School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2015

Final Project Report

Group 16 Members: Jing Chen (260553228) and Brian Zhou (260381251)

**Introduction**

This report presents a component-based distributed travel reservation system, which is an adaption of the project of CSE 593 of the University of Washington. The system is a multiple client and multiple server system, which simulates the services provided by online travel reservation agency. The function of the system includes: **(1) Distribution**: any client would send reservation requests (e.g., flight/car/room reservation request) to corresponding servers (e.g., flight/car/room server) via a middleware server, which is responsible for processing, and distributing requests from clients to servers. The middleware layer is made transparent to clients, so that clients are not aware of the distribution function provided by the middleware.

**(2)** **One-phase concurrency control (1PC)**: multiple clients would request to update same flight/car/room objects on RM servers concurrently. The bunch of requests issued by each client is regarded as a transaction. Thus, concurrency control mechanisms (e.g., two-phase lock & optimistic concurrency control) should be implemented to guarantee those transactions are serializable.

**(3) Two-phase concurrency control (2PC)**: if one transaction involves multiple objects which are distributed among multiple RM servers, then 2PC enables those RM servers to vote whether they are ready to execute the transaction. If all distributed RM servers are ready, then the transaction would be commit successfully. Otherwise, it would be abort locally. In this case, a transaction manager is responsible for sending vote requests, collecting vote results and sending decision commands. Since any servers are possible to crash during this procedure, shadowing mechanism should be implemented to keep data persistence upon crash and restart of each server.

The whole project is implemented as two ways in Java, one is by using of web-service, and the other is based on TCP socket programming. The web-service design only implemented **distribution function**. At the mean time, the TCP socket design realized all three functions mentioned-above: **distribution, 1PC and 2PC**. The following sections are organized as: Section I describes the architecture of web-service design, while details of the TCP socket design is introduced in Section II.

