AUBURN UNIVERSITY

Programming Assignment 1

Objective of this assignment:

- To get familiar with developing and implementing TCP or UDP sockets.

What you need to do:

1. Implement a simple TCP Client-Server application

Objective:

The objective is to implement a simple client-server application using a safe method: start from a simple **working** code for the client and the server. You must slowly and carefully *bend* (modify) little by little the client and server alternatively until you achieve the ultimate goal. You must bend and expand each piece alternatively the way a black-smith forges iron. From time to time save your working client and server such that you can roll-back to the latest working code in case of problems. Not using this "*baby steps*" strategy may leave you with a ball of wax hard to debug.

For this programming assignment, you are advised (*not mandatory*) to start from the *Friend* client and server to implement this simple application.

Programming Assignment (TCP)

(100 points): TCP "*Billing System*" Client-Server

Implement the following Client-Server application that will use two programs: a client program myFirstTCPClient.java and myFirstTCPServer.java. The server will help a client build a bill. The client will send ***n*** pairs (**Q_i**, **C_i**) where **Q_i** and **C_i** are the quantity and code for Item i, respectively. The server returns the description D_i and unit cost CS_i for each item i. This information allows the client to display a bill with descriptions and costs for all items. Client and Server must successfully run on different Tux machines.

a) **Client: myFirstTCPClient.java**

This program must take two command line arguments: a hostname $\boldsymbol{H}$ and a port number $\boldsymbol{P}$. The hostname $\boldsymbol{H}$ is a name or a decimal dotted-quad IP address of the server S_v. The port number $\boldsymbol{P}$ is any valid port number where the server S_v binds to. On Tux machines, a valid TCP port number is in the range 10010-10200. Each team must use Port number 10010+TeamNumber. For example, if your team is Team17, your server S_v must use Port # 10027 (= 10010+17) for the server.

This client program must complete all these operations:

1) Create a TCP client socket connected to the server $S_v$running on the machine with hostname (or IP address) h bound to Port number P.

2) Repeatedly: prompt the user to enter a quantity number **Q_i** and a description code ***C_i*** until the user enters **Q_i** = -1. **Q_i** is a positive short integer in the range 0-32767. ***C_i*** is a positive short integer in the range 0-32767.

3) Build an array ***A*** of bytes with all ***n*** pairs (**Q_i**, ***C_i***) in the network byte order (*big endian*) terminated with -1 (equal to the short number 0xFFFF) following this protocol:

Request #	TML	Q_1	C_1	Q_2	C_2	…………	Q_n	C_n	**-1**

where

a) **Request #** is an identifier assigned to the request by the client. You can start from some random number and than increment the request number with each user billing request

b) ***TML*** is the *Total Message Length* in bytes including the bytes of the TML field.

4) Display in hexadecimal byte per byte the array ***A*** built in the previous question. You should display something like: *0x00 0xE2 0x4B* ………………….*0xFF 0xFF*. The actual values will depend on the pairs (**Q_i**, ***C_i***) entered by the user. If you have any doubt, check with your TA that you are forming the right array ***A*** complying with the required protocol and that you are correctly displaying the array A as expected.

5) Send to the Server S_v the array ***A***.

6) Wait for a response from the server.

As soon as the client receives the response, it must display byte per byte in hexadecimal the array of bytes received from the server. It should display something like: *0x01 0x3F 0x4A* ………………….*0xFF 0xFF*. The format of the response will follow this protocol:

Request #	TML	**TC**	L_1	D_1	CS_1	Q_1	L_2	D_2	CS_2	Q_2	…….	L_n	D_n	CS_n	Q_n	**-1**

where

a) ***Request #*** is the request number related to the request sent by the client. This field allows the client to match requests in case it issued multiple successive requests. Request # is a short integer (2 bytes).
b) ***TML*** is the *Total Message Length* in bytes including the bytes of the TML field. TML is a short integer.
c) ***TC*** *is the total cost of the bill* = $\sum_1^n CS_i \times Q_i$. The total cost is computed and inserted in the message by the server S_v. **TC** is an integer (4 bytes).
d) L_i is the length in bytes of the string D_i (see below) for Item i. L_i is one byte.
e) D_i is a string that is the description of the product that has the code C_i. The server S_v retrieves the string D_i from a csv file ***data*** it hosts. Its length L_i will not exceed 255 bytes. The encoding scheme for the string is the standard "default" encoding scheme (refer to the string representation in Module 2 Socket Programming). If the code C_i is not in the csv file data, then D_i is set to the string "Article Not Available".
f) CS_i is the unit cost of the product that has the code C_i. The server S_v retrieves the unit cost CS_i from a csv file ***data*** it hosts. CS_i is a short integer. If the code C_i is not in the csv file data, then CS_i is set to 0.

