COMP512 - Distributed Systems
Project Final Report

Zhaoqi Xu <zhaoqi.xu@mail.mcgill.ca>
Zhiguo Zhang <zhiguo.zhang@mail.mcgill.ca>

November 2017

Contents

1 General design and architecture

1.a RMI-based system architecture and middleware design

Figure 1.1 shows the RMI-based system architecture.

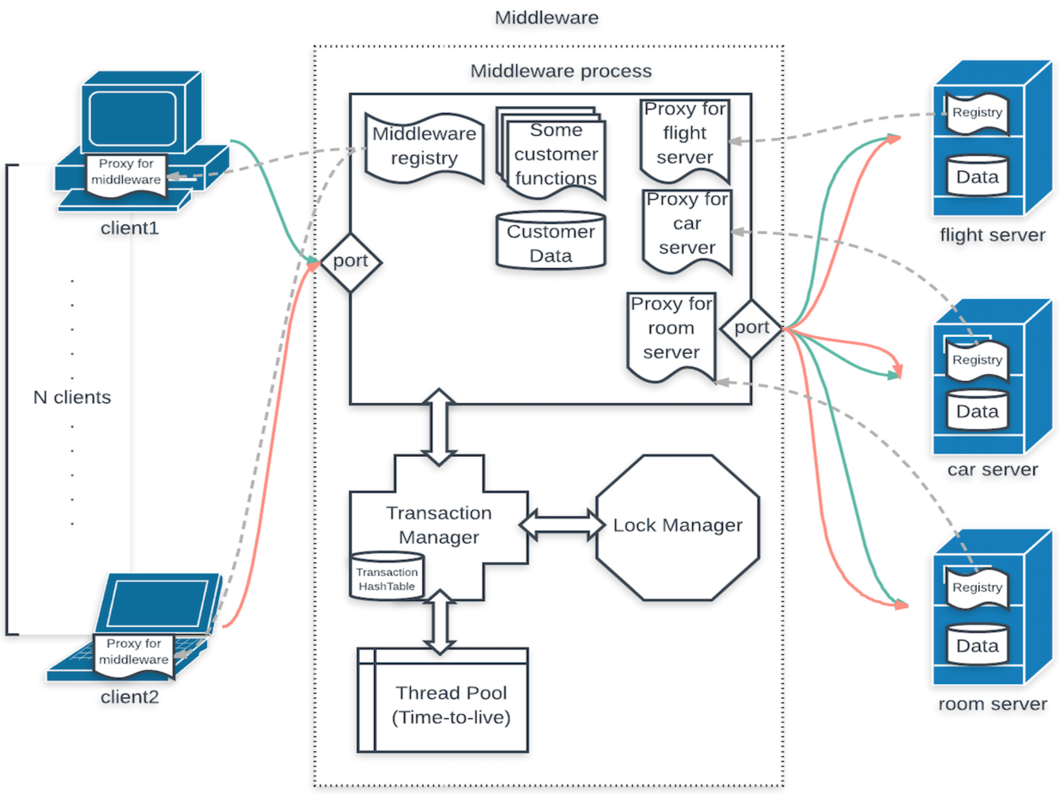

Figure 1.1 RMI-based Travel Reservation System

Flight, car and room data are stored separately on three *Resource Manager* servers while the customer data is stored on the middleware. Besides, each RM server holds a registry correspond to its service. The registry of customer class is keeped on the middleware. The middleware knows the server name and port number of each Resource Manager. When the middleware is launched, it automatically looks up the registry on each Resource Manager and creates a proxy for all remote objects on that RM. Middleware also creates a registry for customer remote objects. The clients know the server name and port number of middleware. When a client is launched, it will do the same thing as middleware does and create a proxy for all the remote objects on middleware. In this way, we get the clients and servers communicate through a global middleware. The Transaction Manager, Lock Manager and Thread Pool were designed for concurrency control which we talk in *Components and features implementation* section. Figure 1.2 shows the working flow of a complete transaction in our distributed system. We took command *'newroom'* as an example.

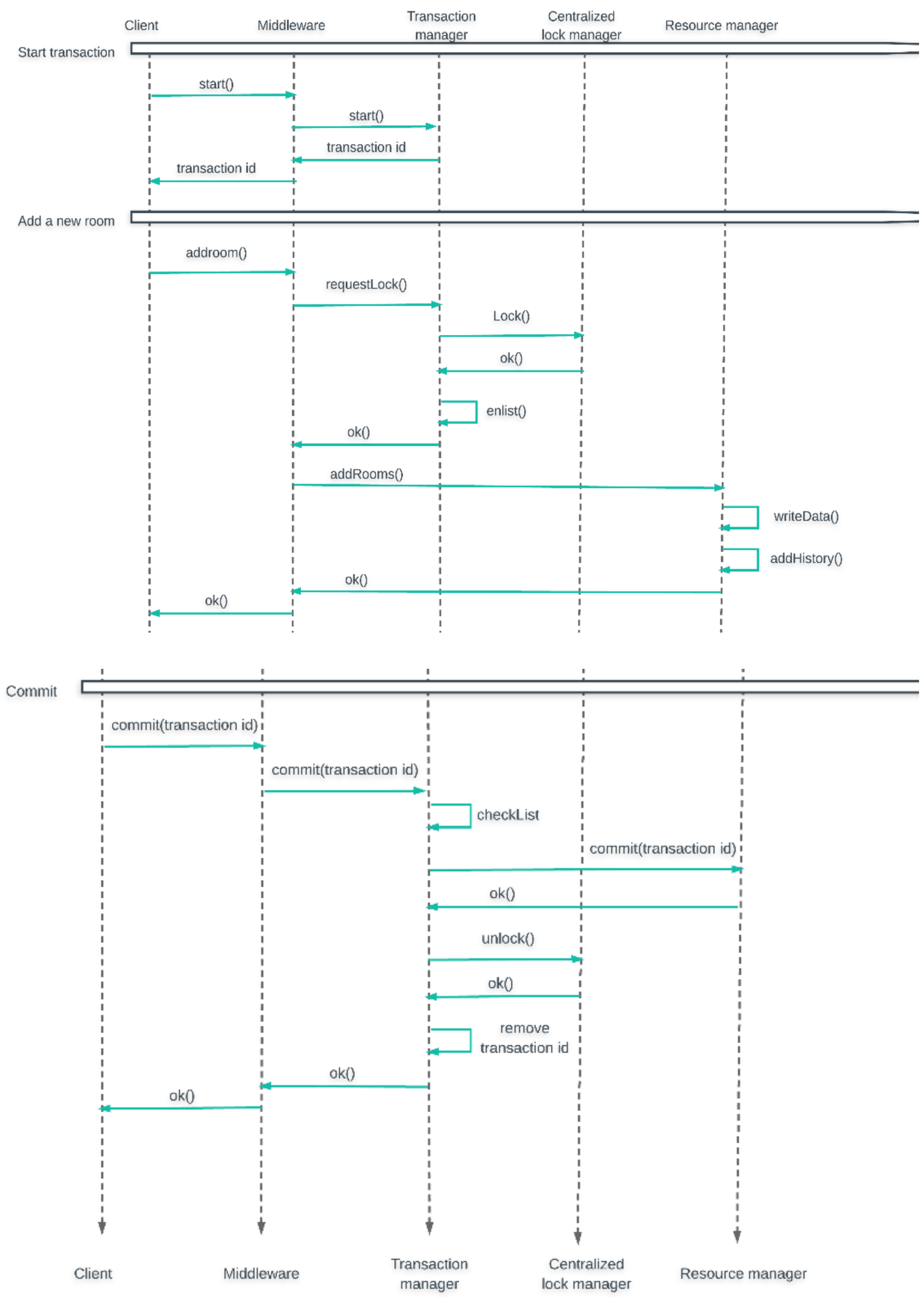

Figure 1.2 Add a new room example

1.b TCP-based system architecture and middleware design

Figure 1.3 shows the TCP-based system architecture.