**Section I: Web-Service Design**

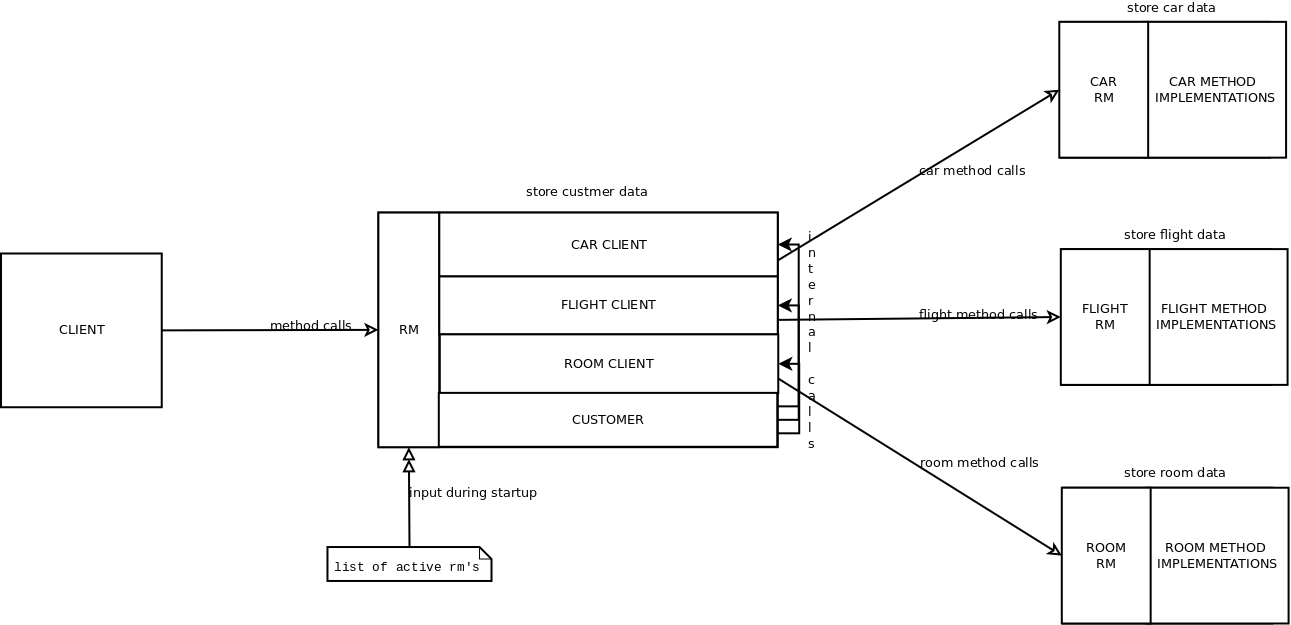


Figure 1: Architecture of Web-Service Design

**Section 1.1: Design Description for Distribution**

Figure 1 illustrates the overall architecture for our web-service design. By observation, there are one client, one middleware server and three resource manager (RM) servers. Since the specified travel system only provides reservation service for flights, cars and rooms (hotels), three RM servers are required. The middleware server is implemented as front end of RM servers, which acts as an interface for multiple clients under web-service protocol. Each client would make reservation requests relating to particular objects (e.g., flight, car, room) on RM servers which host and handle that particular objects. The middleware server hosts customer objects and implements methods/calls pertaining to customer objects. If a client reservation request involves one or many of flight, car and room objects (e.g., itinerary), then the middleware server would handle all requests pertaining to the client objects locally, and make internal calls to the middleware clients when information on the objects are required. After that, the middleware clients would make calls to the corresponding RM servers to fulfill the requests. For example, itinerary request requires adding reservations to the customer object and updating the corresponding ReservableItem count on RM servers. Thus, the middleware server would first retrieve the customer object locally, and then informs the internal middleware clients to call the methods on RM servers to decrease the count for the ReservableItem. After that, the middleware would add the reservation to the customer object, which is saved locally. Particularly, the middleware server would automatically retrieve the host and port of flight, car and room RM servers on start up via a user editable file.

**Section 1.2: Testing Strategy**

**1.2.1 Goal:** Test if our web-service design fulfills distribution function required by specification. That is, test if new added middleware server is able to correctly forward reservation requests from multiple clients to the corresponding servers. Moreover, test if the three RM servers are able to make updates on ReservableItem count specified by clients’ requests.

**1.2.2 Test Cases:**

(1) reserveflight

newflight,1,1,10,100

newcustomerid,1,1

reserveflight,1,1,1

middleware response: 1 customer object is created, and 1 flight reservation is made

flight RM response: flight count is decreased by 1

(2) reservecar

newcar,1,montreal,20,200

newcustomerid,1,1

reservecar,1,montreal

middleware response: 1 customer object is created, and 1 car reservation is made

car RM response: car count is decreased by 1

(3) reserveroom

newroom,1,montreal,30,300

newcustomerid,1,1

reserveroom,1,montreal

middleware response: 1 customer object is created, and 1 room reservation is made room RM response: room count is decreased by 1

(4) itinerary

newflight,1,1,10,100

newcar,1,montreal,20,200

newroom,1,montreal,30,300

newcustomerid,1,1

itinerary,1,1,1,montreal,true,true

middleware response: 1 customer object is created, and 1 flight reservation is made, 1 car reservation is made, and 1 room reservation is made

flight RM response: flight count is decreased by 1

car RM response: car count is decreased by 1

room RM response: room count is decreased by 1

(5) deletecustomer

newflight,1,1,10,100

newcar,1,montreal,20,200

newroom,1,montreal,30,300

newcustomerid,1,1

itinerary,1,1,1,montreal,true,true

deletecustomer,1,1

middleware response: customer object is deleted

flight RM response: flight count is increased by 1 compared with test case (4)

car RM response: car count is increased by 1 compared with test case (4)

room RM response: room count is increased by 1 compared with test case (4)

