Comp512 Project 3 Report

Benjamin Sprott and Yasaman Sedaghat

December 7, 2010

Abstract

The project concerns a distributed travel reservation system. Herein we describe the system in detail but highlight two new features namely Two-Phase Commit and Recovery.

Introduction

The project concerns a distributed travel reservation system. The system has a basic client, Middleware and server architecture. These connect via Java RMI. The system already consists of Middleware, client, server, transaction manager and lock manager architecture. Further to this, we have completed the implementation of both the Two-Phase Commit protocol(2PC) and Recovery at the Resource Managers (RM). The system functions on the *transaction* paradigm in which requests from a client are grouped together logically, submitted together in some sequence, and then either committed or aborted.

Commitment is a process whereby write requests are stored in permanent memory at a server. 2PC ensures that when a transaction is ready for commitment, but not all servers can perform the commit, the transaction itself must be completely canceled. Thus, a voting phase is implemented to gather from each server the confirmation that they can commit the transaction. This ensures the atomicity of 2PC. If a transaction is aborted, the state of any server or database is rolled back to original values. 2PC is accomplished through the implementation of a few stages. From figure 4 we see the details of 2PC.

Recovery is a method for handling server crashes during the normal processing of a transaction. For instance, suppose that during the execution of a transaction, a server which hosts the crucial data undergoes a fatal crash. In this case, if the transaction was meant to be committed, and all other servers involved were ready to commit the atomicity property would not be respected. Thus, data is stored in permanent storage using a method called shadowing. In figure 3, we can see where the status is saved at each phase. This allows a resource manager to know for each transaction what

stage of the two phase commit protocol it is in. If a crash occurs, then a server may be restarted and the status of each transaction, along with the crucial data, can be read. A server can remember to send a vote, or that it was waiting for votes or that it was waiting for a vote request.

2 System Description

The travel reservation system is meant to mimic the reservation of flights, hotel rooms and rental cars in a manner similar to Expedia. We imagine that separate companies run the flight, hotel and car rental servers and thus the servers are tested each on their own machine, truly exposing the distributed nature of the system. From the client console, a user can create cars, hotel rooms and flights and further reserve these objects. These objects are reserved by customer objects. The cars, flights and hotel rooms reside on separate servers while the customer objects reside on the Middleware server. Below are the detailed descriptions of the various parts that went into making a fully distributed transactional system with high concurrency.

2.1 System Architecture

The basic architecture is that of client, Middleware and servers and we can see the general layout in figure 1. The system consists of any number of clients running the client process on their local machine, along with four other processes distributed on various remote machines. The Middleware layer is a client/server process, while the three resource managers (one each for Rooms, Flights and Cars) are more or less server processes. The system is built on RMI. The basic functioning of RMI is that a client holds a proxy for a server and the proxy acts like a local object with various methods which are specified by an interface. When the client requests a method of the proxy object, the information is packed and sent to the server where the method is invoked remotely. For our system, there is a basic three link chain as in figure 2. This kind of connectivity presents certain puzzles when attempting to implement each of the subsystems.

2.2 Transactions and the Transaction Manager

Transactions are the logical grouping of actions, where actions are understood as read or write. Transactions are handled in an atomic way, in that if any one of the writes are successfully committed to memory, then all the writes at every other server involved in the transaction are also committed successfully to memory. If any transaction is forced not to commit the changes, i.e. it aborts the transaction, then each other system must also abort the transaction. These groupings are handled by a transaction manager. The transaction manager is implemented as an object in Middleware

