DOP: 17/01/23 SANJANA ASRANI
DC LAB01 D17B/01/BATCH B

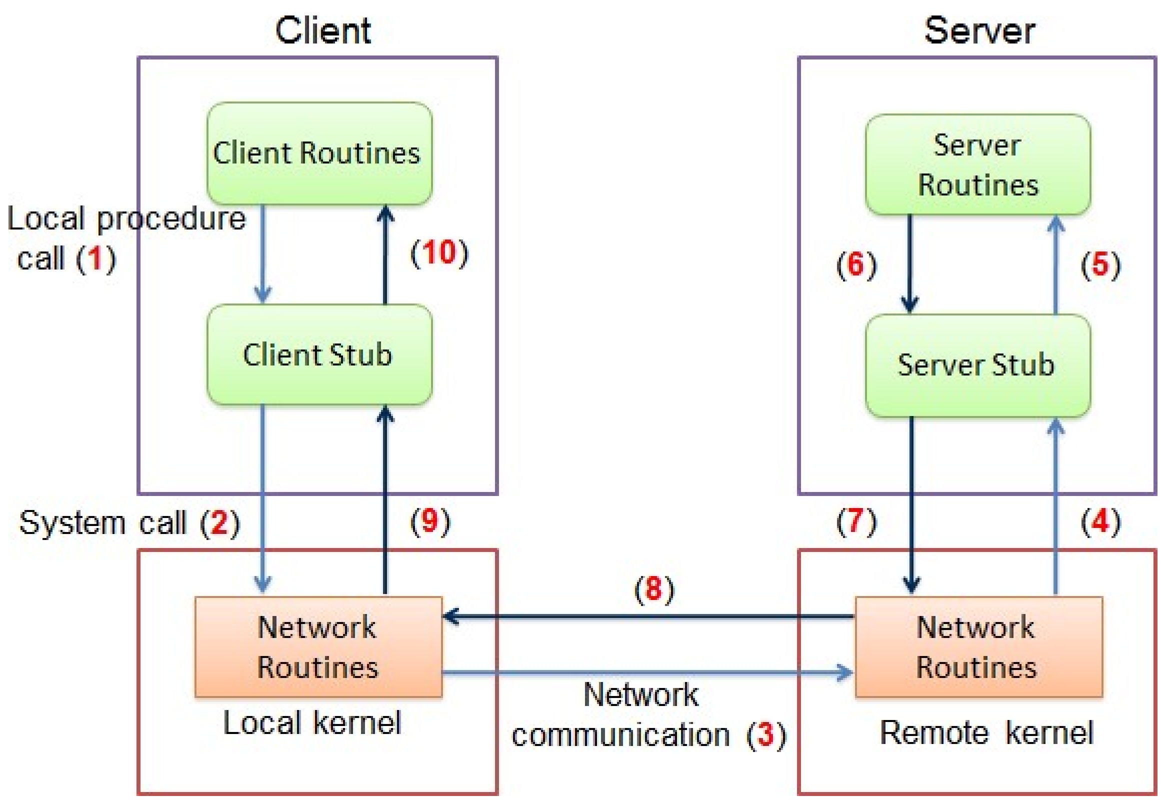
AIM:
Write a program to demonstrate Inter-process communication in Client Server Environment using RPC/RMI .
Unary RPC- Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.

THEORY:

Inter-process Communication (IPC):
Inter-process communication refers to the mechanisms and techniques that enable communication between different processes or programs running on a computer or across a network. IPC allows processes to exchange data, synchronize their activities, and coordinate their execution.

