PG - 1
COMP-512: Distributed Systems, Fall 2017

Distribution, Transaction Management & Replication

Shabir Abdul Samadh
McGill University
Student ID: 260723366
shabir.abdulsamadh@mail.mcgill.ca

Jianhua Li
McGill University
Student ID: 260351181
jianhua.li2@mail.mcgill.ca

ABSTRACT

This report provides an overview of the architecture and design choices related to building a complete distributed system from its standalone version. The report covers 3 major phases of the complete build-up:(1) Distributing the standalone `(single client-server model)` into a ***multi-server*** setup; (2) Enabling ***transaction support*** with *lock management*; (3) Ensuring *fault tolerance* with ***replication***. The application used here is a simple reservation system that allows the user to add, remove, update & query information related to its first class entities. These entities are `Cars, Flights, Hotels (Rooms)` and `Customers`. The standalone application provides a simple **interface** along with a **client** program and a corresponding **server** implementation and it uses an **RMI** registry to enable the client to invoke methods on the server. The initial step of distributing a stand-alone application was experimented using two different communication paradigms: **TCP Sockets** & **RMI Calls**. We have distributed this standalone system in such a way that each first-class entity of the system `(i.e.Car, Flight, Hotel & Customer)` are managed by separate instances of the server. We have bridged these separate instances via a **Middleware Server**. Given that the `Customer` is common and is related to all three other entities, we have managed customer related processing in our middleware server. Going forth from the initial phase we have based our development only on **RMI** based setup. The system is completely written in `Jave`.

Keywords

Distributed-Systems; Middleware; TCP Sockets; RMI; Transaction Management; Replication; Lock-Management

1. DISTRIBUTED SETUP

1.1 Introduction

The report describes the development of a distributed architecture from a stand alone application that has a **server** component and a **client**. These two components are glued together by a common **interface**. The server provides the implementation of all the methods in the **interface** whereas the client uses the interface as the stub to invoke the implemented methods on the server. The default setup uses an **RMI-registry** allowing the server to register its implementation of the interface with this locally run registry. The server, on startup, creates an instance of the *interface-implementation* and registers this instance against some pre-agreed reference on the **registry**. The client on the other hand contacts the **registry** to obtain the reference to the server object *(which is an implementation of the common interface)*. It then invokes methods on this object as if it was on the local *JVM*.

The application is a simple reservation system that has `Cars, Flights, Hotels(Rooms) & Customers` as first-class entities. The current implementation handles all information and logic related to all of these entities in a single server; in a common data-structure that holds data using an abstract object type. The server implementation provides methods related to all these entities. Thus, calls from multiple clients are handled & stored centrally by this single server. Our work has been focused on extending this setup into a distributed deployment based on two specific technologies available.

1.2 Overall Architecture & Design

We extend this simple **RMI** based application into a distributed setup. We do this by recreating the server implementation per entity. That is, logic & information related to `Cars, Flights and Hotels` are handled by 3 separate server instances. We call these separate server instances as **Resource Managers(RM)**. Each of these RMs only maintains data related to a specific entity. In addition, we deploy a **Middleware(MW) Server** between the clients and these RMs, to manage & distribute calls from multiple clients. We have moved all processing and storage related to the `Customer` entity to the middleware-server, given that it is a common class that relates to all of the other 3 entities. We also have implemented an additional method **reserveItinerary()** that allows `Flights, Cars & Hotels` to be booked against a single `Customer`. We have developed this distributed deployment based on two implementation paradigms: **TCP Sockets** based communication and message exchange *Vs* **RMI** based remote object-method calls. However, it is noteworthy that the way we handle `Customer` information is different in the RMI based implementation. In the RMI based distributed setup, the `Customer` details are duplicated in each of the RMs to overcome response delays for client requests. It is further explained under the section for **RMI** implementation.

1.2.1 Client to MW & MW - Server Communication

In both implementation methodologies *(TCP and RMI)* all client to MW server communication are blocked until the MW server returns with a response. However, calls from the MW server unto the RMs are handled in a non-blocking

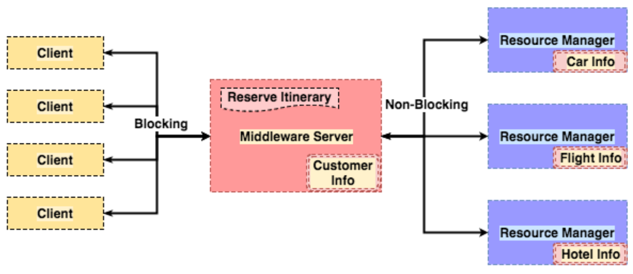


Figure 1: Deployment Architecture of the system

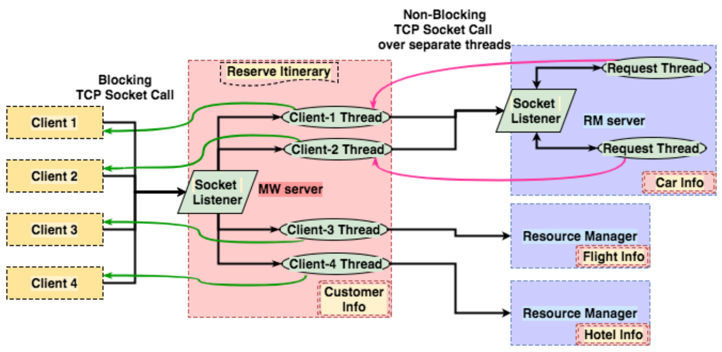


Figure 2: Architecture using TCP sockets

fashion. This enables concurrent requests from multiple clients to be handled with equal priority without having to be queued. That is, even though the call from the clients are blocked by the MW server, all such requests are forwarded to the RMs immediately and are handled separately and in-parallel by the RM. This is achieved by handling each client request on a separate thread within the MW server and the RM. Also, **explicit locking** *(in the case of RMI)* and **implicit locking** *(by means of Java ConcurrentHashMaps in the case of TCP)* are used to ensure thread safety during concurrent updates to the data-structures.