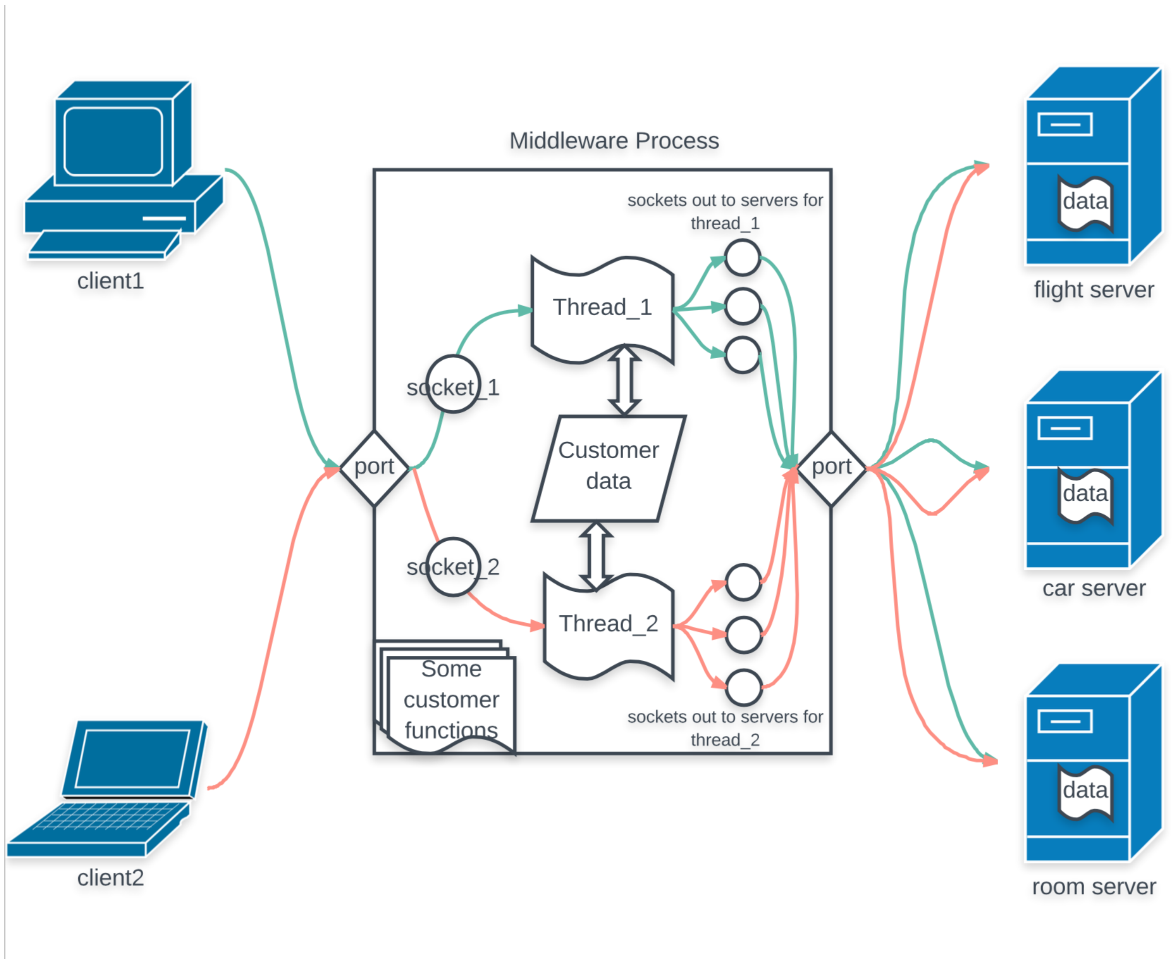

Figure 1.3. TCP-based Travel Reservation System

The basic structure of TCP-based system is very similar to the RMI-based version. The flight, car and room is handled separately on three *Resource Manager* while the customer resides on the middleware. The servers and middleware are multi-threaded. The server is listening requests on a port. When a request arrives, the server creates a new socket and passes it as a parameter to the new created thread. This way, a connection between server and client is established in the thread and the newly created socket gives the server destination information. The implementation of middleware is more complicated. It not only has to handle the request from client as the server does, but also need to create three sockets for the Resource Manager respectively in a thread (we assumed that the middleware knows the server name and port when it is launched). Therefore, each middleware thread has one server socket connected to client and three sockets connected to RMs. Once we have all the sockets created, we can bind data streams on the sockets and send query through the data streams to achieve communication between clients and servers. We did not implement concurrency control for this system because the phase 2 and 3 was based only on the RMI-based system.

1.c Customer design

We implemented customer functions and stored customer data on middleware server in both systems. The main reason we designed the system this way is that we can reduce data transfer frequency and improve system performance compared to handling the customer by an additional server. When doing operations corresponding to customers like *reserveflight*, *reservecar*, *reserveroom* and *deletecustomer*, we need to query data from multiple resource managers, centralize these data on middleware and then forward the processed data back to the resource managers. In our system, we only need to store the processed customer data locally on the middleware. If the customer is handled by an additional server, we will have to forward it through WAN. The data transfer speed through hard drives is much faster than the speed through WAN, thus we can make sure that the system performs better under this architecture.

2 Data design

2.a Log file

We implemented a serializable class called *LogFile* for convenience of saving log to and loading log from disk. The structure of *LogFile* is very simple. It contains a transaction id which is used to notify which transaction this log belongs to and a ArrayList of string that contains the necessary records.

2.b Transaction

Transaction is also implemented as a class. Each transaction contains a transaction id, a operation counter which tracks the number of operations that occurs in this transaction, a stack of vectors of string used to save the transaction history. It also has three functions *addHistory(), addSubHistory(), pop()* that are responsible for managing the transaction history.

2.c Master record

For convenience of transferring data between memory and disk, master record of shadowing is also maintained in a serializable class called *MasterRecord*. A flag indicating the active copy and the transaction last committed are maintained in this class.

3 Components and features implementation

3.a Centralized lock manager

In our project, we decided to use a singleton lock manager that has all lock information including customer, cars, flights, and rooms. The reason to use a single lock manager is that whenever the middleware needs to request a lock, it only needs to request it locally. This reduces unnecessary hops between middleware and resource managers. The details of implementation of lock manager will be discussed in three parts.

1. *Lock conversion*
   The first task in phase 2 is to add lock conversion to the given lock manager implementations. The two functions modified are *Lock()* and *LockConflict().* In *Lock* function, if *bConvert* is set to 0, then we remove the existing *READ* lock and add new *WRITE* lock on the data item. In *LockConflict* function, in the case that the transaction already has a lock and is requesting a *WRITE* lock, there are two cases to analyze i.e. the type of lock already holding. If the transaction already had a *READ* lock, we need to check if there are other transaction holding any *READ* lock on the item(No other transactions can have *WRITE* lock). If some other transactions also had *READ* lock, *LockConflict* returns *true*. Otherwise, *bConvert* is set to 0 and function returns *false*. If the transaction already has a WRITE lock, new *WRITE* lock request is redundant and the functions throws *RedundantLockRequestException.* Then, lock function catches the redundant lock exception and returns true, i.e. ignores the new lock request.

