Lessons from a massive Distributed System Application: Travel Reservation System

Nirmal Kanagasabai, Akash Singh
School of Computer Science
McGill University
{*nirmal.kanagasabai, akash.singh*}@cs.mcgill.ca

Introduction

The goal of the project is to design and develop a component-based reservation system. System supports multiple client and multiple server Travel reservation system via RMI and part of it in TCP as well. RMI results and methodology is mentioned in part 1 whereas TCP is mentioned in Part 2 of this report. In all there are three Resource Managers (RM) and a middleware server. The middleware can connect to multiple clients. In addition, lock manager is implemented for proper data access during various transactions. Shadowing was used for persistence of data items in various resource managers. Further, this report also explains the implementation of crash management system. The report will cover all these features in addition to performance analysis.

Part 1: Remote Method Invocation

Objective

Pass remote objects over RMI for communication between clients and servers. The way different layers work in RMI is reflected in Figure 2. In Java, using its RMI (Remote Method Invocation) facility, a piece of code running on one machine can invoke a method located on another machine. In such a method call, the machine containing the calling code is referred to as the local machine and the machine containing the called method as the remote machine. Such a method call is alluded to as remote call and such a method as remote method or remotely accessible method. Also the object containing such a method is called remote object or remotely accessible object.

RMI Methodology

(Pitt and McNiff 2001) The application layer sits on top of RMI system. RMI consists of three layer: stub/skeleton, remote reference layer, and transport layer. Building an RMI application superficially requires the following steps: 1. Define the interfaces, 2. Implement the server, and 3. Implement the Client. Running a RMI application needs the following steps:

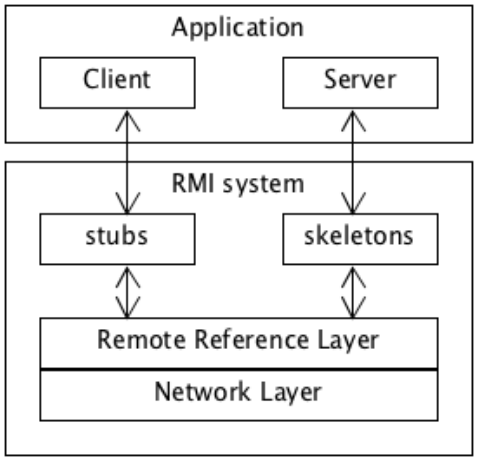

Figure 1: Remote Method Invocation system diagram.

1. Run the resource manager for car, hotel and flight, 2. run the server, and 3. run the client. Stub is the representation of the server object on the client. Stub is the proxy for the remote object. When a client makes a call on the remote object, the stub then passes that request to the RRL layer. RRL decides if the server is unicast or multicast. RMI is a unicast server. Unicast means that you cannot replicate the object. "Remoteref" in RRL contains a method that is invoked by the stub to pass across the request to the server side. When a client makes a request on an object. The methods could be taking parameters in the form of primitive types or object type or remote object type. If the parameter is of primitive type, then stub performs marshling, meaning putting together parameters and add the header. then pass it to RRL via remoteref. If its an object, then it cannot send an object across JVMs, we need to serialize the object and send it across. so, during the marshaling it serialize the object. If it is a remote object, then it has to serialize the stub object and passes it to RRL via remoteref. Skeleton is responsible for unmarshaling.