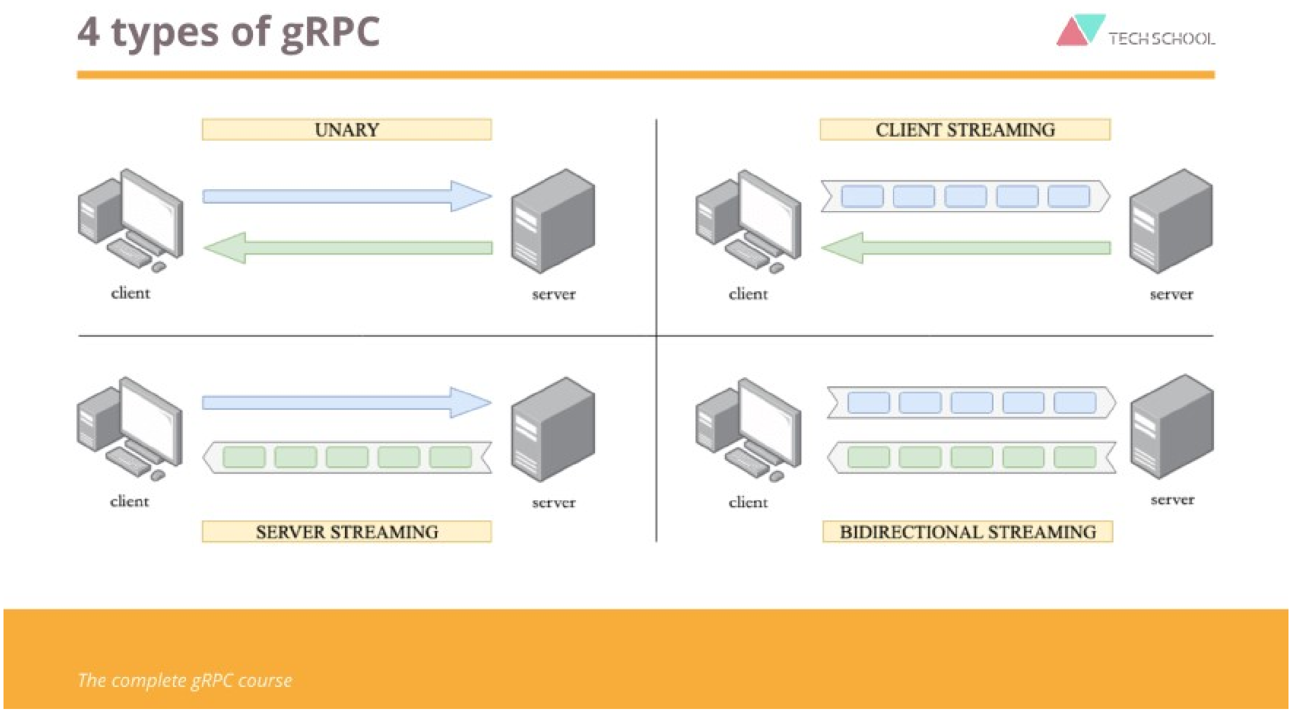
Client-Server Environment:
Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order. For example, a server can wait until it has received all of a client's messages before writing its messages, or the server and client can play "***ping-pong***" – the server gets a request, then sends back a response, then the client sends another request based on the response, and so on.

Remote Procedure Call (RPC) Model:
RPC is a communication model that allows programs to execute procedures or functions on a remote server as if they were local. It abstracts the communication between processes over a network, making it appear as if a procedure is being called locally even though it is executed on a remote server.



1. When a client routine performs a remote procedure, it calls the client stub, which serializes the input arguments.
2. This serialized data is sent to the server using OS network routines (TCP/IP)
3. The data is then deserialized by the server stub
4-5. presented to the server routines through stubs with the given arguments.
6. The return value from the server routines is serialized again
7-8-9. It is then sent over the network back to the client where it's deserialized by the client stub and presented to the client routine.

This remote procedure is generally hidden from the client routine and it appears as a local procedure to the client. RPC services also require a discovery service/host-resolution mechanism to bootstrap the communication between the client and the server.



Unary RPC (Remote Procedure Call):
Unary RPC is a type of RPC where the client sends a single request to the server and receives a single response in return, similar to a regular function call. It follows a simple request-response model, where the client invokes a remote procedure on the server, which processes the request and sends back the result to the client. Unary RPCs are commonly used for synchronous communication between distributed systems, resembling traditional function calls in programming.
Characteristics: Resembles a traditional function call, where the client invokes a remote procedure and waits for the result.

Client and Server Stub in RPC:
Client and server stubs are code components that facilitate communication between the client and server in an RPC system. The client stub acts as a proxy for the client, marshaling parameters and sending them to the server. The server stub unmarshals the parameters, invokes the actual procedure, and marshals the results back to the client. Stubs abstract the complexity of network communication, enabling seamless interaction between the client and server.