1.3 (R)emote (M)ethod (I)nvocation

The RMI implementation is rather simple since Java RMI is already multithreaded, so we don't need to take care of that aspect ourself. We chose to use the three resource managers to manage customers data. In each RM, we keep the same data about customers, except informations about their reservations. One type of RM only keeps its own type's reservation informations. For example, in *Car's RM*, we only keep a customer's car reservations. This implementation requires some details to be taken care of. First of all, when we add a customer, we need to add it in all three RMs, each with the exact same customer ID. In order to achieve this, when adding a new customer, we call *newCustomer(id)* to one particular RM first, upon sucessful execution, it will return a new unique customer ID; then we use this customer ID to add the same customer into the other two RMs using the function *newCustomer(id, cid)*. Same goes for deleting customers, ie. deleting in all three RMs. When we need to reserve an itinerary. We first reserve the flights with the *Flights* RM; if necessary, we then reserve the cars and rooms with their respective RMs. When querying customer's informations, again we have to query all three RMs then concatenate the returned strings into one string.

1.4 TCP based implementation

Figure 2 shows the distributed setup implemented using **TCP Sockets**. The implementation is such that all calls from the client to the MW server are blocked. However, calls from the MW server to the RMs are all executed immediately without having to be queued. This is achieved by handling each client request in a separate thread from within the MW server. The RMs also forks a new thread per client request to ensure incoming requests are addressed immediately and concurrently. This way we ensure that even-though requests from clients to the MW server are blocked, any number of client requests are forwarded to the RMs *(from the MW server)* immediately and are processed in parallel.

1.4.1 *MW Server & Customer Information*

The middleware server acts as the middle-man between the clients and the RMs. Whilst, we maintain 3 separate RMs for each of the entities - `Cars, Flights & Hotels`, our **TCP** based design holds information related to the `Customer` at the MW server. We have made this design choice given that the `Customer` entity is related to all of the other entities whereas they individually are independent of each other. Moreover, calls for any of those 3 entities could only be related to the `Customer` entity. Hence, maintaining such a common attribute at a central location via which all communication flows through makes the setup efficient. The other design choices of either having a separate RM for the `Customer` entity or having the customer information duplicated in all 3 RMs would both introduce a high performance penalty as a result of increased back-and-forth communication. Either of those two setups will necessitate more than one network call to separate servers for every client request that is a combination of `Customer` and one of the other 3 entities. The MW server also provides an additional functionality - **reserveItinerary()**. This function enables a combination of all three entities `(Cars, Flights and Hotels)` to be booked against a single customer. Upon receiving a call for this function, the MW server distributes the call to all 3 RMs to make the booking and finally stores this information in the data-structure holding customer information *(maintained by the MW server)*.

1.4.2 *Message representation and flow*

An important design choice in terms of the **TCP** based implementation is how the messages are communicated over sockets. We needed to have a generic representation that is understood by all three parties involved in the communication: *Client, MW server & RM.* Given that all the components of the system are written in `Java`, we came up with a generic `Java` class representation for the request and response messages. We implemented two `Java-Serializable` classes: `RequestMessage & ResponseMessage`[1]. We have also introduced a `MsgType` *enum* to denote the type of request message being sent. The `RequestMessage` class carries an attribute of the *enum* type `MsgType` as an indication of the type of request incoming from the client. These 3 shared message representation models `(RequestMessage, ResponseMessage & MsgType)` are implemented as part of

[1]The class implementation can be found in *Appendix*-A

a common library named **interface**, which is used by the client, MW-server and the RMs.

For every request that the client sends, it wraps the request in an instance of the `RequestMessage` class and sends it over the socket channel to the MW server. The client uses the `MsgType` attribute of the `RequestMessage` object to set the correct request-type. The MW server assigns a new thread per new client-connection immediately. The assigned client thread *(denoted by the class `MiddlwareRequestHandler`)* starts by de-serializing the incoming bytes *(over the socket)* into an instance of the `RequestMessage` object. Then it checks the type of the request.

- If the request is related to `Customers` then it is processed by the MW server itself;
- If it is a request specific to any of the 3 entities *(`Car, Flight` or `Hotel`)*, then it is forwarded to the corresponding RM;
- If it is a reserve request for a specific entity *(`Car, Flight` or `Hotel`)*, then it is initially processed by the MW-server to validate customer, then is forwarded to the corresponding RM and finally *(upon receiving a response from RM)* the customer is updated with the reservation info;
- If it is a `reserveItinerary` request, then the MW-server first validates the customer, then forwards request to the `Flight-RM` to reserve the flights, then upon receiving a response checks if a car and a hotel needs to be booked. If **YES**, then the request is forwarded to the `Car-RM` first and upon receiving a reply from it, the request is forwarded to the `Hotel-RM`.

The final processing point of the incoming-request *(either the MW-server or the RM)* wraps the results in an instance of `ResponseMessage` and sets **status** attribute to either `1:SUCCESS` or `0:FAILURE`. If the response was generated at the RM then the response message is received by the MW-server, a *status message* is added to the response and is forwarded to the client. This flow is depicted by Pseudocode 1.

1.5 Testing

Both implementations *(TCP & RMI)* were tested for the correct execution of all the *Client Interface* methods. They were also tested against cases where these interface functions could be called erroneously. *Deleting a non-existing `Customer`, Querying a non-existing `Car, Flight` or `Hotel`, Trying to reserve an entity which has 0 availability* are some such erroneous invocations which were tested. However, the two most important test-cases discussed below.

1.5.1 Concurrent client request handling:

This test ensures that the MW-server and the RMs immediately serve client-requests that arrive late, even-though an earlier client's request is still being processed. This test was setup as follows:

- each client sends it's `Client ID` along with its requests
- the MW-server and the RMs are configured to `sleep` on the requests from clients with specific `Client IDs`.

So, this way we setup certain clients as long-running-request clients and the rest as normal. We were able to to validate that even-though the MW-server or the RMs were sleeping on requests from these clients, all other requests were responded to immediately without delay.

1.5.2 Reserve Itinerary:

This test validates the correct execution of the `reserve itinerary` function. That is making sure that the MW-server carries out the booking of `Flights`, `Cars` & `Hotels` in order and a failure of one results in reverting the booking all-together. We carried out this test by issuing *itinerary* requests with different combinations of the 3 entities and with/without some of the entities in the itinerary. We were able successfully execute all such test scenarios whilst preserving the correctness of the data.

