Distributed Application

Overview – RMI Architecture

Remote Method Invocation (RMI) was the architecture utilized for the final implementation of the distributed application. The RMI architecture consists of 3 parts (each with distinct roles): Client host, Middleware Server host, and Resource Manager Server hosts.

The Client host (Figure 1) is responsible for connecting to the middleware server and executing client commands. Upon bootup, the client host has several responsibilities before a user can execute a command. Initially, the **RMIClient** locates the RMI registry on the middleware server (using the server hostname and a specific port that the registry is located at) and performs a lookup of the **IResourceManager** remote object. Any communication failure or lag in the middleware binding the remote object's stub will result in the client waiting until these issues are resolved. Upon a successful connection, the Client provides a command-line interface (CLI), parses user input, and invokes the appropriate command using the remote object's stub. The stub object marshals the message and sends it to the skeleton object residing on the middleware server.

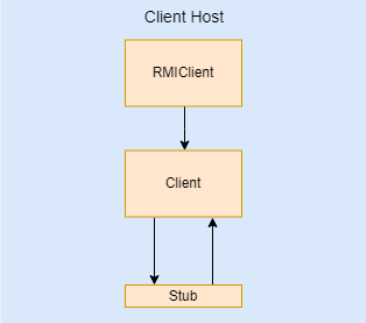

Figure 1

The Middleware Server host (Figure 2) is the mid-point between the client host and the resource managers. All client commands are sent through the Middleware. As such, the first responsibility of the middleware is to create the **IResourceManager** remote object and register it at the RMI registry. Now, the client can lookup the **IResourceManager**, utilizing the stub object to send messages to the middleware. The skeleton object, located at the middleware, unmarshals client messages and sends them to the remote object. Secondly, the Middleware locates the registry for each of the Resource Managers and obtains **IResourceManager** references (stub objects) for the *Flight*, *Car*, and *Room* resource managers (The Middleware is also a Resource Manager for the *Customer* data type). When a message is received, the Middleware performs any Customer actions and forwards any *Flight*, *Car*, and *Room* actions to the appropriate Resource Manager server (using the stubs).

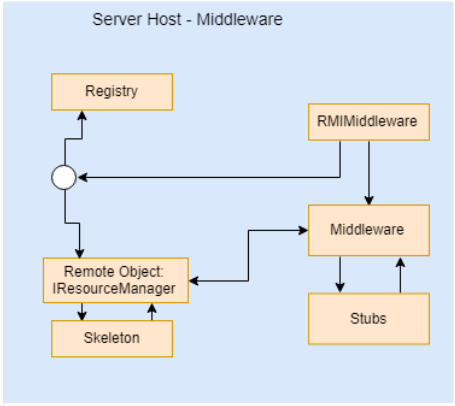

Figure 2

The Resource Manager Server hosts (Figure 3) perform actions related to the *Flight*, *Car*, and *Room* data types (*Customer* actions are performed at the Middleware Server). At each Resource Manager, a **IResourceManager** remote object is created and registered at the RMI registry. The Middleware server can lookup the remote objects, for each server, and utilize the stub object to send *Flight*, *Car*, or *Room* messages to their respective Resource Manager. Upon receiving a message, the remote object's skeleton unmarshals the message and forwards the message to the **IResourceManager** object for command execution.

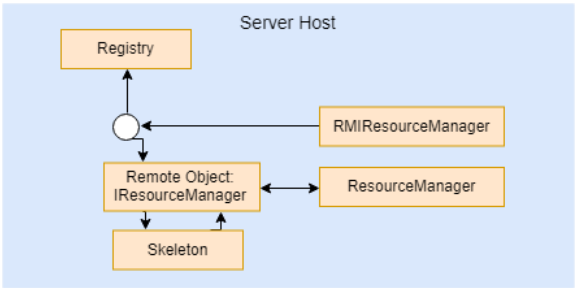

Figure 3

Implementation

Distribution and Communication

As outlined in *Overview – RMI Architecture*, communication between the Client Server host and the Middleware Server host and between the Middleware Server hosts and the Resource Manager Server hosts is facilitated by the Remote Method Invocation (RMI) architecture.

