

### Objective of this assignment:

- Develop and implement a simple application using UDP sockets. The client must be implemented in Java. The server can be implemented in Java or in your **preferred**<sup>1</sup> language (different from Java) for a **5 points bonus**. Insure that your preferred language is already installed and available on Tux machines. It is your responsibility to check ahead: do not check/test at the last minute.

### What you need to do:

#### I. Implement a simple UDP Client-Server application

##### Objective:

The objective is to implement a simple client-server application using a safe method: start from **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 your ultimate goal: meeting all requirements for this assignment. 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 bugs. Not using this "baby steps" strategy may leave you with a ball of wax hard to debug.

For this programming assignment, you are advised (optional, not mandatory) to start from the Friend client and server code (see lecture and resources) to implement this simple application. The Friend app was presented during the lectures. Consider using *Wireshark* to check whether the protocols you implement meet this assignment requirements. If using a language other than Java for the server, you are on your own. Ensure that your preferred language is already available on Tux machines. It is your responsibility to timely check. Start ahead. Do not wait until the last minute.

**Hint:** look at how to “How to get started?” below

#### Part A: Datagram socket programming

The objective is to design a **Calculating Server (CS)**. This calculating server performs bitwise Boolean and arithmetic computations requested by a client on signed integers. Your server must offer the following operations: 1) addition (+), subtraction (-), 3) bitwise **OR** (|), 4) bitwise **AND** (&), 5) division (/), and 6) multiplication (\*). A **client** will form a request following this protocol (byte per byte):

Field	TML		Operand 1	Operand 2	Request ID	Op Name Length	Op Name
Size (bytes)	1	1	4	4	2	1	Variable

#### Where

1) **TML** is the Total Message Length (in bytes) including TML. It is an integer representing the **total** number of bytes in the request.

2) **Op Code** is a number specifying the desired operation following this table

Operation	+	-		&	/	*
OpCode	0	1	2	3	4	5

- 3) **Operand 1** is a signed number making the first operand
- 4) **Operand 2** is a signed number making the second operand
- 5) **Request ID** is the request ID. This number is generated by the client to differentiate requests. You may use a variable randomly initialized and incremented each time a request is generated.
- 6) **Op Name Length** is the length in bytes of the operation name (see below what the *Operation Name* is). Pay attention: the length is the number of bytes used to encode the string. Recall that some encoding schemes use two bytes per character.
- 7) **Op Name** is name of the requested operation: "addition", "subtraction", "or", "and", "division", and "multiplication". The Op Name string must be encoded using **"UTF-16"**.

<sup>1</sup> The language must be available on Tux Machines. Check before developing/implementing.

Operands are sent in the **network byte order** (i.e., big endian).

**Hint:** create a class object *Request* like "Friend", but with the fields needed for a request ...

Below are two examples of requests. Bytes in the array are hexadecimal numbers.

**Request 1:** suppose the Client requests to perform the OR operation **240 | 4**: (This is the 5th request)  $(240)_{10} = 0xF0$  and  $(4)_{10} = 0x04$ . We omit the "0x" prefix for each byte expressed in hexadecimal.

13	02	00	00	00		00	00	00		04		06	F		00		00	72
										00			E					
										05								

**Request 2:** suppose the Client requests to perform the operation **227 & 183** (if this is the 12<sup>th</sup> request):

15	03	00	00	00	E3	00	00	00	B7	00	0C	08	FE	FF	00	61	00	6E	00	64
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

The **Server** will respond with a message with this format:

Total Message Length (TML)	Result	Error Code	Request ID
one byte	4 byte	1 byte	2 byte

## Where

- 1) **TML** is the Total Message Length (in bytes) including TML. It is an integer representing the **total** numbers of bytes in the message.
- 2) **Result** is the result of the requested operation
- 3) **Error Code** is **0** if the request was valid, and **127** if the request was invalid (Message length not matching TML)
- 4) **Request ID** is the request ID. This number is the number that was sent as Request ID in the request sent by the client. This will allow the client to match the results to the appropriate request.