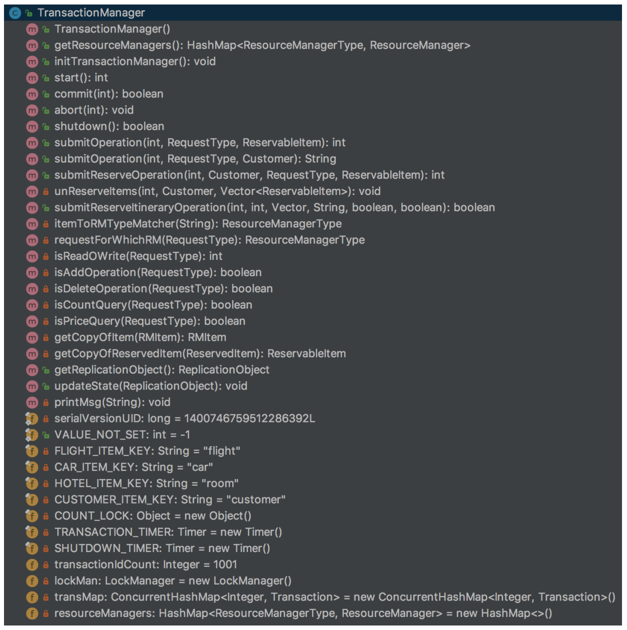
Algorithm 1 Message Processing by MW-Server

```
1: procedure RECEIVE_REQUEST_MESSAGE
2:     msgType ← RequestMessage.getMsgType
3:     response_message ← new ResponseMessage()
4:
5:     if msgType.equals(Customer_Related) then
6:         response_message ← processMsg();
7:
8:     if msgType.equals(Entity_Related) then
9:         response_message ← callCorrespondingRM();
10:
11:    if msgType.equals(Reserve_Request) then
12:        validateCustomer();
13:        response_message ← callCorrespondingRM();
14:        reservationInfo ← response_message.info;
15:        updateCustomer(reservationInfo);
16:        response_message ← generateResponse();
17:
18:    if msgType.equals(Iteinerary_Request) then
19:        validateCustomer();
20:        flightInfo ← callFlightRM();
21:
22:        if bookCar then
23:            carInfo ← callCarRM();
24:
25:        if bookHotel then
26:            hotelInfo ← callHotelRM();
27:        updateCustomer(reservationInfo);
28:        response_message ← generateResponse();
29: return response_message
```

2. TRANSACTION MANAGEMENT

The experimentation on the two communication paradigms (**TCP** & **RMI**) proved that the **RMI** based approach seemed easy in terms of having to not worry about message representation and parsing. Also, there was far less, communication handling to be done given that the *RMI* libraries takes care of it. Hence, we decided to stick with **RMI** for the next phases of the project. Even-though we were moving forward with the **RMI** approach; when it comes to handling transactions, we wanted to stick with the approach of having a common `customer` store. Hence, the `customer` information was moved to the **middleware** - kept common to all **RM**s.

With *transaction* support implemented, the client requests are grouped together as separate transactions. Each transaction is identified by a `TransactionID` and its life is determined by `start()` & [`commit()` or `abort()`] invocations of the client. The `TransactionManager` class keeps track of all the current transactions. All invocations made *(by the clients)* to the **Middleware** interface implementation of the *middleware-server* is routed to the *TransactionManager* who then takes care of how to handle it.



```
TransactionManager
    TransactionManager()
    getResourceManagers(): HashMap<ResourceManagerType, ResourceManager>
    initTransactionManager(): void
    start(): int
    commit(int): boolean
    abort(int): void
    shutdown(): boolean
    submitOperation(int, RequestType, ReservableItem): int
    submitOperation(int, RequestType, Customer): String
    submitReserveOperation(int, Customer, RequestType, ReservableItem): int
    unReserveItems(int, Customer, Vector<ReservableItem>): void
    submitReserveItineraryOperation(int, int, Vector, String, boolean, boolean): boolean
    itemToRMTypeMatcher(String): ResourceManagerType
    requestForWhichRM(RequestType): ResourceManagerType
    isReadOWrite(RequestType): int
    isAddOperation(RequestType): boolean
    isDeleteOperation(RequestType): boolean
    isCountQuery(RequestType): boolean
    isPriceQuery(RequestType): boolean
    getCopyOfItem(RMItem): RMItem
    getCopyOfReservedItem(ReservedItem): ReservableItem
    getReplicationObject(): ReplicationObject
    updateState(ReplicationObject): void
    printMsg(String): void
    serialVersionUID: long = 1400746759512286392L
    VALUE_NOT_SET: int = -1
    FLIGHT_ITEM_KEY: String = "flight"
    CAR_ITEM_KEY: String = "car"
    HOTEL_ITEM_KEY: String = "room"
    CUSTOMER_ITEM_KEY: String = "customer"
    COUNT_LOCK: Object = new Object()
    TRANSACTION_TIMER: Timer = new Timer()
    SHUTDOWN_TIMER: Timer = new Timer()
    transactionIdCount: Integer = 1001
    lockMan: LockManager = new LockManager()
    transMap: ConcurrentHashMap<Integer, Transaction> = new ConcurrentHashMap<Integer, Transaction>()
    resourceManagers: HashMap<ResourceManagerType, ResourceManager> = new HashMap<>()
```

Figure 3: TransactionManager

Figure 3 depicts the skeleton of the *TransactionManager*. It has a *map* of all (currently) alive transactions. Each transaction is denoted by a `Transaction` object. The *Transaction* object corresponding a transaction keeps track of the items **accessed** by that transaction.

2.1 Transaction Object

TTL.

The **T**ime **To L**ive metric of the transaction is initialized at the creation-time of the new transaction. It is set to be **2 minutes & 30 seconds** by defualt. The *TransactionManager* runs a periodic service *(every 5 seconds)* to update the TTL value for all transactions and to check for any transaction that has expired. If it finds any transaction that had lived passed its TTL, then it is aborted and is removed from the *map* of all transactions. However, whenever a client request is received for a specific transaction, then its TTL is reset to the default value.

accessedItems.