2. *Requesting lock from transaction manager*
   Whenever a client sends request on any data items, the middleware invokes *requestLock()* to transaction manager. Then, transaction manager calls *Lock()* to Lock manager. Depending on whether there is a lock conflict, the lock manager will return true for lock can be granted, return false if some arguments are invalid, or wait for lock conflicts to be solved(with possibility of deadlock existence). The *requestLock()* function also increments the operation counter of the given transaction ID. This helps to reset the time-to-live timer, which will be discussed in 3.c.

3. *Deadlock*
   The lock manager handles deadlock through timeout mechanism. That is, whenever there is a lock conflict, the lock manager wait until the lock is released or the timeout is reached. In the meanwhile, the client is blocked. If the transaction has been waiting for a period longer than timeout period, the lock manager throws deadlock exception. The transaction manager catches the deadlock exception and aborts the transaction which caused the deadlock exception.

3.b Transaction manager

The transaction manager is a singleton object inside the middleware server, it maintains an active transaction hash table mapping transaction ids to transaction objects. This allows the middleware to know which transaction that a operation belongs to. Transaction class consists of the transaction identifier, operation counters, and a stack of transaction histories. The operation counter is used for time-to-live mechanism. The stack is used for the recovery process if the transaction is aborted. The reason to use stack is that the last operation is recovered first. Transaction manager also enables extra features for middleware including *start(), commit(), abort()* and *shutdown()*.

The start function increments the transaction counter and returns the current transaction counter as a new transaction id. Moreover, the start function also creates a transaction object that is stored in hash table and starts a TimeThread for time-to-live mechanism. Since we are using strict 2PL, the commit function is implemented as a one-phase commit.

The commit function checks the validity of transaction id. If the id is valid, the transaction manager sends commit request to related Resource managers. The Resource managers execute the commit function locally and update transaction hash table. Afterwards, in transaction manager, the commit function unlocks all the locks that transaction holds, and removes it from active transaction hash table.

For the abort function, middleware and resource managers handle the record recovery using the local transaction history stack and transaction manager does the validation check and unlocks locked data items.

For the shutdown function, the transaction manager checks if there are any active transactions. If there is no active transactions, the transaction manager returns true and the middleware shuts down other resource managers and middleware. An extra thread returns true to client if middleware shutdown succeeds.

3.c Time-to-live mechanism

For each transaction, we start a separate thread(*TimeThread*) to inspect the time-to-live values of that transaction. If one has been idle for more than 60 seconds, the transaction will be aborted. We keep an operation counter in each transaction. When an operation involving this transaction is carried out, the operation counter is incremented and the time to live is reset. The *TimeThread* keeps checking whether the operation counter is incremented and time to live expires. If the operation counter is not incremented within the time to live, then the transaction is aborted.

3.d Shadowing

Shadowing is implemented the same way as we have seen in class. We keep two copies of database: one committed version and one working version. We also keep a master record that indicates the latest committed version. Master record is updated every time the transaction is successfully committed. Let's say we have two copies A and B of database and A is the where the master record currently point to. Upon execution of T, we update main-memory copy. Upon commit request of T, We first write main-memory copy to B and then write master with pointer to B and transaction id to disk. Upon abort request of T, we simply discard main-memory copy and read A into main memory. If crash before writing master record, A remains latest committed copy. After restart of system, A will be loaded into main memory and T will abort. If crash after commit of T but before commit of any other transaction, master record contains T and points to B. Upon restart of system, B is loaded into main memory.

3.e Logging

To recover from crash, the processors must know their state to be able to tell others whether they have reached a decision and the status must be available after failure and restart. This is achieved by writing a log record to disk for every state change in protocol and every transaction. As mentioned in 2.a, the log file is saved as a serializable object in the disk which contains states and a transaction id. To make the log easier to use for recovery, we keep a log per transaction per site. For example, if we have a transaction T with transaction id 5 involving two resource managers *Car* and *Room*, before T is successfully committed or aborted, there will be three log files existing in our system. One named *'Car_5.log'* is on the Car RM, one named *'Room_5.log'* is on the Room RM and one named *'TM_5.log'* is on the middleware which is the log of Transaction Manager. The log file will be automatically deleted from the disk once the transaction is completed, namely, either abort or commit. Figure 3.1 and 3.2 show when and what status is written into the log file for resource managers and coordinators(transaction manager) during Two Phase Commit process. The details about how the system recover from crash with the help of the log file will be introduced in 5.c and 5.d.

