CS 421 – Computer Networks
Programming Assignment II
Application Level Routing

Due: 21 December 2018

I. Introduction

In this programming assignment, you are asked to implement a program in Java. The program is a simple load balancer program which is an extended and modified version of the routing program that you implemented in the first programming assignment. The program which sends the original data is called the **client**. Client has an arbitrary length array of integers, each of which should be fed into a sequence of functions, which is again determined by the client. The servers to which the client wants to send the messages are called **operators**. Unlike the first programming assignment, each operator is identical (they do not have separate functionalities). However, the client does not have any information about where these operators are and how busy they are. Hence, we need a routing program called **load balancer** which distributes the integers to the operators according to their levels of occupancy.

Client and operators use a custom application layer protocol slightly different than the protocol in the first programming assignment; so, be sure to follow the specifications of the protocol. We will provide you with the tools to test your load balancer program. Design specifications for this assignment do not require any multi-threading. The assignment should be done using the Java programming language using TCP sockets.

Constraints For this programming assignment, you **MUST NOT** use the wildcard import keyword (e.g., "`import.net.*;`") and instead explicitly import all necessary classes you use.

The goal of the assignment is to make you familiar with the application layer and TCP. You have to implement your program using the Java Socket API of the JDK. If you have any doubt about what to use or not to use, please contact your teaching assistant.

When preparing your project please mind the formatting as this project will be auto-graded by a computer program. Any problems in the formatting can create problems in the grading; while you will not get a zero from your project, you may need to have an appointment with us for a manual grading. Problems caused by Windows/Linux incompatibility will have no penalty. However, errors caused by incorrectly naming the project files and folder structure will cause you to lose points.

II. Design Specifications

a. Running the Program

Your program must be **a console application** (no graphical user interface, GUI, is allowed) and should be named as `LoadBalancer` (i.e., the name of the class that includes the `main` method should be `LoadBalancer`). Your program should run with the command

```
java LoadBalancer <Addr> <OpAddr1> <OpAddr2> … <OpAddrN>
```

where "`< >`" denotes command-line arguments. These command-line arguments are:

- `<Addr>` [Required] The IP address and port number to which your load balancer program will bind. You should use the same value for starting up the testing code. The format of this argument will be like `127.0.0.1:9999`
- `<OpAddri>` [Required] The IP address and port number of operator `i`
  Your program should be able to take an indefinite number `N` of `OpAddr` arguments for `i` = `1` to `N`, where `N` ≥ `1`.

e.g.,

```
java LoadBalancer 127.0.0.1:9999 127.0.0.1:10000 127.0.0.1:10001
127.0.0.1:10002 127.0.0.1:10003 127.0.0.1:10004
```

In the above example, the load balancer is bound to `127.0.0.1`, i.e., localhost, on port `9999`. It also knows that there are 5 operators bound to ports `10000`, `10001`, `10002`, `10003` and `10004` with the same IPs respectively.

When a user enters the above command, your program will start to listen for incoming connections at `127.0.0.1:9999`. Now you can start the test program to test your load balancer program with the following command **(you should start the test program after you run your load balancer program)**:

```
python Client.py <Addr>
```

where <Addr> is the same as <Addr> in the java command above to run LoadBalancer. Just like the first assignment the testing program will launch the client and 5 operators at localhost (i.e., 127.0.0.1). See the table below of all operators and their ports.

Type	IP	Port
Client	127.0.0.1	9999
Operator	127.0.0.1	10000
Operator	127.0.0.1	10001
Operator	127.0.0.1	10002
Operator	127.0.0.1	10003
Operator	127.0.0.1	10004

These are the default ports on which the program will run. If you have port conflicts with some other program, you can start the tester program with

```
python Client.py <Addr> <port1> <port2> <port3> <port4> <port5>
```

to manually set the ports of the operators.

b. Application Layer Protocol

The custom application layer protocol defined for this assignment is an extended and modified version of the protocol for the first assignment.

Basic format of a message is as follows:

<type>:<message>\n

where <type> is one of the 5 predefined strings given below, <message> is the content of the message (which can be empty for some of the types) and \n is the newline character indicating end of the message. Please note that every message in this custom protocol should be sent as a **string**, encoded with **UTF-8** encoding.

There are 5 possible message **types** in the protocol:

1) `DATA`: Used for sending integer sequences. Some example uses:
   - `DATA:12\n` (for sending a single integer)
   - `DATA:12,65645,34236,9800,1,2,0,345\n` (for sending an array)

   Note that if more than one value is sent, each integer should be separated with a comma. Unlike the first assignment, the numbers can have an arbitrary number of digits.