6) Check if the value TC is equal to $\sum_1^n CS_i \times Q_i$ where CS_i is the value received in the response and is the value sent in the request. If you do not have equality, you must display "Error: the total cost in the response does not match the total computed by the client.

7) If the check was successful, display the bill in a format understandable by an average Facebook user (see below). Note that the actual values will depend on the pairs (**Q**$_i$, CS_i) where **Q**$_i$ is the quantity for Item i sent in the request and CS_i is the unit cost received in the response.

Item #	**Description**	**Unit Cost**	**Quantity**	**Cost Per Item**
1	Pencil #HB	$1	3	$3
2	Nutrition Bars	$2	5	$10
….	…..	….	…..	…..
n	Chocolate Bars	$4	6	$24
			Total	**TC**

To implement the client myFirstTCPClient.java. **The client must be implemented in Java.**

The server is described below.

b) **Server: myFirstTCPServer.java**

This server program must take one argument: a port number P. The port number P can be any valid port number in the range 10010-10200. Each team must use Port number **PN** = 10010+TeamNumber. For example, if your team is Team17, your server S_v must use Port # 10027 (= 10010+17) for the server. The TA may use a port number different from **PN** when grading the work.

This server program must:

1) Create a TCP server socket
2) Wait for a client to connect……. When the server receives a request (message):
3) display the IP address and port # of the client,
4) **display byte per byte in hexadecimal the *request* array of bytes received**. The server should display something like: *0x00 0xE2 0x4B …………………0xFF 0xFF*. The actual values will depend on the pairs (**Q**$_i$, **C**$_i$) contained in the received byte array:

Request #	TML	Q_1	C_1	Q_2	C_2	…………	Q_n	C_n	-1

5) Build the server response (byte array)

Request #	TML	**TC**	L_1	D_1	CS_1	Q_1	L_2	D_2	CS_2	Q_2	…….	L_n	D_n	CS_n	Q_n	-1

as follows

a) Fill the Request # field with the Request # field in the received request
b) For each pair (**Q**$_i$, **C**$_i$), retrieve from the file F the description D_i and the unit cost CS_i, compute the running cost TC (Total Cost) and fill accordingly the response byte array.
c) For each string D_i, fill its length L_i
d) Fill each quantity **Q**$_i$ in the order received in the request.
e) Fill the trailer of the response with -1(0xFFFF)
f) Fill ***TML*** with the total number of bytes making the message including the bytes for the field TML itself.
g) **display byte per byte in hexadecimal the array of bytes of the response you just built**. The server should display something like: *0x01 0x3F 0x4A …………………0xFF 0xFF*. Note that the actual values will depend on the pairs (**Q**$_i$, **C**$_i$) received in the request.

4) **Error**: If the server receives a number of bytes not equal to TML, server must send a response that contains only Request # and -1:

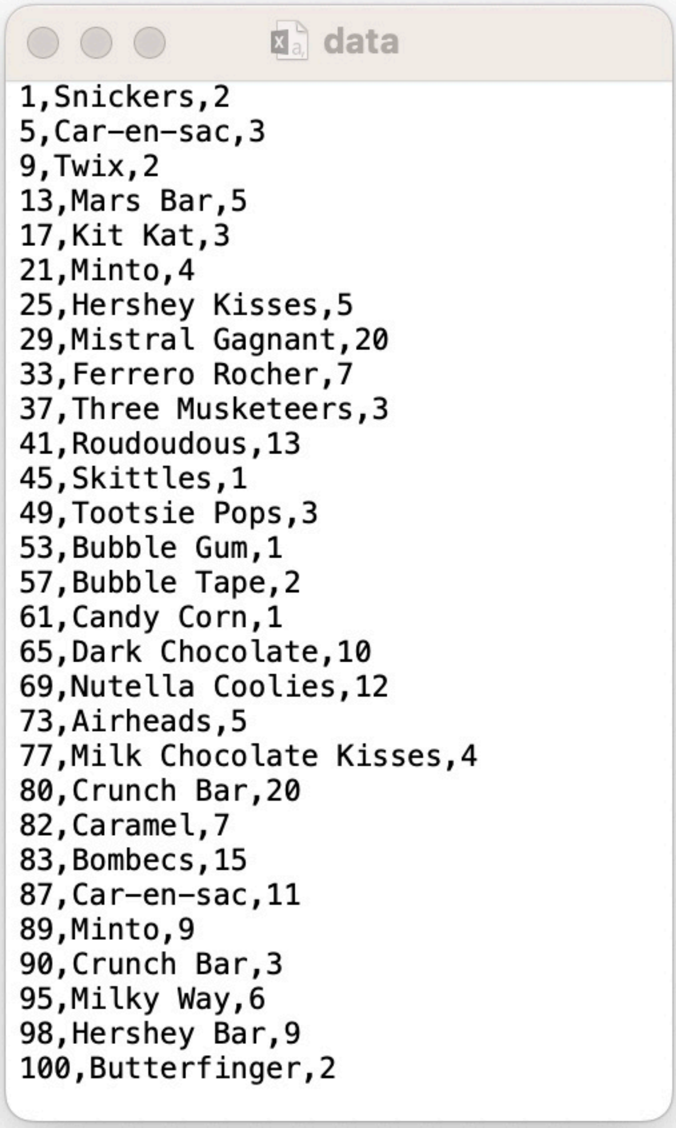
Request #	-1

To implement the server myFirstTCPServer.java, you should *consider* starting with the provided *Friend* code.

Bonus: If you implement the server in a language different from Java, you will get **5 points Bonus** points. For the language other than Java, the only constraint is that the language must already **be available and installed on Tux machines** (do not expect the TA to install any package to compile and execute the code you will submit). Check a Tux machine for the chosen language before starting to implement this protocol.

Data file

The data file $data$ uses the csv () format and is available for download on Canvas. This file must be accessible for the server Sv. The data file $data$ contains values (C_i, D_i, CS_i). The data file $data$ may have many more lines and actual values may be different from the figure below.



```
1,Snickers,2
5,Car-en-sac,3
9,Twix,2
13,Mars Bar,5
17,Kit Kat,3
21,Minto,4
25,Hershey Kisses,5
29,Mistral Gagnant,20
33,Ferrero Rocher,7
37,Three Musketeers,3
41,Roudoudous,13
45,Skittles,1
49,Tootsie Pops,3
53,Bubble Gum,1
57,Bubble Tape,2
61,Candy Corn,1
65,Dark Chocolate,10
69,Nutella Coolies,12
73,Airheads,5
77,Milk Chocolate Kisses,4
80,Crunch Bar,20
82,Caramel,7
83,Bombecs,15
87,Car-en-sac,11
89,Minto,9
90,Crunch Bar,3
95,Milky Way,6
98,Hershey Bar,9
100,Butterfinger,2
```

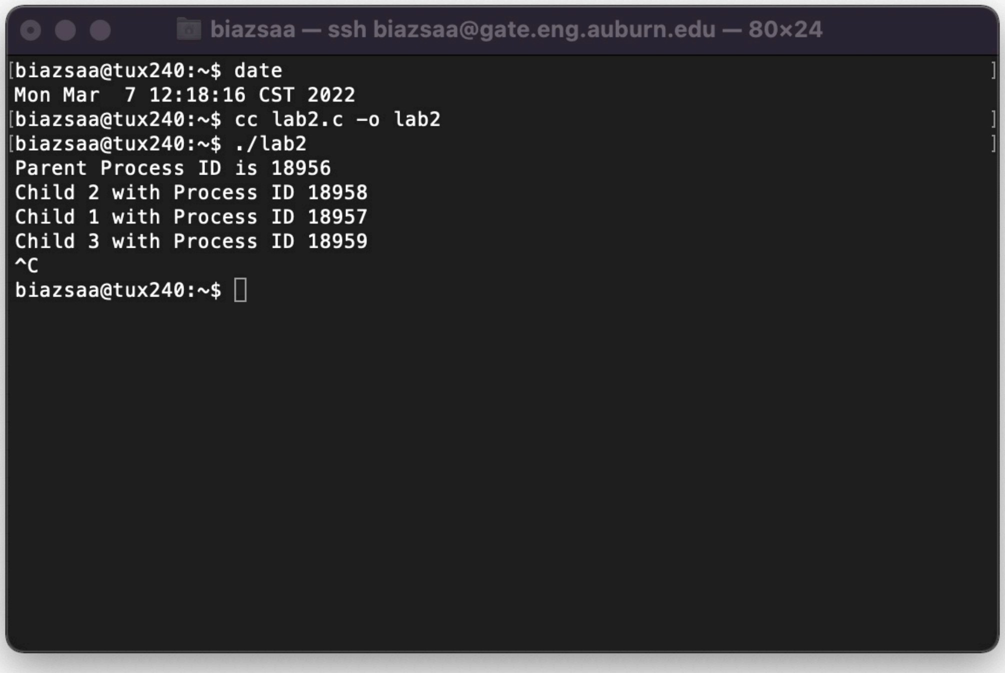
The grading and turn in instructions are described below.