In response to **Request 1 (240 | 4)** below

13	02	00	00	00	F0	00	00	00	04	00	05	04	00	6F	00	72
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

the server will send back: (. We omit the “0x” prefix for each byte expressed in hexadecimal. )

	00	00	00		00	00	05
--	----	----	----	--	----	----	----

In response to **Request 2 (227 & 183 )**,

15	03	00	00	00	E3	00	00	00	B7	00	0C	06	00	61	00	6E	00	64
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

the server would send back:

08	00	00	00	A3	00	00	0C
----	----	----	----	----	----	----	----

- a) **Repetitive Server:** Write a datagram **Calculating Server (ServerUDP.xxx)**. This server must respond to requests as described above. The server must bind to port (10010+ *TID*) and could run on any machine accessible on the Internet. *TID* is your Canvas team #. The server must accept a command line of the form: **prog ServerUDP portnumber** where **prog** is the executable, **portnumber** is the port the server binds to. For example, if your Team ID (GID) is Team 13 then your server must bind to Port # 10023.

Whenever a server gets a request, it must:

- display the request one byte at a time in hexadecimal (for debugging and grading purpose)
- display the request in a manner convenient for a typical Internet user: the request ID and the request (operands and required operation)

- b) Implement a datagram **client (ClientUDP.xxx)**

- Accepts a command line of the form: **prog ClientUDP servername PortNumber** where **prog** is the executable, **servername** is the server name, and **PortNumber** is the port number of the server. Your program must prompt the user to ask for an **Opcode**, **Operand1** and **Operand2** where **OpCode** is the opcode of the requested operation (See the opcode table). **Operand1** and **Operand2** are the operands. For each entry from the user, your program must perform the following operations:
  - form a request as described above
  - display byte per byte in hexadecimal the request that will be sent
  - send the request to the server and wait for a response
  - display the server's response byte per byte in hexadecimal (for debugging and grading purpose)
  - display the response of the server in a manner convenient for a typical Internet user: the request ID, the response and the error code (display Ok when error code is 0)
  - display the round trip time (time between the transmission of the request and the reception of the response)

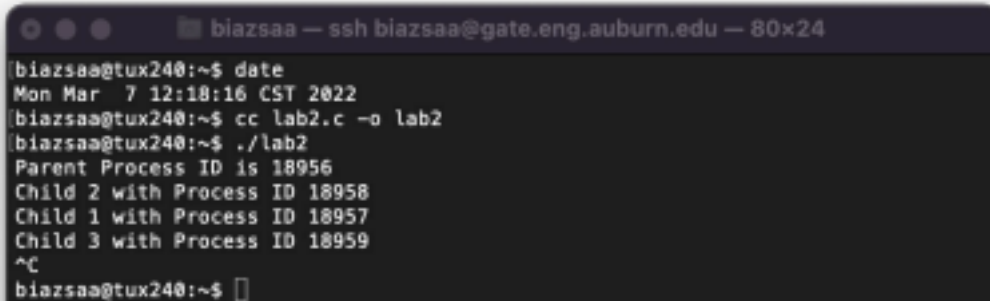
- viii. prompt the user for a new request. (Use some way to allow a client to quit. Just be explicit about how to quit)

To implement the server, you should consider (optional) starting with the Friend code. **If you implement a successful 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 it must already be installed on Tux machines. Check the Tux machines for your chosen language before you start implementing.**

**Your code must be neat and pleasant to read. Comment the code appropriately. If starting from some other code, delete all unnecessary instructions (do not just comment out the unnecessary instructions). A code not neat or pleasant will be penalized up to -30 points.**

### Data collection and analysis

For the client, report separately the min, average, and max round trip time. Include screenshots of your client and server executing on the Tux machines. Screenshots on machines other than the Tux machines will not receive any credit. **To receive any credit, the screenshots must clearly show the Tux machine name, the username of one of the classmates, and the date.** In other words, if any information (username, date, or tux machine name) is missing, the assigned credit for the assignment will be 0. **You must have two screenshots: one for the server and one for the client.** Here is a screenshot containing the Tux machine, a username, and a date. **Avoid screenshots too small. If screenshots are not easily and conveniently readable, they will be considered missing. Screenshots must be easily and conveniently readable.**



```
biazsaa — ssh biazsaa@gate.eng.auburn.edu — 80x24
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:~$
```

### How to get started?

1) Download all files (UDP sockets) to run the "Friend" application used in Module 2 to illustrate how any class object can be exchanged: Friend.java, FriendBinConst.java, FriendEncoder.java, FriendEncoderBin.java, FriendDecoder.java, FriendDecoderBin.java, SendUDP.java, and RecvUDP.java.