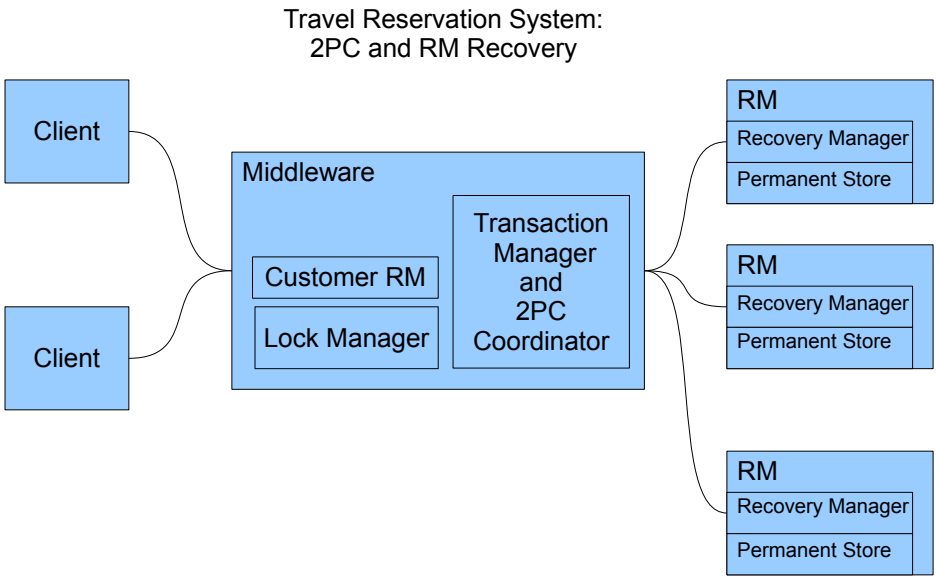

Figure 1: System Architecture

as seen in figure 1. The job of the transaction manager is to create and handle transactions with the servers. The Transaction manager has several important methods:

1. start()
2. enlist)
3. commit()
4. abort()

The first method is Start(), which creates a transaction by producing a unique id (an increment from the previous transaction id), and then loads this transaction into a hash table. The second important method is enlist() which is used when a read or write request is sent form the client. When a read or write method is called remotely from the client, the Middleware

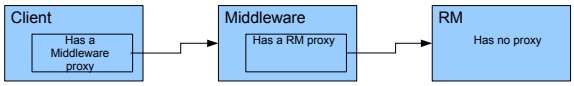

Figure 2: The chain of proxies

must first notify the transaction manager so that it may update a table which records which transaction is using which resource manager. It does this by storing a boolean array in the hash table at the key corresponding to the transaction id. The boolean array has three entries, one for each of Flight, Room and car. A true value in the Flight slot indicates that this transaction enlists the use of the Flight resource manager. Commit() commits the transaction by contacting each of the resource managers enlisted for the transaction and requests that they commit the given transaction. Abort() is similar except that the transaction manager requests that each enlisted resource manager abort the given transaction.

2.3 Concurrency and the Lock Manager

From the perspective of the Middleware or a Server, concurrency is a central theme in the basic operation of that system. Concurrency refers to the ability of a computer system to process multiple blocks of code, also called threads, a the same time. While these actions may be performed by the same processor by an interleaving of the operations, it seems from the users point of view to be done by different machines. High concurrency allows many clients of a server to make requests at the same time.

To have high levels of concurrency, a system has to properly control access to data. This is achieved through the use of a Lock Manager. The lock manager is logically related to the Middleware. That is, it is the Middleware that consults the lock manager and the transaction manager to organize requests into transactions and get locks. As in figure 1 the Middleware has a local object of type LockManager and uses it to organize access to the base data which lives on the resource managers. Any time a remote method is invoked at the client, the Middleware contacts the lock manager to get a lock. This is done in a fairly straightforward fashion, in that the call to the lock manager includes a transaction identifier and a string which includes

the a resource manager identifier and a location identifier. Thus strings such as "Flight1", "Car2", "Room3" are passed to the Lock method which then acquires a lock on the particular server at the particular location. Thus, when a request is made of the Flight server for a particular flight number, the hash table for that flight number at that server is locked and the lock is given to the requesting transaction. When the transaction is committed, via a call from the client, the lock on that server is released. Redundant lock requests cause exceptions to be thrown and when a read lock is held by transaction T and transaction T then requests a write lock, the lock is then upgraded.