**Section II: TCP Socket Programming Design**

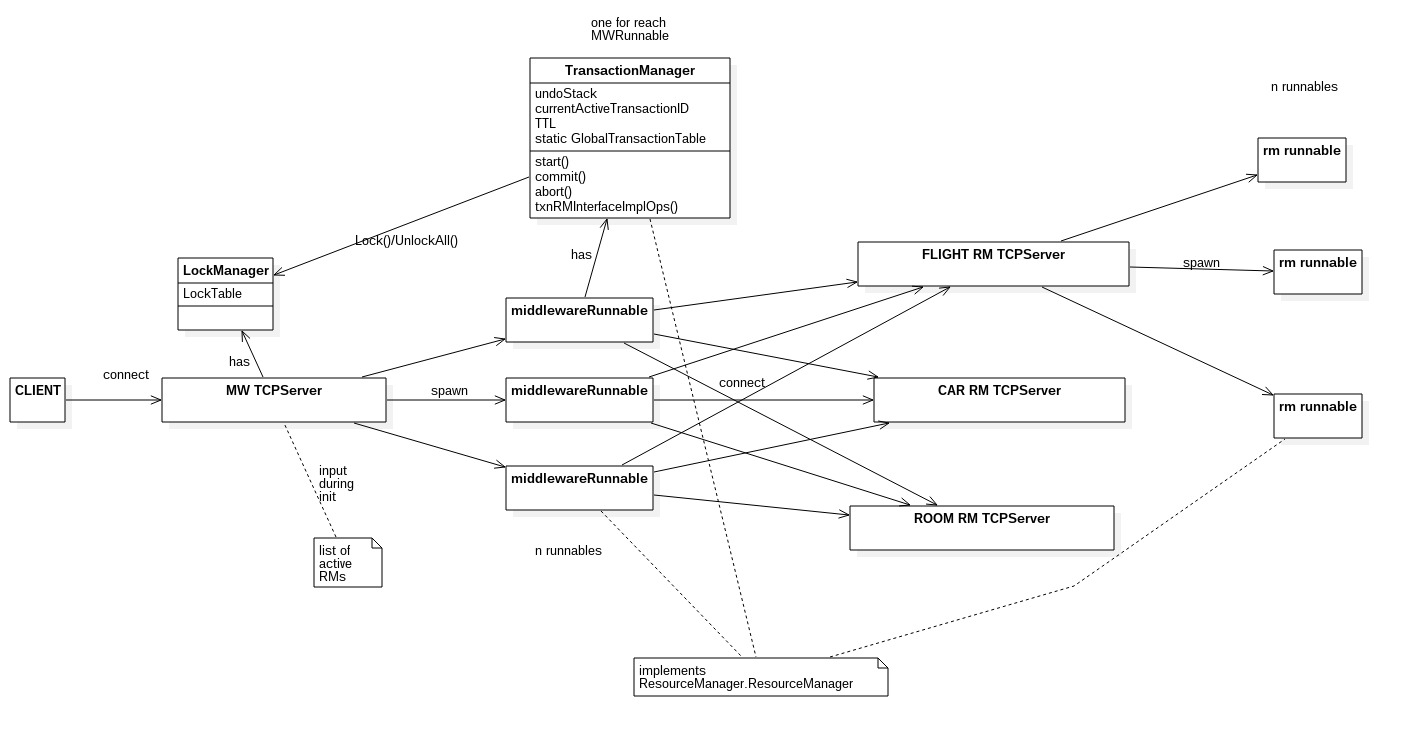
****

Figure 2: Architecture of TCP socket Design

**Section 2.1: Design Description for Distribution**

In our TCP socket programming implementation, the middleware and RM servers are implemented under TCP protocol. Whenever clients come in, a bunch of runnables would be spawned, which are used to handle clients’ connection. For example, the middleware runnables are created for each end user client connection. They normally keep expecting client commands as long as the client is remained as connected. Upon creation, the middleware runnables normally establish 3 different connections of its own, one for each of the RM servers under TCP protocol (short for TCP server). Once a middleware runnable makes a connection with an RM TCP server, the RM TCP server would create a RM runnable to handle this connection. Since it is indeed those runnables that are actually handling the connections, it would be them who implement the RM interface. This principle enables us to be able to achieve parallel execution both at the middleware layer level and at the RM server level. Again, the middleware server is responsible for handling customer data only and making calls to RM servers when info on car, flight and room objects are required. The only difference for calls between web-service and TCP socket is they no longer be web-service interface defined remote calls. Instead, they are TCP string messages that contain the command we want to execute. The RM servers take these commands and parse them, and then invoke the appropriate local method calls. The result of the local execution is sent back to the middleware runnables via the same TCP connection.

**Section 2.2: Design Description for One-Phase Concurrency Control**

**2.2.1 The Centralized LockManager**

We have implemented the LockManager at the middleware layer level as a static field. All lock requests have to go through the middleware in order to lock and unlock items.

