Assignment 1 Handout

Programming Assignment 1: Simple Client-Server

Due: February 3 at midnight

In this assignment, you will implement a simple client-server application (written in Python). The application is a car rental application, in which a client manages a set of car rental reservations. Clients can contact the server to retrieve information about available cars, dates, check existing reservations for a specific car, and make and withdraw reservations. We will use this application as a running example throughout the whole term, making it increasingly more sophisticated (and realistic), by adding, among other things, support for multiple clients and servers, data replication, etc.

In this assignment, you have to implement a single client and a single server that communicate over the network using sockets. You can choose whether you want to use datagram sockets (UDP) or streaming sockets (TCP) for the interprocess communication, as discussed in class and in the labs. However, in future applications, we will work with datagram sockets only. Also, to keep the implementation flexible, your server should expect a command-line parameter that provides the port number it should listen on, and the client should expect two command-line parameters that tell it where to find the server (i.e., a machine name and a port number).

Server Code

The server stores its information persistently in four local text files. You can download examples of these files from the course website. Each file contains a specific set of data:

• *cars.txt*: lists all available cars (such as **HondaPilot** or **BMWX3**), one car per line
• *dates.txt*: lists all dates for which a reservation can be made (**Thursday-2023-01-26**), one date per line
• *reservations.txt*: lists all existing reservations, one reservation per line

The server, upon starting up, has to read in the information from these files. The information in the first two files will not change during the server's lifetime. Reservations will come and go as the client interacts with the server. Ultimately, when the application terminates, the file needs to reflect the set of reservations at that point, so if the server is restarted, it can get an up-to-date view of the existing reservations. In this assignment, we are not (yet) worried about servers crashing or multiple servers having to coordinate their car rental reservations, so it is sufficient to update the reservation file when the server program terminates.

The server will receive client requests, process them, and send replies back to the client. In particular, the server has to process the following set of requests:

• *cars*: returns a list of all cars for which reservations can be made
• *dates*: returns a list of all dates for which a reservation can be made

• *check* <*car*>: returns all existing reservations for <car>
• *reserve* <*car*> <*date*>: enters a reservation for <car> for date <date>
• *delete* <*car*> <*date*>: delete a reservation for <car> for date <date>
• *quit*: server updates file *reservations.txt* and quits

The server should implement checks to ensure that requests are sensible. For example, a *reserve* request can only be made for a valid car, date, and should not conflict with an already existing reservation. Similarly, deleting a reservation should result in an error message to the user if such a reservation does not exist.

Running the Server

Assuming your server implementation is stored in a file *Server.py*, you will run the server by executing the following command:

python Server.py port

where port is the port number the server listens on. Remember that you have to pick a port number greater than 1024, because only processes running with root (administrator) privilege can bind to ports less than 1024.

The Client

You should write a client that basically works as the user interface to the application. The client allows a user to enter commands, sends them to the server, receives the replies, and displays the results to the user. As such, in this assignment, the client is probably simpler than the server. Assuming your client is implemented in a file *Client.py*, then as mentioned above, you should write the client so that it starts with the following command:

python Client.py host port

where host is the name of the computer the server is running on and port is the port number it is listening to. Note that you can run the client and server either on different machines or on the same machine. When running the client and server on the same machine, you can use either the IP address assigned to the PC or use the generic loopback IP address 127.0.0.1.

The primary task of the client is to establish and manage the connection to the server. One of the error cases to worry about in a client-server application is the loss of messages over the network or the partial failure of one component. How to deal with these errors depends on the type of sockets you choose to use. For datagram sockets, using UDP as the underlying transport layer protocol, the client has a bit more work to do explicitly than with streaming sockets. Because UDP is an unreliable protocol, some of the packets sent to the server may be lost, or some of the packets sent from server to client may be lost. For this reason, the client cannot wait indefinitely for a reply to a message. You should have the client wait up to one second for a reply; if no reply is received, then the client should assume that the packet was lost during transmission across the network. You

Assignment 1 Handout

will need to research the API for DatagramSocket to find out how to set the timeout value on a datagram socket (see also Lab 1).
When developing your code, you should run the server on your machine, and test your client by sending packets to *localhost* (or 127.0.0.1). After you have fully debugged your code, you should see how your application communicates across the network with a server run on a separate computer, across a network (if you have access to more than one PC in your location).

Message Format

The application requires the client and server to exchange requests that embed commands from the client to the server and replies (from the server to the client) that either contain the results of executing that command on the server or an appropriate error message/indication. You are free to design whatever message structure you choose. Some options are to exchange strings, to exchange more structured data types that separate commands from (optional) parameters, to encode commands with integers, etc. Your solution will not be tested with other solutions, so only your client and server implementations need to agree on the structure of messages.

Submission Requirements

Submit your solution using Brightspace. Your solution should, at the very least, provide an implementation of the client (in file Client.py) and the server (in file Server.py). Do NOT submit a ZIP file or any other archive, rather submit individual files. You also do not need to submit the input *.txt files. Your programs should run as is with the current version of Python (3.10.1), and the code should be well documented. Marks will be based on:

- Completeness of your submission
- Correct solution to the problem
- Following good coding style
- Sufficient and high-quality in-line comments
- Adhering to the submission requirements

The due date is based on the time of the Brightspace server and will be strictly enforced. If you are concerned about missing the deadline, here is a tip: multiple submissions are allowed. So you can always submit a (partial) solution early, and resubmit an improved solution later. This way, you will reduce the risk of running late, for whatever reason (slow computers/networks, unsynchronized clocks, etc.).

Assignment 1 Handout

Sample outputs