**YAO’S MILLIONAIRE PROBLEM**

# 

Table of Contents

# 

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

[**THE PROBLEM STATEMENT**](#_st8ul0i9l3yr) **1**

[**SOFTWARE IMPLEMENTATION**](#_mwodb6x8l8cp) **2**

[Two Party setup and execution](#_ueasjwqe349c) 3

[Communication Layer](#_581l1naiy6aq) 6

[Creation of shares based on inputs](#_i5u8o6fnofn9) 8

[Operations on shares](#_uv8hx9w4cz8e) 11

[The choice of ABY world for operations](#_805dxykdqy1b) 12

[**CONCLUSION**](#_apyhl9x4mur3) **12**

# 

# 

# 

# 

# 

# 

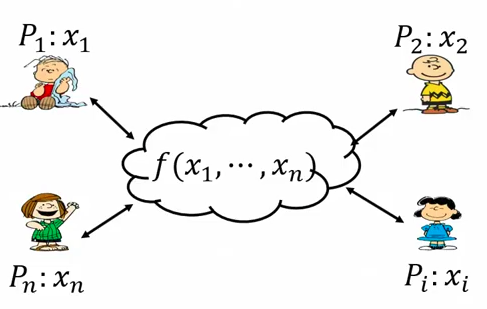
# **INTRODUCTION**

We can say that the application of cryptography is used to provide secure communication between two people sharing information. This is done by encrypting the data and passing it through a secure channel such that only the two parties know what the encryption means since they would be able to decrypt it on receiving it. Any cryptographic algorithm should satisfy the basic properties of **privacy**, **correctness**, **authenticity** and **integrity** [1]. Now we consider a situation where we have N people or users who wish to communicate with each other, but do not completely trust each other with their sensitive information. This type of application requires a distributed system to share data between the N-users.

A distributed application additionally requires **confidentiality** and **availability** of sensitive data. The situation we have in the example above is a case of a distributed application involving N number of mutually distrusting parties. This means that though information is to be shared amongst each other, no user other than the respective owners of their sensitive information is allowed to know the contents of it to ensure maximum security. The outcome should be an output through jointly performing a computation while retaining the privacy of their inputs. This is where we can introduce SMPC into our application.

# **THE PROBLEM STATEMENT**

Let us assume another example where we have two millionaires who wish to find out who is wealthier without revealing how much each one of them are worth. In SMPC, we would call each of these millionaires a party participating in this setup, hence we assume the two parties to be P0 and P1. The value of N discussed earlier now becomes 2. Their inputs are given into the system upon encryption, using the method of oblivious transfer [2]. By doing so, the input from both parties are received in a format understandable only to the computation engine performing the necessary operations, in this case checking which is the greater input value and sending this as the output back to both parties.



The above is a reference to one of the most important scenarios in the world of SMPC known as Yao’s millionaire’s problem [3]. It has allowed numerous technical advances in SMPC and more efficient implementations over distributed systems. Some of these implementations adhere to special protocols that provide a standard method of execution for all parties using it. One such protocol that we use is the ABY2.0 protocol [4], a unique method that performs operations in the **A**rithmetic, **B**oolean and **Y**ao world interchangeably. In the following project, we make use of all these concepts in our implementation of a privacy preserved system.

# **SOFTWARE IMPLEMENTATION**

Implementation of the Yao’s millionaires problem requires the following functions to work smoothly:

1. Two parties setup and execution
2. Communication layer
3. Creation of shares based on inputs
4. Operations on shares
5. Choice of Arithmetic, Boolean or Yao world for operations

The implementation is executed using C++, a special thanks to the encryptomotion group for giving us the MOTION2NX repository to work with. As mentioned earlier this follows the ABY protocol to perform operations, as well as an optional setup for using GMW protocol instead (doc). Let us get into the functionalities of the setup.