**2.2.2 The Centralized TransactionManager**

The TransactionManager (TM) is implemented as an instance associated with each middleware runnables. As clients connect to the middleware TCP servers, a middleware runnable is created to establish this connection, and then this middleware runnable would spawn a TransactionManager instance that is specific to this connection. When client submits non-transaction requests, the middleware runnable would call its implementation of the RM server interface methods. When the client issues start (), the TM of that connection will start the TTL countdown, generates a system wide unique transaction ID and return that to the client. Until this transaction is aborted or committed, all requests would go through the TM's implementation of the RM server interface methods instead of the middleware runnable’s. The TM's implementation of the RM server interface methods simply attempts to acquire all necessary locks, it then builds an undo command to undo the current request in case of abort and save this undo into the undoStack of the current transaction. It then goes on to call the middleware runnable implementation of this command, which actually performs the appropriate, reads and writes of data. The TM then saves the involved RM servers’ into the global transaction table that is shared by all TM's. If a client issues the commit () command, the undoStack and all other state related to the current transaction is wiped from the TM so that it can be ready for the next transaction. If instead the abort () command was triggered, then TM would execute each command in the undoStack in order to revert all the effects in the current transaction. After an abort or commit execution, the list of involved RM's for this transaction is taken off the global transaction table. Each TM will allow 1 active transaction at a time for its client. After typing start (), entering a request with an XID other than the one returned to the client will result in that particular request being ignored.

**2.2.3 Time To Live (TTL)**

After a start () command is received by the TM, it would start a countdown thread, which sleeps for TTL amount of time. Upon wakening, this countdown thread would call abort (). Following the start of the countdown thread, each request received by the TM from the client will reset this countdown thread, hence renewing its time to live.

**2.2.4 System Shutdown**

Once a middleware runnable receives a shutdown () command from the client, it would first consults the global transaction table to see if there are any active transactions still going on. If so it would ignore the shutdown command, otherwise it will first issue shutdown commands to the RM TCP servers, and then it would shut itself down.

**2.2.5 Difficulties**

The most difficult part of one-phase concurrency control was how to figure out the undo process once a transaction is aborted. We opted for an undoStack that execute each undo command one after the other in stack order to revert any changes upon abort. Another difficult part would be lock request for deleteCustomer () and newCustomer (). DeleteCustomer requires locking of reserved items as well so we need a specific method just for that. NewCustomer () method was especially difficult because between the time it generates a unique customer ID and the time it creates a customer with that ID, another transaction might use newCustomerID () using the same customer ID that the system has generated for newCustomer (). Therefore the solution was to lock on the generated customer ID before newCustomer () can create the actual customer using it.

**2.2.6 Testing Strategy**

**2.2.6.1 Goal:** Test if our TCP socket design fulfills one-phase concurrency control (1PC) function claimed by specification. That is, test when two clients both update same objects on RM servers, only one client who first issues update request than the other would be able to do the update.

**2.2.6.2 Test Cases:**

(1) Test Lock Manager (two transactions both read same info, shared locks are granted by lock manager, no deadlock is happened)

(2) Test Lock Manager (two transactions, one first reads, the other then writes to same info. Deadlock is happened, the transaction that tries to write will work after the deadlock timeout after 30 sec)

(3) Test Lock Manager (two transactions, one first writes to, the other then reads

same info, deadlock is happened)

(4) Test Lock Manager (two transactions, both write to same info, deadlock is happened)

(5) Test Transaction Manager (abort undo operations within a transaction)

(6) Test Transaction Manager (commit makes transaction implemented)

(7) Test Timeout (transaction will be forced to abort after 60 seconds if user types no new command lines)

**2.2.7 Performance Evaluation**

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

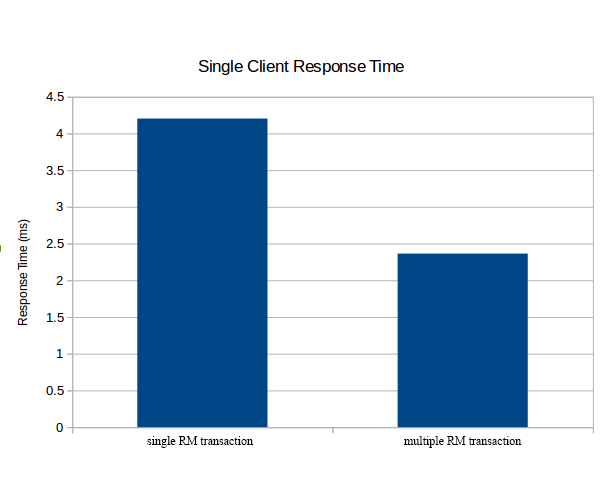


Figure 3: Single Client Response Time

**(1) Goal:** We are simply would like to see for a single client in the system how much time a single RM request takes versus how much time a multi-RM request takes. We just collect enough single RM request response times to come up with an average, and compare that to the average of response time of running a multi-RM request a bunch of times correct.