Read locks can be upgraded to write locks as long as only one transaction holds the read lock. This is done simply by upgrading the lock in the lock table.

In the case of a deadlock, an exception is thrown by the lock manager which is then passed to the Middleware. At the Middleware, the transaction is aborted and a TransactionAbortedException is thrown which is passed back to the client.

2.4 The Two-Phase Commit Protocol

Two-Phase commit is implemented at the Middleware and the Resource managers. The 2PC is atomic in that any aborted transaction must have all of it its operations undone. This was accomplished through the use of four separate hash tables. There is one table for each of Car, Hotel, Flight and Customer. In the event that a write operation is requested on any of these types of data, the permanent memory is checked to see if there are already rooms, flights, cars or customers. If there are, this data is loaded into a temporary hash table and then updated to reflect the write request. In detail, this is accomplished by using a new writeData method which receives a string describing which data type is being written and uses it to update only the table responsible for that data type. The table is a hash table and contains hash tables which are keyed on the location, and flight number appropriately.

A new readData method is also created that servers two functions. First, it uses a string to see which of the different tables of temporary data to check. If the sought after data item is not found in the temporary data, the permanent storage is checked and in either case the data is returned. Second, any time data is read, it is also cached in the temporary hash tables and a reference to the local data is passed back.

At commit time, all the tables are scoured for any data. All the data is then written to the permanent storage and the tables are emptied. At abort time, the tables are simply emptied.

To ensure further atomicity, a voting phase is implemented in which the coordinator (in this case the Transaction Manager) constructs three separate

threads, one for each of the resource managers. In each thread, the vote request method called *prepare* is called. At the RMs, this method induces them to flip a coin weighted to vote to commit with probability 80 %. If the vote is no, then the RM returns false and aborts locally. If the vote is yes, then the RM will return true. During this time, the Transaction Manager is waiting to receive all the votes. It can time out waiting for votes if it waits longer that 30 seconds. If it times out, it will abort the transaction. If it does receive votes from all the participants the Transaction Manager will simply gather the result of the votes and if no on has voted to abort it will send the command to commit the transaction. Otherwise it will send the command to abort.

The RM will wait for both the vote request and also the result of the voting phase. In either case, if an RM has waited too long for either a vote request or the vote result, it will time out and the timeout time is 35 seconds. It will then abort the transaction. If the Transaction Manager then does a vote request, it will receive a vote to abort.

2.5 Recovery

In the event of a crashed server, it is imperative that a reboot is administered and the data and status of all transactions is correctly remembered. For instance, if during the execution of a transaction, a server crashes just before voting to commit. If that server is able to commit, it is important that it is able to restart and send its vote to commit. To handle this we implement data shadowing. In data shadowing we keep temporary and committed data in permanent storage. This data consists of hash tables of data objects like cars or flights. Since hash tables implement serializable, the crucial data is stored using serialization and object output streams. As noted in the section on 2PC, there is a temporary table and a committed table.

Three total files are stored, fileA, fileB and a master record. The master record is also a class called RecoveryManager which implements serializable. The record keeps track of which of fileA or fileB is storing temporary data and which is storing committed data. It also keeps a hash table of what we called Wait4VoteRequestElements and these store the status of each transaction as well as a timeout counter for each transaction allowing the transactions to timeout and abort while waiting for vote requests or vote results. When an RM commits, it stores both the temporary and committed tables and also changes the record indicating which of fileA or fileB is the committed versus the temporary data. Also, any time a write operation is performed at an RM, the temporary data is stored in permanent storage.

