**THE CLIENT SERVER SETUP**

# 

Table of Contents

# 

[**INTRODUCTION**](#_jhosz9fyrc6) **1**

[**THE CLIENT-SERVER SPLIT**](#_byd622423xz) **1**

[**CLIENT SIDE OPERATIONS**](#_jdm26fi9smjm) **3**

[Client Functions](#_z8w4lrimodhj) 3

[Headers and Structures](#_demdyopywjf) 4

[Accepting the Inputs](#_hg32go6q148u) 5

[Generation of Shares](#_rrdq59euq04a) 6

[Communication with the Servers](#_cvr343nlc93k) 9

[**SERVER SIDE OPERATIONS**](#_r6toohhlp9au) **11**

[Server Functions](#_puw2cctsr5c8) 11

[Variable Mapping and Communication Setup](#_g19ylpf61yi6) 12

[The Compute Server file ( compute\_server.cpp )](#_3jepgbu9uo8v) 12

[Building the Dot Product Operation](#_8cjwugnajidk) 13

[**STEPS FOR EXECUTION**](#_d110spmp4n93) **13**

| **NOTICE:**  The following documentation is created purely based on the interpretation and supervision of the MOTION2NX repository by IUDX, and can be subjected to changes based on correction in existing information, addition of new discoveries and removal of redundant data. |
| --- |

# 

# 

# 

# **INTRODUCTION**

As we have seen in the previous setup of Yao’s Millionaires problem, we have two mutually distrusting parties communicate with each other to find out who is worth more than the other. We used the **Greater than** operation through a **circuit** to find the solution to the problem. But what if we have parties that wish to add or multiply certain values or arrays? And what if we can create an environment where the server acts as an independent entity instead of side by side as a party? The following document explains how a four-way client-server setup can be created to perform the same functions having all the entities independent of each other.

# **THE CLIENT-SERVER SPLIT**

In the millionaires problem, we saw that two parties exchange input values in the form of shares without the aid of an external server. In an ideal scenario, this may look like a negligible violation in terms of the operations performed being dependent on the presence of at least one party. To overcome this, we simply separate the server operations from the parties, and create an independent client-server setup for our model.

The result of this is a 4-way communication model as displayed in Fig 1, where we have 2 servers and 2 parties. The parties, also called **data providers**, are responsible for providing the input values to the servers. The conversion to shares can be done at this level too since it would remove the concern of creation of private shares by an external party other than the party itself. The public shares can be broadcasted to the servers and the private shares sent separately for operations.

Two servers are required in this case because the operations that are performed ideally occur between two different parties and hence the two servers act as the virtual parties that hold the shares given to them by the Data Providers. In the following document, we shall discuss the implementation of a simple dot product example using this 4-way setup.

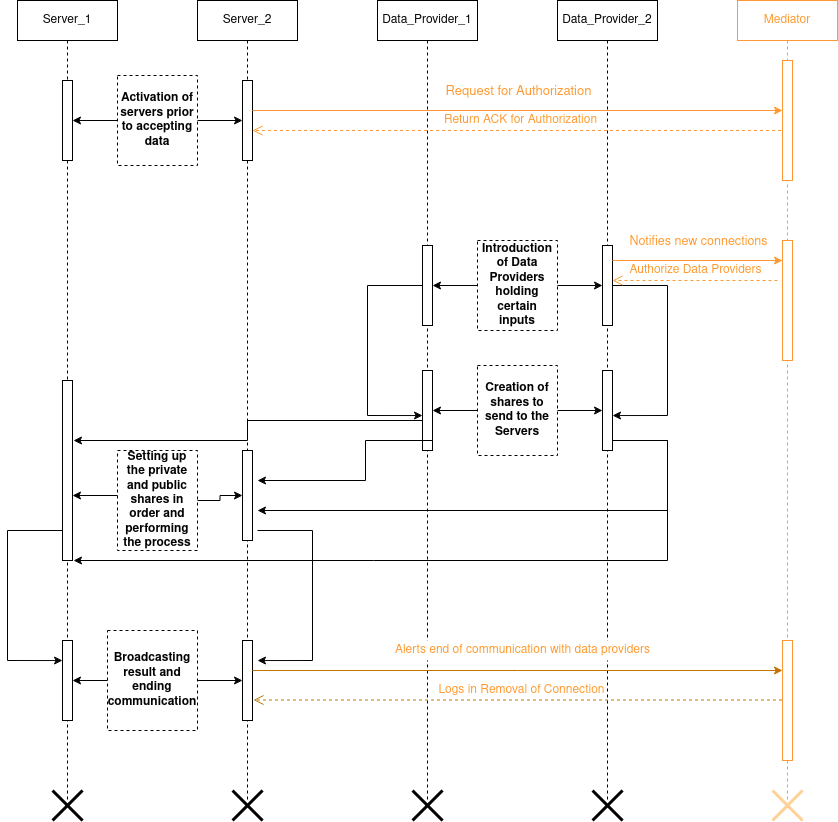


Fig 1: The four way client-server timing model (proposed)

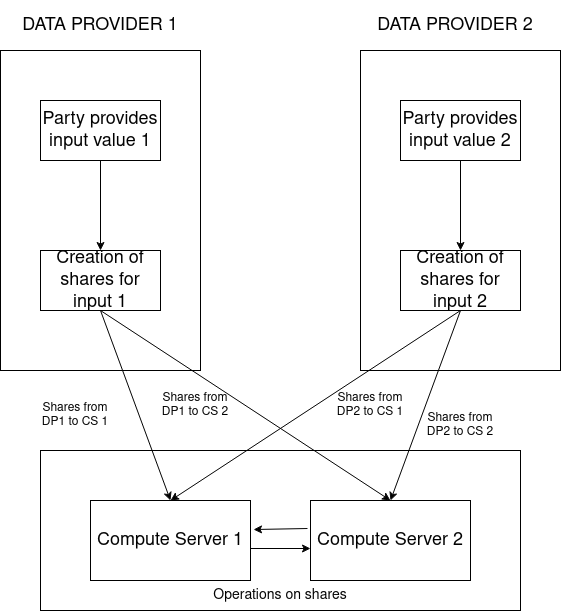


Fig 3: The four way setup architecture

# **CLIENT SIDE OPERATIONS**

This example is a simple implementation of multiplying two given arrays of elements to obtain their dot product. The operation requires a combination of multiplication and addition, that will be performed by the servers, whilst the creation of shares from the provided inputs is taken care of by the data providers themselves as seen in Fig 3.