**(2) Description:** Figure 3 plots a figure of response time with respect to single RM transaction and multiple RM transaction. We observe single RM transaction requires more (almost two times) response time on average than that of multiple RM transaction.

**(3) Analysis:** In our testing program, single RM transaction normally contains four types of requests on a single RM. However, multiple RM transaction also contains four types of requests, but they are distributed on different RMs (e.g., flight, car, room). Thus, multiple RM transaction could execute on parallel, while single RM transaction could only execute serially. Moreover, single RM transaction needs to send requests for more times, which increase the communication between middleware and RM and accounts for more response time. For example, itinerary request needs to be sent one message from client to RM servers, while to achieve same function, one needs to send three requests, which are reserveflight, reservecar and reserveroom.

* + - 1. **Response time when there are many clients in the system**

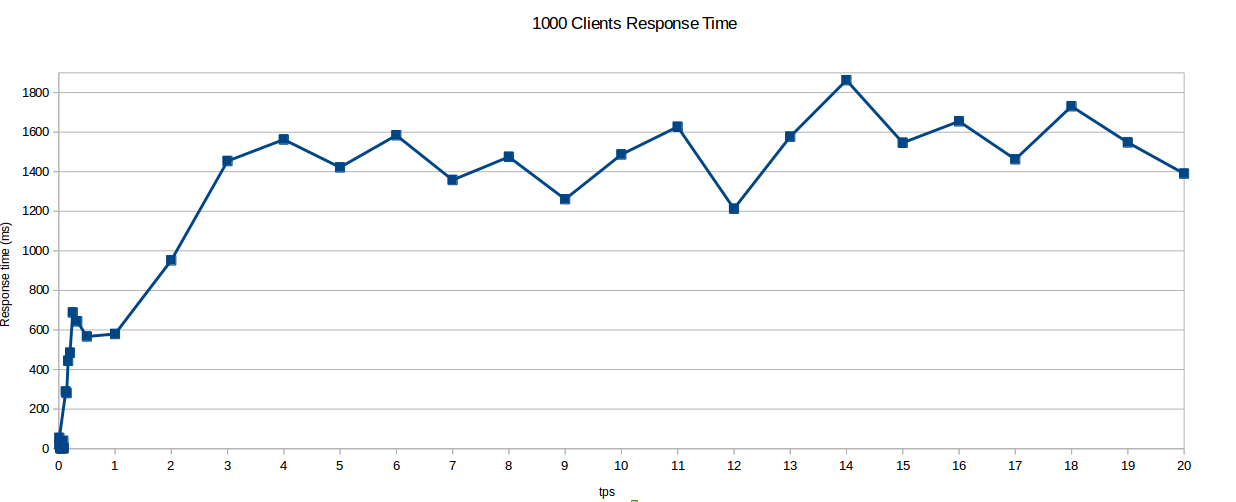


Figure 4.1: 1000 Clients Response Time

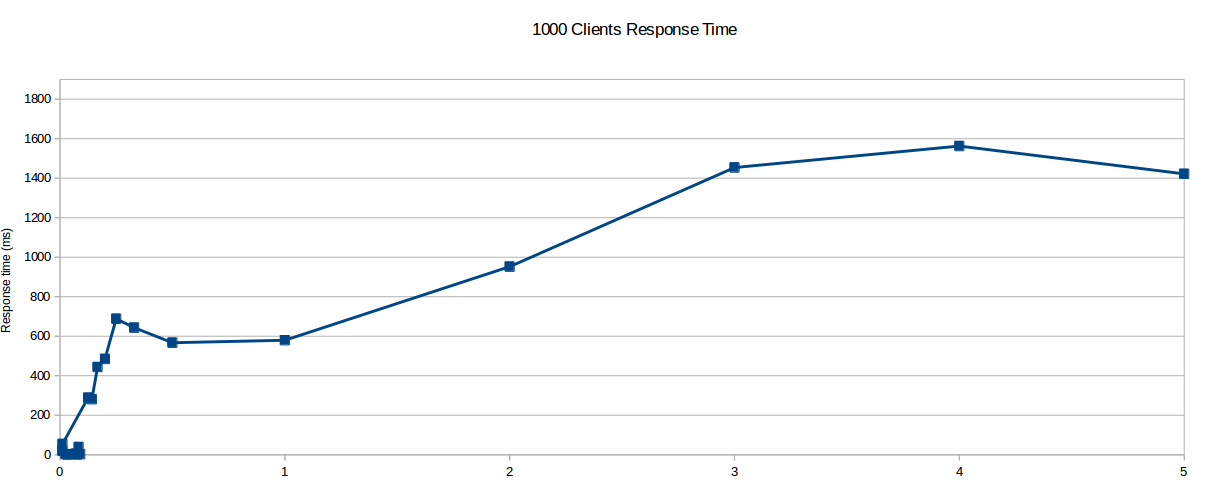


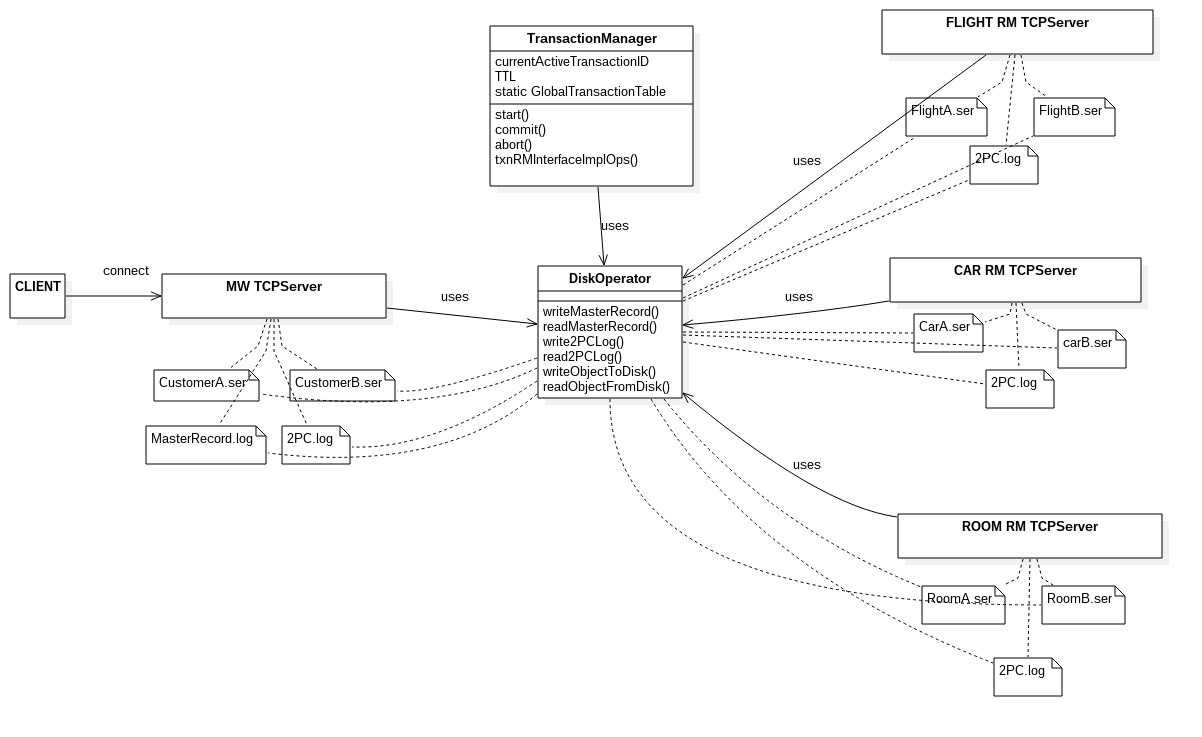
Figure 4.2: 1000 Clients Response Time from 0tps to 5tps

**(1) Goal:** We try to measure throughput, so we want to show how my response time vary as we increase load (transaction per second, short for tps). We run a fix amount of clients and submitting requests at say 1tps, get an average response time, then run these same clients but at 2tps, get average response time, then 5 tps, then 10tps so on and on and plot average response time versus tps.