Each *Transaction* has a `<key, value>` pair **map** that keeps track of all accessed items by this particular transaction. Whenever, a new client request is submitted, the *TransactionManager* first checks whether the **accessedItems** map of this specific *Transaction* has the item involved with this request. If it does then the incoming request is processed on this item. If the item is not found in the *map*, then the corresponding *resource-manager* is contacted to fetch the item first, then is put into this *map* and finally the request is processed. Hence, during the lifetime of a transaction, any of the *resource-managers* is contacted for a specific item just once. Subsequent requests *(pertaining to the same transaction)* involving the same item is served using the item already fetched during its first-access. This ensures that all updates [`write() & delete()`] on all items happens on a copy of the item kept in the middleware. Thus, if the transaction is `abort()`ed at any time then, there is no change that needs to be propagated to the *resource-managers* since no data was updated at that end. The middleware copy of *accessedItems* is flushed and the aborted transaction is removed. Only when the client calls `commit()` on the transaction the items in this *map* are written back to the corresponding *resource-manager(s)*.

writeLists and deleteLists.

In addition to the *accessedItems map*, each transaction also maintains two lists (for writes & deletes) of **item-keys** for items which were either updated or deleted. Also these lists are maintains separately per *resource* (`car, flight, hotel & customer`). At the `commit()` time of the transaction, these lists are traversed and per key the corresponding item is fetched from the *accessedItem map* and is added to the resource manager.

2.2 Transaction Manager

2.2.1 Lock Manager

The *TransactionManager* also has a `LockManager` that is global to all transactions. This ensures that access-control for a specific item is maintained across multiple transactions that attempt to access this item.

2.2.2 Reserve Operation

When we call *submitReserveOperation(tId, customerItem, requestType, resourceItem)*, first we retrieve the *Transaction* object. Then we have to acquire the necessary lock for the operation. After getting the lock, first we check whether the customer has already been accessed within the same transaction. If so, we just directly use the customer object that's inside the *Transaction* object. If not, then we have to start a mini transaction where we try to get the customer object from customer manager which is inside the middleware, once we get the *customer*, we commit this mini transaction. Then we need to actually reserve the item for the customer, for which we have to call *submitOperation(tId, requestType, resourceItem)* first. Inside this method, it will also first check whether the item has already been accessed by looking inside *transaction.getAccessedItemSet()*, only has it not been accessed will it try to retrieve the reservable item from the resource managers. Eventually it returns the desired value so that we can proceed with the reservation.

2.2.3 Modify Customers

If we want to modify customers, the principle is similar to a reserve operation. We just have to call *submitOperation(tId, requestType, customerItem)*. We first check whether the cusomter has already been accessed. If so, we just access the customer object that is already inside the transaction. If not we make a mini transaction in order to get the customer object.

3. PERFORMANCE ANALYSIS

In terms of analyzing the performance of this system we considered two specific cases - load produced by *a single client* & by *many clients* concurrently. The performance metric that we were specifically targeting was the **R**ound **T**rip **T**ime *(RTT)* of the transactions introduced by the client(s). In addition our analysis was setup in such a way that the following scenarios were test for:

- *Read* operations on a single **R**esource **M**anager (RM).
- *Read* operations on multiple RMs.
- *Write* operations on a single RM.
- *Write* operations on multiple RMs.
- Transactions with a combination of *Read & Write* operations on a single RM.
- Transactions with a combination of *Read & Write* operations on multiple RMs.

For the case of testing the system on **multiple-concurrent-clients**, the transactions produced per client was a random mix of *Read & Write* operations.

3.1 Setup & Results

For the test scenarios given above, the *Read* operations were **randomly** picked on each iteration to be either *queryResource* or *queryResourcePrice*. For the write operations, they were picked to be one out of *addResource*, *deleteResource*, *reserveResource* or *reserveItinerary* at each iteration. Also for each test case some randomly generated resources were added to the RMs to ensure they are not empty during the test. Identifiers to the randomly generated items were kept locally by the **analysis-client**. The client randomly picks an item from this list to carry-out one of the operations discussed above. Also, the client was setup such that it randomly picks on which RM the operation is going to be performed in random. Thus, multiple runs of the *RTT* analysis would not result calculations biased to a single RM.

For the last 2 *(out the 6)* listed above, the client generates a random number between 1 and 10, which will denote the number of operations per transactions. Subsequently for each of the operation in the chosen number of cycles, the client randomly decides to do a *Read* or a *Write* operation. Finally, based on this decision it picks the operation also in random as mentioned above. For the *single client* scenarios we run each test for 100 loops and we vary the test by varying the transaction per sec **(load)** on the system. For the *distributed client* setup, we let each client submit 20 transactions *(per test)* on an interval decided by the load to be submitted to the systems. We also varied the number of clients between 10 and 40, to identify the impact of increased concurrent clients.

The figure above shows the RTT results for the 1st 4 cases listed earlier. It is noteworthy that the average RTT for *Writes* are higher compared to *Read* operations. This is directly intuitive by the fact that for *write* involved commits the *middleware* must contact the corresponding *RM*, whereas *reads* does not require the *middleware* to do any further processing. It addition we can also conclude from the above results that continuous *reads* on a single *RM* seems to

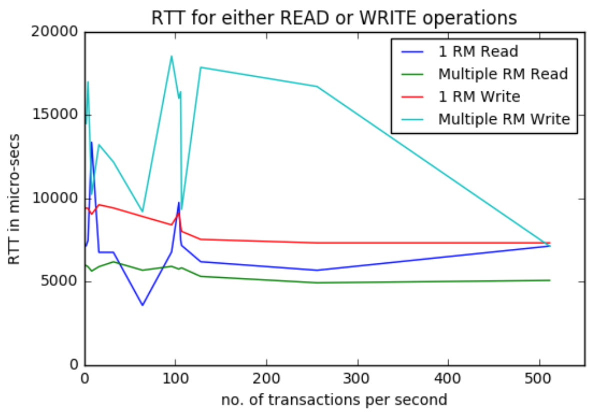


Figure 4: RTT for either Read or Write operations

be taking longer to respond that *reads* on multiple *RMs*. We believe that this primarily given that the load that each *RM* has to handle is distributed with *multiple-RM-reads* compared to *single-RM-reads*. However, this is not the same for the case of *writes*; *multiple-RM-writes* takes more **RTT** than *single-RM-writes*. This can be explained by the fact that with multiple *RMs* involved the proxy-stub invoked by the *middleware* is continuously changing for subsequent requests. Hence, we believe this does not allow the *JVM* to optimize the calls efficiently compared to the case of a *single-RM-invocation*.