The status of a transaction is updated at three points during its lifetime. We can see in figure 3 just where these status updates are made. A transaction starts its life in the PREPARED status, waiting for new read/writes or a vote request. If at any time the transaction commits or aborts, the

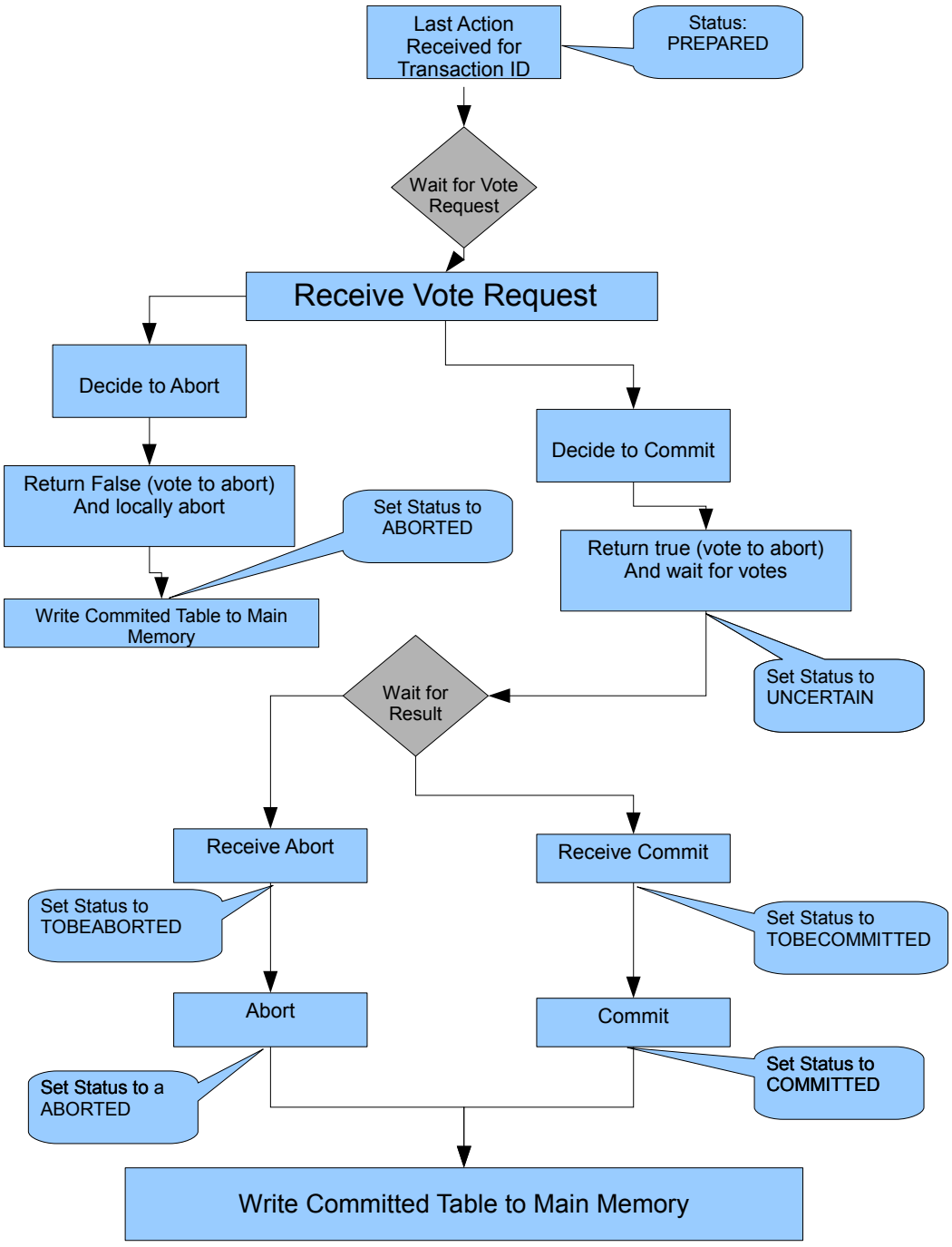

Figure 3: Demonstrating the flow for a participant in 2PC