**(2) Description:** Figure B.1 plots a figure of response time with respect to transaction workload varying from 0 tps to 20 tps (transaction per second) when there is 1000 clients in our system. Each client sends requests to multiple RM with mixed-workload (mixed-type-of-requests, mixed-type-of-operation) each time. We observe that, in the beginning, when the transaction workload is fairly low (from 0tps to 0.5 tps), response time increases very fast. After 0.5 tps, the increase of response time is moderate, but still fast. When transaction workload reaches 5tps, response time basically saturates between 1600ms to 1800ms. To clarify, Figure B.2 is made as part of Figure B.1 from 0 tps to 5 tps.

**(3) Analysis:** As transaction workload increases, communications among clients, middleware and RMs are congested, which possibly makes response time longer. Furthermore, middleware and RMs need more time to handle increasing number of transactions. For example, a lot of transactions may need to query or reserve flight, thus lock manager may need to lock and unlock flight resources, which make transaction execution more time-consuming.

**Section 2.3: Design Description for Two-Phase Concurrency Control**

****

**2.3.1 Shadowing**

In order to handle crashes, our system first needed to implement data persistence. This is achieved by using shadowing. We would use 2 images in order to make sure our system always have valid data. Each copy would take turn to represent the valid state of the system that we can always fall back on if any current modifications invalidates the data. If the current modifications are acceptable, we can then save this state into the other copy and with 1 I/O indicate to the master record that our latest reliable data resides on the newly updated copy. The old copy now becomes a rough draft on which we can attempt to write new data on. If the current modifications are not acceptable, then we fall back on the old copy and any future modifications would override the modifications saved on the latest written copy, thus overriding the unaccepted modifications.

**2.3.2 The Distributed Transaction Manager (TM)**

As a consequence of shadowing, the undoStack used in deliverable 2 is no longer needed as effects of transactions are no longer reverted upon abort. Instead, we would create before and after images corresponding to the before and after states of the data for a transaction. If a transaction is committed, the after image is loaded into memory and the before image is loaded if the transaction is aborted. All writes to objects during a transaction is done on a local copy that was read from the main memory of the node the specific object resides on. If a transaction is committed, the local written object then overwrites the copy in main memory during commit time and main memory is then written to one of the 2 shadow files. If the transaction is aborted, the local modified object copy is simply discarded and main memory copies remain unchanged. There is nothing to write neither at this stage because the shadow representing the latest reliable data is the same as the data in main memory. Reads are performed on local copy and if the local copy does not exist, then one would be created from the main memory copy and the read is then performed on the newly created local object.

**2.3.3 Failure Model**

With persistence and transactions achieved with shadowing, we can now use 2PC to implement fault-tolerance distributed commit. Our first phase involves the middleware asking all the Rms and itself to get ready, meaning preparing the before and after images. The second phase would mean indicating to all nodes which image would become the latest valid image that all nodes should have in main memory. Our system is implemented using sockets, so detecting crashes is fairly simple. The nodes in our system communicate using strings. If there is failure between 2 node due to crashes or network partition, a null will be returned instead of a string. Thus we do not need to detect failures using time out. In case of failures, our system needs to determine if the mw has made a decision by checking if its 2pc log has a commit record. Once it knows that it has made a decision, then no matter which nodes fails, upon restart, it will surely have the correct image loaded into memory. That image is the one indicated by the master record residing at the middleware. If the middleware has not made a decision, then the system will take the conservative choice to abort the transaction which means that any node that crashes, after the mw recovers, would revert to the before copy pointed to by the master record. For MW crashes, recovery simply involves reading the master record and loading the right image, then telling all active rm's to load the right image by sending that info in a message. For RM crahes, when performing the recovery, the RM checks whether the MW is up, if it is not, then the RM does nothing as it has no idea what is the correct copy to load. If the MW is up, then the RM asks from the MW which copy to load.