How is object passed as reference in RMI:
In RMI (Remote Method Invocation), objects are passed by reference, allowing methods to be invoked on remote objects as if they were local. When an object is passed as a parameter or returned from a remote method call, a reference to the actual object is transmitted, enabling the remote client to interact with the object located on the server. This mechanism facilitates seamless communication between distributed systems, as the client can manipulate remote objects without the need for explicit handling of low-level details like object serialization and deserialization.

Call by Reference and Call by Value in RPC:
In RPC, parameters can be passed by reference or by value. For call by reference, the reference to the data is passed, allowing the server to access and modify the actual data. For call by value, a copy of the data is passed, ensuring that modifications do not affect the original data. The choice depends on the specific requirements of the RPC and the nature of the data being transmitted.

Object Passing as Reference in RMI:
In RMI, objects are passed as references. When a client invokes a method on a remote object, the object reference is passed to the server. The server then operates on the actual object instance, allowing changes made on the server side to reflect on the client side. This reference passing mechanism is fundamental to RMI's ability to work with distributed objects.

Lightweight RPC:
Lightweight RPC refers to a simplified and efficient implementation of RPC mechanisms. It focuses on minimizing the overhead associated with remote communication, reducing latency, and optimizing resource usage. Lightweight RPC is designed for performance and efficiency in scenarios where a lean communication model is preferred.

Callback RPC:
Callback RPC involves a mechanism where a server can initiate a call back to the client. In this scenario, the client provides a callback function or procedure to the server. The server can then invoke this callback on the client side, allowing for more interactive and event-driven communication between the client and server.

Call Semantics of RPC:
RPC supports different call semantics, such as:

- **Synchronous**: The client waits for the server's response before continuing.
- **Asynchronous**: The client continues its execution while waiting for the server's response asynchronously.

Handling Failure in RPC:
RPC systems typically handle failures through mechanisms like:

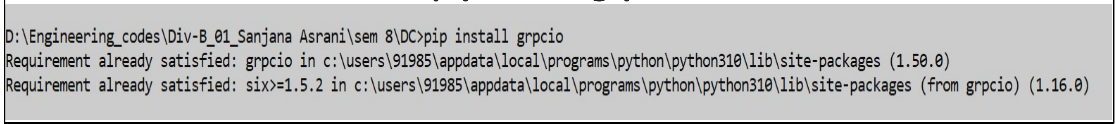
- **Retry**: If a communication failure occurs, the client or server may attempt to retry the RPC call.
- **Timeouts**: Setting a timeout ensures that if a response is not received within a specified time, the RPC is considered unsuccessful.
- **Error Handling**: RPC systems define error codes and mechanisms to handle errors gracefully, providing feedback to the calling party about the success or failure of the remote procedure call.

Remote Method Invocation (RMI):
Remote Method Invocation is a mechanism that allows a program to invoke methods on objects that exist in a different address space, typically in a remote machine. RMI enables distributed computing, allowing objects to interact across a network as if they were local. It involves a client making method calls on remote objects, and the RMI system handling the communication and execution of these remote method invocations.

CODE & OUTPUT:
Steps:

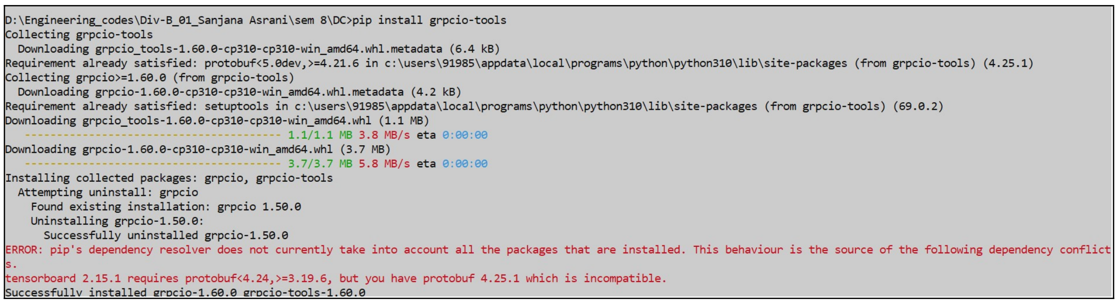
1. install the packages by running the following cmds:

pip install grpcio



```
D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC>pip install grpcio
Requirement already satisfied: grpcio in c:\users\91985\appdata\local\programs\python\python310\lib\site-packages (1.50.0)
Requirement already satisfied: six>=1.5.2 in c:\users\91985\appdata\local\programs\python\python310\lib\site-packages (from grpcio) (1.16.0)
```

pip install grpcio-tools

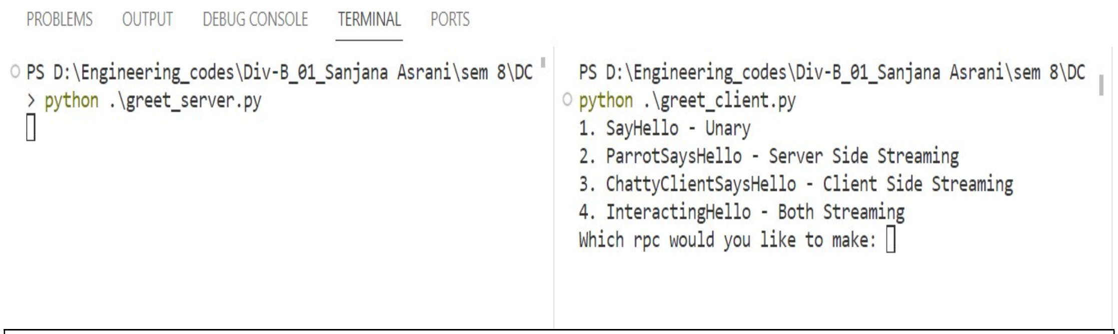


```
D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC>pip install grpcio-tools
Collecting grpcio-tools
  Downloading grpcio_tools-1.60.0-cp310-cp310-win_amd64.whl.metadata (6.4 kB)
Requirement already satisfied: protobuf<5.0dev,>=4.21.6 in c:\users\91985\appdata\local\programs\python\python310\lib\site-packages (from grpcio-tools) (4.25.1)
Collecting grpcio>=1.60.0 (from grpcio-tools)
  Downloading grpcio-1.60.0-cp310-cp310-win_amd64.whl.metadata (4.2 kB)
Requirement already satisfied: setuptools in c:\users\91985\appdata\local\programs\python\python310\lib\site-packages (from grpcio-tools) (69.0.2)
Downloading grpcio_tools-1.60.0-cp310-cp310-win_amd64.whl (1.1 MB)
   ---------------------------------------- 1.1/1.1 MB 3.8 MB/s eta 0:00:00
Downloading grpcio-1.60.0-cp310-cp310-win_amd64.whl (3.7 MB)
   ---------------------------------------- 3.7/3.7 MB 5.8 MB/s eta 0:00:00
Installing collected packages: grpcio, grpcio-tools
  Attempting uninstall: grpcio
    Found existing installation: grpcio 1.50.0
    Uninstalling grpcio-1.50.0:
      Successfully uninstalled grpcio-1.50.0
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflict
s.
tensorboard 2.15.1 requires protobuf<4.24,>=3.19.6, but you have protobuf 4.25.1 which is incompatible.
Successfully installed grpcio-1.60.0 grpcio-tools-1.60.0
```

2.
Run the cmd:
python3 -m grpc_tools.protoc -I protos --python_out=. --grpc_python_out=. protos/greet.proto

This generates 2 files: **greet_pb2_grpc.py** & **greet_pb2.py**
i.e. the server and client stub

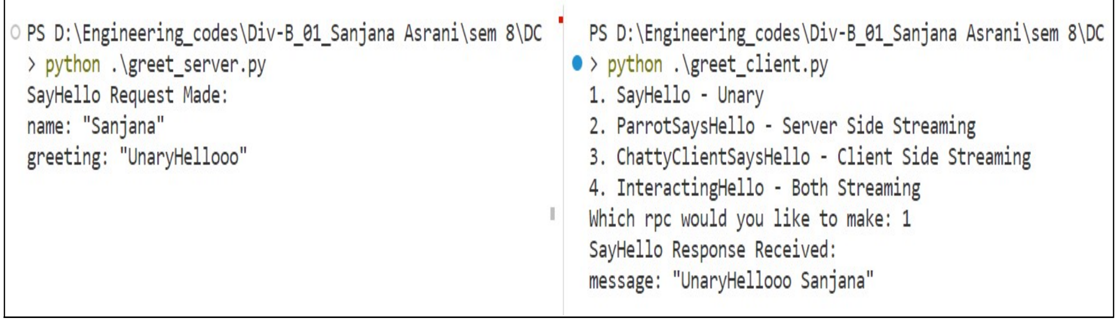
3. create server code file
4. create client code file
5. open terminal and run the server code so it starts listening
open a split terminal and run the client code.



```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC
> python .\greet_server.py
```

```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC
python .\greet_client.py
1. SayHello - Unary
2. ParrotSaysHello - Server Side Streaming
3. ChattyClientSaysHello - Client Side Streaming
4. InteractingHello - Both Streaming
Which rpc would you like to make:
```