Project Description and Methodology

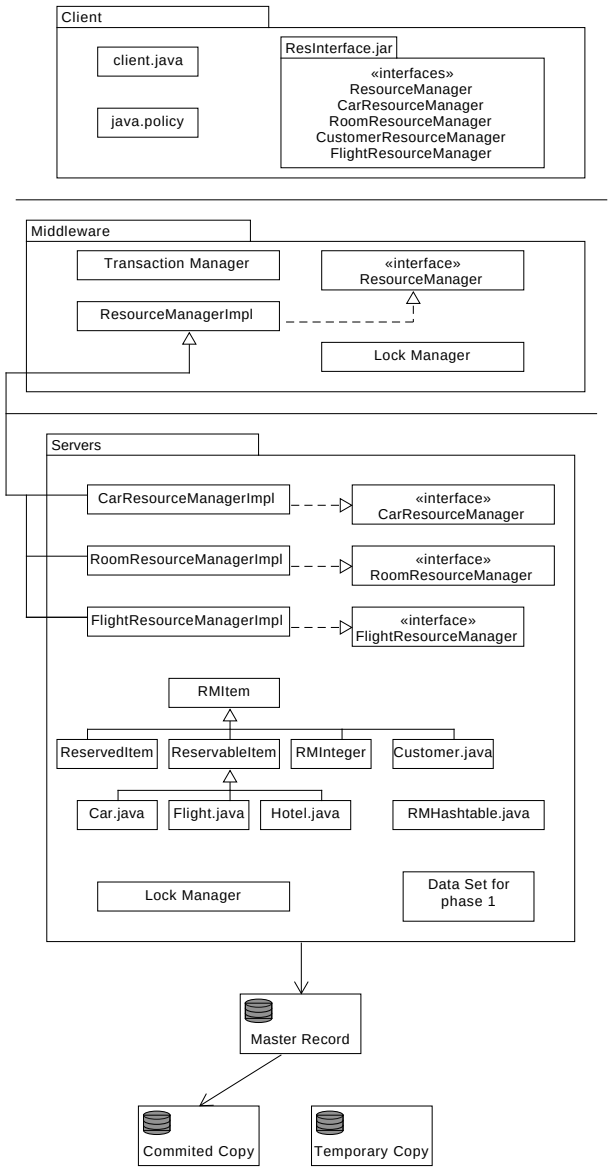

Figure 2: RMI complete package diagram.

Flight, car, and room reservations are handled by three different server $Group23\_Car$, $Group23\_Room$, and $Group23\_Flight$. Customers are handled inside the middleware because the ItemManager interface doesn't map well to customers as the middleware does not implement that interface. Consequently, customers cannot be handled by the same code that handles the other resource managers. This causes code to be duplicated in many place to handle customers properly.$Group23\_ResourceManager$ acts as the middleware server providing interface for the all the functionality of the servers. The link between the middleware and the resource managers is meant to convey that the resource manager does not always know about the middleware; only when it needs to recover from a crash do we give it a reference to the middleware, so that it can perform a query to know how what to do with transactions inside the persistent working set. All these are individual servers deployed either on different machines or on same machines but different ports. The client, however, does not see any of this distribution, instead interacts with the middleware for all the requests. The middleware server provides an additional functionality of *itinerary*, which books a set of flight, a car and a hotel at the final destination. This is a high level operation associated with the middleware server. The items to be registered aer synchronized, such that they can receive concurrent request for registration. This has been handled in the middleware server.

Test cases

The following operations were tested:

- newflight
- newcar
- newroom
- newcustomer
- newcustomerid
- deleteflight
- deleteroom
- deletecustomer
- queryflight
- querycar
- queryroom
- querycustomer
- queryflightprice
- querycarprice
- queryroomprice
- reserveflight
- reservecar
- reserveroom
- itinerary
- start transaction
- commit transaction
- abort transaction
- crash resource manager
- crash middleware
- recover resource manager
- recover middleware

The above evaluation was performed using 1 to 3 clients instances running on different servers separate from the middleware node, which was running on a server separate further from the hosting location of the RMs. The tests were conducted using machines in the lab 2 of the Trottier building. Following are some of the components of this simulation:

Lock Manager

Implementation of lock conversion tasks: LockManager.java is modified to implement lock conversion (from readonly lock to read-write lock). The two methods we had to modify were Lock and LockConflict.

- If we already have a write lock and we are requesting for a WRITE lock on the same data then throw an exception.
- If we already have a READ lock, we count how many dataObj have a lock on the same dataName. If there is only one, that means this transaction is the only one with a lock, and we can upgrade. If the count is greater than one, it means at least one other transaction has a lock, and therefore we can not do the lock upgrade.
- In Lock, if the bConvert bit is set, we remove the TrxnObj and DataObj from the manager and put in objects with the same properties, except the LockType has been changed from read to write.

In LockConflict, if the transaction wants a write lock on a data object and that this transaction already has a read lock, we set the bConvert bit to true. If the transaction already has a write lock on the data object, we throw a RedundantLockRequest exception. Every resource manager has a LockManager instance to control access to its resources.