## **Two Party setup and execution**

The first step to execute the millionaire’s problem is to have two parties trying to achieve a certain result. As mentioned earlier, these parties can be mutually distrusting parties, which creates a **semi-honest** environment, ideal for executing this problem. We assume the parties to be **party\_0** and **party\_1**.

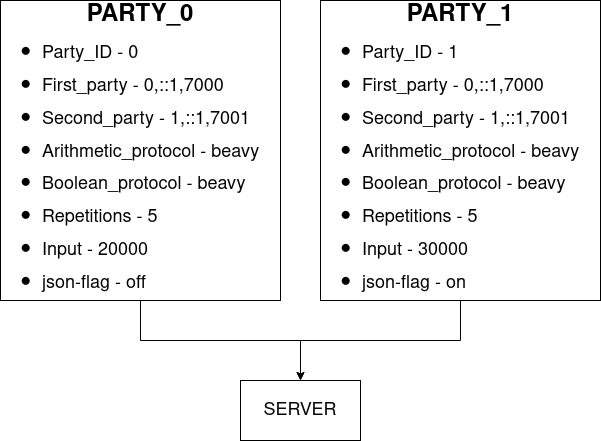


Fig1. Two party setup

As we can see in Fig.1, we have the two parties participating in the operation, which in this case is to find out who is worth more. We assume that party\_0 is worth 20,000 units of currency, while party\_1 is worth 30,000 of the same units. Note here that both parties are unaware of each others’ worth and hence use this operation to discreetly find out who is worth more without revealing any exact values.

To start with, we require a linux system with MOTION2NX and its dependencies setup (doc), and two terminals. For this example we shall execute the operation locally on a single computer, but shall be using two terminals to be the two parties respectively. First we traverse to the example in the folder.

**EXECUTION:**

**cd MOTION2NX/<build\_folder\_name>**

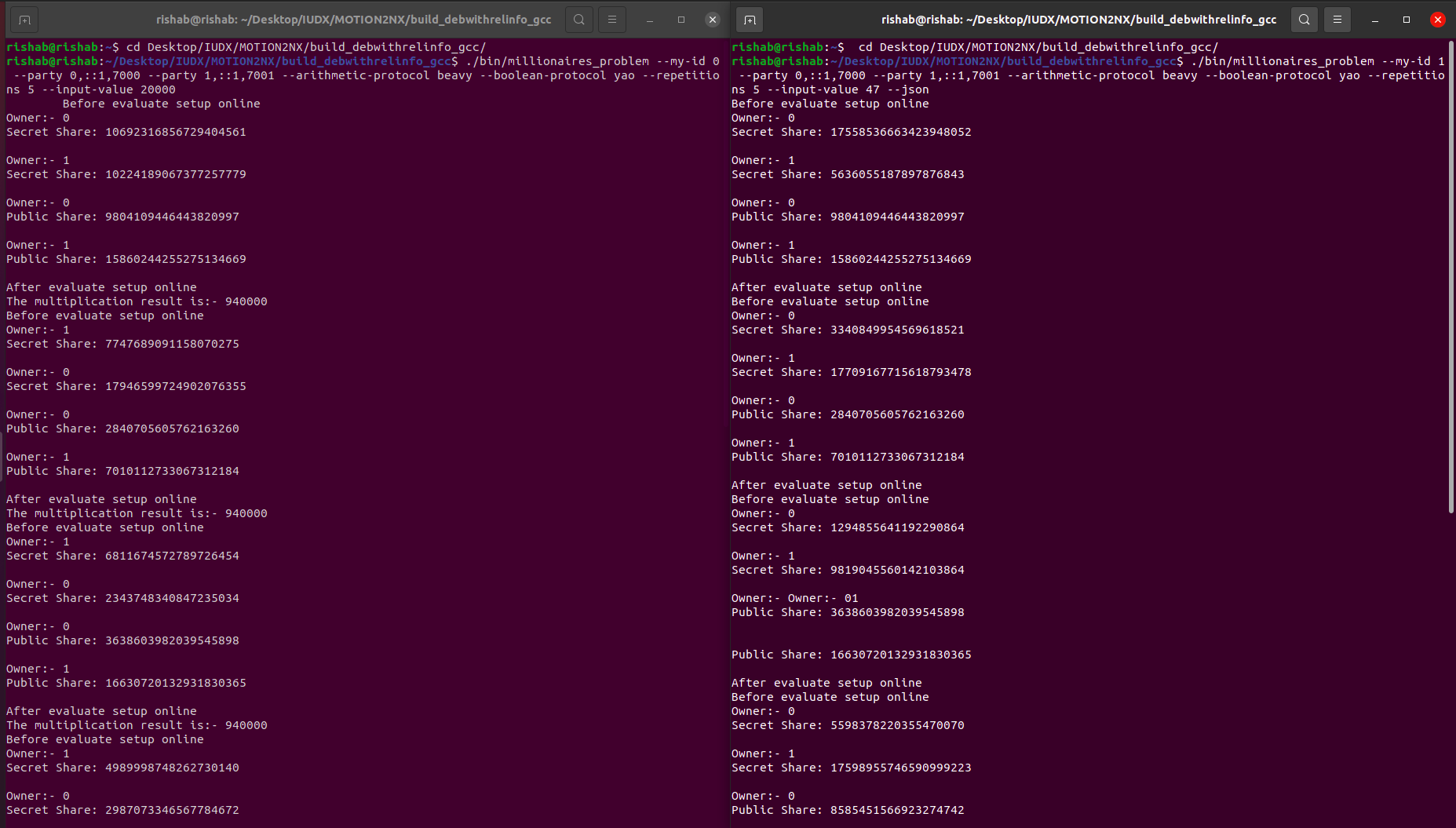
**Terminal 1: ./bin/millionaires\_problem --my-id 0 --party 0,::1,7000 --party 1,::1,7001 --arithmetic-protocol beavy --boolean-protocol yao --repetitions 5 --input-value 20000**