- 2) Compile these files and execute the UDP server and client. Ensure they work.
- 3) Create a new folder called Request and duplicate inside it ALL files related to the Friend class object
- 4) Inside the Folder Request, change ALL occurrences of "Friend" with "Request" including the file names.
- 3) Adapt each file to your application. Replace the fields used by Friend with the fields used by a request.
- 4) Aim to have the client send one request and have the server understand it (just like what we did with a friend object).
- 5) When your server will receive and print out correctly a request, then you need to send back a response...
- 6) Create a class object Response....

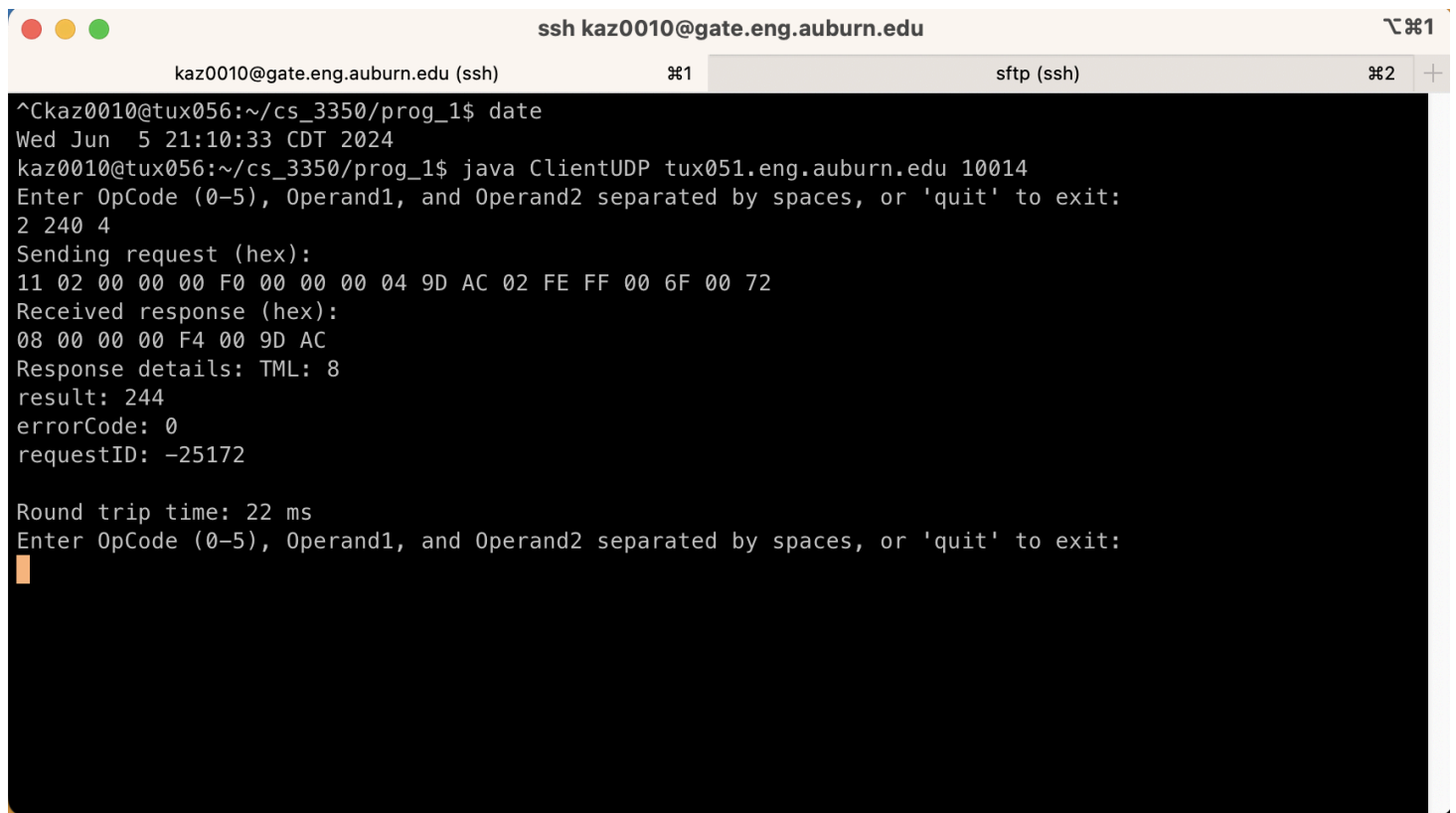
#### Screenshot 1:

```
ssh kaz0010@gate.eng.auburn.edu
kaz0010@tux239:~$ ping 131.204.14.051
PING 131.204.14.051 (131.204.14.41) 56(84) bytes of data.
64 bytes from 131.204.14.41: icmp_seq=1 ttl=128 time=3.31 ms
64 bytes from 131.204.14.41: icmp_seq=2 ttl=128 time=0.525 ms
64 bytes from 131.204.14.41: icmp_seq=3 ttl=128 time=0.472 ms
64 bytes from 131.204.14.41: icmp_seq=4 ttl=128 time=0.461 ms
64 bytes from 131.204.14.41: icmp_seq=5 ttl=128 time=0.450 ms
64 bytes from 131.204.14.41: icmp_seq=6 ttl=128 time=0.506 ms
64 bytes from 131.204.14.41: icmp_seq=7 ttl=128 time=0.519 ms
64 bytes from 131.204.14.41: icmp_seq=8 ttl=128 time=0.725 ms
64 bytes from 131.204.14.41: icmp_seq=9 ttl=128 time=0.697 ms
64 bytes from 131.204.14.41: icmp_seq=10 ttl=128 time=0.466 ms
64 bytes from 131.204.14.41: icmp_seq=11 ttl=128 time=0.515 ms
64 bytes from 131.204.14.41: icmp_seq=12 ttl=128 time=0.458 ms
