COMP 512 - Distributed Systems

Team: V for Vim

Amir El Bawab -260645260

Archit Agnihotri - 260777127

Description

The software discussed in this report is a command line based application that allow one or more clients to simultaneously configure customers profiles, flights, cars and rooms services. The purpose of this deliverable is to transform a simple Client/Server architecture into a distributed design by introducing a middleware server and domain specific resource managers.

Several technologies are available to accomplish a distributed system design. In this report we discuss Remote Method Invocation (RMI) and Transmission Control Protocol (TCP).

RMI

Description

An RMI based system relies on a registry that plays an intermediate role in invoking the request from the clients to the server. The server has to register an object into the registry with a unique key. Once an object is referable from the registry, a client can connect to the registry and perform requests on the registered object using its unique key.

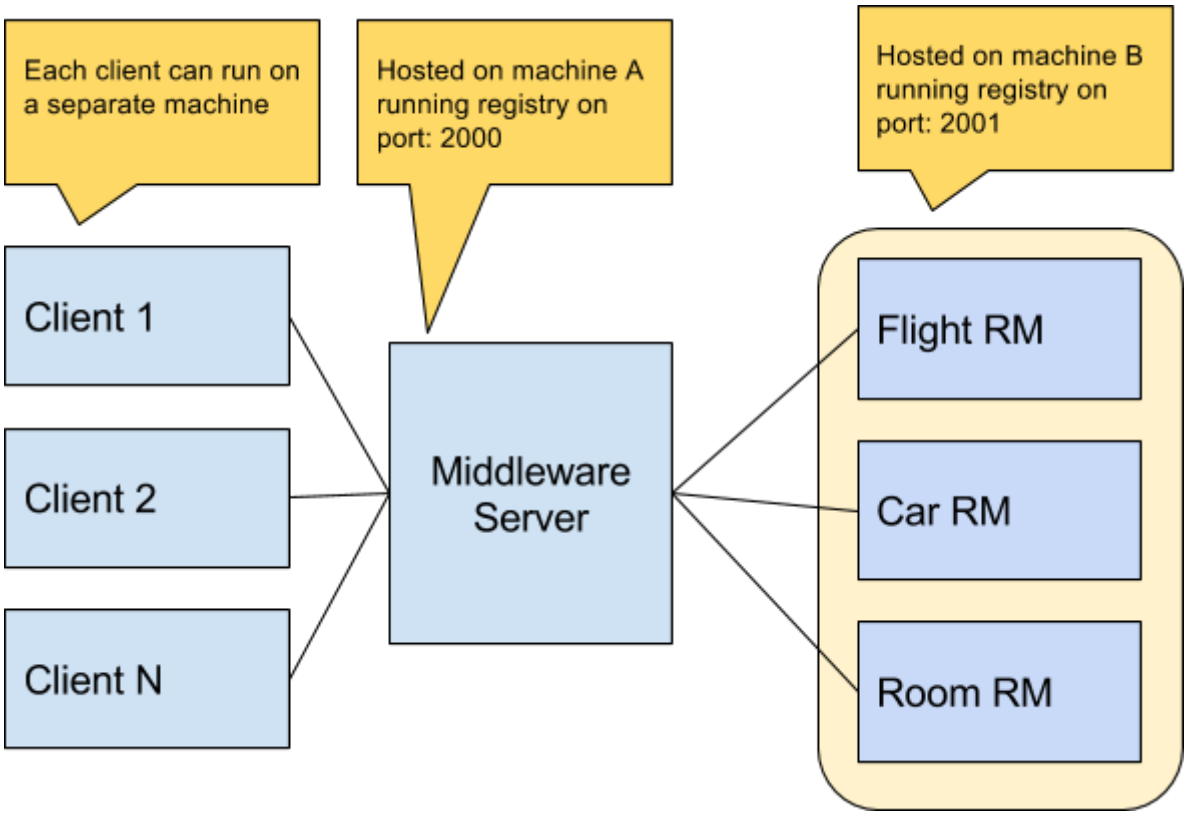
Multiple clients

By design, RMI technology allows multiple clients to connect to the registry and invoke methods on the object that lives on another Java Virtual Machine (JVM). However, RMI is not thread safe when multiple clients are connected to the registry from different JVM, therefore a locking mechanism has been integrated into the project which guarantees that one client at a time can invoke a method where concurrency of calls can potentially damage the data.