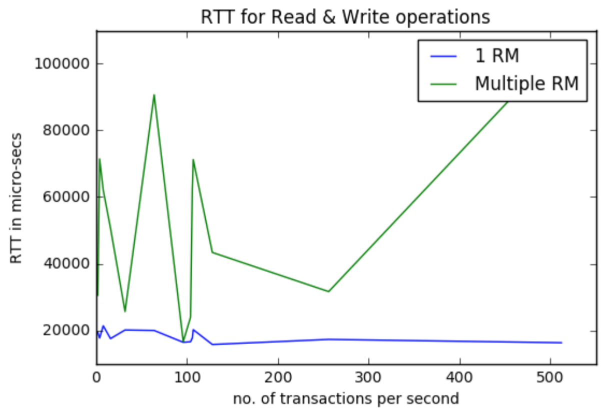


Figure 5: RTT for Read+Write operations

Figure 5 shows the **RTT** variation for transactions involving a random set of *Read & Write* operations. It was tested for two cases: *Read+Write* operations on (1) a single *RM* and (2) on multiple *RM*s. We can see that the average **RTT** is much higher when multiple *RM*s are involved. We can explain this by a similar deduction as before: multiple *RM* based transactions require interactions to varying *RMI-registry* **proxy-stubs**. This weakens the level of optimization that the *JVM* could provide to the system. However, with a single *RM*, the calls fall within a specific set of elements and the system could support this much faster & efficiently.

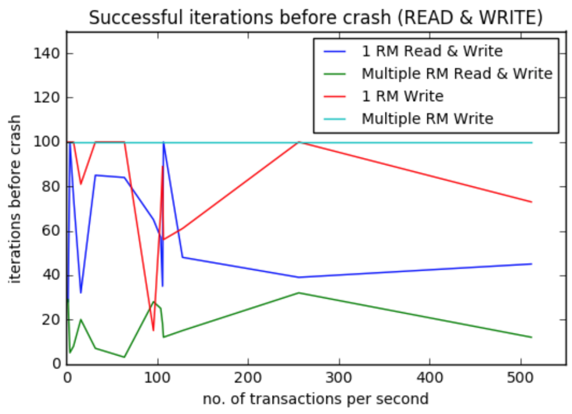


Figure 6: Successful iterations (Write & "Read+Write" operations)

Figure 6 is a depiction of *loop-counts* before **crash** *(or exception scenario)* for 2 cases discussed above: (1) *Write operations* and (2) *Read + Write operations*. It can be noted that the success-rate `(without failure)` is higher in the case of transactions involving only *write* operations compared to *read + write* combined transactions. Whilst this can be pretty intuitive by the fact that a combinations of operations will definitely result in a longer commit period. However, what is interesting here is that for *writes-only* test, *multiple RMs* seems to be handling crashes better than *single RM* scenario. On the other-hand for the *read-write combined* test, a *single-RM* performs better than with *multiple-RMs*. We believe to have introduced some sort-of error into our test resulting in what we observe.

Table 1: Concurrent Clients invocation

Load	10 clients	40 clients
1 t/sec	36 ms	58 ms
2 t/sec	33 ms	48 ms
10 t/sec	28 ms	22 ms

Table 1 shows our results for **RTT** for transactions when *concurrent-clients* produced load. We see that with increasing no of clients the **RTT** increases. However, we also note that for a fixed client number, increasing load sometimes reduces **RTT**. We describe this as an error we made in adding a `ZERO` to the average calculation for iterations throwing a `DEADLOCK`.

4. REPLICATION & FAULT TOLERANCE

Finally, we introduce *replication* to make the system fault tolerant. We create separate clusters per `server-type` (resource manager or middleware) and initiate a **master-node** per cluster. The **master** of the cluster acts as receiver of all requests. It then transfers this information to all the replicas in its cluster. We make use of the `JGroups` open-source library to manage the underlying group communication between the replicas of a cluster.

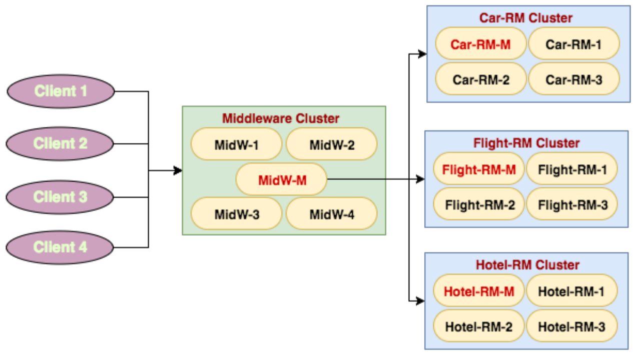


Figure 7: Cluster setup of MW & RMs

Figure 7 shows the architecture of how different clusters are setup. The nodes in a cluster that are denoted in **red** are the master-nodes. It is also noteworthy that the master of the *middleware-cluster* is attached to the clusters of the resource managers and also each client is part of the middleware cluster as well. The reason for doing this is to notify the *middleware-master & the clients* about the failure of the *master node* of the resource manager cluster & the middleware cluster(s) respectively. This is explained in the section that follows.

4.1 Design Choices

4.1.1 Intra-Cluster Communication

The communication between the nodes in a cluster are managed using the `JGroups`[2] group communication library written in `Java`. `JGroups` takes care of informing all the cluster nodes about changes local to the cluster. These include *a new node joining the cluster*, *an existing node leaving/failing* and *the appointment of a new* ***leader*** *for the cluster as a result of the current leader failing.* `JGroups` also maintains the **view** of the cluster which shows all the membership details: who is in the cluster and who is the current leader. Moreover, `JGroups` takes care of **leader-election** for the cluster. The default leader-election protocol is that the node which joined the cluster earliest is picked as the leader. Hence, the first node to boot-up the process becomes the **leader** by default. If the current leader fails then the node that joined next would become the new leader.