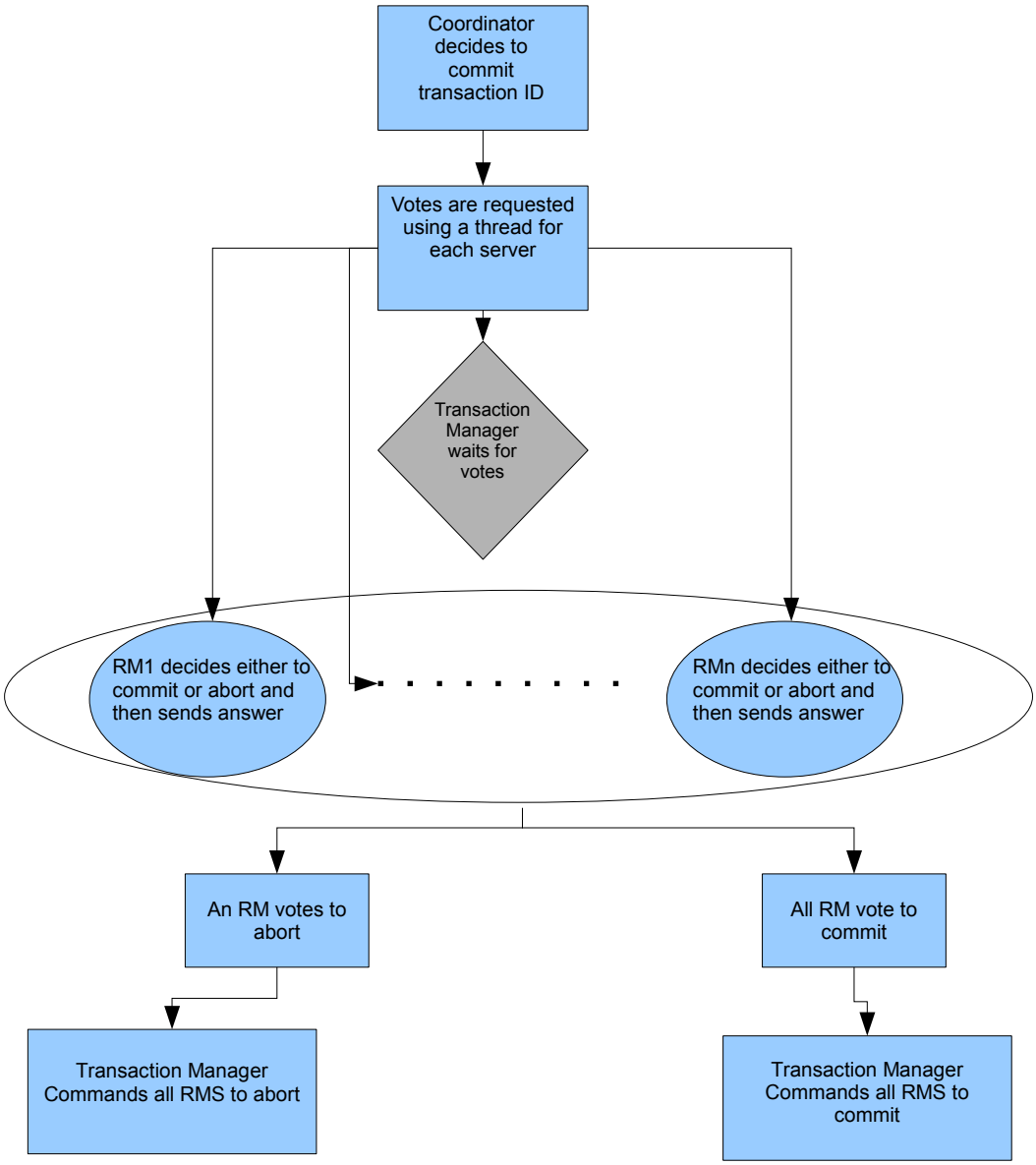

Figure 4: Outline of 2PC for a coordinator

status for that transaction is set to COMMIT or ABORT. If, after the RM receives a vote request, the RM sends a vote to commit, it will wait with its status set to UNCERTAIN. Each time a transaction changes its status, the master record is saved with this updated status.