**Terminal 2: ./bin/millionaires\_problem --my-id 1 --party 0,::1,7000 --party 1,::1,7001 --arithmetic-protocol beavy --boolean-protocol yao --repetitions 5 --input-value 47 --json**

To explain what each step means, the first line with ‘cd’ traverses to the **build** folder as created from the setup (doc). This has to be done on both terminals, and upon reaching the folder, we enter the code for terminal 1 and terminal 2 on separate terminals. Then we just wait for completion of execution as shown in Fig 2 and Fig 3.

The explanation of each flag is as shown below:

1. **--my-id:** This flag is used to indicate the ID of the party. ID 0 stands for party\_0 and 1 for party\_1 in this case.
2. **--party:** This is an important flag that allows us to define each party in a way the system can understand. It involves 3 parts:
   1. Party\_id – This is used for similar reasons as the first flag, but in this case acts as a label more than an ID for the next two parts.
   2. IP address – This is an important detail that is used by the communication layer to locate and connect with the party for communication purposes.
   3. Port number – Aids the IP address in communication. **Note:** If we run the two parties on the same system, we must have the same IP address and different port numbers. Else the IP addresses will vary based on systems, but port number can be changed based on the user's choice.

Fig 2: Terminal Execution for Millionaires problem

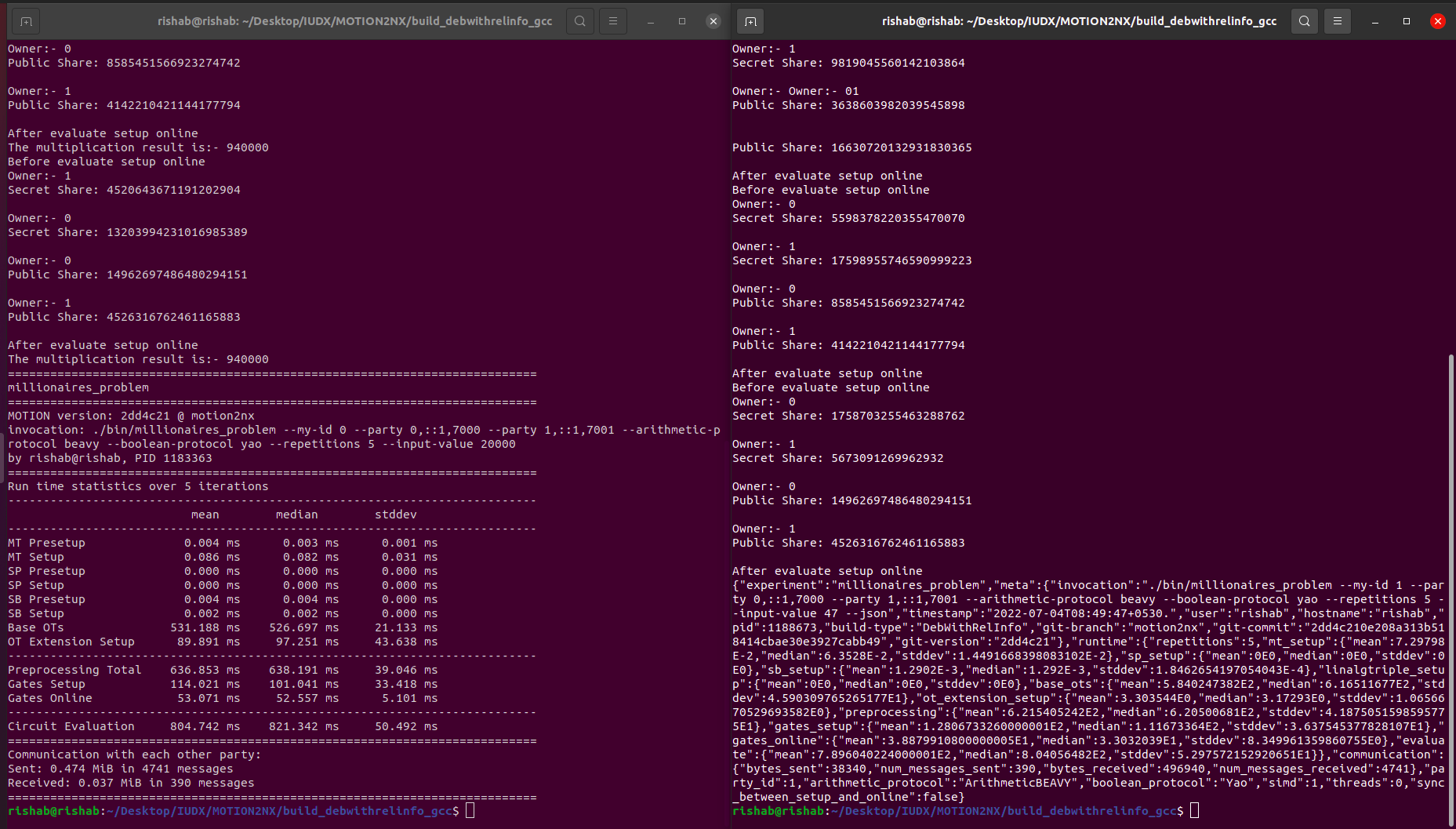


Fig 3: Variation from normal format to json format

1. **--arithmetic-protocol:** An important flag, this determines the world used by the ABY2.0 protocol. Though the default used is ‘beavy’ which is an arithmetic format for operations, we are allowed to enter gmw as well as yao based on preference.
2. **--boolean-protocol:** An important flag again, same as above, we can use any of the options based on our preference of operations performed. In this case we once again use ‘beavy’.
3. **--repetitions:** This flag determines the number of times the program repeats its operation. Mostly used to cross check the time of execution and correctness of the operation over multiple executions.
4. **--input:** This is used to provide the input values into the system, here being the net worth of each millionaire. Only one input is provided per party / terminal.
5. **--json:** An optional flag that is used to convert user understandable input into a JSON format. This is useful for combining with APIs in the future if necessary.