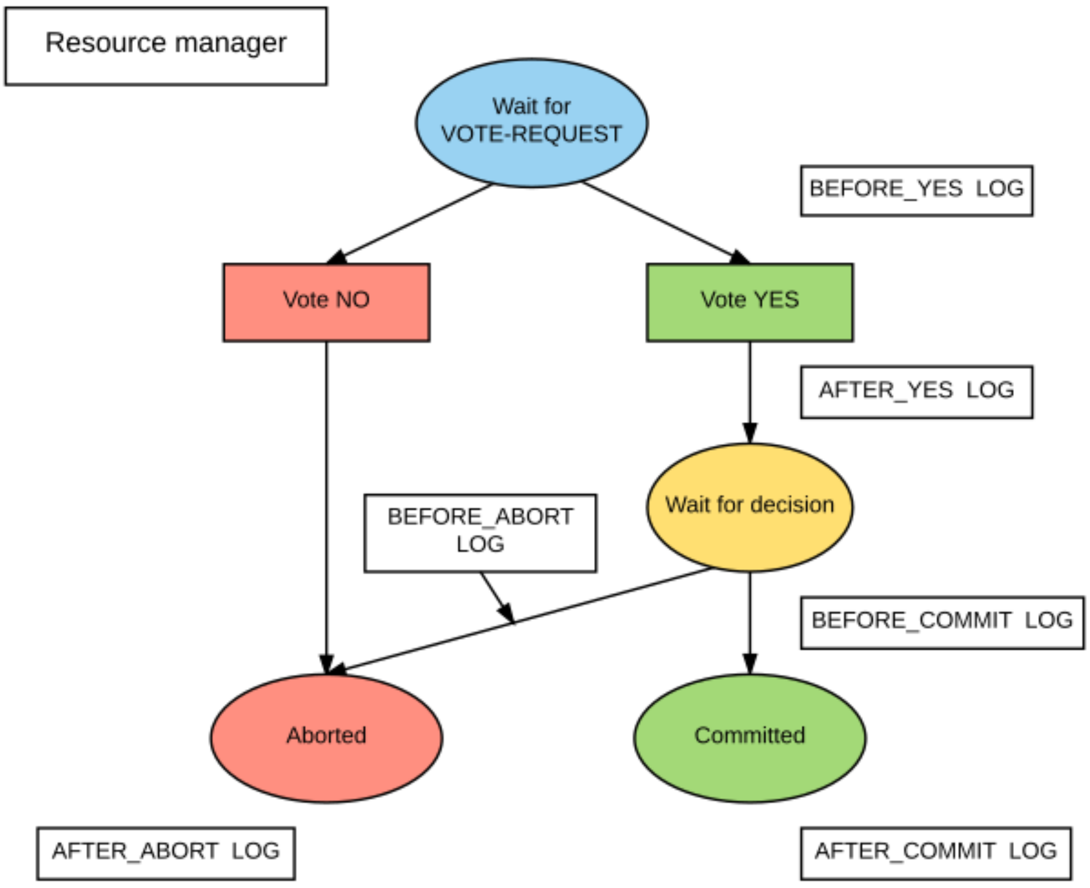

Figure 3.1 Log points on resource manager for Two Phase Commit

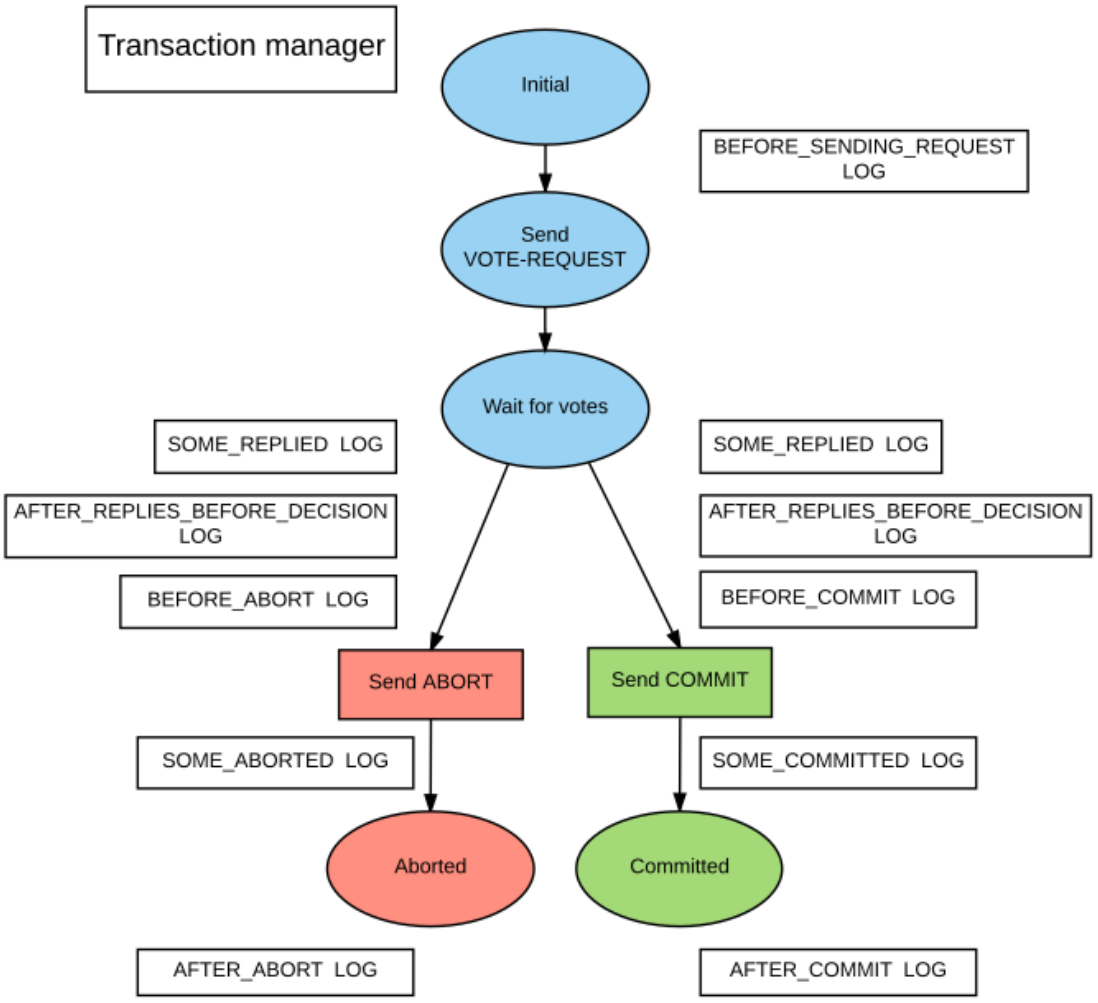

Figure 3.2 Log points on transaction manager for Two Phase Commit

3.f Two Phase Commit

We will briefly describe how Two Phase Commit works here, since it is clearly explained in slides. Firstly, the coordinator (transaction manager) starts the commit process and sends a VOTE-REQUEST to all participants (resource manager). Upon receiving a VOTE-REQUEST, the participants sends a vote with YES or No. If vote is NO, it aborts the transaction locally. On the coordinator side, if all vote YES, it sends COMMIT decision to all participants and commits locally, otherwise, it sends ABORT decision to all participants and aborts locally. Upon receiving decision, the participant commit or abort accordingly. A function *prepare()* was added in both middleware and resource manager for the phase one. Function *commit()* was also adapted to Two Phase Commit protocol.

3.g Methods implementation on middleware adapted to distributed environment

1. *Design of reserve* method*
   Firstly, in the RMI-based system, once the middleware gets a request from a client, it checks existence of the give customer ID. Then, the middleware sends a *reserve* request on the corresponding resource manager(e.g. *reservecar* to *Car-RM*) through function call on the resource manager proxy . The corresponding resource manager handles the remaining validation of the *reserve* request, namely the existence of requested item. If the item can be reserved, resource manager will update the local hash table of reservable items. The system will propagate a boolean variable back to the client.
   On the TCP socket based system, the customer validation is the same. However, in order to update the customer information stored on middleware, the middleware queries the price of the item before sending the reserve request. The reason is that the resource manager can only send back a boolean statement for the reserve request. If the reservation succeeds, middleware customer information needs to be updated for the reserved item with queried price. On the client side, there is no difference between RMI-based and TCP-based system.

2. *Design of deletecustomer method*
   We added an *updateItem* subroutine in resource manager as a function(RMI-based) or request(TCP-based) for middleware to handle restoration of reserved items when deleting a customer. After restoring all reserved items by the given customer ID, middleware can safely delete the customer information. The items then can be reserved by other customers.

3. *Design of itinerary method*
   The first part of the implementation is merely to check the availability of all flights and the car and/or room at the given location through remote method invocation(RMI-based) or message passing(TCP-based) from middleware to resource managers. This will prevent partial reservation issue where there is an runtime error that something cannot be reserved but some flights are already reserved. After successfully checked the availability, the middleware will call reservation functions(RMI-based) or send reserve request messages(TCP-based) to respective resource managers.

4 Concurrency control with Two Phase Lock

The concurrency control is achieved by Two Phase Lock and Two Phase Commit. A lock guarantees exclusive use of a data item to a current transaction. In other words, transaction T2 cannot write on a data item that is currently being used by transaction T1. A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed so that another transaction can lock the data item for its exclusive use. More details of lock can be found in section 3.a. Besides Two Phase Lock, Two Phase Commit protocol plays an important role in concurrency control as well. It guarantees that if a portion of a transaction operation cannot be committed, all changes made at the other sites participating in the transaction will be undone to maintain a consistent database state.

5 Correctness

5.a Basic functionalities of resource manager and middleware

To test the correctness of basic functionalities of the system, we have run several tests on both interactive clients and non-interactive clients. Our test consists of multiple stages which are in a sequential order.