2) `FUNCS`: Used for sending a sequence of function names or a single function name. Some example uses:
   - `FUNCS:f1\n` (for sending a single function name)
   - `FUNCS:f1,f2,f3,f2,f2,f2\n` (for sending a sequence of function names)

   Function names such as f1, f2, f3 correspond to different functions implemented by the operators. Note that the functions are identical in each operator. In this assignment, we have 3 functions:

   - `f1`: Multiply by 17
   - `f2`: Take the square
   - `f3`: Add 53

   Therefore, a message like `FUNCS:f1,f3,f2,f2\n` corresponds to the following operation:

   $$((x \times 17 + 53)^2)^2$$

   where x is the input. Also note that, although only 3 functions are implemented, **your program should be working with an arbitrary number of functions**, i.e., **we are going to test your programs with different numbers of functions and different functions**.

3) `GETOCC`: Used by the load balancer to get the occupancy levels of the operators. This message type does not have a `<message>` part. Example usage:
   - `GETOCC:\n`

Please note that even if there is no `<message>`, there is still a colon (:) before the newline due to the definition of the protocol.

4) `OCC`: Used by the operators to report the occupancy levels. `<message>` part includes an integer in the range [0, 100], which indicates the percentage of how busy the operator is. For example, 0 indicates that the operator has no other job to do, and 100 indicates that the operator is fully occupied with other jobs. Your load balancer program will use this information to decide how much data should be distributed to each operator. Some example uses:
    - `OCC:0\n`
    - `OCC:100\n`
    - `OCC:12\n`
    - `OCC:58\n`
5) `END`: Used by the load balancer program to tell operators to close their connections. This should be sent after all the operations are done and the results are gathered from the operators. Like `GETOCC`, this message type does not have a `<message>` part as well. Example usage:
    - `END:\n`

    Please note that even if there is no `<message>`, there is still a colon (:) before the newline due to the definition of the protocol.

c. LoadBalancer Design Specifications

The scenario for this assignment is summarized as follows:

1) Client has an arbitrary length sequence of integers and an arbitrary length sequence of function names. It wants each integer in the sequence to be processed by the given function sequence in order. It connects to your LoadBalancer program and sends a `DATA` message to send the numbers and a `FUNCS` message to send the function names **in no specific order**. That is, it can first send the `DATA` message, then `FUNCS` message, or vice versa (you need to check the message type to determine the type of the message).
2) LoadBalancer receives the sequence of numbers and the sequence of functions names from the client. Then, for each operator, it sends a `GETOCC` message and receives an `OCC` message indicating the occupancy level of each operator.

3) LoadBalancer divides the number sequence into different length sequences for each operator according to the procedure that will be explained soon.
4) For each function in the function sequence,

   For each operator,

   i. LoadBalancer sends the corresponding portion (calculated in step 3) of the number sequence to the operator using `DATA` message,
   ii. sends the current function with a `FUNCS` message,
   iii. receives the result array from the operator.
5) After all the numbers in the input sequence are processed with the desired function sequence, LoadBalancer sends an `END` message to each of the operators.
6) LoadBalancer sends the resultant number sequence to the client with a `DATA` message.

For the calculation in step 3, LoadBalancer should use the following formula:

$$p_i = \frac{\frac{1}{u_i + 1}}{\sum_{j=1}^{N} \frac{1}{u_j + 1}}$$

$$(No.\ of\ elements\ for\ i^{th}\ operator) = floor((No.\ of\ all\ elements) \times p_i)$$

where u_i is the level of occupancy of the i^{th} operator, N is the number of operators and $floor$ is a floor function that takes the integer part of a floating-point number (you can use integer casting or Math.floor() in Java). This formula basically indicates that each operator should receive an amount of data inversely proportional to its occupancy level. Note that, due to the floor operation, **some of the data can remain undistributed**. If that happens, **you can send the remaining data to any of the operators**.

d. Example

The table below shows an example to further clarify the design specifications. In this example we have a client (Client), two operators (Operator1, Operator2), and a load balancer (LoadBalancer). We have a short sequence of numbers of length **11** and a series of **2** functions (f1, f3), functionalities of which are described in the Application Layer Protocol section. Occupancy level of Operator1 is **75** and occupancy level of Operator2 is **25**. If we apply the formula above to obtain p_1 and p_2, we have:

$$p_1 = \frac{\frac{1}{75+1}}{\frac{1}{75+1}+\frac{1}{25+1}} = 0.25$$

$$p_2 = \frac{\frac{1}{25+1}}{\frac{1}{75+1}+\frac{1}{25+1}} = 0.75$$

Using these values, we can determine how much data we should send to each operator:

$$(No.of\ elements\ for\ \ Operator1) = floor(11 \times 0.25) = 2$$

$$(No.of\ elements\ for\ \ Operator2) = floor(11 \times 0.75) = 8$$

Note that 2 + 8 = 10, but we have 11 inputs. As described in the previous section, we can distribute that remaining input to any of the operators. Let us choose Operator1 for example. Then, we should send 2 + 1 = **3** numbers to Operator1 and **8** numbers to Operator2.