Transaction Management

Transaction management is performed by cloning the hashtable in the main memory after checking the validity of transaction. Once the transaction is complete, it checks for conflicts with the locked objects, and aborts or commits the transaction. This is done by creating a TestData in the local without performing the operations directly to the HashTable. A TestData contains three concurrent hash maps for storing:

- vector of commands,
- transaction ids to locations, and
- mapping of the current state of the object

Transaction manager is also the supervisor for two-phase commit protocol. A table mapping transaction IDs to their two-phase commit status: NonCommitted, Phase1, or Phase2. On the contrary, TransactionManager class is implemented as a singleton class contained within the ResourceManager. Every time a transaction is instantiated, the following information is maintained in three different concurrent HashMaps:

- vector of RMs associated with a transaction,
- time of creation for the transaction, and
- status of the transaction

TransactionKiller class is responsible for checking the time-to-live values and if it notices a transaction being idle for more than 25000ms, it aborts the transaction.

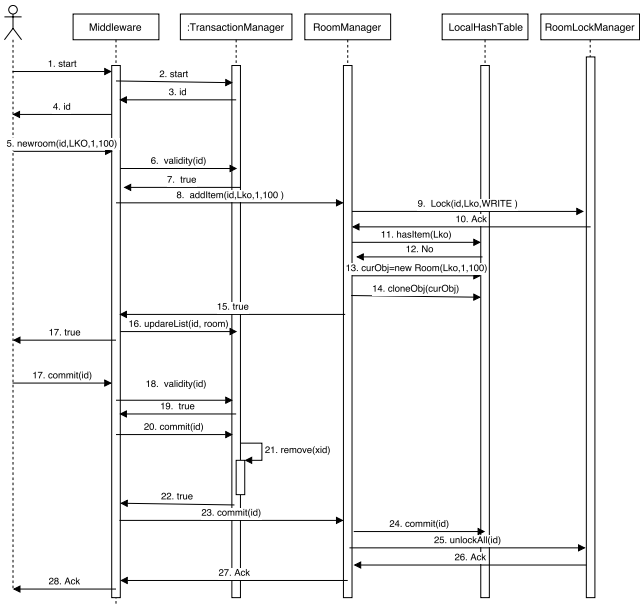

Figure 3: Sequence diagram for transaction process in RMI.

Crash Implementation

Crashing a Resource Manager When a resource manager crashes, either its working set has been saved to disk ./'pwd'/Group23/[RM].ws or it isn't. If the data is not present in this directory then no recovery is possible and the transaction will be aborted. If it exists on disk, we load it into memory, and we initiate the recovery procedure. When a resource manager is started in recovery mode, it has a reference to the middleware, and will ask it what to do (i.e. abort or commit) with every transaction in the working set. Once the recovery procedure is completed, the resource manager works as usual. The following cases are mentioned below and also shown in Figure 4

- **Crash before saving working set:** no recovery is possible for this participant, and other participants in the transaction are instructed to abort.
- **Crash after saving working set:** When the resource manager restarts, it'll query the middleware to know what to do with the transactions. Because the RM hadn't sent its response, the middleware aborted the whole transaction, and thus the resource manager will abort.
- **Crash after sending response:** The resource manager will query the middleware, and the middleware will tell it whether it has to commit or abort the transaction.

Middleware crashes If the middleware crashes during recovery, it'll re-send the decision to all the resource managers. Again, on the resource manager side, the effect of receiving a duplicate decision is benign; if the

