Milestone 1: Distributing an Application

Group 3
Nicole Ho
Younes Boubekeur

Introduction

This project revolves around developing a Travel Reservation system with booking and management support for flights, cars and rooms. Given a single-client, single-server implementation, our goal in this milestone is to distribute the server first with Java's Remote Method Invocation (RMI) API and later with the TCP protocol.

RMI implementation

Overview

The server is distributed over three `ResourceManager`s, one each for flights, cars and rooms. An intermediary server called `Middleware` sits between the client and the server, relaying messages back and forth. To keep our system simple, each resource manager implements the entire interface and the middleware calls only the respective methods for each manager. Both the `Middlware` as well as the `ResourceManager`s can be run on an arbitrary combination of physical machines, also referred to as nodes. For example, we can run each server on a separate node. We could also run `Middleware` and the flight `ResourceManager` on one node and run the car and room `ResourceManager`s on another.

Implementation details

The bash script that is used to run the servers starts the RMI registry on each machine. The Java code binds each server to the registry with a server name, eg `group3_Middleware` and a port number. This allows for the server to accept remote invocations of the methods it exposes via its public interface. The client nodes connect to the servers by getting a reference to the RMI registry and looking up the appropriate server name and port, which they obtain from command line arguments. For more details, please refer to the submitted code.

Customer handling

Our first design decision was to have the middleware manage the customers since it does not directly deal with any of the other resources. However, the purpose of our middleware is act to as a layer of communication and distribute the requests. With this in mind, we chose to decentralize customer handling through replication at each resource manager. Each manager maintains its own list of customers. When the client invokes a `customer`-command, the middleware calls the method on all resource managers.

Bundles

As each resource manager maintains a copy of the same list of customers, the middleware simply calls `reserveFlight()`, `reserveCar()`, and/or `reserveRoom()` on the corresponding `ResourceManager` based on the input paramemters.

TCP implementation

Overview

Our next task was to re-implement the system using the TCP protocol. Since remote method invocation was no longer possible, it was necessary to use generic message passing over TCP. This requires that the machines package their messages (requests and responses) into sendable objects, which would be unpackaged on the other side and acted upon asynchronously.

Architecture

In our implementation, all requests and responses are sent as serializable objects. To send commands over TCP, we designed `UserCommand`, a simple custom class that implements `Serializable` and can therefore be packaged into an `ObjectOutputStream`. A `UserCommand` has an numeric `id`, the `Command` to run, and its arguments as a String array.

On both the middleware and the resource managers, a server socket is continuously listening on the port for new connections. For every incoming request, a new thread is spawned in a fixed thread pool. This allows for reasonable concurrency while limiting the total number of threads to avoid server overload.

The general algorithm for the `TCPClient` has common elements with its RMI counterpart. It takes a command string from the human user (eg, `addFlight,1,2,3,4`), performs basic validation, and parses it to determine the appropriate action to execute. In the case of TCP, this command is encapsulated into the `UserCommand` object mentioned above and sent to the middleware via an `ObjectOutputStream` bound to the appropriate socket. The Middleware then reads the `Command` attribute of the `UserCommand` received over its `ObjectInputStream` and determines the correct `ResourceManager`(s) to call asynchonously and forwards the `UserCommand`. When it receives a response, it sends it back to the client caller. The `ResourceManagers` behave in a similar manner. The format of the responses, from the `ResourceManagers` to `Middleware` and from `Middleware` to the `Clients`, is a primitive type wrapped in an object. This is possible since all methods of the `IResourceManager` interface return either `boolean`, `int`, or `String`.

Additional functionality: Implementing Concurrency using new Java 8 features

To simplify our codebase and make it more efficient, we used the following relatively recent Java features:

1. **Executors**: To manage a fixed size of concurrent threads automatically with minimal syntax.
2. **Lambda functions**: We used these thoughout to express event handling code in a more expressive functional style. Concretely, this means we can specify an action (method) as a parameter to another method, so it can be run asynchronously. Other benefits of the Java lambda interface include better parallel performance and the ability to use `CompletableFuture`, detailed next.
3. **CompletableFuture**: This is the most powerful feature used in this milestone, since it gives us a better API for programming reactively, without forcing us to write explicit synchronization code needed when using thread primitives. Put simply, a `CompletableFuture` is equivalent to a JavaScript `Promise`, a piece

of code does something that could take time (such as contacting a remote server) and promises to return the result at a later time. In the meantime, the executing thread is free to do other things, such as handle other requests. Thanks to this asynchronous behavior, multiple requests can be serviced concurrently, thereby ensuring that the server is non-blocking.