Choose 1: **UnaryRPC**
as soon as 1 is selected from client side, it is receieved on the server side



```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC
> python .\greet_server.py
SayHello Request Made:
name: "Sanjana"
greeting: "UnaryHellooo"
```

```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC
> python .\greet_client.py
1. SayHello - Unary
2. ParrotSaysHello - Server Side Streaming
3. ChattyClientSaysHello - Client Side Streaming
4. InteractingHello - Both Streaming
Which rpc would you like to make: 1
SayHello Response Received:
message: "UnaryHellooo Sanjana"
```

CONCLUSION:
I have successfully navigated a menu-driven program, opting for the Unary RPC option and implementing the chosen task. Understanding of Unary RPC, Stubs, RMI, and diverse RPC types is done along with practical implementation of the unary RPC

Sanjana Asrani **D17B/01/Batch B** **DC Lab-01**

AIM:
Write a program to demonstrate Inter-process communication in Client Server Environment.- gRPC
A. Server streaming RPCs
B. Client Streaming and
C. Bidirectional streaming

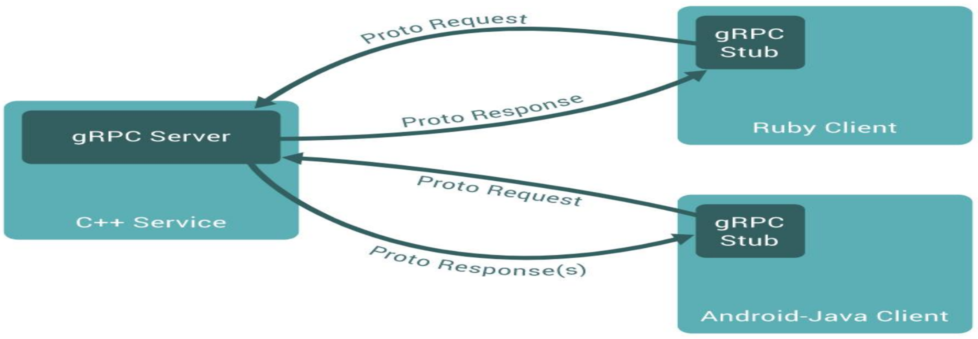
THEORY:

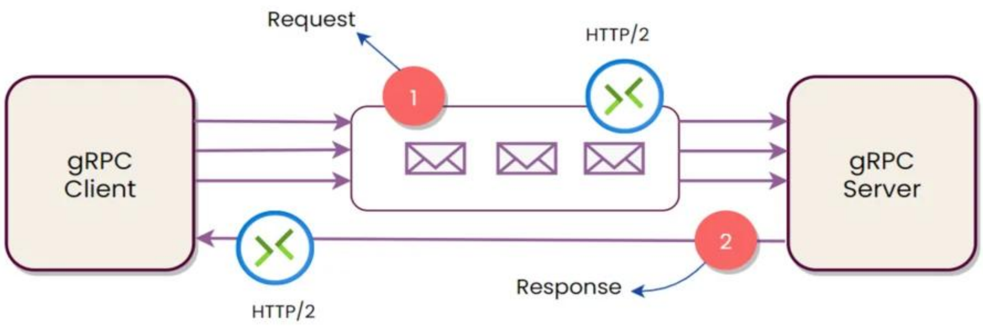
Server Streaming RPC:
A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client's request. After sending all its messages, the server's status details (status code and optional status message) and optional trailing metadata are sent to the client. This completes processing on the server side. The client completes once it has all the server's messages.
Characteristics: Useful when the server needs to send a sequence of data to the client, and the client can start processing the data as it arrives.