`JGroups` also allows for a newly joined node to fetch the current state of the existing nodes. The newly joined node sends a *broadcast-request* asking for the server-state, which can be responded to by any of the nodes in the cluster. We have implemented the application-layer *(on top of JGroups)* to make use of this feature for newly joining nodes/replicas to fetch state from the others and become up-to date as instantly as possible.

4.1.2 Inter-Cluster Communication

The only cross cluster call that happens is from the `middleware` to the `resource-managers`. All such calls happen

[2]JGroups is a toolkit for reliable messaging. It can be used to create clusters whose nodes can send messages to each other.

only between the *cluster-leaders*. The **middleware cluster leader** calls the leader of each of the resource managers to cater client requests. Similarly the clients directly call the middleware cluster leader to submit it's requests. However, it is important to notice that these communication happens using the **RMI** based communication established in the initial phase of this project. Hence, we recall that the clients submit requests via the `proxy-object` of the *middleware*, whilst the `middleware` calls the *resource managers* using the `proxy-object` for the resource manager(s). So in this extended architecture only the respective cluster leader registers its object reference with the **rmi-registry**. Hence, when the middleware or the client obtains the *proxy-reference* from the rmi-registry, it will be the object reference specific to the leader of the cluster in question. Thus, when the node that failed is the *leader* of that cluster, then the newly elected leader (middleware or resource manager) must register its object with the *rmi-registry.* Subsequently, the calling node *(client or the middleware-master)* should `rebind` with the rmi-registry to fetch the new proxy-reference.

In order for the client or the middleware-master to rebind itself with the **rmi-registry** upon a cluster-leader failure, they need to be notified of this event. We handle this by making the clients be part of the middleware cluster and the middleware-master be part of the resource-manager cluster(s). This way the clients keep track of who the current middleware-master is and the middleware-master keeps track of who the leader is for each resource-manager cluster. Thus, when the leader of any of the clusters (*middleware or resource-manager(s)*) fails, the listening node (*either a client or the middleware-master*) would notice this event by way of the `JGroups` **view-change** notification. At this point the listening node (*either a client or the middleware-master*) would contact the rmi-registry and rebind its object reference against the newly registered master's proxy. This ensures that no node invokes an **RMI** call on a proxy-object of a non-existing node.

Moreover, giving membership to an external node (*client or middleware-master*) within a cluster (*middleware-cluster or resource-manager-cluster*) creates an additional scenario to be handled. Like already explained `JGroups` elects the cluster-leader based on the **time-joined** to the cluster. Thus, there will be a point where if all the nodes that joined a particular cluster - prior to the *outsider* (listening-node) - fail, then this *outsider* would become the leader. This is unacceptable, since it is the leader of the cluster who accepts requests from all outsiders. So the leader being an outsider-node would render the system to crash. We handle this scenario by **forcing a new-leader election** in the event of an outsider becoming the leader. This will result in the next oldest insider-node from the cluster being elected as the new leader.

4.1.3 Replication messages

The leader of each cluster is given the responsibility of forwarding replication messages to all the other nodes. All update actions are propagated to the replicas whereas `read` queries are not. State sharing between the replicas happen whenever a new node joins the cluster via the `JGroups` group communication protocol. This underlying communication is supported by implementing the `getState()` method in each node's application layer to receive the latest state from the other members of the cluster. This state request is served by the other nodes by implementing the `setState()` method.

Resource-Manager Cluster.
The resource-manager leader(s) forward an `UpdatedItem` message *(to their replicas)* for every `write()` or `delete()` call they receive. *read()* calls do not require any replication messages to be forwarded. In addition replication messages are forwarded for all **transaction** related calls (`start()`, `commit()` & `abort()`) too. The contents of the `UpdatedItem` message is shown in Figure 11 of *Appendix* B. For state-update requests from new members to the cluster, the `RMHashtable` data-structure of the **master** is passed over.

Middleware Cluster.
The middleware-manager forwards replication messages *(to the other nodes)* for every request it receives from the clients. This is necessary because, each request from a client changes the state of the **Transaction Manager**. The replication messages are sent in `ReplicationObjects` (*as seen in Figure 12 of Appendix B*) that contains the complete state of the **Transaction Manager** of the master. The same object structure is used to share state information with a newly joined node.

5. CONCLUSION

From the two choices **TCP & RMI** for handling communication between different components of the distributed setup, **RMI** definitely proved easier to build on top of. This reduced the burden of having to worry about message representation and thread handling. Moreover, the approach to delay updates to *resource-managers* until the transaction is `commit()`ted proved very smooth in-terms of handling `abort()`s and failures efficiently. The delayed update ensured a consistent state across *resource-managers* if a client `abort()`s or *times-out.* However, this had a marginal overhead of having to make a copy of every-item fetched from the corresponding *resource-manager.* This is because the *RMI* invocation to a *resource-manager* returns a reference to the item and any changes made to it is reflected in the *resource-manager.* considering the complexity involved in `abort()`s as the system grows to incur more components, we believe this is an acceptable cost to bare.

Replication handling was primarily built on top the `JGroups` group communication. The replication messages within the *middleware cluster* is far more than the ones within the *resource-manager cluster.* This is because the *resource managers* are only contacted during transaction `start(), commit(), abort()` and at **first-access** of any item. However, every client invocation unto the *middleware* causes a state change in the *Transaction Manager* as well as the *Lock states.* Hence, to handle failures efficiently, it is important that every change is reflected on all middleware replicas.