64 bytes from 131.204.14.41: icmp_seq=13 ttl=128 time=0.558 ms
64 bytes from 131.204.14.41: icmp_seq=14 ttl=128 time=0.453 ms
64 bytes from 131.204.14.41: icmp_seq=15 ttl=128 time=0.524 ms
^C
--- 131.204.14.051 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14001ms
rtt min/avg/max/mdev = 0.450/0.709/3.310/0.699 ms
kaz0010@tux239:~$ date
Wed Jun 5 21:23:53 CDT 2024
kaz0010@tux239:~$
```

#### Screenshot 2:

```
ssh kaz0010@gate.eng.auburn.edu
kaz0010@tux239:~$ ping 131.204.14.056
PING 131.204.14.056 (131.204.14.46) 56(84) bytes of data.
64 bytes from 131.204.14.46: icmp_seq=1 ttl=128 time=0.844 ms
64 bytes from 131.204.14.46: icmp_seq=2 ttl=128 time=1.75 ms
64 bytes from 131.204.14.46: icmp_seq=3 ttl=128 time=0.588 ms
64 bytes from 131.204.14.46: icmp_seq=4 ttl=128 time=0.966 ms
64 bytes from 131.204.14.46: icmp_seq=5 ttl=128 time=0.987 ms
64 bytes from 131.204.14.46: icmp_seq=6 ttl=128 time=0.546 ms
64 bytes from 131.204.14.46: icmp_seq=7 ttl=128 time=0.723 ms
64 bytes from 131.204.14.46: icmp_seq=8 ttl=128 time=0.938 ms
64 bytes from 131.204.14.46: icmp_seq=9 ttl=128 time=0.723 ms
64 bytes from 131.204.14.46: icmp_seq=10 ttl=128 time=0.467 ms
64 bytes from 131.204.14.46: icmp_seq=11 ttl=128 time=0.951 ms
64 bytes from 131.204.14.46: icmp_seq=12 ttl=128 time=0.576 ms
64 bytes from 131.204.14.46: icmp_seq=13 ttl=128 time=0.509 ms
64 bytes from 131.204.14.46: icmp_seq=14 ttl=128 time=1.21 ms
64 bytes from 131.204.14.46: icmp_seq=15 ttl=128 time=0.730 ms
^C
--- 131.204.14.056 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14008ms
rtt min/avg/max/mdev = 0.467/0.834/1.753/0.320 ms
kaz0010@tux239:~$ date
Wed Jun 5 21:21:36 CDT 2024
kaz0010@tux239:~$
```