Client Streaming RPC:
A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message. The server responds with a single message (along with its status details and optional trailing metadata), typically but not necessarily after it has received all the client's messages.
Characteristics: Suitable for scenarios where the client needs to send a significant amount of data to the server, and the server processes the data as it arrives.

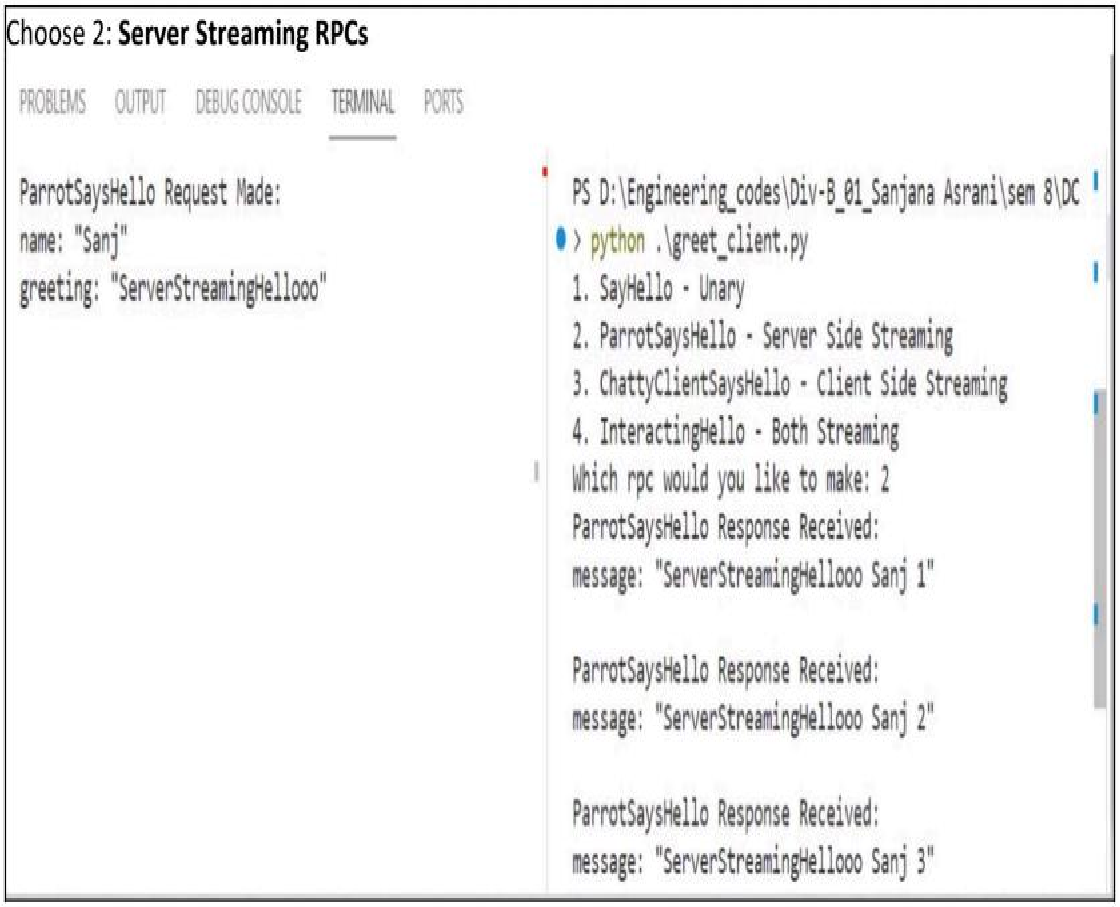
Bidirectional Streaming RPC:
In a bidirectional streaming RPC, the call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline. The server can choose to send back its initial.
Characteristics: Enables continuous communication between the client and server, allowing for interactive and real-time data exchange.

gRPC: Remote procedural calls working:





Output:


Choose 2: **Server Streaming RPCs**

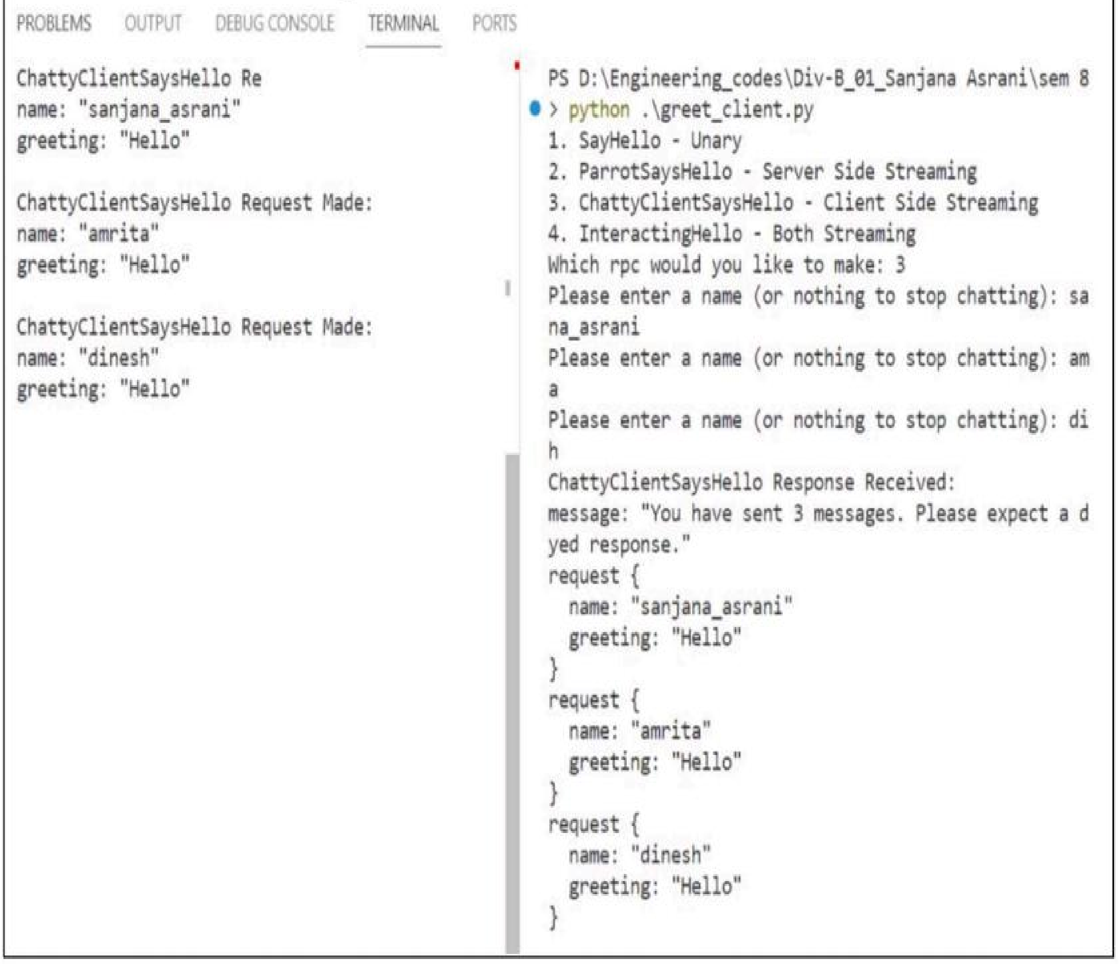
```
ParrotSaysHello Request Made:
name: "Sanj"
greeting: "ServerStreamingHellooo"
```

```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8\DC
> python .\greet_client.py
1. SayHello - Unary
2. ParrotSaysHello - Server Side Streaming
3. ChattyClientSaysHello - Client Side Streaming
4. InteractingHello - Both Streaming
Which rpc would you like to make: 2
ParrotSaysHello Response Received:
message: "ServerStreamingHellooo Sanj 1"

ParrotSaysHello Response Received:
message: "ServerStreamingHellooo Sanj 2"

ParrotSaysHello Response Received:
message: "ServerStreamingHellooo Sanj 3"
```