DEMO

The user will enter the following pairs: (41,3), (87,5), (29,1), (21, 5), (83, 4). You must execute the client and the server on two different Tux machines and collect two screenshots for the client and the server, respectively.

Screenshots on machines other than Tux machines will not receive any credit. **To receive any credit, the screenshots must clearly show 1) the Tux machine name, 2) the username of one of the classmates, and 3) the date**. In other words, if any information (username, date, or tux machine name) is missing, there will be a 25 points penalty. **You must have two screenshots: one for the server and one for the client.** Below is a template screenshot containing the Tux machine, a username, and the date. **Avoid screenshots too small. Do not place the two screenshots side by side. If screenshots are not easily and conveniently readable, they will be considered missing. Screenshots must be easily and conveniently readable.**

Recall that the screenshots must use the pairs: (41,3), (87,5), (29,1), (21, 5), (83, 4) entered by the user.



```
biazsaa@tux240:~$ date
Mon Mar  7 12:18:16 CST 2022
biazsaa@tux240:~$ cc lab2.c -o lab2
biazsaa@tux240:~$ ./lab2
Parent Process ID is 18956
Child 2 with Process ID 18958
Child 1 with Process ID 18957
Child 3 with Process ID 18959
^C
biazsaa@tux240:~$ 
```

Insert Here the Screenshot for the Client (make a billing demo using the pairs: (41,3), (87,5), (29,1), (21, 5), (83, 4)) →

Insert Here the Screenshot for the Server (billing demo using the pairs: (41,3), (87,5), (29,1), (21, 5), (83, 4)) →

Report (a missing report incurs a 35 points penalty)

- Your report is **this** file description and must contain the following information:
    - whether the programs work or not (this must be just ONE sentence unless your program does not work as expected)
    - the directions to compile and execute your program (only the compilation/execution commands as the TA knows how to move your files and log on Tux machines)
    - the required demo screenshots of the execution of TCP client and server. **To receive any credit, the screenshots must clearly show the Tux machine, the username of one of the classmates, and the date**. To get the date, just run the command date before executing your program. **Each** missing screenshot will result in **a 50 points penalty**. A screenshot missing any information (tux machine name **OR** username OR date) will get a **25 points penalty.**

What you need to turn in:

- Electronic copy of your source programs (standalone, one by one, separately, in other words **NOT** in a zipped folder). This allows the grader to access separately on the Canvas grader each source program

AND

- A zipped folder that contains all source programs

AND

- Electronic copy of **this** file (including your answers) (standalone). Submit the file as a PDF.

For example, if you have 5 source programs, you must submit on Canvas 7 files (5 source code files, one zipped folder, and one PDF file).

Grading

1) The TCP client is worth 40% if it works well. "*Works well*" means that your code

a) **meets** the **protocol** specifications (20%) and the user interface (10%)

b) communicates correctly with YOUR server (10%). Furthermore, screenshots of your client and server running on Tux machines must be provided. The absence of screenshots or screenshots on machines other than Tux machines will incur **50 points penalty** per missing screenshot. A screenshot missing any information (tux machine name **OR** username OR date) will get a **25 points penalty**.

2) The TCP client is worth 10% if it works well with a **working** server from any of your classmates. The working server will be selected by the TA after all assignments are turned in. The faculty and the TA will make sure that the selected server meets all protocol requirements. While it is a good idea to check your client/server against server/client of other teams, it is not mandatory.

The server is graded the same as the client (20% + 10% + 10% + 10%).