### Screenshot 3:

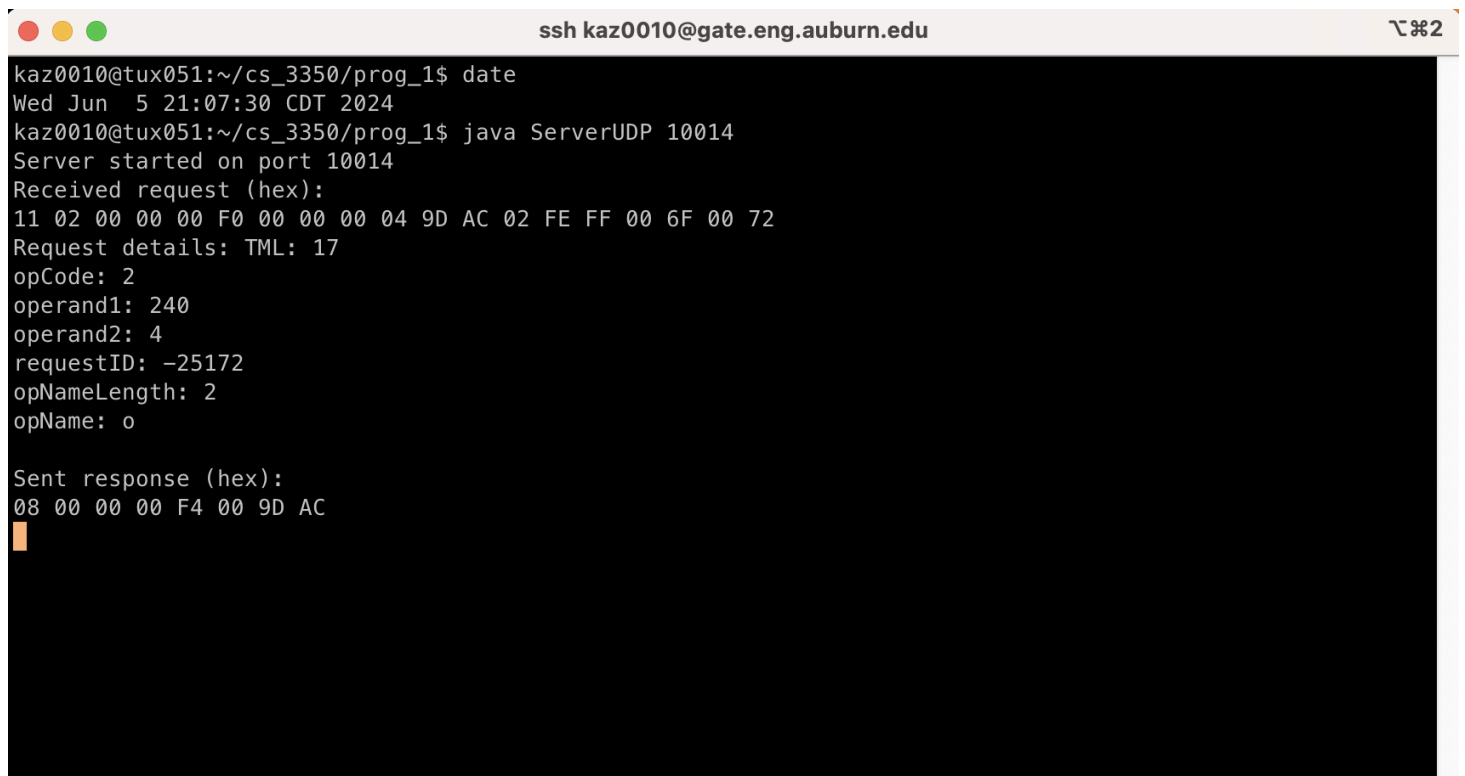


A terminal window titled "ssh kaz0010@gate.eng.auburn.edu" with a tab labeled "kaz0010@gate.eng.auburn.edu (ssh)". The terminal shows the following commands and output:

```
^Ckaz0010@tux056:~/cs_3350/prog_1$ date
Wed Jun  5 21:10:33 CDT 2024
kaz0010@tux056:~/cs_3350/prog_1$ java ClientUDP tux051.eng.auburn.edu 10014
Enter OpCode (0-5), Operand1, and Operand2 separated by spaces, or 'quit' to exit:
2 240 4
Sending request (hex):
11 02 00 00 00 F0 00 00 00 04 9D AC 02 FE FF 00 6F 00 72
Received response (hex):
08 00 00 00 F4 00 9D AC
Response details: TML: 8
result: 244
errorCode: 0
requestID: -25172

Round trip time: 22 ms
Enter OpCode (0-5), Operand1, and Operand2 separated by spaces, or 'quit' to exit:
█
```

### Screenshot 4:



A terminal window titled "ssh kaz0010@gate.eng.auburn.edu" with a tab labeled "kaz0010@gate.eng.auburn.edu". The terminal shows the following commands and output:

```
kaz0010@tux051:~/cs_3350/prog_1$ date
Wed Jun  5 21:07:30 CDT 2024
kaz0010@tux051:~/cs_3350/prog_1$ java ServerUDP 10014
Server started on port 10014
Received request (hex):
11 02 00 00 00 F0 00 00 00 04 9D AC 02 FE FF 00 6F 00 72
Request details: TML: 17
opCode: 2
operand1: 240
operand2: 4
requestID: -25172
opNameLength: 2
opName: o

Sent response (hex):
08 00 00 00 F4 00 9D AC
█
```

**Report (a missing report incurs a 30 points penalty)**

- Write a report that will report your results..
- Your report must contain the following information:
  - whether the programs work or not (this must be just ONE sentence)
  - the directions to compile and execute your programs