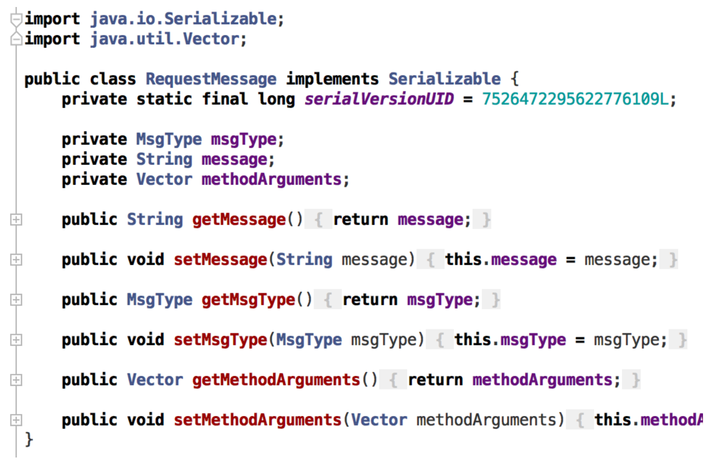
6. FUTURE WORK

As an extension to *Replication & Fault Tolerance*, **Load-balancing** needs to be introduced. This will ensure the middleware layer can handle more clients efficiently. In addition `2 Phase Commit` also needs to be integrated into the system to ensure **recovery**.

APPENDIX

A. TCP MESSAGING MODEL

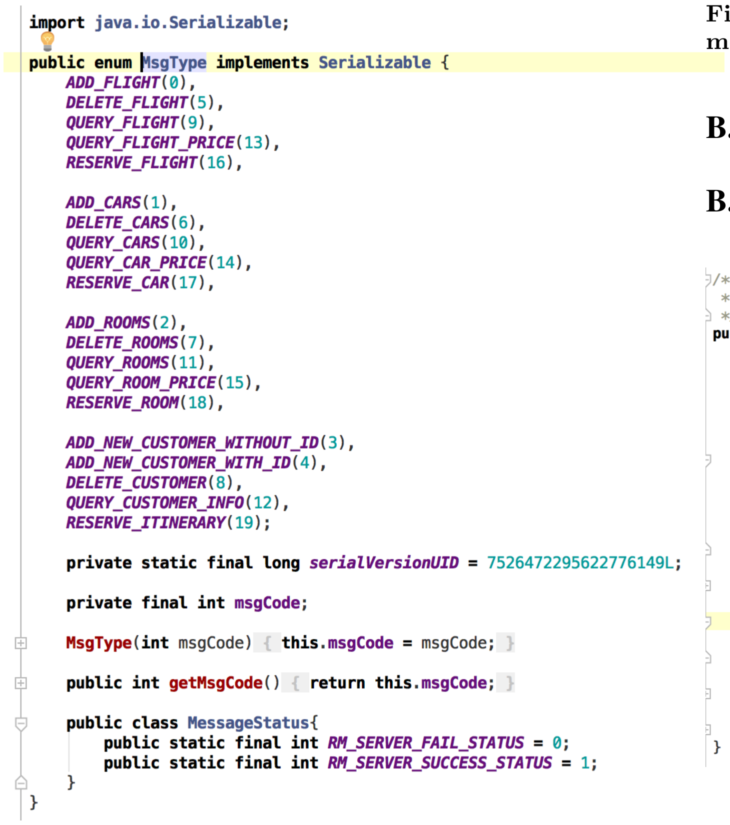
A.1 Class RequestMessage



```
import java.io.Serializable;
import java.util.Vector;

public class RequestMessage implements Serializable {
    private static final long serialVersionUID = 7526472295622776109L;

    private MsgType msgType;
    private String message;
    private Vector methodArguments;

    public String getMessage() { return message; }

    public void setMessage(String message) { this.message = message; }

    public MsgType getMsgType() { return msgType; }

    public void setMsgType(MsgType msgType) { this.msgType = msgType; }

    public Vector getMethodArguments() { return methodArguments; }

    public void setMethodArguments(Vector methodArguments) { this.methodA
}
```

Figure 8: Message representation class for request messages

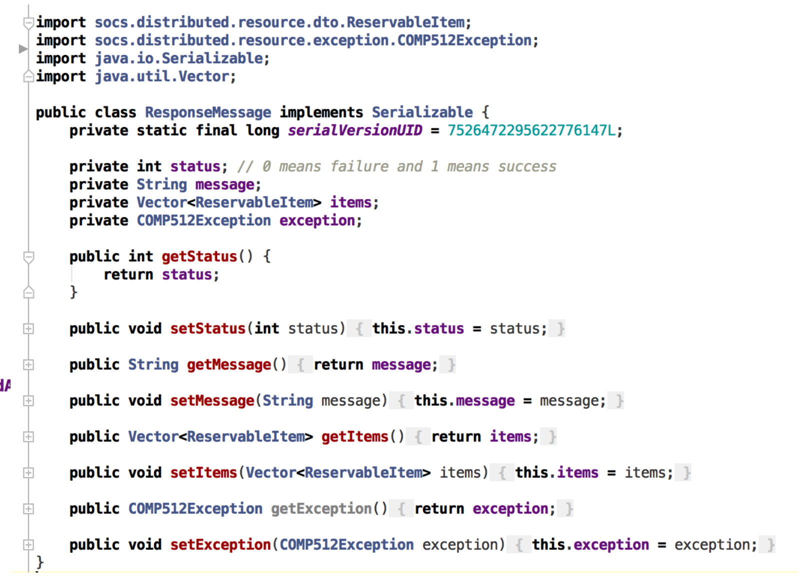
A.2 Enum MsgType



```
import java.io.Serializable;

public enum MsgType implements Serializable {
    ADD_FLIGHT(0),
    DELETE_FLIGHT(5),
    QUERY_FLIGHT(9),
    QUERY_FLIGHT_PRICE(13),
    RESERVE_FLIGHT(16),

    ADD_CARS(1),
    DELETE_CARS(6),
    QUERY_CARS(10),
    QUERY_CAR_PRICE(14),
    RESERVE_CAR(17),

    ADD_ROOMS(2),
    DELETE_ROOMS(7),
    QUERY_ROOMS(11),
    QUERY_ROOM_PRICE(15),
    RESERVE_ROOM(18),

    ADD_NEW_CUSTOMER_WITHOUT_ID(3),
    ADD_NEW_CUSTOMER_WITH_ID(4),
    DELETE_CUSTOMER(8),
    QUERY_CUSTOMER_INFO(12),
    RESERVE_ITINERARY(19);

    private static final long serialVersionUID = 7526472295622776149L;

    private final int msgCode;

    MsgType(int msgCode) { this.msgCode = msgCode; }

    public int getMsgCode() { return this.msgCode; }

    public class MessageStatus{
        public static final int RM_SERVER_FAIL_STATUS = 0;
        public static final int RM_SERVER_SUCCESS_STATUS = 1;
    }
}
```