First part of the test is whether the functionality of servers and middleware behaves as expected for only one client connected to middleware. The tests on creating an new item and querying both the number and price of an item are trivial and therefore omitted in this report. The focus of this part is on the test of reserving an item, deleting a reservable item, deleting a customer who has several reservations, and itinerary method. For the *reserve* function, both RMI-based and TCP-based system behaves as expected. If the item actually can be reserved, the system will reserve the item with correct data updates on both middleware and resource manager without data corruption. If the item or customer does not exist (or the remaining item is zero), the failure message will be returned to client application and the data segment on both middleware and resource manager do not change. For the *deletecustomer* function, after execution, the data on resource managers are updated and items become available to reserve. The customer information is removed from middleware. For the *itinerary* function, we have tested the case where (a) flight, room, or car does not exist (b) repeated flight numbers (c) correct input arguments. The output behaves as expected.

The second stage of the test concentrate on the system architecture design. First of all, we tried to connect multiple clients to our middleware and send requests concurrently. There is no data corruption when multiple clients try to write on the same data segment. This is handled by java built-in *synchronized* blocks on the data. Then, we tested the case where one of the client is blocked. The trick we used is to intentionally create a malfunctional method that takes a very long time to proceed in resource manager. One of the clients calls the method indirectly and gets blocked from executing. At the same time, other clients in our system are working without any interruption. Furthermore, we have tested the system stability when some components are disconnected from the system i.e. force quit *Car-RM* while other components are running. In this example, the system loses connection to car resource manager without crashing the whole system. Moreover, a client that is waiting for response from *Car-RM* gets blocked and hopefully can get a response later. Clients that are not sending requests to *Car-RM* can run normally on other modules, e.g. reserve a room or flight, query customer information, etc.

In addition to manual testing, we implemented a non-interactive client (*client_test*) for testing purpose. The *client_test* also print out the current data on each *RM* and middleware (by sending queries to middleware). One of the noticeable discoveries from automated testing is that TCP-based build takes slightly longer time than RMI-based build. Our discussion on this observation concludes that the remote method invocation is slightly faster than message passing by socket.

5.b Two Phase Commit

No matter whether there is failure or not, Two Phase Commit(2PC) should keep data consistent among all components. The test of 2PC without failures is straightforward. The test involved several type of transactions such as transaction involving only one resource manager and multiple resource managers, transaction ending with commit or abort, transaction consisting of only write operations, only read operations and both read and write operations. It turned out that the results were exactly as expected, which means the data is always consistent among the components. The correctness of 2PC with failures will be discussed in following subsections.

5.c Crash and Recovery at Resource Manager

The status of sites before failure can be recovered by scanning the log file. There are totally 5 possible cases for resource manager crash including (1) crash before client calling commit, (2) crash after commit called(received vote requests) before sending answers, (3) crash after sending answers but before receiving decisions, (4) crash after received decisions before commit/abort, (5) crash after commit/abort. Due to the fact that Customer resource manager resides within Middleware and depends on Transaction Manager, we separate the cases for Customer resource manager with other Regular resource managers. For regular resource managers, we have the following actions during recovery of the resource manager respectively.

For case (1) at regular resource managers, we simply abort the transaction. This is because that once the client invokes commit to the Middleware, the Middleware will send prepare to each involved resource managers. Since the resource manager is crashed, it cannot send answer to the transaction manager. As a result, all the resource managers will be aborted(as all_votes_yes is false). Therefore, when the resource manager is recovering, it should abort such transaction. For the Customer resource manager, if crash happens before 2PC is called, there will be a transaction manager log for this transaction id with empty records. At recovery stage, transaction manager will abort such transactions. For case (2) at regular resource managers, the transaction is aborted since the answers received at transaction manager is not all vote yes. At Customer resource manager, we handle this case in the same way. For case (3) and (4) at regular resource manager, the transaction is not committed or aborted. The action depends on the decision from transaction manager. Therefore, the way to handle case (3) and (4) is to ask transaction manager to resend the decision and commit/abort accordingly. However if the crash site is at Customer resource manager, the transaction manager also crashes. Hence, as the ordering in the implementation, Customer is always the first one who receives decision. Furthermore, transaction manager waits for Customer resource manager's response before sending more commit or abort to regular resource managers. In this way, no other resource manager can commit/abort before Customer resource manager. As a result, it is safe and consistent to abort the transaction at all sites. We will see that case (3) and (4) is a special case for crash at transaction manager. For case (5) at regular resource managers, recovery simply removes the log files as the operations are already saved to stable storage after commit/abort. If the crash happens at customer resource manager after customer committed/aborted, then the decision for this transaction is known and some resource managers are committed/aborted already. Therefore, at recovery of middleware, we can use the decision to finish the commit or abort at all sites.

In testing, we tested above cases in 3 modes by automated crashing. For the first case, this is hard to automate without flags in basic methods e.g. *addCars* etc. We want to avoid massive copy and paste crash code in those methods. However, it's much easier to test by simply force quitting the JVM process(Ctrl + C). In such way, we can set any crash point before commit is called. For case (2) - (5), since they are inside an atomic commit operation at client interface, we have 3 crash mode, namely mode 1 for case (2), mode 2 for case (3) and (4), mode 3 for case (5) based on the action to take during recovery. Figure 5.1 shows the correspondence between crash cases and crash modes (case 1 is omitted as it is not part of 2PC).

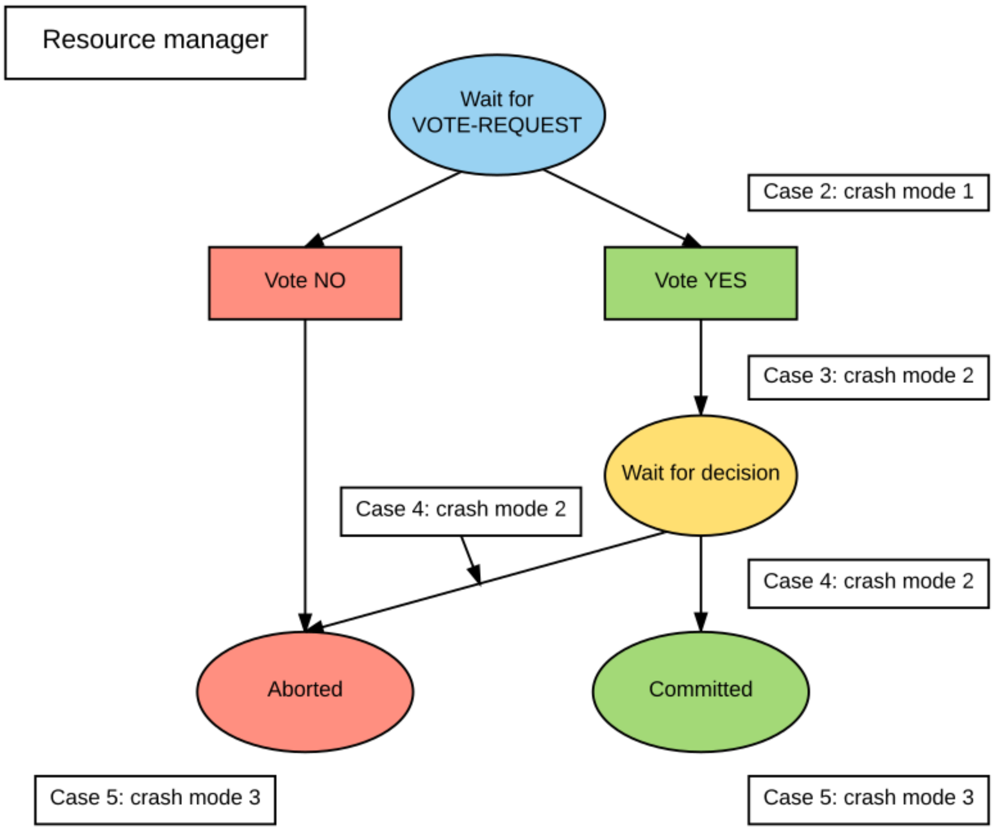

Figure 5.1: Crash and Recovery at Resource Manager

5.d Crash and Recovery at Transaction Manager

Transaction manager has more possible ways to crash. Nevertheless, it is relatively easier to recover the transaction manager. We continue the numbering in part 5.c for different cases. We have the following cases for transaction manager crash: (6) crash before sending vote request (7) crash after receiving some replies but not all after sending vote request (8) crash after receiving all replies but before deciding (9) crash after deciding but before sending decision (10) crash after sending some but not all decisions (11) Crash after having sent all decisions.