## **Client Functions**

To understand how the client operates, we first need to understand a part of the communication layer as explained in the millionaires problem (doc). It uses a boost library called **asio** which is responsible for the implementation of asynchronous TCP processes. This allows us to communicate directly with the two servers without having to communicate with the other party. Along with establishing communication, the shares are generated through a key created using the pseudo-randomness generating algorithm called **Mersenne twister** algorithm. This key works nearly identical to the randomness\_generator() we have seen in the millionaires example, and is guaranteed to introduce a high level of randomness.

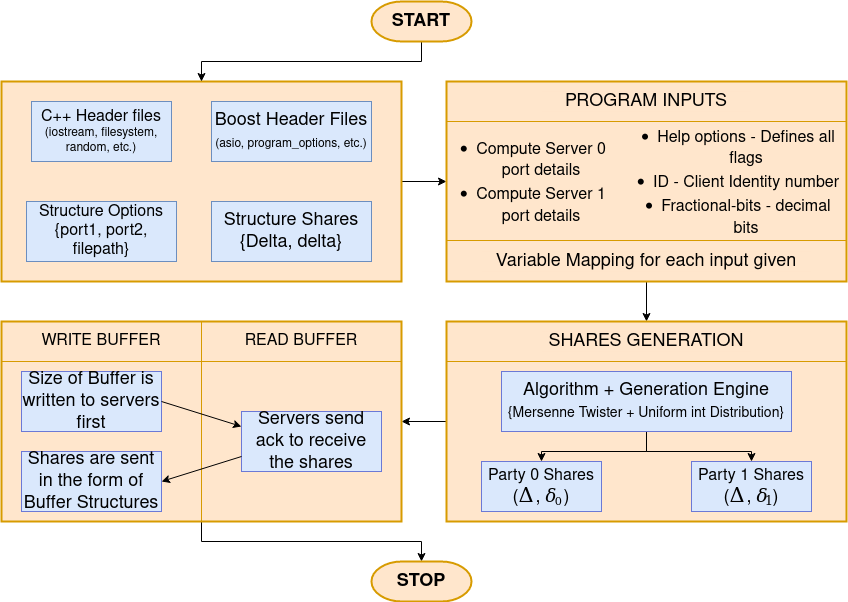


Fig 4: The Client side program flow

## **Headers and Structures**

Here we shall go through the code step by step to get a deeper understanding of what each function is responsible for. As mentioned earlier, the entire codebase is written in C++, where we make use of both standard as well as boost libraries.

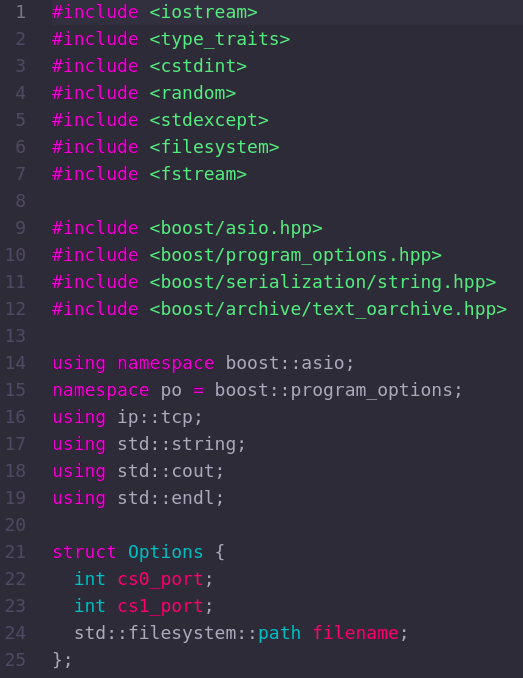


Fig 5: Headers and Structure for client code

As we can see in Fig 5, The client side uses multiple files meant for a variety of processes ranging from reading inputs, files, connecting the clients and servers and so on. The **Options** structure is used to store the ports of the two clients and the file that each client reads the input values from.

## **Accepting the Inputs**

We come to the next part of the code, which is parsing program options. This is a part of the boost library, and it allows us to provide a more structured format of giving command line arguments. Fig 6 gives us the function used for this purpose. We can see that there are a total of 4 flags that should be provided by each client for execution, and an additional help flag that displays how to execute the client side program.

The first part having **desc** creates the declaration and purpose of each flag, then the variable map or **vm** is used to define the operation of each flag and map them to their respective flag names.