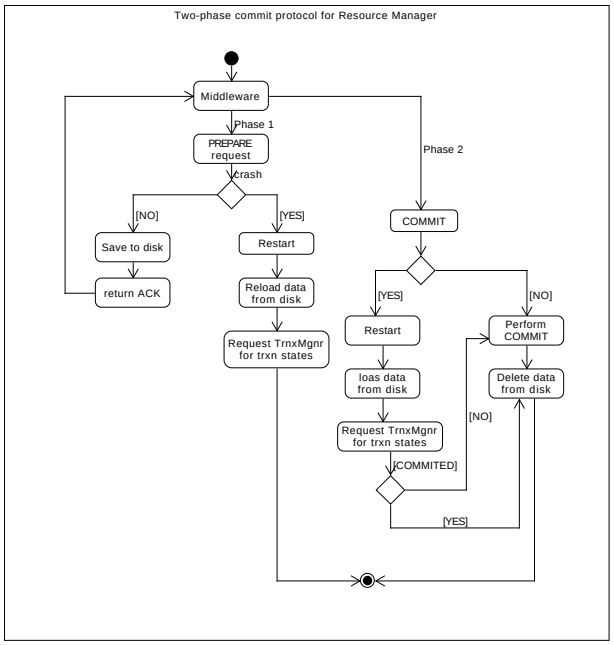

Figure 4: Activity diagram for resource manager crash cases.

transaction had already been processed, the new decision will be ignored, otherwise it'll be processed.

Performance Analysis

For the performance analysis, a client program was built to submit requests in a loop. Two different tests were carried out namely, Single Client analysis and Multiple Client analysis. The response.

Single Client Test In case of single client, as there was no concurrency in the system, there was no need for the threads to sleep between the transactions. Two different transactions types where chosen for this study. The first transaction type involved only one Resource Manager and the second transaction type involved all the Resource managers. In each of these transaction types, the number of operations that were performed was the same. This was done to ensure that they are comparable in overhead.

The 8 operations that were used in the single client study are: AddCustomer, AddFlight, 4 * QueryFlight, ReserveFlight and DeleteCustomer. The 8 operations that were used in multiple client study are: AddCustomer, AddFlight, AddCars, AddRooms, QueryRooms, QueryCars, ReserveItinerary and DeleteCustomer. The response time of these two cases transaction types for the single client was studied against the number of transactions involved. It can be seen in Figures 2 and 3.

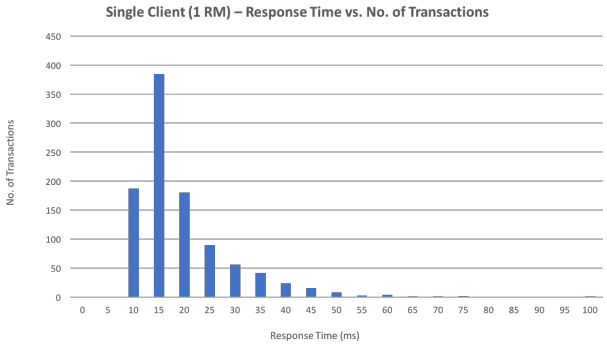

Figure 5: Single Client (1 RM) – Response Time vs. No. of Transactions.

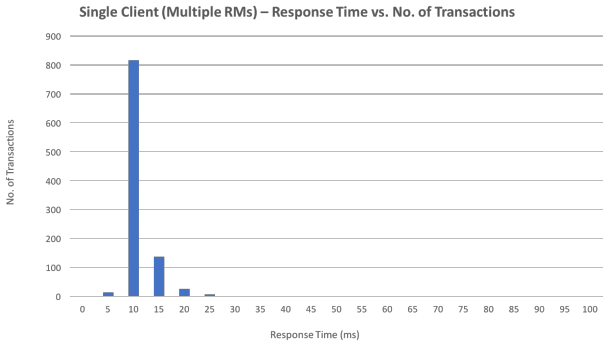

Figure 6: Single Client (All RMs) – Response Time vs. No. of Transactions..

Multiple Clients Test In case of multiple clients, 5 clients were considered for the study. The operations involved in the transactions include: AddCustomer, AddFlight, AddCars, AddRooms, ReserveFlight, ReserveCars, ReserveRooms, etc. The response times for each of these clients are studied against the number of transactions and can be seen in Figures 4 to 8.

Behavioural Analysis

From the graph 2 and 3, it can be interpreted that, lesser the response time, more are the number of transactions that are performed. In case of Single Client (1 RM) transaction type, a majority of the transactions (752 transactions out of 1000) was executed within the range (10 to 25 ms) constituting about 75% of the total number of transactions under study. A mix of both read and write operations were performed in this study. In case of Single Client (All RMs) transaction type, it was surprising to note that the response time for 816 transactions(81.6%) was 10 ms. This was lesser than the average of Single Client 1 RM analysis. In order to explain this phenomenon, we decided to study the operations we used within the transactions.