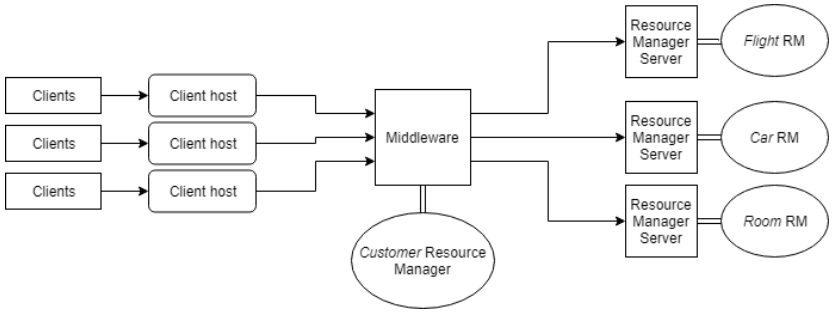

Figure 4

Moreover, Figure 4 provides a high-level picture of the application distribution. Users utilize Client Server hosts to execute commands. Commands involving the *Customer* data type are executed at the Middleware (since the Middleware is, itself, a Resource Manager). Commands involving the *Flight*, *Car*, or *Room* data types are executed at the appropriate Resource Manager Server.

Transactions and Locking

For handling transactions, the Middleware coordinates with the Transaction Manager, Lock Manager, and Resource Managers (Figure 5). For commits, transaction local copies are flushed to the global **RMHashMap**. For aborts, the local copies are simply thrown away.

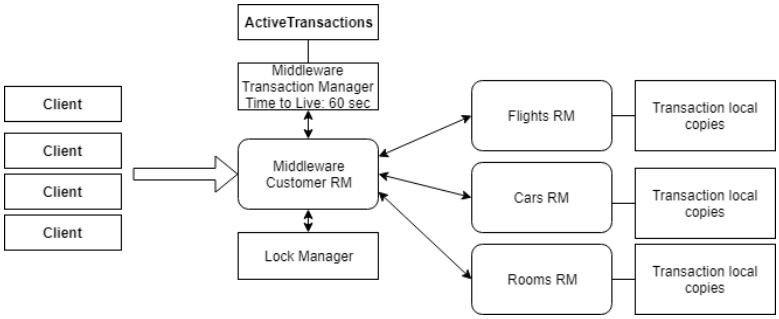

Figure 5

Locking

The Lock Manager is implemented at the Middleware Server, thus providing a **centralized 2-Phase Locking** solution to the clients. When a client performs a query on item *x*, the locking manager grants a READ_LOCK for *x* only if no other transactions (besides the current transaction) has a WRITE_LOCK on the *x.* When a client modifies the state of the data, the locking manager grants a WRITE_LOCK for *x* only if no other transactions (besides the current transaction) has a WRITE_LOCK or READ_LOCK on the *x.* Moreover, a transaction unlocks all their locks at the end of the transaction (after an abort or commit). 2-Phase Locking prevents any data corruption and allows for the correct coordination of transactions in the distributed environment.

The Lock Manager required lock conversions; this entailed detecting that transaction *t* is requesting a WRITE_LOCK on a data item for which *t* already has a READ_LOCK and no other transaction has a lock on the item. If this is the case, the READ_LOCK is converted to a WRITE_LOCK; otherwise, transaction *t* is blocked until the conflict is resolved (either by another transaction releasing the conflict locks or by throwing a *deadlock* exception).

Transaction Management

Here's an illustrative example of a sample transaction (assume *flight-1* was created in t_1; t_1 was committed)

Start t_2,t_3 → *queryFlight,2,1* → *addFlight,3,1,1,1* → *commit,2* → *commit,3*

Start

When a client executes the *Start* command, the middleware asks for a new transaction id (xid) from the Middleware Transaction Manager, and the middleware returns this to the client. In the background, the Transaction Manager creates a new *Transaction* object (xid=*i)* and adds this object to the HashMap of active transactions. It is important to note that, for active transactions that are stale (i.e. no command involving the transaction has been executed in a while), the Transaction Manager will assume that the connection to the Client host was lost and the transaction will be aborted.

queryFlight,2,1

t_2 wants to query *flight-1*. At the Middleware, check for the existence of the active transaction and request a READ_LOCK on *flight-1*. This is granted, since there are no other active locks on *flight-1*. Thus, the *queryFlight* command is executed and returned to the client.

addFlight,3,1,1,1

t_3 wants to modify *flight-1*. At the Middleware, check for the existence of the active transaction and request a WRITE_LOCK on *flight-1*. This is **not** granted, since there are other active locks on *flight-1*; t_3 is blocked (NOTE: if this blocking lasts too long, a *Deadlock* exception is thrown. In this case, t_3 would be aborted).