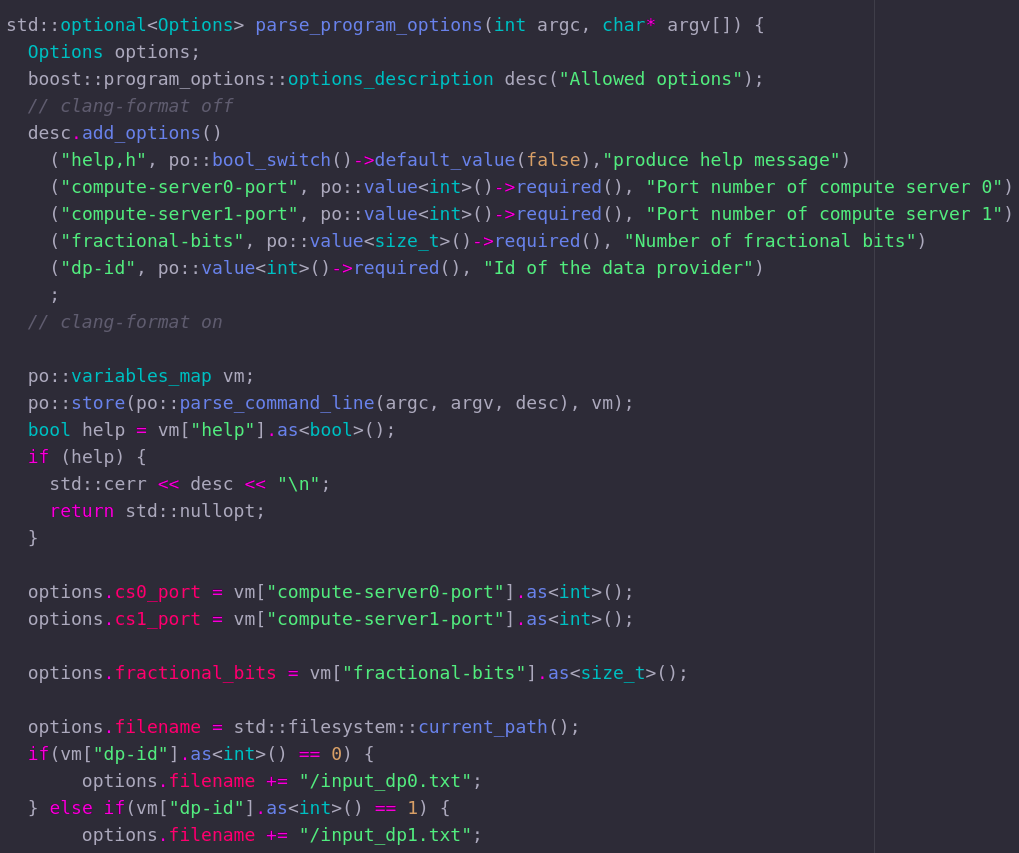


Fig 6: Parse Program Options

## **Generation of Shares**

With the inputs taken care of, the next step is to create shares of each of these inputs. As mentioned earlier we use the Mersenne Twister algorithm provided to us as a function from the standard C++ library. As seen in Fig 7, the **share\_generation** function takes in 5 arguments: The input file, the number of elements to be read, the shares to be created for the first and second clients, and finally the number of fractional bits (Number of bits after the decimal point).

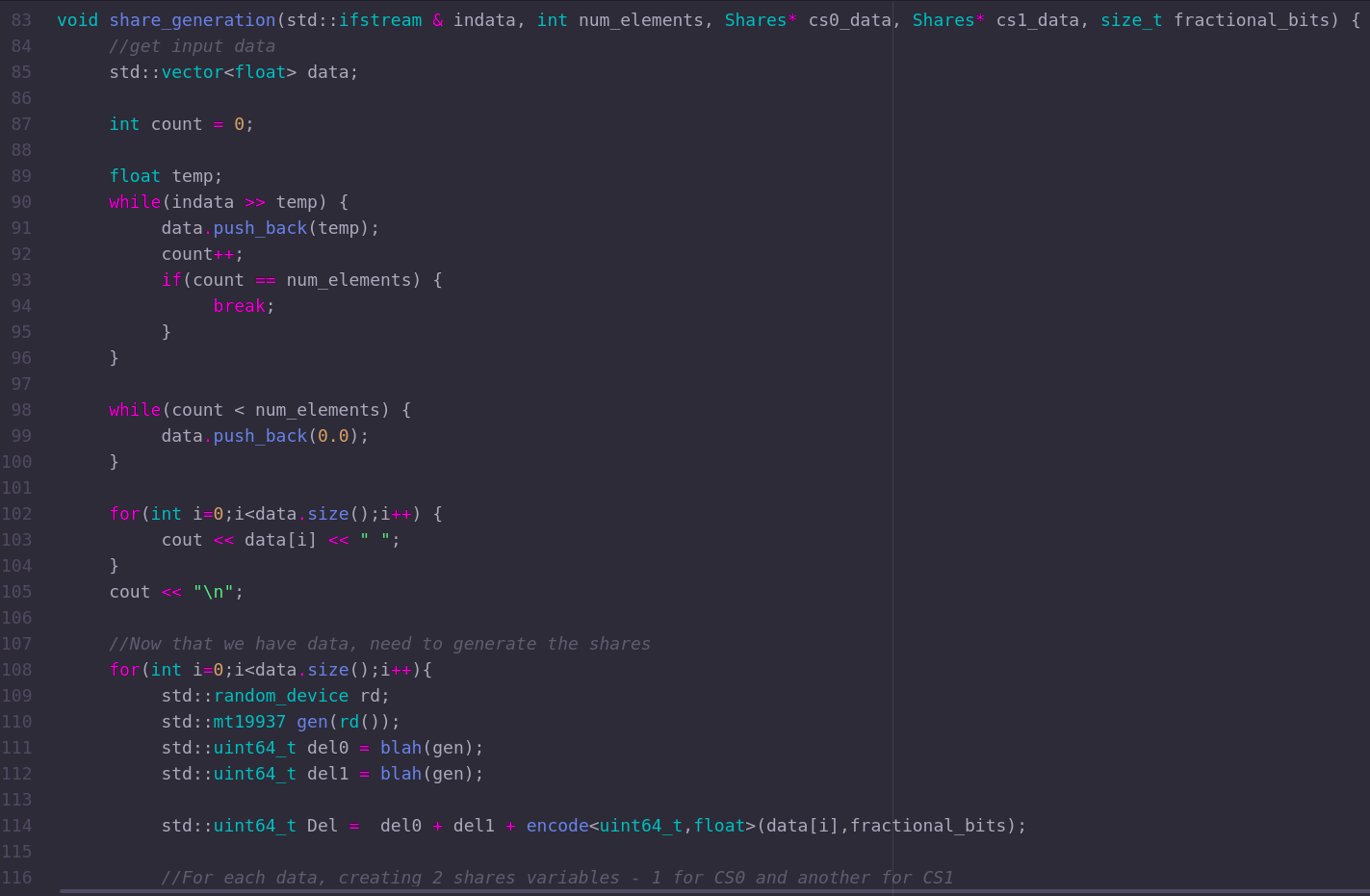


Fig 7: The shares generation function

The input values are stored into a temporary variable **temp** and the while loop adds in the extra required values to maintain correct matrix dimensions, that is, if we have a 3 x 2 matrix but only has 5 elements, this provides a padding by adding 0 as the sixth element to maintain consistency.

We then load up the **random\_device** which produces non-deterministic random numbers. As an addition to this, we enhance this by using the twister algorithm using the **std:mt19937**  to generate a random key value. This is then used to convert the input values of each client into their respective shares.

The **blah** function (Best function name ever) as shown in Fig 7 is used to create the *δ* shares for each client. It uses the algorithm to create these shares, which is then added up to create the Δ shares using the variable Del. These shares are then transferred to each party as shown in Fig 8, where each party is a structure holding its own public and private shares respectively. This is then returned to the main function, which then connects the two parties to the servers.