The reason all of this data logging takes place is that when an RM starts, it can check permanent storage for the presence of stored data and the master record. In our implementation, when the resource manager is instantiated, it does a recovery phase. In this phase, the recovery manager is reconstituted from permanent storage using a ObjectInputStream. Also, both the temporary hash tables and the committed hasH table are likewise restored to the state before crashing. Furthermore, if it is discovered that any transactions are in the TOBECOMMITTED status, the RM knows that it had received the commit for this transaction but never actually committed. It therefore goes ahead and commits this transaction. Likewise, if it finds the TOBEABORTED status, it will abort this transaction.

3 Testing For Correctness

In previous deliveries, we tested the basic message passing through RMI, the Lock Manager and Transaction Manager. Here we show the correctness of 2PC and Recovery.

3.0.1 Correctness of 2PC

At first it is assumed that there is no failure for any component of the system. So we start with one transaction. The client tries to create data in each of the three resource managers and then ask to commit the transaction. The first step is for the middleware to decide if it wants to commit or not, and it is simulated by using a random variable. If middleware decides to commit the transaction, it will send the vote requests to all of the resource managers and will wait for their answers. The same idea of flipping a coin to decide whether to commit or not has been used in the resource managers. At the end, if all of them vote YES, the middleware would send commit to the resource managers and the transaction can commit successfully. This process was tested by starting another transaction who would query the same data right after committing. However, during the voting if any of the resource managers vote NO, the whole transaction would be aborted and no results of the created data can been seen after that.

The following is one of the actual scenario that was tested on the system:

Test the 2PC with one client, three resource managers

client1:
start

newcar,1,1,1,1
newflight,1,1,1,1
newroom,1,1,1,1
commit/abort,0

client2:
querycar,1,1
queryflight,1,1
queryroom,1,1

One can repeat the same scenario a few times, to see different results of using 2PC protocol.

3.0.2 Correctness of Recovery

In this project, the recovery has been implemented only for the resource managers. So in order to test that, we need to force the resource manager (or middle-ware) to crash at every possible point during the voting, or even after committing/aborting and then restarting that resource manager to get the results.

We considered a set of possible crashes (each identified with a unique number) and in our implementation it is the client who decides which crash should happen. Simply whenever the client wants to commit or abort a transaction, it would also enter a number that shows the type of the crash. At every possible point of crash in the middle ware or resource manager, this number would be checked, therefore at the right place the system would call the *SelfDestruct* method in order to shut down itself.

Since we have stored the current status of all the transactions within the RecoveryManager in the resource managers, whenever the system restarts, the statuses are known, hence the resource manager can perform accordingly.

The following are two of the actual scenarios that were tested on the system to shows how the different crashes are handled:

Testing recovery of the RM with two clients
and also crash #8 : crash after receive vote request but before sending answer

client1:
start
newcar,2,2,2,2

client2:
start

newcar,3,3,3,3
commit,8

restart the carRM → it will recover and remembers the changes that client1 did.

client1:
commit,0 → it should be able to commit

client3:
start
querycar,2,2 → will see the results
querycar,3,3 → will see the results

Testing crashes of the RM with two clients
crash #10 : crash after receiving decision but before committing/aborting
crash # 9 : crash after sending answer but before receving the decision

client1:
start
newcar,11,11,11,11
newflight,11,11,11,11

client2:
start
newflight,22,22,22,22
newcar,22,22,22,22

client1:
commit,10 or 9
restart the carRM → it will recover and remembers the changes that client1 and 2 did
restart the flightRM → it will recover and remembers the changes that client1 and 2 did

client2:
commit,0

client3:
start
querycar,11,11 → will see the results
queryflight,22,22 → will see the results