Commit,2

t_2 wants to commit. Using Middleware Transaction Manager, move the local **RMHashMap** data associated with the transaction to the global **RMHashMap** for all resource managers involved. After the commit, t_2 unlocks its locks and the Transaction Manager moves t_2 from active to inactive(commit=True),

Commit,3

Assuming the READ_LOCK on *flight-1* was unlocked before a *Deadlock* exception was thrown, the t_3 commit is similar is execution to *Commit,2*.

Logging

For the logging feature as part of the final milestone, I implemented the *Logger* class which leverages the Java *JSON* library. The *JSON* library provides the following classes and APIs:

- *JSONObject*: A key-value format which can be stringified for simple communication and storage
- *JSONArray*: Array of values
- *JSONParser*: Parses JSON-formatted files and converts the file to a Java Object in memory
- *JSONObject*.*toJSONString()*: which stringifies the JSON formatted object or array.

For the Middleware and the Resource Managers, 3 JSON files are maintained: *committed.json*, *in-progress.json*, and *master.json*.

The *master.json* files stores the last committed transaction id (xid), any locking information so the locks can be restored upon recovery (only at the Middleware), the last crash mode (so we can crash during recovery), and information about 2PC.

The *in-progress.json* file stores information required to restore the in-progress transactions and the Transaction Manager (at the Middleware).

The *committed.json* file stores the information required to restore the global **RMHashMap** (i.e. committed transactions).

Data Shadowing

During the execution of the active transactions, any changes read from and modify local **RMHashMap** data, rather than the global data. As active transactions execute, these modifications on the local **RMHashMaps** are flushed to the *in-progress.json* file for both the Middleware and the Resource Managers. This allows the Transaction Manager and the active transactions to be restored, should the server crash. If the transaction is aborted, the transaction's local data is freed from memory and the active transaction is removed from the *in-progress.json*.

If the transaction is committed, the following events occur:

- The local **RMHashMap** of the transaction is flushed to the global **RMHashMap**,
- A new *committed.json* file is written, using the xid of the transaction to write the file (i.e. *committed_{xid}.json*). Thus, our old global copy (located in the old *committed.json)* isn't overwritten until the *last Committed* field of the *master.json* is updated to point to the new file
- The *master.json* > *last Committed* key-value pair is updated with the xid and flushed to file. The old *committed_{previous xid}.json* can be removed
- In memory: the transaction manager moves the transaction from active to inactive (changes flushed to file) and the *Prepared* part of the *master.json* file (see *2PC* below for a description of this key-value section) is removed

Recovery (2PC)

The recovery algorithm depends on whether it is the Middleware or Resource Manager that is being recoved; thus, I'll go through these separately:

Middleware

Upon a call to *commit(xid)* at the Middleware, a *Prepared* section of the *master.json* file is created, and it will store recovery information for the transaction. The creation of this *Prepared* section in the JSONObject for the transaction is used in lieu of a "Before Voting" log.

As the Middleware calls the *prepare* function of the Resource Managers, the votes of the resource managers are flushed to the *Prepared* section of the *master.json*. Upon recovery, the number of votes will signify which Resource Manager will still need to vote. With all the votes, a decision can be made, and abort/commit messages are sent. Any failures here will result in (upon start-up): re-formulating the decision (which is inexpensive since we have all the votes) and resending the decision (if the decision is redundant, i.e. the resource manager has already acted on the decision, nothing happens)

Resource Manager

Upon a call to the *prepare(xid)*, a *Prepared* section of the *master.json* is created, and it will store the recovery information for the transaction. The creation of this *Prepared* section is used in lieu of a ”Vote Request Received” log.

The Resource Manager can come to 2 decisions (which are logged): abort (the transaction has already been aborted or does not exist) and commit. This decision is stored; if the Resource Manager fails here and the decision is abort, the recovery procedure will simply abort the transaction. If the Middleware re-calls the *prepare(xid)*, then the vote will be “No” and the Middleware will abort or if the Middleware has a timeout the transaction will have aborted anyway. If the decision is “Yes”, the Resource Manager checks (with timeout) whether the transaction is active at the Middleware. If yes: let the Middleware re-call *prepared(xid)*, and the “yes” vote will be returned. Otherwise, if “No” or timeout, abort the transaction.

If, after sending the answer, the Resource Manager fails: if the answer is “yes”, the Resource Manager must wait indefinitely for the Middleware to send the the answer (the Resource Manager can’t make any decisions based on its local state). Otherwise, the Resource Manager can abort the transaction (since it voted “No”; even upon restart, the Middleware will see the “No” response and abort).

Upon receiving the commit/abort decision from the Middleware, the Resource Manager logs the Middleware decision and continues with *commit(xid)* or *abort(xid)*. Should there be any crash, the recovery will simply re-try the Middleware decision (exceptions for already committed/aborted transactions will be caught and ignored).