Figure 9: MsgType enum that lists all available message types

A.3 Class ResponseMessage

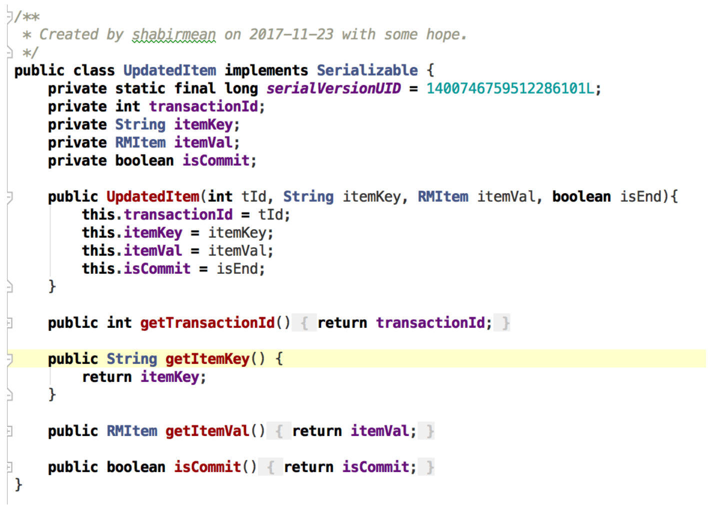


```
import socs.distributed.resource.dto.ReservableItem;
import socs.distributed.resource.exception.COMP512Exception;
import java.io.Serializable;
import java.util.Vector;

public class ResponseMessage implements Serializable {
    private static final long serialVersionUID = 7526472295622776147L;

    private int status; // 0 means failure and 1 means success
    private String message;
    private Vector<ReservableItem> items;
    private COMP512Exception exception;

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) { this.status = status; }

    public String getMessage() { return message; }

    public void setMessage(String message) { this.message = message; }

    public Vector<ReservableItem> getItems() { return items; }

    public void setItems(Vector<ReservableItem> items) { this.items = items; }

    public COMP512Exception getException() { return exception; }

    public void setException(COMP512Exception exception) { this.exception = exception; }
}
```

Figure 10: Message representation class for response messages

B. REPLICATION

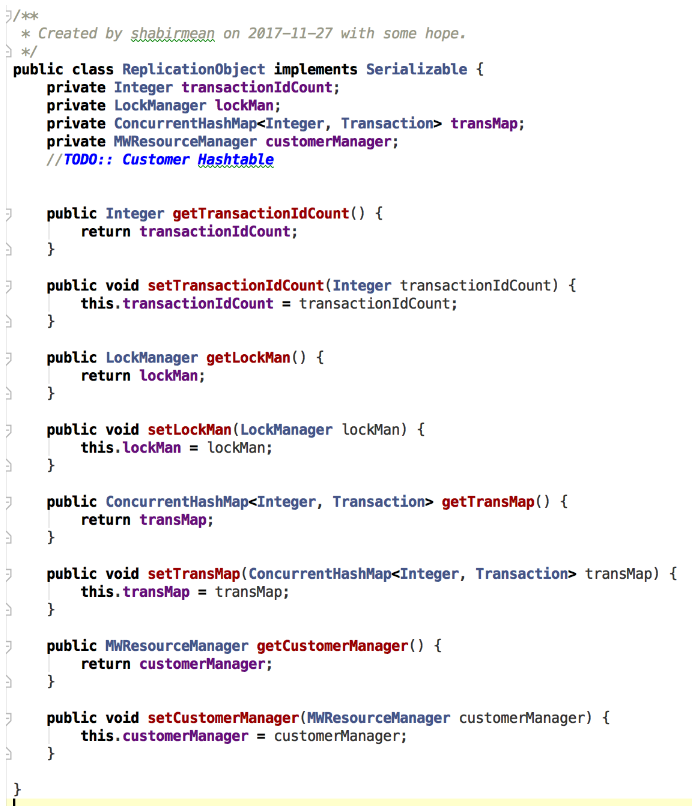
B.1 Class UpdatedItem



```
/**
 * Created by shabirmean on 2017-11-23 with some hope.
 */
public class UpdatedItem implements Serializable {
    private static final long serialVersionUID = 1400746759512286101L;
    private int transactionId;
    private String itemKey;
    private RMItem itemVal;
    private boolean isCommit;

    public UpdatedItem(int tId, String itemKey, RMItem itemVal, boolean isEnd){
        this.transactionId = tId;
        this.itemKey = itemKey;
        this.itemVal = itemVal;
        this.isCommit = isEnd;
    }

    public int getTransactionId() { return transactionId; }

    public String getItemKey() {
        return itemKey;
    }

    public RMItem getItemVal() { return itemVal; }

    public boolean isCommit() { return isCommit; }
}
```

Figure 11: Replication message for an updated item

B.2 Class ReplicationObject



```java
/**
 * Created by shabirmean on 2017-11-27 with some hope.
 */
public class ReplicationObject implements Serializable {
    private Integer transactionIdCount;
    private LockManager lockMan;
    private ConcurrentHashMap<Integer, Transaction> transMap;
    private MWResourceManager customerManager;
    //TODO:: Customer Hashtable


    public Integer getTransactionIdCount() {
        return transactionIdCount;
    }

    public void setTransactionIdCount(Integer transactionIdCount) {
        this.transactionIdCount = transactionIdCount;
    }

    public LockManager getLockMan() {
        return lockMan;
    }

    public void setLockMan(LockManager lockMan) {
        this.lockMan = lockMan;
    }

    public ConcurrentHashMap<Integer, Transaction> getTransMap() {
        return transMap;
    }

    public void setTransMap(ConcurrentHashMap<Integer, Transaction> transMap) {
        this.transMap = transMap;
    }

    public MWResourceManager getCustomerManager() {
        return customerManager;
    }

    public void setCustomerManager(MWResourceManager customerManager) {
        this.customerManager = customerManager;
    }

}
```

Figure 12: Replication object shared by the Middleware-master