**2.3.4 Special Feature**

Our implementation deviates slightly from the standard implementation but achieves the same effect as intended by the instructions of this deliverable. This difference is that the master record that points to the right image only resides on the middleware. As a consequence, no matter at what stage a node crashed, it must ask the middleware to know which copy to load upon recovery. This means that if the mw is down, then a recovering node cannot know what copy to load until the mw is back up and running. However this does not cause any trouble for our deliverable because the system cannot function without the middleware so the rm's need not to be correct until the mw has come back up. When the mw does come back up, it will automatically tell all the active RM's which copy to load. This means that when the middleware sends the decision to the participants, it would also embed the version of the file that needs to be loaded into the decision message. This also means that all nodes would use the same version to save its portion of the valid data at all times. An additional feature we implemented is a slight modification of the 2PC log. In case when the mw crashes after writing a commit decision to the 2PC log but before changing the master record, there might be confusion when it recovers. At recovery time, it would not be able to tell if the master record is indicating the right version to load because it does not know if it has updated the master record before crashing or not. To be able to tell this, we would embed in the 2pc commit record the version of the image that needs to be loaded. So in a failure case described above, if the version in the master record is the same as the 2pc commit record, then we must have updated the master record before crash, so the master record is pointing to the correct image version that we should load. If the 2 record differs, then we follow the 2pc log and update the mater record and load the right image. The 3rd deviation is that our RM's do not require a vote request time out. This is because before an RM is asked to prepare for a commit, the main memory on the RM holds the data from the previous validated commit. Any active transactions involving objects of this RM only reads from this RM. Writes are performed on local copies from these reads. The writes do not take effect until commit time. In addition, the RM has no concept of transactions or locking as this information is handled at the middleware level, so our RM's do not distinguish between a state where its waiting for a vote request with another other state where its idle and waiting for instructions from the MW.

**2.3.5 Difficulties**

This deliverable was not particularly difficult. Once we made sure that each feature is built solidly, building the next feature on top using the previous feature as a foundation made things easy. However, at this stage of the project, the code has become very messy. The code is much harder to maintain and a simple pointer problem has caused us many hours of debugging.

**2.3.6 Testing Strategy**

To test the correctness of our system in crash and in recovery, we have to make sure that the crash at the nodes result in correct decisions made at each node. The nodes would also need to make the right calls when it recovers from the crash. To test this, we can crash situations then check whether the remaining nodes behave correctly through console output and log. We can then test correct recovery by restarted the crashed node and check if all parties behave correctly.

**2.3.6.2 Test Case:**

Crash before sending vote request:

start

newflight,1,1,1,1

newcar,1,1,1,1

newroom,1,1,1,1

newcustomerid,1,1

commit

start

deleteflight,2,1

deletecar,2,1

deleteroom,2,1

deletecustomer,2,1

mwdiebeforevoterequest

commit

result: all RM's would have data of transaction 1 in main memory and upon mw recovery the mw decide to abort and tell everyone to load data representing transaction 1

crash after sending vote request but before receiving any replies:

start

newflight,1,1,1,1

newcar,1,1,1,1

newroom,1,1,1,1

newcustomerid,1,1

commit

start

deleteflight,2,1

deletecar,2,1

deleteroom,2,1

deletecustomer,2,1

mwdieaftervoterequest

commit

result: Some RM's may have made before and after copies, some may not have. The middleware upon recovery will decide to abort and tell all nodes to load the copy representing transaction 1.

RM crash receiving vote request but before sending answer:

crash after sending vote request but before receiving any replies:

start

newflight,1,1,1,1

newcar,1,1,1,1

newroom,1,1,1,1

newcustomerid,1,1

commit

start

deleteflight,2,1

deletecar,2,1

deleteroom,2,1

deletecustomer,2,1

rmdieafterrequest, flight

commit

result: MW would have made the decision to abort as it has not received all the votes, thus all the correct nodes would not commit. Upon recovery of flight rm, it would ask the mw for the decision and also load the image representing transaction 1.

RM crash after sending decision:

crash after sending vote request but before receiving any replies:

start

newflight,1,1,1,1

newcar,1,1,1,1

newroom,1,1,1,1

newcustomerid,1,1

commit

start

deleteflight,2,1

deletecar,2,1

deleteroom,2,1

deletecustomer,2,1

rmdieaftersendanswer, flight

commit

result: mw would have made the decision of either commit or abort, and all the correct nodes would have loaded the appropriate image. Upon recovery of the flight rm, it would ask the MW for which copy to load and do so.