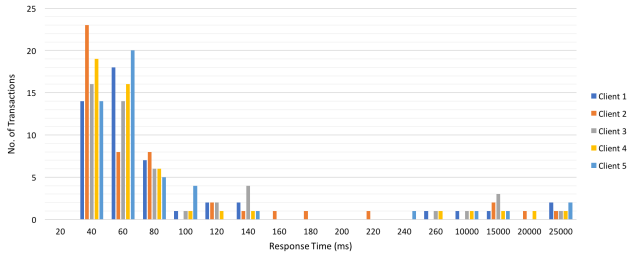

Figure 7: Multiple Clients (All RMs) – Response Time vs. No. of Transactions..

Table 1: Single Client (1 RM and All RMs) - Average Response Time vs. Type of Operations

Type_Of_Operations	1 RM	All RMs
Read-Only	7.184	8.295
Write-Only	12.984	10.273
Both	14.931	16.146

All those operations which we considered for the Single Client (All RMs) study was write-only (Add/Reserve) operations. We decided to perform different set of operations - Read-Only, Write-only and both Read and Write operations. The response times for these operations on the different transaction types used in Single Client study can be observed in Table 1. It is really surprising to see that when write-only operations are performed, accessing all the 3 RMs is faster than accessing only one. In other cases, the average response time for single RM was relatively lesser than the 3 RMs.

While investigating the response times we obtained, one interesting observation was that the system performed better (in executing the commands quickly) after the first few transactions. Initially, few transactions took longer (say 20 to 25 ms) before reaching the numbers usually below 15 ms. We speculate this change in behaviour to be because of the Java Just-In-Time Interpreter which starts optimizing the operations after executing them for quite a number of times.

In case of multiple clients, the same graph, response times vs. no. of transactions for each of the clients were plotted. It could be observed that, a majority of transactions are performed in the 40 to 60 ms range. This response time is higher compared to the response times obtained by Single client study. The possible explanation for this increase in response time is the introduction of concurrency control mechanisms in the system. Ideally, there are multiple clients competing for the same resource and there is also a waiting time for the transactions for the resource to be released by the other transactions holding a lock on it.

Few transactions starved for the resource longer than others and was aborted as it was inactive for a long time. And few of them happened to be in a deadlock and was aborted. This explains the outliers in Figure 4 where the response times (10000, 15000, 20000 and 25000) had few transactions plotted against them.

From the above graphs, it can be safely inferred that the number of clients had a big impact on the system. For future works, a study on the response times vs. no of transactions where more number of clients (say, 10, 50, 100, 500 and 1000) with different sets of operations (Read-only, Write-only and both Read-and-Write) needs to be done to draw concrete inferences about the performance.

Part 2: TCP/IP

Objective

The objective of this exercise is to distribute the reservation system using TCP Sockets. TCP, which stands for Transmission Control Protocol, is a connection-oriented service which offers reliability as the clients and servers are connected throughout.

Design Considerations

Typically, a TCP client, after sending a request to the server, expects an acknowledgement in return. Meanwhile, the server processes the request and responds to the client. However, for the sake of simplicity and practicality, there will only be one acknowledgement back to the client and that would be the service offered. Also TCP, by default, is a blocking protocol. After the client has sent a request, until the request is processed, it is not allowed to send another request. We will retain the default nature of TCP at the client-middleware interaction. However, in the middleware-RM interaction, there will be a TCP's non-blocking nature that will be established.

Architecture

In this implementation as seen in Figure 3, we decided to implement a middleware which will receive the requests from multiple clients. It then decides to which RM the request needs to be sent to and then forwards it accordingly. Also, it gets back the response from the RM and forwards it back to the corresponding client who requested for a service.

The client and the middlewareServer classes will have the Socket implementations which facilitate the communications between them. Likewise, the connection between the middlewareServer class and the RMs is established by creating a new RMManager class. The RMManager, similar to MiddlewareServer class, establishes a ServerSocket in it.