  - the information this assignment asks you to report (minimum, average, and maximum round trip times) and the required screenshots of the execution of the 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 programs. **Each** missing or incomplete screenshot will result in a **50 points penalty**.

Yes, my programs work on the Tux machine, in the above screenshot you can see examples of my client and server communicating with each other!

To compile and execute the program you must do the following do the following:

For each file run javac <FileName>.java

File Name:

ClientUDP

Request

RequestDecoder

RequestEncoder

Response

ResponseDecoder

ResponseEncoder

ServerUDP

To run the files use the java <FileName> command.

So above in screenshot 1 shows a ping being executed against my running server as you can see 15 packets were sent with min/avg/max are 0.450/0.709/3.310 respectively. Screenshot 2 shows a ping being executed against my running client as you can see 15 packets were sent and the min/avg/max are 0.467/0.834/1.753.

**What you need to turn in:**

- Electronic copy of your source programs (standalone, i.e. **NOT** in a zipped folder)
- Electronic copy of the report (including your answers) (standalone, i.e. **NOT** in a zipped folder). Submit the file as a Microsoft Word or PDF file.
- **In addition**, put all files in a zipped folder and submit the zipped folder.

**Grading**

1) Client is worth 40% if it works well:

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 the Tux machines will incur 50 points penalty per missing screenshot

2) UDP client is worth 10% if it works well with a working server from any of your classmates.

The server is graded the same as the client (30% + 10% + 10%).