NOTE: Since the architecture now involves connection timeouts, there can be situations where active transactions get into a bad state (i.e. During *Bundle*, if we have already reserved the flights, but lose connection to the *Cars* Resource Manager and timeout, the application needs to abort the transaction – to ensure bundle is atomic). Thus, upon Resource Manager recovery, active transactions also ask the Middleware if the transaction is still active. If “yes”: do nothing; else if “no” or timeout abort.

Special Functionality

Special commands *Summary* and *Analytics* were implemented and extend the default commands for the CLI. *Summary* takes in the xid and it outputs a summary of the client-resource allocation. For each client, the resources they have reserved will be printed. *Analytics* takes in the xid and an upper bound. Any resource with remaining quantities less than or equal to the upper bound will be printed. Thus, *Analytics,1,0* will print all those resources with no remaining items for t_1, *Analytics,1,1* will print those with 1 or 0 remaining items for t_1, etc.

NOTE: Since I am working alone, Alex allowed me to skip the custom functionality for Milestone 2 and 3.

Development and Performance Issues

From a development perspective, issues I had revolved around code maintenance and manageability. For big projects, especially projects with large and overlapping subsections (i.e. Milestones), careful planning can reduce development time, as code reuse and comprehension is maximized. Although the Distributed Application implementation works as desired, the code is quite difficult to understand. In the future, I hope to refactor the code.

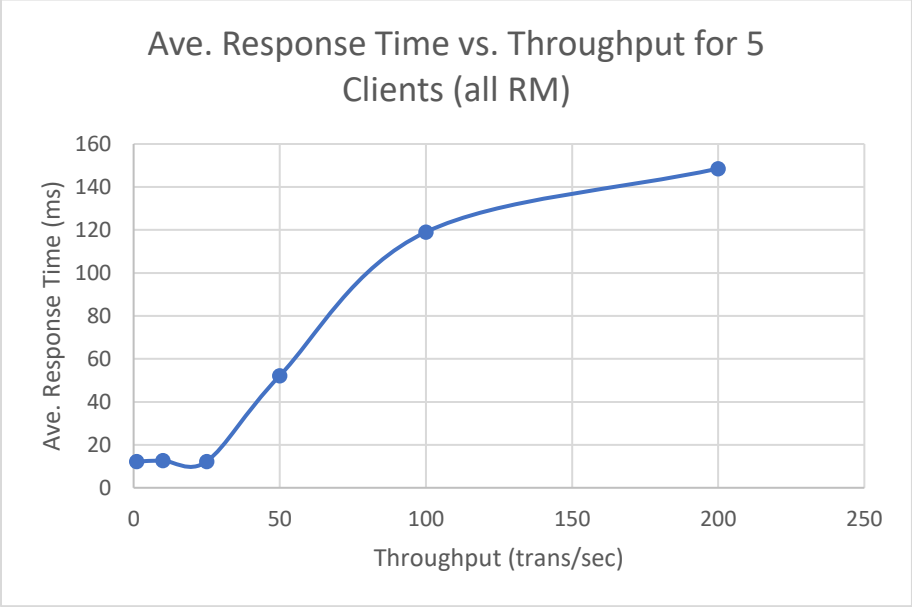

Figure 6

With regards to performance, there seemed to be a bottleneck at the middleware, and this is an issue for response time. Referring to Figure 6, the saturation point seems to be around throughput = 25 trans/sec. After this point, response time increases rapidly with throughput. After an analysis of the data, the average response time increase was determined to be due to an increase in time spent at the Middleware. The transaction type (which involved the reservation of a *Flight, Car,* and *Room*) relied on the *Customer* data and the coordination of the 3 Resource Managers, and so, the middleware will under a lot of strain during a high throughput time.

Correctness

To guarantee correctness throughout the project, I created a RMI*ClientTest* class that executed commands in a loop. Thus, I was able to create and maintain a test suite. This allowed me to:

1. Ensure future Milestones of the project didn't break the functionality of previous Milestones (i.e. *addFlight,1,1,1,1* has the same expected outcome in all stages of the project)
2. Expand the test suite to incorporate and test new features (i.e. for Milestone 2, the concept of transactions was introduced; thus, the test suite was extended to include transaction concepts like commit, abort, etc.)

The *RMIClientTest* class utilizes the *Client* class and is very similar to *RMIClient*. However, instead of a CLI, commands and arguments are listed in a String array and executed in a loop.