For the cases (6) - (9), since no resource manager actually commit or abort at this point, the action during recovery should be aborting all active transactions. For case (10), depending on the decision sent, if "SOME_COMMITTED" is found in log, transaction manager should commit the transaction at all site, while if "SOME_ABORTED" is found in log, transaction manager should abort the transaction. For case (11), because Customer first receives the decisions and transaction waits for Customer resource manager response, if all decisions are sent, Customer resource manage must have done commit/abort. Therefore, it is safe to simply delete the log, as we assume other resource managers are less likely to crash at the exact same time. The 6 cases for transaction managers are automated through crash mode 4 - 9 as shown in Figure 5.2.

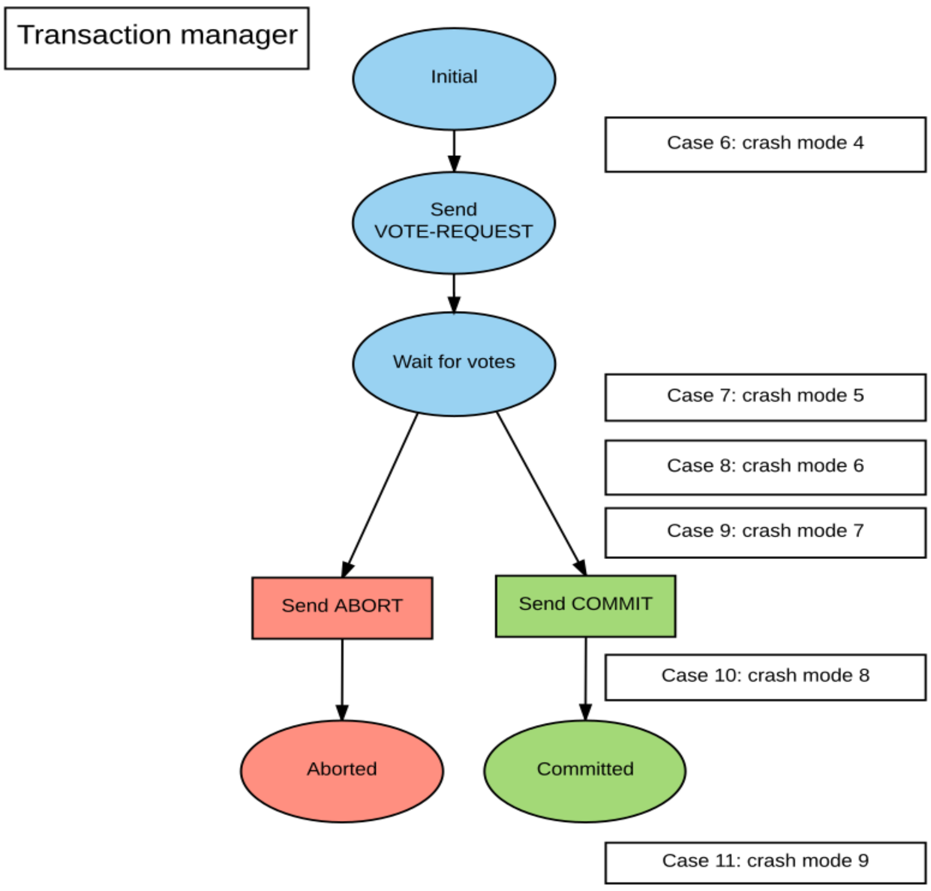

Figure 5.2: Crash and Recovery at Transaction Manager

6 Difficulties

6.a Customer design

The first difficulty we have encountered in this project is the way to design customer related functionalities. More specifically, the delete customer function is one of the most difficult parts in project phase one. The difficulty comes from the requirement to release the reserved items and update corresponding resource managers accordingly. As we do not have existed interface for such operation, we implemented extra interface functionalities for this issue.

6.b Abort recovery

The next issue that we spent lots of time to solve is the abort recovery. Since in our design all intermediate operations are written on the hash table in memory directly, on the occasion of abort, we need to recover all previous operations done in memory. The way we address this problem is to store a history for each transaction. The history is implemented by stack data structure. There are two main reasons. First is that stacks preserve the order of executions. This property is extremely important for transactions in which different ordering of operations can lead to different results. The second reason is that stack is easy to reverse the order in case we want to redo the operations (which is used in crash recovery where we need to redo operations on the data after restored from disk).

6.c Resource Manager reconnection to Middleware

In project phase three, crash on any site can happen. Therefore, sometimes crash can isolate some sites due to the dependency of existed system. For instance, if a car server crashed and restarted, the middleware is not connected to the newly started car server, due to the fact that at the start of middleware, the middleware seeks all resource managers through RMI and the connection is fixed once middleware is started. In this case, we say that middleware instance depends on the resource manager instances. Hence, even the crashed car server has recovered, the middleware still connects to the old crashed instance of car server. Therefore, an extra hand shaking protocol is required to rebuild the connection between recovered resource manager and the middleware. We found that as long as the middleware resides and runs on the same server, the resource manager can try to connect to that server with specific port number. These informations can be provided at the time of restart. Therefore, once the resource manager finds the middleware, it can ask the middleware to replace the old instance of resource manager by the recovered one. In such way, RMI connection is recovered from crash. On the other direction, if the middleware crashes and recovers, this will not be an issue, because at the start of the middleware, it needs to find those resource managers anyway.

6.d Handle Customer and Transaction Manager crash

Due to the fact that our customer and transaction manager resides on the same middleware, crash of one component would automatically crash the other one. As a result, we have to consider both cases at the same time. In our testing, there are totally six crash mode for transaction manager and three crash mode for customer. This can causes very unpredictable situations and decisions. To handle this issue, we modified the order of some executions to make the execution behavior much more predictable. For example, in the original build we use a for loop to loop through all resource managers involved in a transaction and commit them one by one. However, this is very vague for us to say whether the customer component is committed before the crash or after the crash if the crash happens during the loop. Without knowing this subtle difference, we cannot make a good decision during recovery. This makes the customer component a much more special case than the three other resource manager. As a result, we eliminated such vagueness and forced the executions to be deterministic.

7 Performance evaluation

7.a Single client

We evaluated the marginal performance of the system by executing read only, write only and read+write operations on a single RM and on all three RMs. Each transaction type involves 12 operations, so the transaction types are comparable in overhead. We also ended transactions with both commits and aborts for every transaction type. In a single test of a transaction type, after the system warmed up, we executed this transaction 1000 times. For each transaction type, we did 10 tests. The average latencies of each transaction in milliseconds are given below.