Time	Message Direction	Message	Comments
1	Client →LoadBalancer	DATA:0,1,2,5,6,10,11,1,2,10,3\n	Number sequence
2	Client →LoadBalancer	FUNCS:f1,f3\n	Function sequence
3	LoadBalancer→Operator1	GETOCC:\n	Get occupancy of Operator1
4	Operator1→LoadBalancer	OCC:75\n	Occupancy = 75
5	LoadBalancer→Operator2	GETOCC:\n	Get occupancy of Operator2
6	Operator2→LoadBalancer	OCC:25\n	Occupancy = 25
7	-	-	LoadBalancer calculates p_1=0.25 and p_2=0.75
8	LoadBalancer→Operator1	DATA:0,1,2\n	Send the first 3 numbers to Operator1
9	LoadBalancer→Operator1	FUNCS:f1\n	Send the first function
10	LoadBalancer→Operator2	DATA:5,6,10,11,1,2,10,3\n	Send the next 8 numbers to Operator2
11	LoadBalancer→Operator2	FUNCS:f1\n	Send the first function
12	Operator1→LoadBalancer	DATA:0,17,34\n	Receive
13	Operator2→LoadBalancer	DATA:85,102,170,187,17,34,170,51\n	Receive
14	LoadBalancer→Operator1	DATA:0,17,34\n	Repeat steps from 8 to 13

			for the next function
15	LoadBalancer→Operator1	FUNCS:f3\n	
16	LoadBalancer→Operator2	DATA:85,102,170,187,17,34,170,51\n	
17	LoadBalancer→Operator2	FUNCS:f3\n	
18	Operator1→LoadBalancer	DATA:53,70,87\n	
19	Operator2→LoadBalancer	DATA:138,155,223,240,70,87,223,104\n	
20	LoadBalancer→Operator1	END:\n	Send END message to operators
21	LoadBalancer→Operator2	END:\n	Send END message to operators
22	LoadBalancer→Client	DATA:53,70,87,138,155,223,240,70,87,223,104\n	Send results back to the client

Assumptions and Hints

- Please contact your assistant if you have any doubt about the assignment.
- **Do not forget to check the output of the Client.py program** to see if your code is working (it will display "RESULTS ARE CORRECT") or not (it will print error messages accordingly). Note that Client.py might fail to display all the errors that you make; therefore, you might have to experiment a bit to correct your errors.
- You can modify the source code of the client and the operator for experimental purposes. However, do not forget that your projects will be evaluated based on the version we provide, with possibly **different length integer sequences, different length function sequences, different number of operators and different functions**.
- We have tested that these programs work with the discussed Java-Python combination.
- Do not forget that many of the socket exceptions you will get is probably because your program failed to close sockets from its previous instance. In that case, you can manually shut down those ports by waiting them to timeout, restarting the machine, etc.
- Be careful about the **colon (:)** and **newline (\n)** characters in the custom protocol messages.
- Remember that all the custom protocol messages sent over the sockets (even numbers) should be constructed as **strings** and encoded with **UTF-8** encoding.

Submission rules

You need to apply all of the following rules in your submission. **You will lose points if you do not obey the submission rules below or your program does not run as described in the assignment above.**

- The assignment should be submitted as an e-mail attachment sent to `bulut.aygunes[at]bilkent.edu.tr`. Any other methods (Disk/CD/DVD) of submission will not be accepted.
- The subject of the e-mail should start with [`CS421_2018FALL_PA2`], and include your name and student ID. For example, the subject line must be

  `[CS421_2018FALL_PA2]AliVelioglu20141222`

  if your name and ID are `Ali Velioglu` and `20141222`, respectively. If you are submitting an assignment done by two students, the subject line should include the names and IDs of both group members. The subject of the e-mail should be

  `[CS421_2018FALL_PA2]AliVelioglu20141222AyseFatmaoglu20255666`

  if group members are `Ali Velioglu` and `Ayse Fatmaoglu` with IDs `20141222` and `20255666`, respectively.
- All the files must be submitted in a zip file whose name is the same as the subject line **except the [`CS421_2018FALL_PA2`] part**. The file must be a `.zip` file, not a `.rar` file, or any other compressed file.
- All of the files must be in **the root of the zip file**; directory structures are not allowed. Please note that this also disallows organizing your code into Java packages. The archive should not contain **any file other than the source code(s) with .java extension.**
- The standard rules for plagiarism and academic honesty apply; if in doubt refer to the 'Student Disciplinary Rules and Regulation', items 7.j, 8.l and 8.m.