Internet Programming - Assignment 3

Jayke Meijer, 2526284, jmr251, jayke.meijer@gmail.com

October 17, 2012

Contents

1 Used platform

All code is tested and executed on a machine running Ubuntu 12.04. The used version of *gcc* is 4.6.3. The JDK used is OpenJDK IcedTea6 1.11.4. The Java version is 1.6.0_24.

2 A paper storage server

The first part of the exercise is to write a system for storing papers. This consists of a server were the papers are stored and a client for accessing this server. The communication between these programs is done via RPC - Remote Procedure Calls, using Sun- RPC.

2.1 paperclient

The client supports the following operations: add, remove, list all the papers, give info about a paper and fetch the actual paper. These command are given as arguments and call corresponding functions on the server to handle them. The return value of these functions is passed to the client and if necessary the data is printed.

2.2 paperserver

The server stores the papers and replies to the aforementioned requests by the client. There are a couple of challenges with implementing the server. Two will be discussed in question A and B.

A third challenge is how to store the paper internally. The assignment states that they should be stored in main memory. The choice is made to store them in an ordered linked list. This allows for a flexible storage that only uses the memory that it needs.

The reason the list is ordered is that the assignment states that it is possible there are a lot of remove operations. We also have to appoint an ID to a newly added paper. This means there has to be a way of reusing old ID's. The ordered linked list allows for easy locating of the first free ID, guaranteeing at any time, even after remove operations, a total of `INT_MAX` papers can be stored.

2.3 Questions

Question A To be able to send an arbitrary number of paper detail structures, a linked list is used. Each of the detail structures has a pointer to the next structure. When receiving, the receiver continuously receives these structures, until the next pointer of a structure is `NULL`.

Question B The Sun-RPC requires that you know beforehand how much memory size a datastructure requires. This requires that the code supports arbitrary sizes of data, but that it does keep track of how large this data is. When reading

a paper in the client, this is achieved by checking the size the file has on disk using `stat()`. When using fetch the size of the paper is retrieved from the meta data stored with it.

For the actual communication of the data, the `opaque` datatype from Sun-RPC is used. This creates a data structure in which both the binary data and the size of this data can be stored, allowing the receiver to allocate the required amount of memory to store the data in.

3 A hotel reservation server

The second part of the exercise is to create a hotel reservation server using Java RMI. The way RMI is set up, the implementation is comparable with a normal program. The requests are send implicitly by executing functions on the `Hotel` object, which is on the server, from the client.

The client supports the following commands: list how much rooms of which type are free, book a room of a certain type with the name of the guest and list all the guests.

The server has a Hotel object which has functions that correspond with these commands, which are invoked from the client.

4 Hotel Reservation Gateway

The final part of the exercise is to implement a gateway to this server. This gateway translates requests over a socket into remote method invocations to the earlier created hotel server. The gateway and any client connecting to it take the place of the `hotelclient` program that was created in the previous part of the assignment.

The exercise also requires writing a client in C. This can be seen as an 'example' client, since any client written in any language that follows the used protocol will work with the gateway.

4.1 hotelgw

The gateway program is a derivative of the hotelclient from the previous exercise. The difference is that it does not use arguments for input and does not print its output to stdout. Instead, it uses a socket connection for this. The request comes in over the socket, is 'translated' to be handled over RMI, and the response is translated back to be send over the socket to the connected client.

4.2 hotelgwclient

The created client is written in C. It takes the commandline arguments, concatenates them to a string and sends it over a socket connection it creates to the gateway program. The answer from the gateway is received and printed to `stdout`.

4.3 Questions

Question C The server is implemented as a sequential server. Java does only support this or threaded servers. However, the hotel server of exercise is sequential as well. Since hotelgw is only a translation layer between the client and the server, there is no advantage in making this server threaded. It would only result in threads waiting for access to the server. In addition, when using threads there is need for synchronization, which would only decrease performance since no performance can be gained in this case.

Question D The protocol is defined as a string containing the commandline arguments as they were provided in the first client, without the address of the server. This will allow any client program to simply take the input, concatenate it and send it over the socket connection, without need for translation or interpretations of the given commands. More advanced clients can still easily create a string to send.

The reply is send as a printable string from the gateway. The string that is send is the actual output that the original client gives. Most client programs will simply print this string, but more advanced clients can easily split the data on the `\t` character.

Question E The server needs to have the files associated with the `hotelserver`, which means the `Hotel.class` and the actual server implementation.

The machine that runs the gateway needs the `hotelgw` needs the `Hotel.class` file as well as the files that implement the gateway.

The client using `hotelclient` needs the `Hotel.class` file and the implementation of the client program.

Note that any client connecting to the gateway does not need extra files other than the client program used, making this a far more flexible method than directly connecting to the RMI server. It is not required to distribute the `Hotel.class` file to anyone wanting to use the system in this case.