Results:

Transaction Type	1 RM	3 RMs
12 Read + Commit	16.554	24.278
12 Read + Abort	16.256	24.167
12 Write + Commit	16.615	26.325
12 Write + Abort	17.136	26.194
6 Read + 6 Write + Commit	16.530	25.074
6 Read + 6 Write + Abort	17.030	25.281

We observed from the table above that the write only transaction takes slightly longer time than the read only transaction, and the average response time of read-write mixed transaction type is somewhere in the middle of these two. This conclusion holds for both single RM and three RMs case. Since the number of hops for abort and commit operations are the same in our architecture, the response time will not be affected too much by this factor. The experiment results also proved this inference.

Another fact that can be observed by looking at the table is that the average latencies of transactions involving three RMs are obviously higher than corresponding transactions involving one RM. The explanation of this fact can be that when dealing with three RMs, the communication delay and the overhead of connection establishment are much longer.

In this project, since we did not interact with a real database system, the time spent on read and write operations is not comparable to the time spent on communication delays. Therefore, the time is mainly spent during communication processes in our system. That is also the reason that different transaction types take very similar time to execute.

7.b Multiple clients

For performance evaluation of this part, we used the following parameters:

- A fixed number of clients: 5, 10, 20, 30, 50.
- The sleep interval is either [225, 275] or [475, 525].

We tested how the number of clients impacts the system performance in terms of latency, CPU usage on the middleware and resource manager. The results are given in Figure 7.1, 7.2, 7.3 and 7.4 respectively.

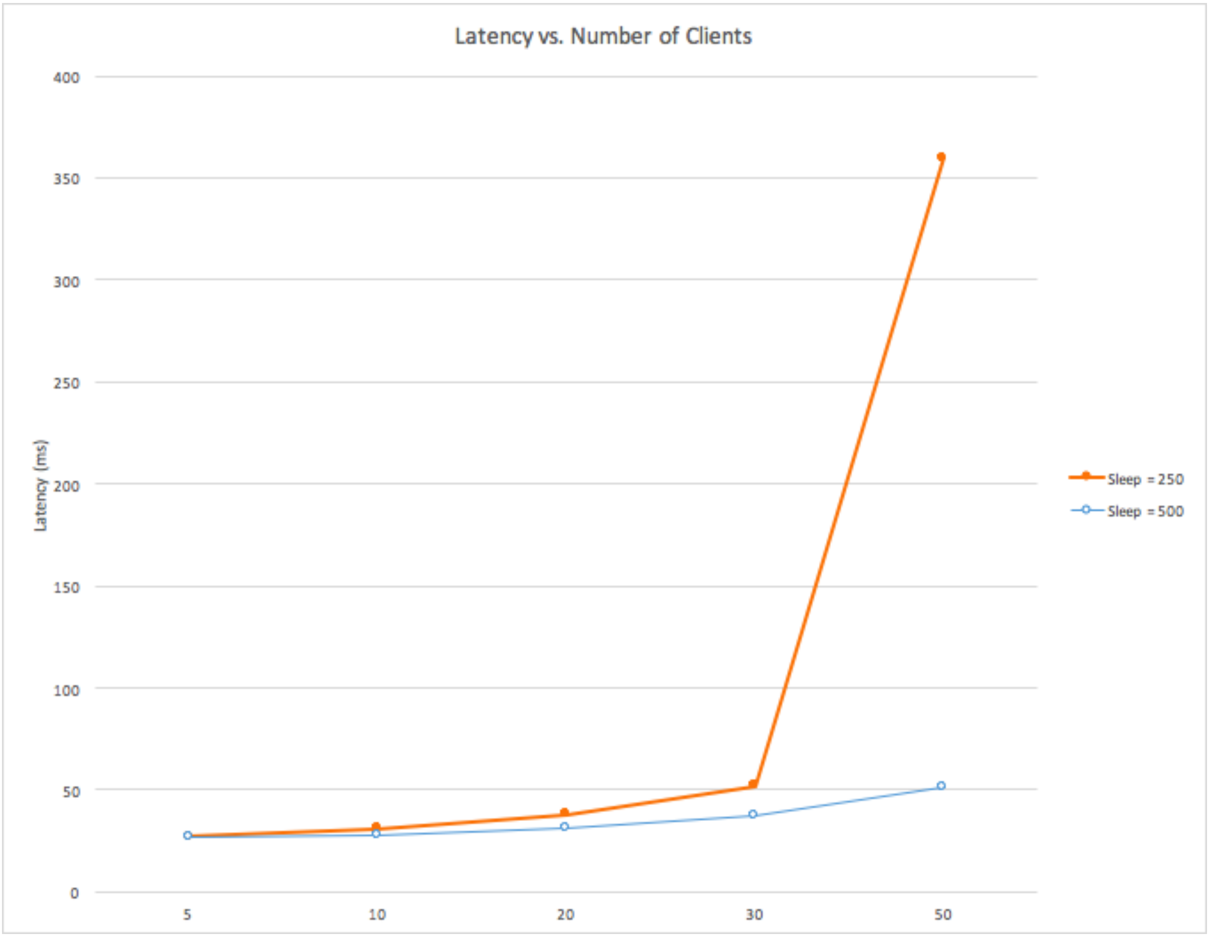

Figure 7.1 Latency vs. Number of Clients

From Figure 7.1, we observed that the latency time grows almost linearly with the number of clients before the number of clients reaching 30. After the number of clients exceeds 30, the latency time grows sharply if the sleep interval is [225, 275], which is to be seen as the saturation point of this system. To make the results more straightforward, we plotted Figure 7.2 which shows the relationship between latency and load injected. When the load injected is around 80, the latency has an sharp increase. Therefore, load injected = 80 is the saturation point of the system. Figure 4 is consistent with Figure 7.1.