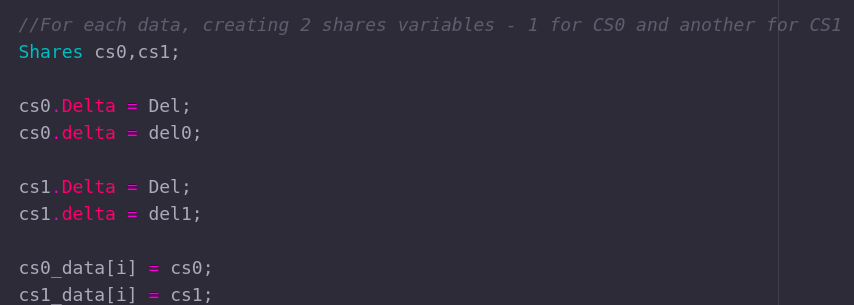


Fig 8: Distribution of Public and Private shares

The blah function as shown in Fig 9 comprises one argument which is the engine. In this program, this is nothing but the twister algorithm format created before, which is used here to create the randomness values. The **uniform\_int\_distribution** allows the program to select any number given the lower and upper limit as defined by **numeric\_limits**. It creates a probability distribution which allows it to pick numbers at random. Here it is used to create the function **dis** (distribution) that uses the engine to create the shares and returns it to the share generation function.

Furthermore a structure named **Shares** can also be seen holding a private and a public share variable. Each client holds one of this type of structure to define its individual private shares, even if the public share values would remain consistent across both clients.

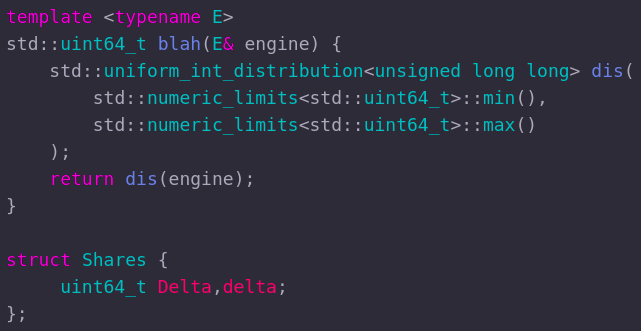


Fig 9: The blah function and Shares structure

## **Communication with the Servers**

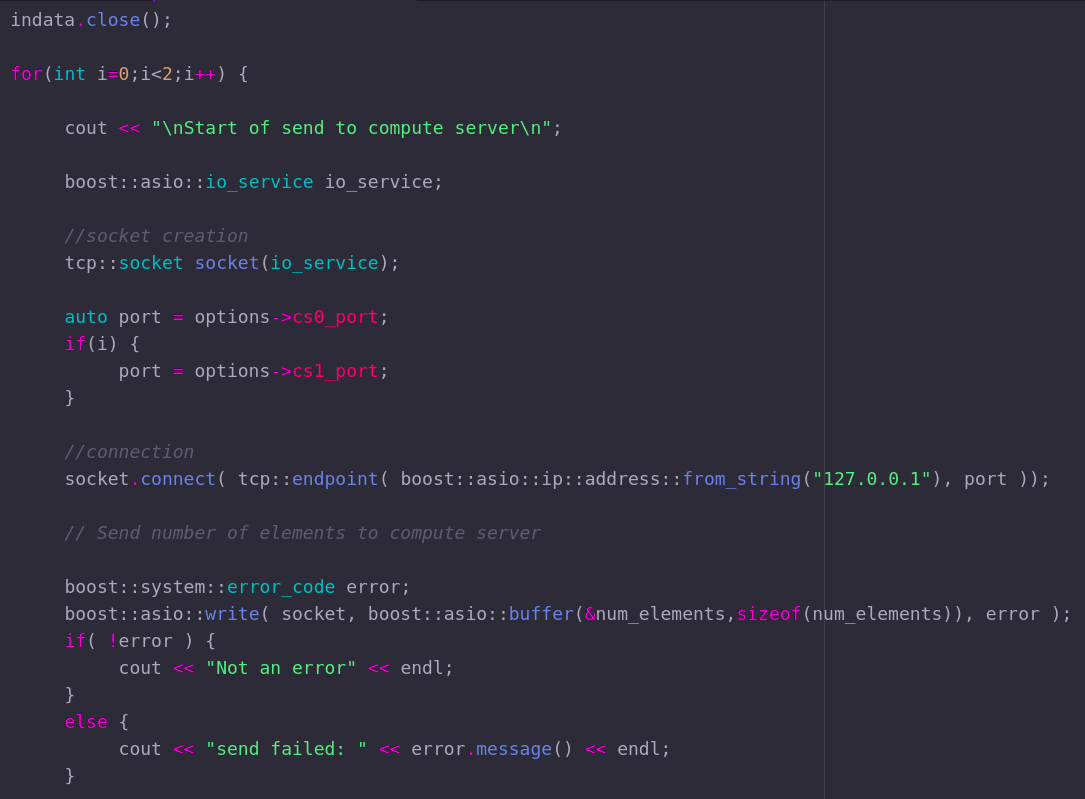


Fig 10: Initialising the TCP process

Once we have the shares created, we move on to the communication segment, where the clients can connect to the servers. The inflow of input is closed prior to beginning the communication. Then for each client (using the for loop), we initialise an asio **io\_service** that allows us to perform the core input-output functionality for the TCP processes.

A tcp **socket** is initialised for each client, which reads the ip address and port using the tcp::endpoint function and binds them to create the connection as shown in Fig 10. Upon receiving no errors, the client now sends the first set of details, which is the number of elements the other party is expected to receive, in the form of a simple buffer to the servers.

So now that we have successfully sent the data, the client can also receive data upon being shared. We use the same buffer system to continue reading until the buffer signals the EOF. This is then type casted into a character array.

