School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2015

Project Part 1: Distributing an Application

Due date: October 5/6

NOTE-NOTE-NOTE: This project deliverable has a maximum of 100 points. If you do a solid implementation where everything is working and you give a good demonstration, then you will receive a maximum of 95 points which reflects a solid A. Of course, if something is missing or does not work correctly, there will be point reductions. 5 points are reserved for extras that are not specifically asked for in the description below. For example a very elegant, modular design of the TCP-implementation, or an outstanding presentation. It's kind of having an A+, something that goes beyond the standards

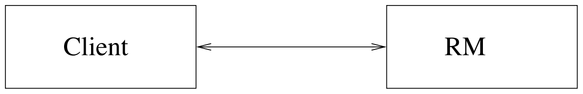
This project is an adaption of the project of CSE 593 of the University of Washington. The goal of this project is to design and develop a component-based distributed information system. The project aims in letting students implement some of the fundamental components and algorithms for distribution, coordination, scalability, fault-tolerance, etc. The example application is a travel reservation system.

The project is organized as a sequence of steps that iteratively add components and system structure, working toward the ultimate goal of a multiple client, multiple server system. Note that the steps are not all of comparable difficulty. Effort required is also not proportional to the length of the specification, so long specs might be easier to implement than short ones.

The interfaces of all server components are specified, so that a common client can attach to and use any project's server. You are provided with the implementation of the sample application and an interactive client. You might want to implement a non-interactive client program for testing, but this will be easy. If you think that the project is not challenging enough feel free to implement your own client GUI. But this will not give you any credit (or maybe the extra 5 points...).

In this first part of the project you have to distribute the simple application in two ways. The first one is based on web-services, the other uses sockets. In the following parts you will add transaction support, fault-tolerance, or other services (you will probably be able to make choices).

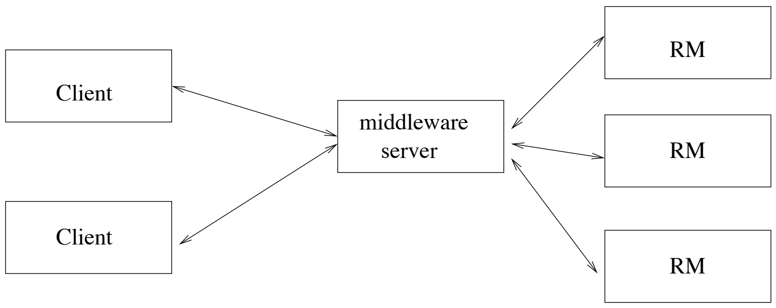
You are provided with a simple web-service based Resource Manager and a simple interactive client connecting to the Resource Manager through the web-service interfacce. The operations that the Resource Manager supports are outlined in the Java interface. The current architecture looks as follows:



The representation and functionality of the reservation system are quite simplifying but will make the project easier. We assume there is only one airline (so a flight identifier is an integer), only one type of car, only one type of hotel room, and only one day. If you follow this approach, then there is only one type of things, and hence, there is only one price. The net effect of addCars(id1, San Diego , 4, $52); addCars(id2, San Diego , 7, $54); would leave 11 cars at either $52 or $54, not 7 cars at $54 and 4 cars at $52. Each customer maintains the list of all reserved items (flights, cars, hotels). `queryCustomerInfo` returns the list of reserved items, and the total cost of the customer. We haven't bothered with an account payment feature. `deleteCustomer` does not only delete the customer record but also cancels all the reservations performed by the customer.

The current implementation ignores the *id* parameter. It might be useful in later parts of the project. Hence, we have added it to the interface. Since cars, hotels, rooms have similar concept and similar methods, our implementation has some abstractions that make further tasks hopefully easier. The implementation does not take into account that several clients might access the resource manager concurrently.

1. **Distribute the systems**. There are several resource managers. The clients, however, do not see anything of this distribution. Instead, a middleware server (providing the interface with the entire functionality), is put between the resource managers and the clients:



   a.) Between the client and the RMs build a global middleware layer. It provides the clients the same interface than the original RM. The middleware server will be given the list of active RMs as command line arguments on startup.

   b.) Flights, cars and rooms should be each handled by separate RMs. (For simplicity, you can keep only one type/implementation of RM providing the entire functionality – the middleware server will simply only call the car-methods at the car-RM, the flight-methods at the flight-RM, etc.) You have to decide how to handle customers (remember that the same customer might have reserved flights and cars). They could be handled by an additional server, the individual RMs (and the middleware might have to collect information), or maybe even by the middleware server (which then would also be some form of resource manager).

c.) The middleware server provides an additional function: *reserveItinerary*, which books a set of flights, and possibly a car and a hotel at the final destination. This itinerary reservation is the sort of high level operation associated with a middleware server.

d.) The middleware and the RMs might receive concurrent requests from different clients. Make sure that such concurrent execution does not damage your data structures.

2. **Use TCP instead of Web-services**. (*Note: also the description below is short, expect that this second part takes a lot of time*). All communication between clients and middleware, and middleware and RMs should be based on sockets. The client can remain blocking (i.e., it sends a request and waits for the reply before sending the next request). However, the middleware should not block when it is waiting for the RMs to execute a request. That is, when the middleware receives a request from the client, it forwards it to the corresponding RM. Then it again accepts client requests. When it receives a response back from the RM, it returns it to the appropriate client. In a similar way, the RM should be able to handle several requests concurrently. That is, you can kind of reimplement web-service functionality on top of sockets, or use another communication paradigm that follows above requirements.

You have to provide the following deliveries:

- A description of your design and a list of tests that you performed to test the systems. The main emphasis will likely be your description of the design of the TCP-based system. Note that for your last deliverable you have to provide a complete report which you can generate by putting together the individual reports. Keep this in mind.
- An online demonstration of your implementation. The demonstration will take between 15-30 minutes (to be determined). Think ahead of time how you best can present your system in that short time-period. The demonstration should include
  - A short outline of your architectures and designs for both parts of the deliverable. Power-point slides are recommended.
  - How you start up the two systems on different servers. Use at least three different machines for the servers and the client program.
  - How you execute certain requests for the two different implementations.
  - Expect questions from the TA on your system throughout the demonstration.

The demonstrations of all groups will take place over two days (likely October 5/6). A sign-up sheet will be put in place so that you can reserve a time-slot. The due date of your report is the time of your demonstration. The TA might ask you to bring a printed copy of your report with to your demonstration.