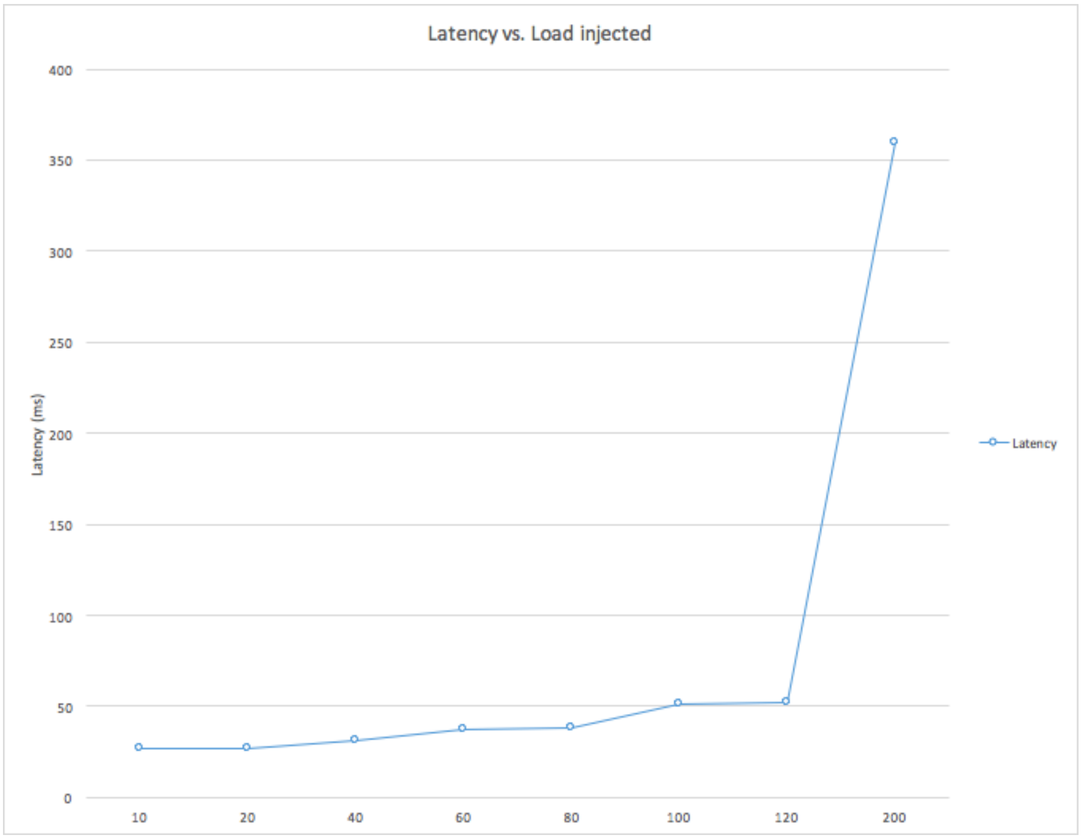

Figure 7.2 Latency vs. Load Injected

From Figure 7.3 and 7.4, we observed that the Middleware CPU usage and Resource manager grows as the number of clients. However the percentage of CPU usage on middleware is much higher than resource manager given that the resource manager and middleware ran on the same CPU model. This observation shows that the CPU on the middleware is the bottleneck of our distributed system. The throughput of the system is largely restricted by the computation ability of the middleware.

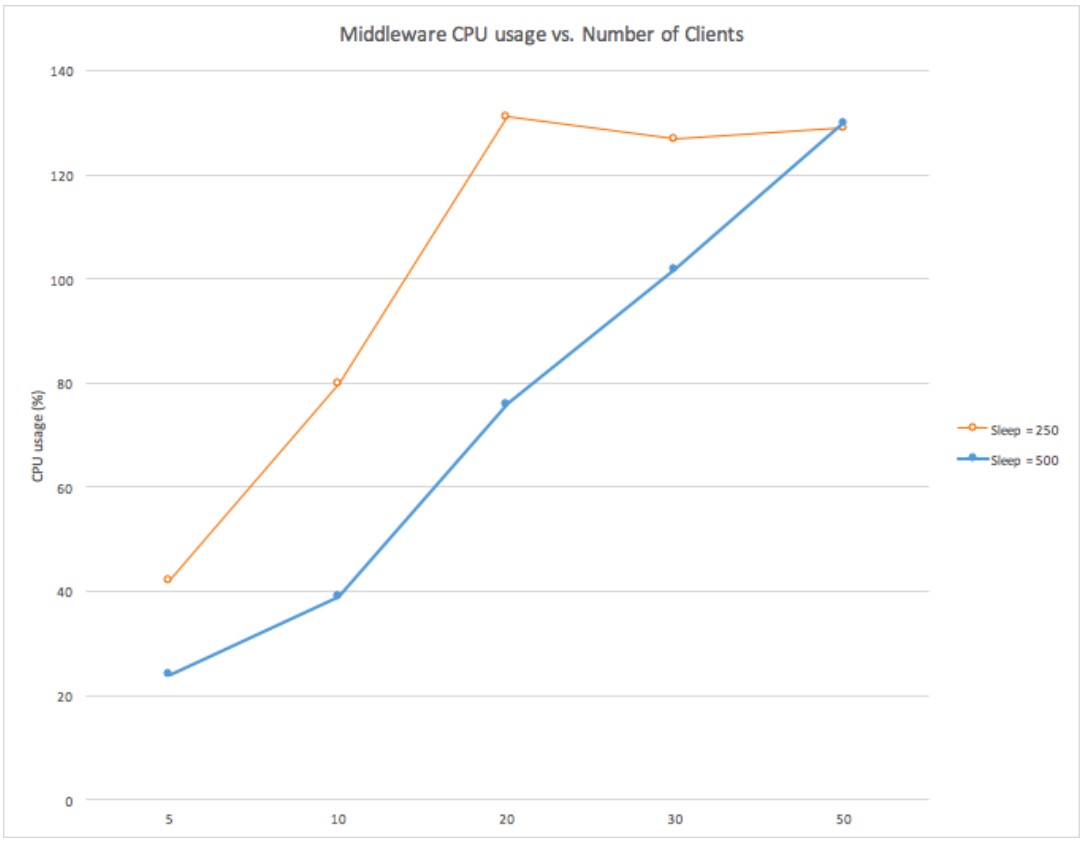

Figure 7.3 Middleware CPU usage vs. Number of Clients

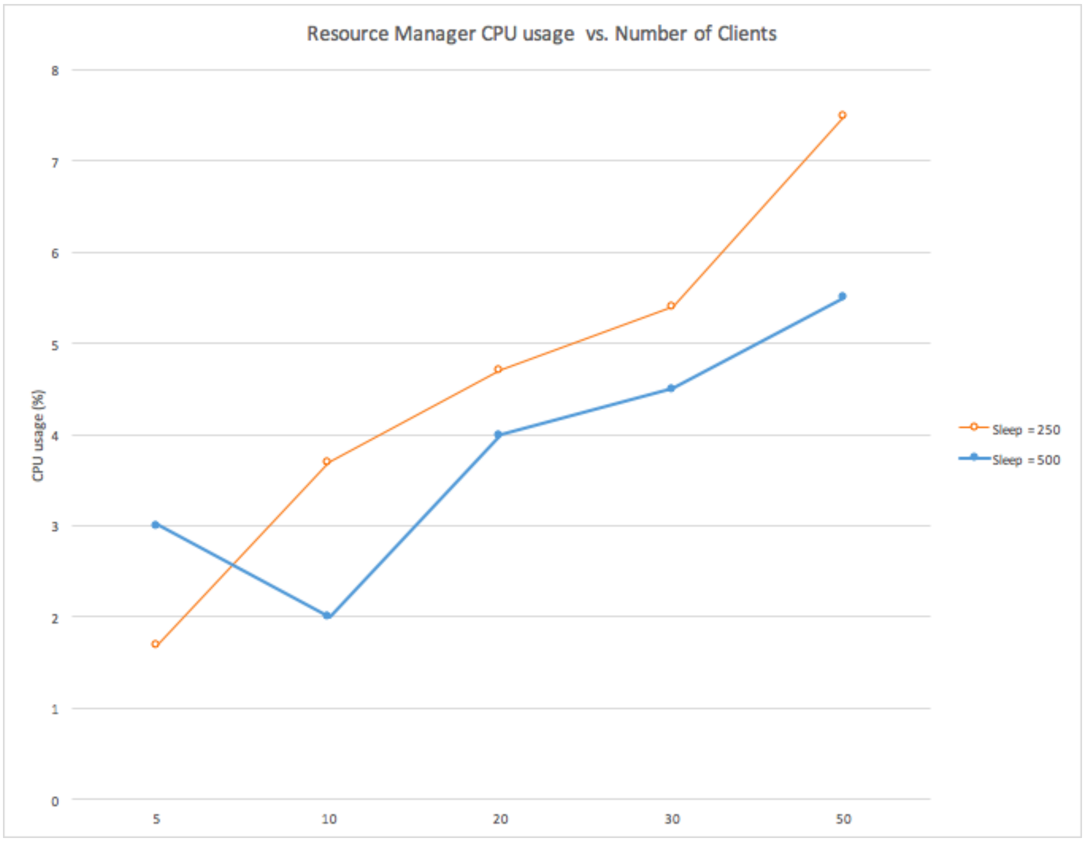

Figure 7.4 Resource Manager CPU usage vs. Number of Clients