Choose 3: **Client Side Streaming**



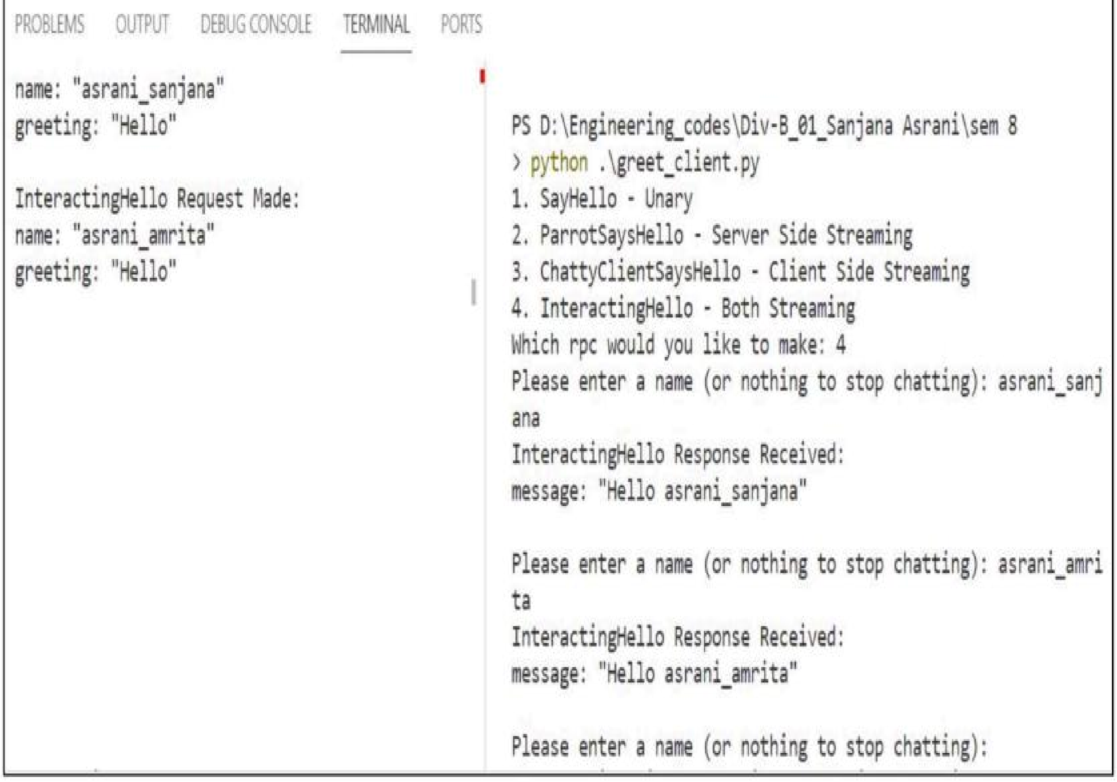
```
ChattyClientSaysHello Re
name: "sanjana_asrani"
greeting: "Hello"

ChattyClientSaysHello Request Made:
name: "amrita"
greeting: "Hello"

ChattyClientSaysHello Request Made:
name: "dinesh"
greeting: "Hello"
```

```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8
> python .\greet_client.py
1. SayHello - Unary
2. ParrotSaysHello - Server Side Streaming
3. ChattyClientSaysHello - Client Side Streaming
4. InteractingHello - Both Streaming
Which rpc would you like to make: 3
Please enter a name (or nothing to stop chatting): sa
na_asrani
Please enter a name (or nothing to stop chatting): am
a
Please enter a name (or nothing to stop chatting): di
h
ChattyClientSaysHello Response Received:
message: "You have sent 3 messages. Please expect a d
yed response."
request {
  name: "sanjana_asrani"
  greeting: "Hello"
}
request {
  name: "amrita"
  greeting: "Hello"
}
request {
  name: "dinesh"
  greeting: "Hello"
}
```

Choose 4: **Both sides are streaming hence Bidirectional**



```
name: "asrani_sanjana"
greeting: "Hello"

InteractingHello Request Made:
name: "asrani_amrita"
greeting: "Hello"
```

```
PS D:\Engineering_codes\Div-B_01_Sanjana Asrani\sem 8
> python .\greet_client.py
1. SayHello - Unary
2. ParrotSaysHello - Server Side Streaming
3. ChattyClientSaysHello - Client Side Streaming
4. InteractingHello - Both Streaming
Which rpc would you like to make: 4
Please enter a name (or nothing to stop chatting): asrani_sanj
ana
InteractingHello Response Received:
message: "Hello asrani_sanjana"

Please enter a name (or nothing to stop chatting): asrani_amri
ta
InteractingHello Response Received:
message: "Hello asrani_amrita"

Please enter a name (or nothing to stop chatting):
```