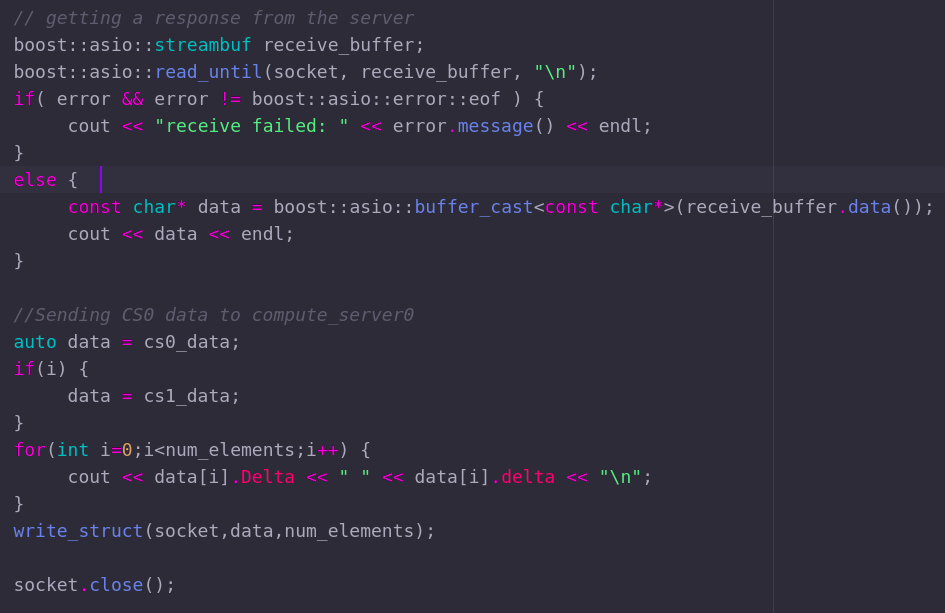


Fig 11: Reading from input buffer

Finally the shares created earlier are written into a buffer structure as done by the **write\_struct** function. Each delta share for each party is initialised into a neat structure format and then sent to the server. This concludes the processes done by the client-side program. The only process left is to receive the final outcome of the process performed by the server-side program. Shown below is the write\_struct function, that initialises the shares for each party and creates a shareable buffer structure for them.

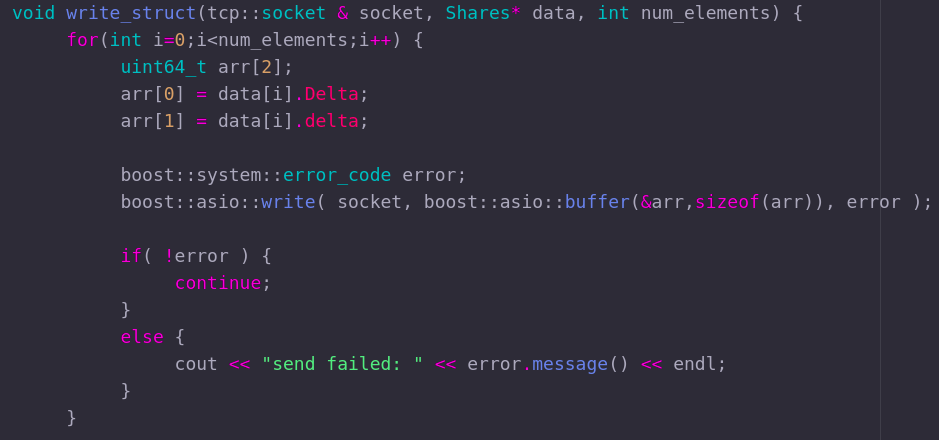


Fig 12: The write\_struct function

# **SERVER SIDE OPERATIONS**

Now that we have an idea as to what takes place on the client side, we come to the server side of operations, where the actual operations will be performed on the inputs provided to them by the clients.

## **Server Functions**

The servers are based on the existing MOTION2NX repository for executing a multiplication process using their in-built **tensors** library that can handle matrix multiplications with ease. The same is used for constructing Neural Network models too. The explanation for tensors has been made in another documentation among these (doc), so in the following document we shall discuss how it has been used.

The inputs are accepted in a similar format as the client side, also resembling the Yao’s millionaires problem input format, where we need to enter the server ID, the IP and port details of the two servers, the arithmetic and boolean protocols being used, the number of fractional bits and the number of repetitions that the program must do. All of them are accepted using the program\_options structure as discussed before.

The difference lies in the flags being used to initialise the servers, and these are listed below:

* **--help, -h** – This is a common flag used to display all the usable flags on the terminal and explain their purpose
* **--my-id** – Used to declare the identity of the servers being initialised
* **--party** – Declares all the server/party IDs participating in the process, their IPs and port addresses in the form of ID,IP,port. Eg: --party 1,::1,1234
* **--arithmetic-protocol** , **--boolean-protocol** – Used to make a choice of protocol using which the operations can take place.
* **--threads** – Though not a commonly used flag, it can be used to create multiple threads for evaluating the gate operations
* **--json** – Used to convert the output into a json format.
* **--num-simd** – Total number of Single Instruction Multiple Data values
* **--repetitions** – Number of times the program repeats the process.

Any other flags found in the code are not used by us,and are easy to understand.

## **Variable Mapping and Communication Setup**

Once the program\_options have parsed all the necessary input values, they need to be stored in the right variables and prepped for further processes. The **variable map** allows us to do precisely this by defining what value each variable would hold, and these variables will be passed to their respective functions by simply calling the **options** structure containing the list of all of them, and specifying which one is required.

Added to this, each party has its own IP address and port number that needs to be handled. It also needs to map the expected port addresses of each client to receive and store their inputs. Once the IP and ports are received, the compute\_server functions ( mentioned below ) are invoked to receive all the shares from the clients. Then the **setup\_communications** function is activated, the same as done in Yao’s millionaires problem, and this allows the servers to perform the necessary operations on the received shares.

## **The Compute Server file ( compute\_server.cpp )**

A small addition as mentioned before is the process of initialising each client’s inputs into their respective inputs. During the parse\_program\_options, the variable map invokes the **get\_provider\_dot\_product\_data** that calls the compute server operations created specifically for the four way setup. This is a separate folder in the **motioncore** section of the MOTION2NX repository.

The function entirely comprises asio functions used to perform the tcp interactions to receive the necessary data. The **io\_services** creates the structure to compile the IP and port numbers for each client and accept the socket connection. The first step is to accept the number of elements as mentioned in the client-side program using the read function from the asio buffer. Any errors it catches would be displayed to the user using the **error\_code** from boost.

The reason for why the clients have a read is to receive from the servers whether they have successfully received the data when no errors occur. Once this is done the servers use the **read\_struct** function to read the structure of shares it receives from the client. Print statements written in between are to cross-check whether the server receives all the values. Finally the shares and the number of elements are returned to the calling function.

## **Building the Dot Product Operation**

When we say “build”, the program is required to create the necessary gate structure in a format that allows the two servers to systematically perform the dot product operation. This begins in the **create\_composite\_circuit** that is responsible for making the gates for the dot product. The first step is to call in the tensors library using the **tensor\_op\_factory** in accordance with the arithmetic protocol of choice ( Use beavy to be on the safe side ). This now allows us to use tensors as well as the **GEMM** operation.

We initialise the shape of the two inputs and the output in the form of a structure ( **gemm\_op** ) followed by which three separate variables are declared to hold each individual dimension. The next step involves the creation of **tensor\_a**  and **tensor\_b** which invokes the **make\_arithmetic\_64\_tensor\_input\_shares** function to sort out the private and public shares of each tensor. Once this is done, we can get the gemm output using the **make\_tensor\_gemm\_op** function that returns the dot product answer, and the output is made to display only on the screen of server\_1 and not server\_0.