The output that follows gives us only a single statement as to which party is worth more than the other. It does not reveal any form of calculation, operation or input values themselves to any of the parties. Followed by this is a list of execution times and resources used by the program to perform the operations, just a measure of efficiency. Values may vary based on multiple factors such as input size, number of repetitions, protocols used and so on.

## **Communication Layer**

Apart from just having parties, a reliable communication must be established between them that would allow not only passive communication but also a discrete and secure format of it too. The underlying software we use is written in **C++** and of the vast ocean of libraries available to us, we make use of the **boost** libraries to provide an asynchronous secure communication link between the participating entities.

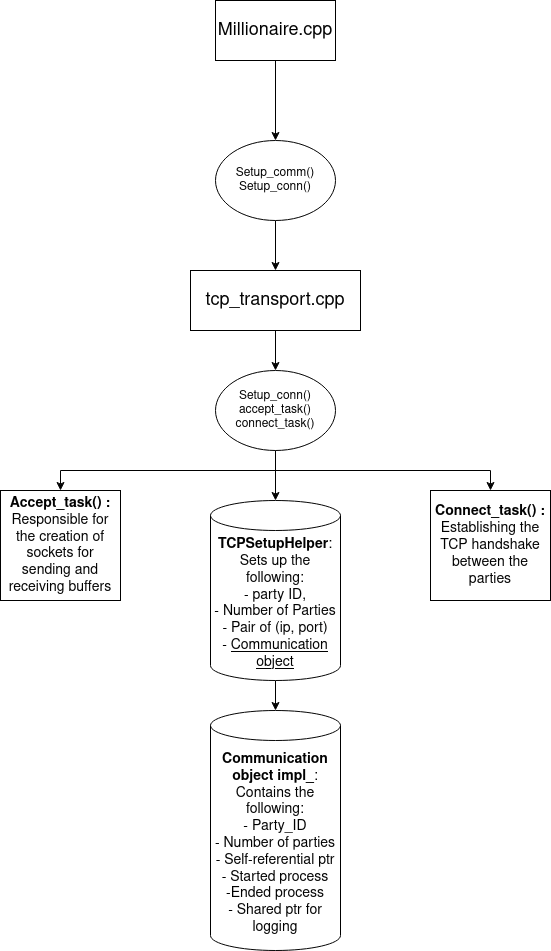


Fig 4: The Communication Layer

As seen in Fig 4, The communication layer consists of multiple structures that store and connect to N-number of parties that join the communication link. Called by the **setup\_communication()** function from that main file, three main functions are accessed in order to perform this process:

1. **setup\_connections():** This function is the primary function of the three which creates the **TCP handshake** setup between the parties wanting to connect. The first step is to call the **accept\_task()** function that initialises the parties that are going to participate in this process. On completion, it sets up the expected number of parties (in this case being 2) and calls the **connect\_task()** to connect all the parties that have been accepted. The final outcome is an array of parties with connected IPs and open sockets ready for communication.
2. **accept\_task():** This function is used to accept a single party to the server that is calling it. In the case of this program, the first party itself acts as the server since the operations are local to the system. It uses a library of boost called **boost::asio** that allows the program to perform asynchronous TCP operations. Further it also distinguishes between the sender and the receiver while setting up the two parties.
3. **connect\_task():** This function is called once all the known parties have been accepted and are to be connected to each other over their socket ports. It creates mutable buffers for each party and performs error handling operations to ensure successful connection.

## **Creation of shares based on inputs**

This is one of, if not the most important step in the process, since the entire process works on the idea of shares (doc). As mentioned before, we shall be using the **ABY shares** in order to perform this operation. A quick overview, we can define them to be a part of additive secret shares, where a value is broken up into segments which on rejoining after operations on it, result in the required answer. In this case the input value of each party is divided into two parts:

1. **Public shares:** This is the share that is available for all parties to see
2. **Private shares:** Only the party that owns the input knows the value of this share, and is obscured from all other parties.