Our design is composed of a flight resource manager (RM), a car RM, a room RM and a middleware server which orchestrates the requests sent by the clients. The middleware server is the gateway to the distributed system which stores data over all the RMs. For instance, a command from the client to the middleware server might be transferred to the domain specific RM, or sometimes broadcasted to several RMs when the request requires information distributed over multiple components. An example of such complex command is

querying information about a customer. The result of this command is stored on multiple RMs and the middleware server is required to perform the necessary action to deliver the correct information while abstracting the complexity of the task from the client.

RMI Architecture



TCP

Description

Unlike RMI, TCP based sockets allow a bidirectional communication directly between the Client and Server. Instead of invoking methods, socket communication allow the transmission of information by sending and receiving bytes. The communicated bytes can be human readable strings but can also be serialized objects. In the current deliverable, the clients commands are sent in the form of human readable strings which are then parsed by the distributed system.

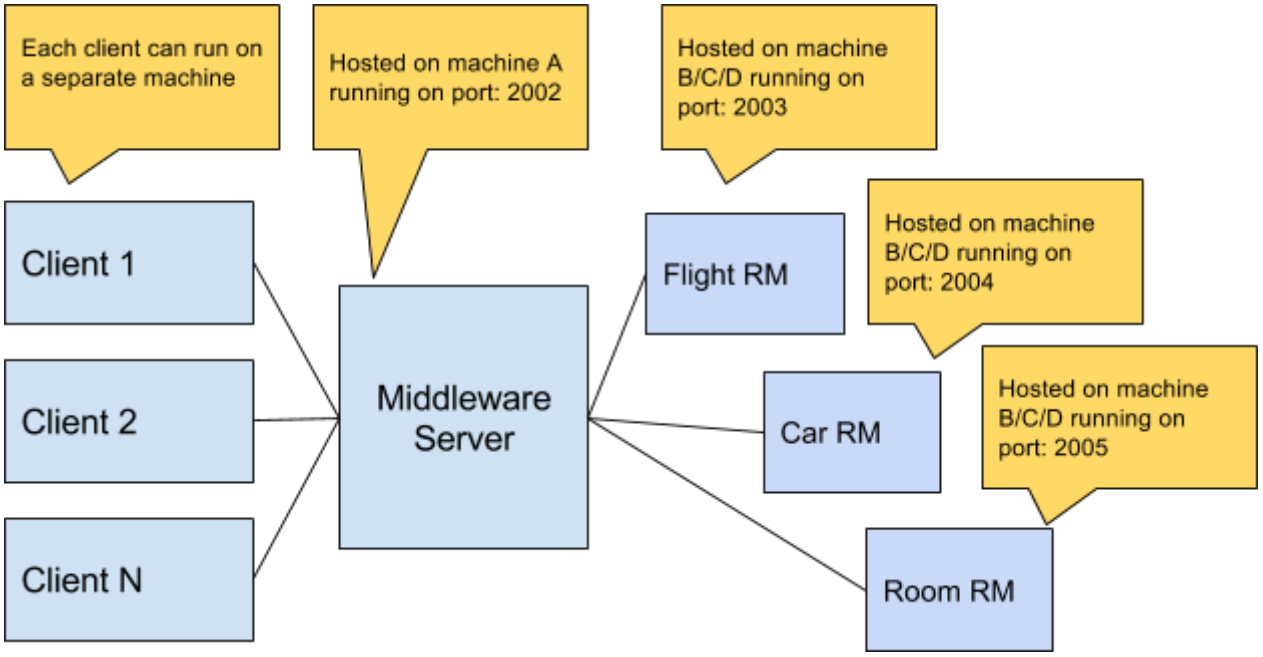
Multiple clients

Each socket connection can connect one client to one server. In order to allow several clients to connect to the same server, the latter must continuously keep listening on a particular port for new clients. To achieve such concurrency, the server must place each accepted socket communication into a child thread which can then run in parallel with the

main thread. Each time a new client is accepted, a new thread will be created. Threading on the server allows several connections to be established between the clients and the server, however having multiple threads running in the same process introduces concurrent shared memory access issues. Since multiple clients might be performing same actions from different ends, returned results might be manipulated by certain context switches. To avoid such data damage, methods on the middleware server side are synchronized.