After the ServerSocket is established in MiddlewareServer, it awaits new connections. As soon as

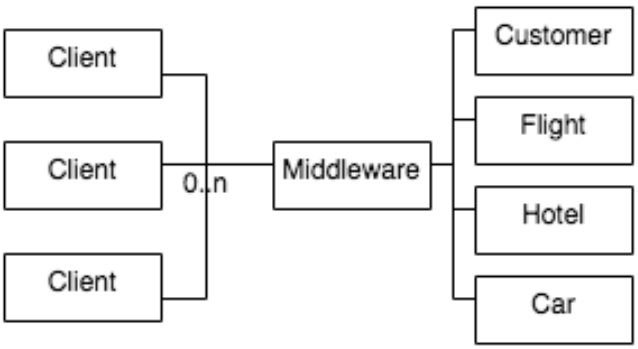

Figure 8: TCP Sockets - Simple Architecture

a new connection is established, MiddlewareThread class takes care of further processing. For every client that gets connected to the Middleware, a new thread is created. Likewise, the serverSocket is established in the RMManager and whenever a new connection is established, the ResourceManagerImpl class is run. This has been made to implement Runnable for facilitating thread handling.

As per the project skeleton, we had three RMs Flight, Hotel and Car. In addition to that, we decided to implement Customer as our fourth RM (as seen in Figure 4). This will not only reduce the functionality on Middleware but also helps us in easily manage the client requests. The burden of keeping track of the customer details will be taken off the middleware making it perform only the minimal functionality. This was done with an idea to make the architecture more flexible for new module additions in between the client and RMs in the future.

As per the current implementation, with every request from the client that involves querying/addition/removal/any other operation on a customer will first be forwarded by the middleware to the customer RM to check the presence or absence of an entry following which the corresponding operation on the RMs will be implemented.

Test Cases

The following operations were tested: 1. Creating new customer
2. Querying a customer who exist
3. Querying a customer who doesn't exist
4. Deleting a Customer
5. Reserving a customer who doesn't exist
6. Creating new items (Flight, Car, Room)
7. Reserving Items (Flight, Car, Room) that exist
8. Reserving Items (Flight, Car, Room) that does not exist
9. Deleting Items
10. Wrong input of arguments
11. Itinerary

Special Features

In addition to the above mentioned implementation, the only thing that our system was lacking was a proper client. The Console-only client was hard to use and the users were expected to key-in the commands every time they wanted to access the system. Also, there weren't any validation checks done at the client side. Every input the user keyed-in, was taken up and then the errors were thrown. For example, for Flight / Car / Room Price, the ideal value that was expected was an integer. If an user typed a string, an error would be thrown and he was expected to re-do the entire typing process.

As a work-around, we decided to build a Graphical User Interface (GUI) for our client. While creating a GUI, we decided to have a 'rich' client which can carry out validation checks and prompt users as and when it is needed. Also, to avoid re-doing the entire step if there was a mistake in the one of the parameters, a small caching mechanism was introduced. This would have the user's last state where from he/she can begin to work with the client. i.e., if an user, who is trying to reserve an itinerary, happened to select all the parameters correctly and entered an incorrect parameter for the number of cars to be reserved, he needn't re-do all the work and just enter the number of cars and reserve an itinerary. The client was built using Java Swing AWT and would automatically connect to the Middleware when a new instance is opened. A snapshot of the client can be seen in Figure 9. New operations for Crashing had been included thereafter.

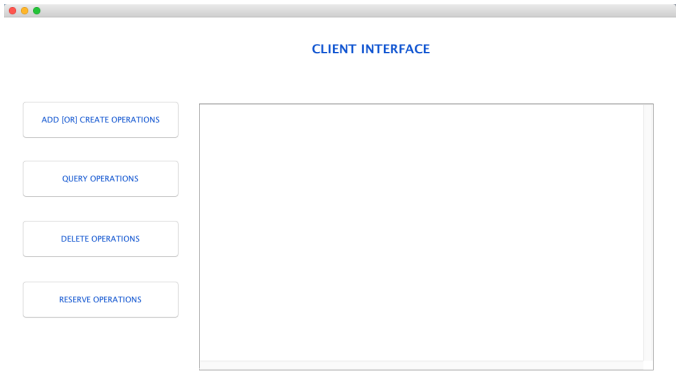

Figure 9: Sample GUI Client

References

Pitt, E., and McNiff, K. 2001. *Java. rmi: The Remote Method Invocation Guide.* Addison-Wesley Longman Publishing Co., Inc.