In the case of this repository, the creation of shares is done using the help of libraries such as **openssl** for creating **aes** keys of varying lengths based on the shares to be generated. This in turn depends upon the size and type of the input being given to the system. The majority of the processes take place in two files:

1) **sharing\_randomness\_generator** 2) **pseudo\_randomness\_generator**

These folders are responsible for the creation of special **keys** that are generated randomly using the aes functions. Each of these are stored in block format (majority start off as 128 bits then they are divided into smaller blocks, commonly 64 bits), and each block holds each bit of the input segments that transfer into shares. The **aes\_128\_ctr\_rng** file and all file names similar to this hold the multiple ways in which the keys can be generated based on user input. It also depends on the operation being performed.

Once the communication between the two parties has been established, we assume that party\_0 takes the role of the server for the time being on the local system. The inputs of both parties are passed into the server section (note that party\_0 is still oblivious to the input of party\_1 even if this happens) as **wires** following a circuit format, and the function to perform the necessary share operation is then invoked ( Here we used BooleanBEAVYInput classes ).

We shall begin with the **server** point of view that is responsible for creating the shares (BooleanBEAVYInputGateSender). The first step is the **setup phase**. Here it is necessary to arrange the inputs into two different storage units, where they can individually be accessed for conversion without ambiguity amongst the input values. We then call the **randomness\_generator()** function and pass the input to this function to generate the public and private shares for each party. This is done using the aes blocks as mentioned before, that act as random keys to encrypt the input value (in a sense) and split it into the respective shares.

The shares are returned to the initial storage structure. Now the server alone holds the private and public shares. This is where we move to the second step, the **online phase**, where the public shares of each party are distinguished and stored separately in a structure to be broadcasted to all parties available. So now, we have the public shares shared by both the parties, and only party\_0 holds its private share.

The above problem is taken care of on the **client** side of the operation (BooleanBEAVYInputGateReceiver), where we see the operations performed from party\_1’s perspective. Similar to party\_0, we start with the **setup phase**,