The TCP based distributed system is composed of a flight RM, a car RM, a room RM and a middleware server which is, in its turn, a client connected to all the RMs servers.

TCP Architecture



Building and running the distributed system

The RMI and TCP implementations of the project are configured using the Gradle build tool. Using a build tool in a distributed systems project facilitates managing complex architectures and complicated dependencies. The delivered project is composed of several Gradle subprojects which can each be built and executed independently.

Relevant Gradle features include:

- Managing dependencies (which is important for a complex distributed project, such as this one)
- Simple integration with third party libraries e.g. JUnit, Apache Logger
- Creating custom tasks in order to build different components in the project while maintaining the execution dependencies.
- Configuring the network details of various components from properties files, which allows the portability of the distributed system without the need to recompile the project files.

Project tree structure

```
root/
├── common
│   ├── client
│   ├── remote
│   └── rm
├── rmi
│   ├── client
│   ├── midserver
│   └── rm
└── tcp
     ├── client
     ├── midserver
     └── rm
```

Commands

The following commands must be executed from the root directory of the repository. The order of execution is important as the servers has to run before the clients start.

For RMI

Start Registry on the middleware server host machine

./gradlew midServerRMIRegistry

Start Registry on the RMs host machine

./gradlew rmRMIRegistry

Start RMs

./gradlew rmi:rm:build rmi:rm:runCar
./gradlew rmi:rm:build rmi:rm:runFlight
./gradlew rmi:rm:build rmi:rm:runRoom

Start Middleware Server

./gradlew rmi:midserver:build rmi:midserver:run

Start a Client

./gradlew rmi:client:build rmi:client:run -x test

For TCP

Start RMs

./gradlew tcp:rm:build tcp:rm:runCar
./gradlew tcp:rm:build tcp:rm:runFlight
./gradlew tcp:rm:build tcp:rm:runRoom

Start Middleware Server

./gradlew tcp:midserver:build tcp:midserver:run

Start a Client

./gradlew tcp:client:build tcp:client:run -x test

Configure network settings

To configure the IP and Port values, edit the corresponding variables in the file build.gradle located at the root of the repository.

Testing

We have thoroughly tested the function calls by stress testing the RM methods, as well as the number of concurrent clients using the system at a given instance.

Unit test

Our unit tests, available for both RMI and TCP clients, have three test levels of {10, 100, 1000} function calls and three threading levels of {2, 4, 6} threads, and each combination is tested. The most stressful test is 1000 function calls per thread, with 6 threads running in parallel trying to acquire access to the synchronised functions.

The functions tested involve adding, reserving, querying and deleting flights, cars and rooms.

Manual test

Since RMI technology does not guarantee thread safety when multiple clients are connected to the same server from different Java Virtual Machines, manual tests has been performed on the distributed system where each client lived on a different machine.

Test case

The following test has been used to test each command of the system on both RMI and TCP based architectures.

addFlight,1,1234,100,299
addCars,1,canada,50,99
addRooms,1,canada,50,99
newCustomer,1
newCustomerId,1,111
queryFlight,1,1234
queryCars,1,canada
queryRooms,1,canada
queryFlightPrice,1,1234
queryCarsPrice,1,canada
queryRoomsPrice,1,canada
reserveFlight,1,111,1234
reserveCar,1,111,canada
reserveRoom,1,111,canada
itinerary,1,111,1234,canada,true,true
queryCustomerInfo,1,111
deleteCustomer,1,111
deleteFlight,1,1234
deleteCars,1,canada
deleteRooms,1,canada

Repository

The code for the project is hosted on a private repository on Github. Digital documentation and a Graphical user interface in progress is available at the following link: https://amirbawab.github.io/V-for-Vim/