running the simulation can then choose to crash the server or not.

In two-phase commit, failures to particularly watch for are 1) in the middleware, a) before vote request, b) after vote request with or c) without having reached a decision and d) after sending the decision and 2) in the resource managers, a) before and b) after receiving the vote request and c) after executing the decision. On recover, in cases 1.a,b,d, the middleware aborts; in case 1.c, the middleware sends the decision again; in case 2.a, the RM aborts; in case2.b, the RM has to wait for an answer from the middleware (web service cannot contact client). In terms of dealing with site failures, we use two mechanisms: exceptions and timeouts. When the middleware waits for vote from the RM, it watches for TransactionAbortedExceptions and InvalidTransactionExceptions and uses a timeout. If an exception of the previous types is returned or if the timeout occurs, then the transaction is aborted. When the middleware sends the decision, it also uses a timeout to make sure that resource managers have received the decision. If the timeout occurs, then the middleware tries to send the decision again after waiting for some time. If a second timeout occurs, then it ignores it. The reason we have to resend the decision is because the RMs cannot contact the middleware directly, only trough the return statement of (or throwing an exception in) a call from middleware to the RMs. Finally, the resource managers could timeout while waiting for a vote request – then if the middleware sends a vote request to that RM, the RM will answer with InvalidTransactionException and the middleware will effectively abort the transaction. If the RM crashes while waiting for decision, then when it will recover, it will wait for the middleware to tell it what the decision was.

**System Test:**

We tested the system mostly manually (by running it) except for the *LockManager* which add some tests that we adapted for our purposes. To test the system, we launched three resource managers, a middleware and a few clients. We knew from part I that except for the transaction system, everything was working. Hence, we focused on testing the transaction functionality. We tested various scenarios, ranging from very simple (e.g. a single transaction wants to read, then convert lock to write, then *commit*/*abort*) to more complex (e.g. three transactions read the same object, one tries to overwrite it (gets a *write* lock), then the others try to write as well but they either abort because after the deadlock timeout or the transaction holding the *write* lock commits/aborts and one of the other gets the lock and is able to commit afterwards.

**Performance Analysis:**

Measurement techniques

We defined a transaction consisting of three operations on each resource manager (plus start/abort). We computed the time before starting the transaction and after finishing the transaction and averaged that over 1000 iterations.

**Results**

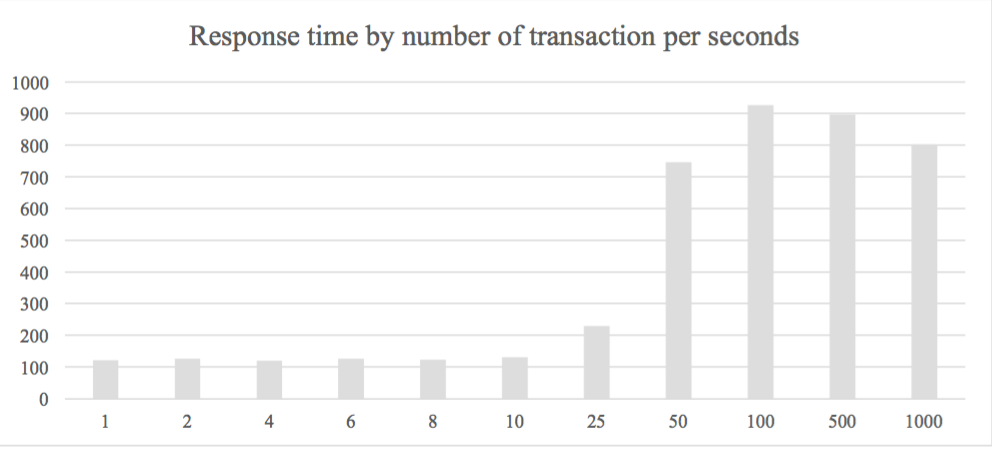
**a) Response time when there is a single client in the system**

Transaction type that involves only one resource manager: 17ms/transaction

Transaction type that accesses all three resources managers: 18ms/transaction

Accessing multiple resource managers doesn’t seem to increase the time by a large margin. To get this data, we defined a transaction type containing three operations and executed it 1000 times. We computed the total time and took the average. According to us, most of the time spent is distributed amongst the middleware and the servers – the reason being that even though the middleware acquires locks, locks are always granted immediately.

**b) Response time when there are many clients in the system**

****

First, observer that as the transaction per second increases, the response time increases as well. It seems that for values of transaction per second between and 1 and 10, the difference in response time is not significant and can probably be attributed to noise. The saturation point of the system seems to be after 50tps where the response time remains at around 800ms/transaction. Throughout the experiment, CPU usage remained fairly low, averaging 7%. Concurrent control certainly plays a role in the increase of the response time. In fact, our transaction types consisted of only three operations, meaning that locks were frequently requested. Hence, many requests had to wait on a lock and possibly fall victim of a deadlock. Hence, most of time seems to be spent in the middleware.

**Problems Encountered:**

Initially, with the basic implementation created, we had problems with the lock manager. Using what seemed to be the simplest and most logical implementation, it was still causing errors. We had to fully read and understand the entire lock manager and associated classes and edit the code appropriately to ensure it was working completely**.**