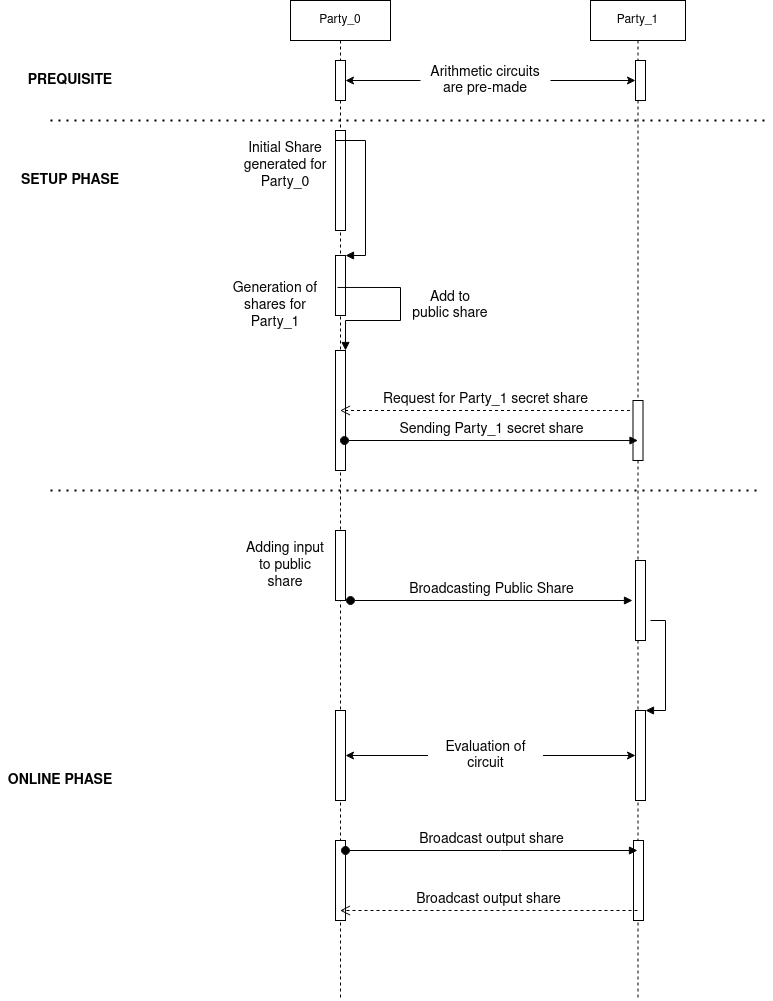


Fig 6: Operation performed in setup and online phases

but here we do not generate any shares, but rather receive the randomness key generated by the server at party\_0, as well as the private share. So one both parties have their respective private share, and the **online phase** for the client side receives the broadcasted public shares in the same format as generated by the server. All of these operations discussed above for creating and sending shares take place using in-built **oblivious transfer** functions. Fig 6 gives us a top-level view of the two phases. The evaluation of circuit is an addition here, which shall be addressed in the next section.

## **Operations on shares**

There are many different operations that can be performed on the shares generated from the earlier section. In this example, we concentrate only on the **comparison** circuit operation, since we require only to check which party has a greater net worth.

Before the shares are generated, it is crucial to first create a circuit that can perform the necessary operation. In the repository, operations are performed using either **gates** or **circuits.** While gates are the simpler of the two where simple operations can be performed a lot faster, circuits on the other hand can consist of multiple gates put together to perform more complex operations, or can perform the simple operations in a more secure format, at the cost of time. In this case, a pre-made **gt\_circuit** is used to perform the greater-than operation on the shares.

The shares are transferred into wires to be fed into the circuit. The circuit can be called and loaded based on the input-size, ranging from 8-bit to 64-bit. The internal operations of a circuit differ from a normal greater than function since here we operate on shares and not direct input values. Once the operation is done, the output is a boolean value, which gives us **1** if party\_0 has a higher net worth and **0** if party\_1 has a higher net worth. This output is broadcasted in a fashion similar to the previous section.

The other operations that can be performed involve replacing the circuit function with other circuits or gates such as:

* BooleanBEAVY***X***Gate: X can be AND, NOT, XOR, INV, Input and Output
* Similar format gates exist to perform addition, multiplication and an array of matrix operations discussed in further documents (doc).

## **The choice of ABY world for operations**

Before we conclude on the entire process, it is important to distinguish the Arithmetic, Boolean and Yao worlds that we use for performing the operations on. The repository uses the Arithmetic world to perform all of its operations on shares and a garbled circuit to give the output, but it is possible to use other worlds when it comes to different approaches to a given problem.

Different examples call for different operations. The set of these present in the repository use different worlds and sometimes a mix of all the worlds based on the demand of operation type. A good example would be Neural Networks, where we can perform matrix operations in the arithmetic world, but complicated operations such as that of non-linear activation functions require the aid of Yao circuits. Comparison of multiple boolean values such as a softmax (future work) can be done in the boolean world as a probability distribution among multiple values.

Hence it is important to never allow the world of operations to be a restriction, at the same time it is important to use them cautiously under the rules and regulations set by the protocols with which we execute our operations.

# **CONCLUSION**

The millionaires problem is only one among the many implementations that make use of the ABY2.0 protocol. The MOTION2NX provides us with a wide variety of examples that can use this protocol in order to perform different operations ranging from array multiplication, matrix operations, neural nets and so on. Each operation uses a different set of files that can be traced down using the header files in each executable. This allows the repository to be flexible with the types of operations that can be performed, and addition of any new functions that abide by the ABY protocol.