Once the entire operation is complete, the log files are created and the sockets are closed, thus concluding the entire process.

# **STEPS FOR EXECUTION**

Now that we know what happens on both sides, we move on to the steps of execution. We assume the entire process is done on a single system and hence use four different terminals, but it is possible to perform the operation using four different computers connected to the same LAN network.

The first step is to prepare the four terminals, two of which are the servers and can be compiled using similar steps mentioned in the Yao’s millionaires problem document. The client side on the other hand comes from another repository, and hence needs to be set up first. As shown in the MOTION2NX Installation guide, we go to the DP-CS Mock Communication repository and obtain the http link of the repository. Once we have this, go onto the desired file location on your system, and upon opening the terminal for that location add the following code:

***git clone https://github.com/<link>/DP-CS-Mock-Communication.git***

Here, <link> is replaced by the location of the DP-CS repository in current use.

Once cloned, the next step is to create the build folder and compile it. This can be done by entering the following lines into the terminal one after the other:

***cmake -S . -B build\_***

***cd build\_/***

***make***

Now we simply cd into the ***dot\_product\_dp*** folder within build\_, and we have the client setup ready! Since we need two, a copy of the current terminal can be created, resulting in two identical independent clients. The same can be done for the server, but for the server, from the MOTION2NX root folder, we do ***cd build\_debwithrelinfo\_gcc/*** after we cmake it, and create another terminal to act as the two servers.

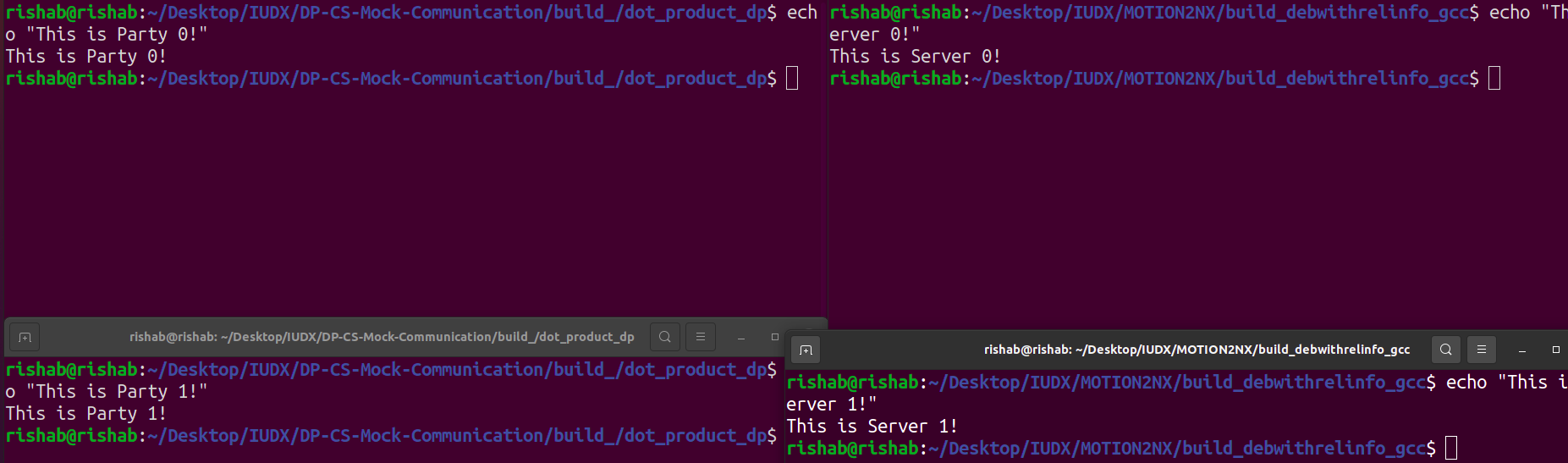


Fig 13: Setting up the two clients or parties

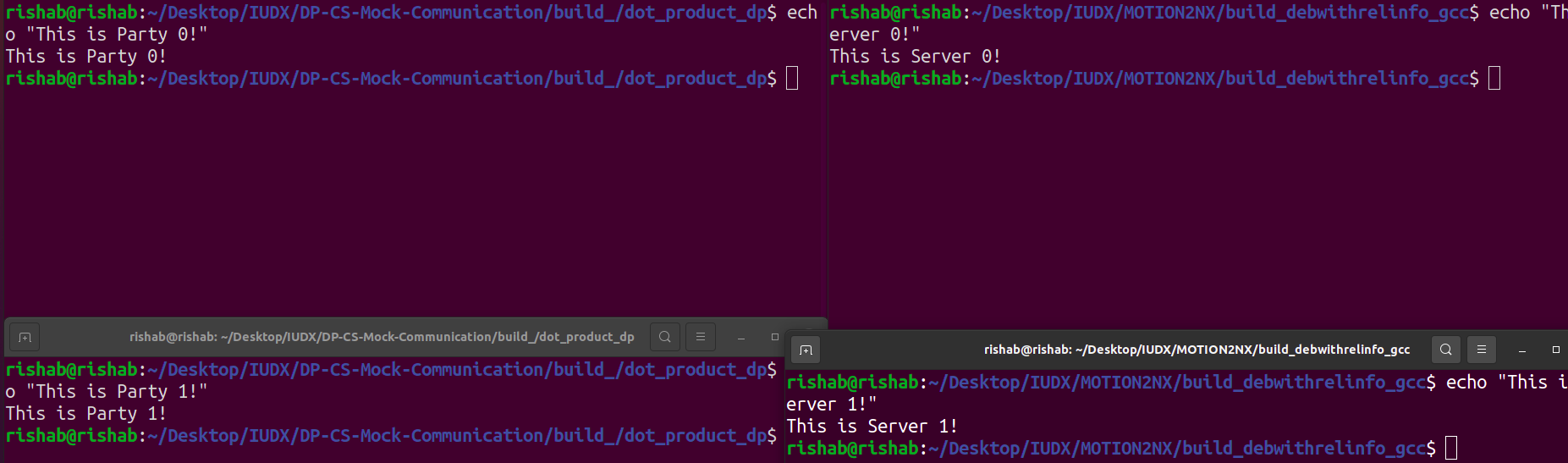


Fig 14: Setting up the two servers

Once we have this setup, we first start up the two servers using the following:

**Server 0:**

***./bin/tensor\_dot\_product --my-id 0 --party 0,::1,7000 --party 1,::1,7001 --arithmetic-protocol beavy --boolean-protocol beavy --repetitions 1***

**Server 1:**

***./bin/tensor\_dot\_product --my-id 1 --party 0,::1,7000 --party 1,::1,7001 --arithmetic-protocol beavy --boolean-protocol beavy --repetitions 1***

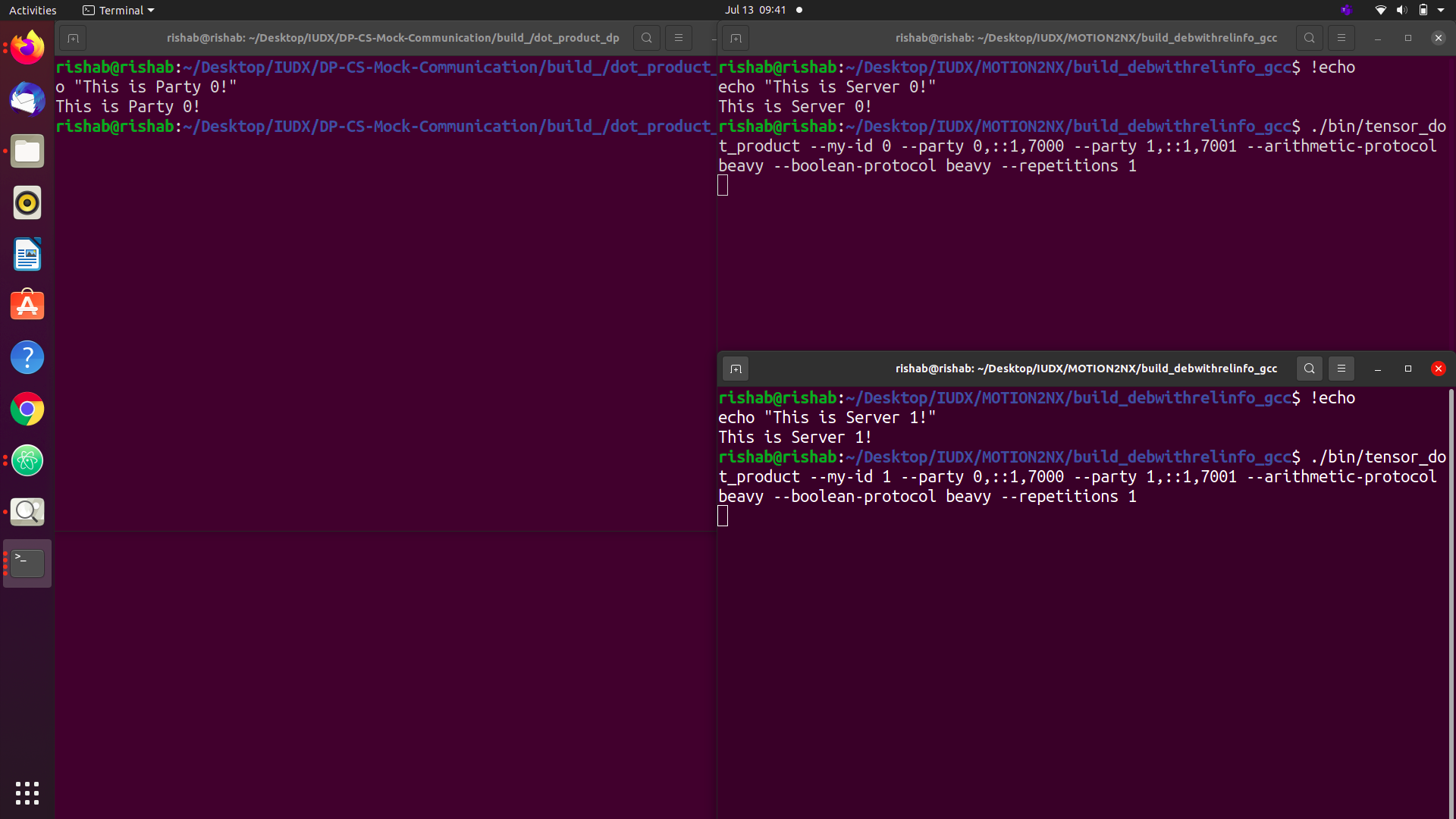
****

Fig 15: Starting up the servers

Now we do the same for the client side. For each client, we enter the following:

**Client 0:**

***./dot\_product\_dp --compute-server0-port 1234 --compute-server1-port 1235 --dp-id 0 --fractional-bits 0***

**Client 1:**

***./dot\_product\_dp --compute-server0-port 1234 --compute-server1-port 1235 --dp-id 0 --fractional-bits 0***

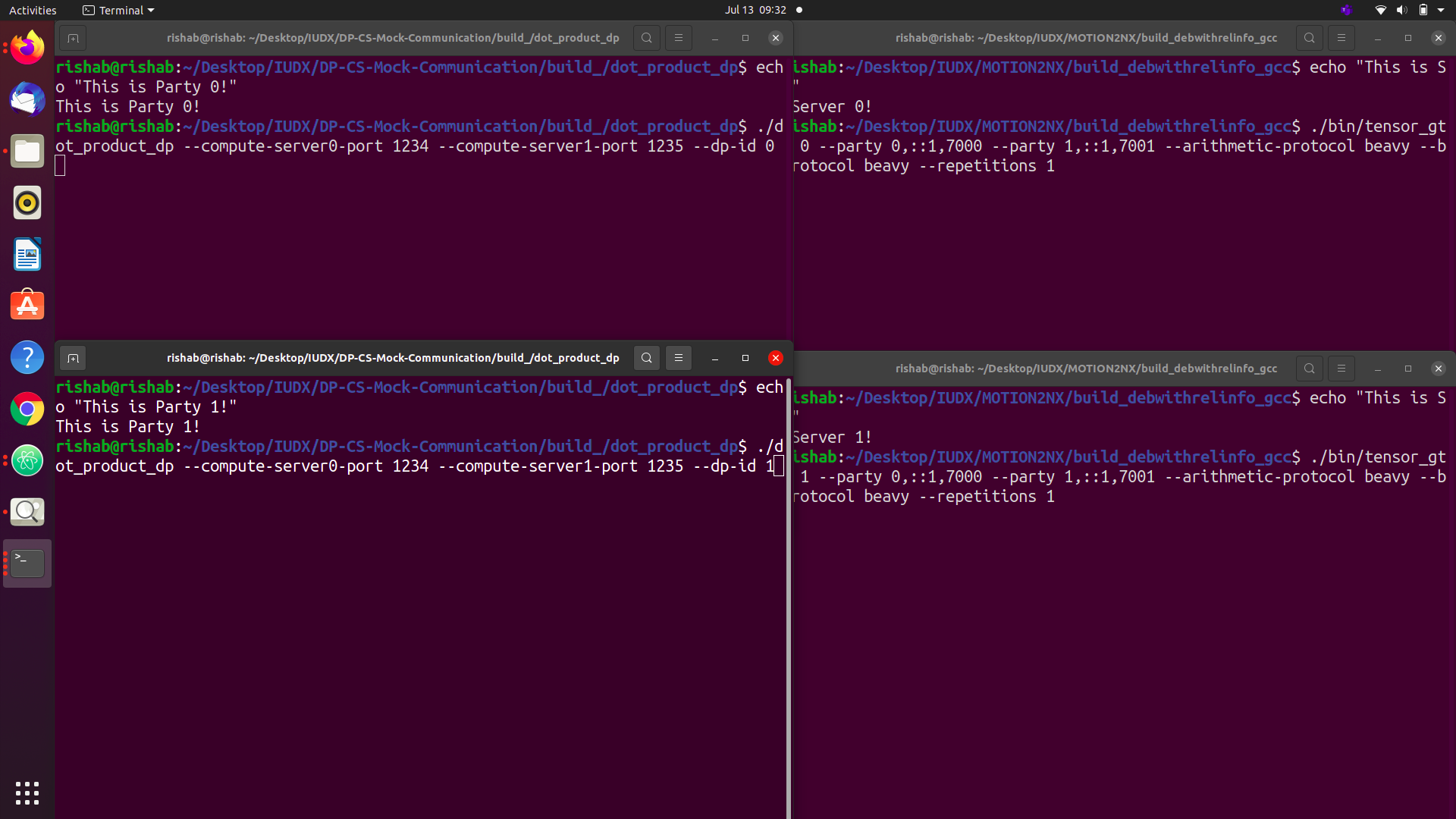
****

Fig 16: Executing the two client side programs

We might wonder where the input is even coming from when we aren’t entering anything! The inputs are pre-made and stored in the form of files named **input\_dp0.txt** and **input\_dp1.txt**. The values can be changed, but it is important to change the dimension values (the first row) in case of addition or removal of values. Once the clients are up and running, we should get an execution occurring on the servers, and the final answer appears as shown.

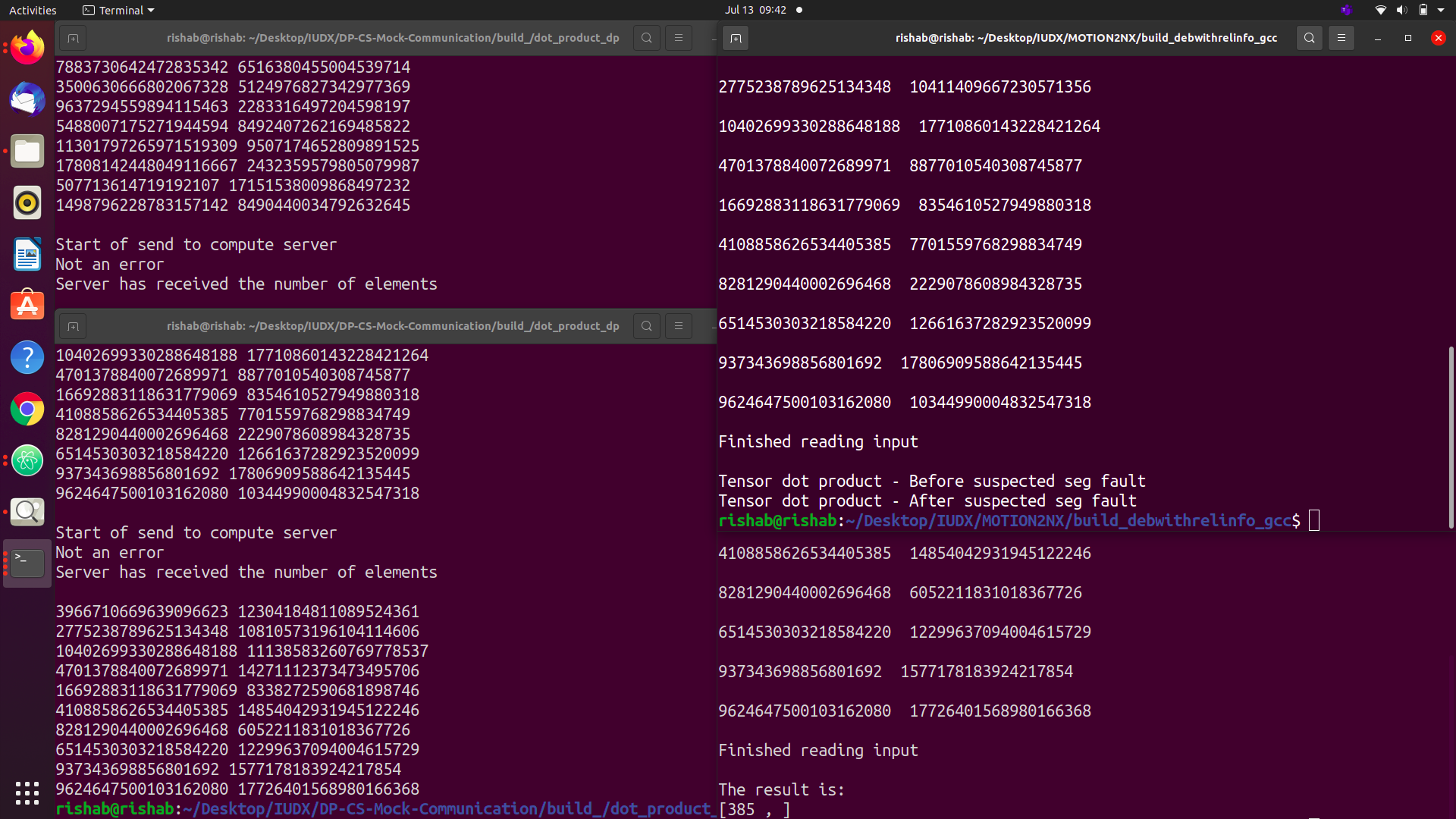


Fig 17: Final answer shown on the servers