**gRPC Notes**

**What is RPC**

* RPC stands for Remote Procedure Call. It is a communication protocol that allows a program on one computer to execute a procedure (or method) on a remote computer. In other words, it enables distributed applications to invoke functions or methods on other systems as if they were local.
* The client sends a request to the server, specifying the procedure to be executed and any necessary parameters. The server receives the request, executes the procedure, and sends back the result to the client.
* The key idea behind RPC is to abstract the network communication details and make remote calls appear as if they were local calls. This allows developers to write distributed applications without having to worry about low-level network protocols, message encoding, or other complexities.
* Note that the client and server can be written in different languages
* A picture containing text, screenshot, font, smartphone

  Description automatically generated

**What is gRPC**

* gRPC (Google Remote Procedure Call) is an open-source high-performance RPC framework developed by Google. (gRPC is a specific implementation of RPC)
* By default, gRPC uses Protocol Buffers (protobuf) for describing the service interface and the structure of the payload (note that other data formats such as JSON can be used too). Protocol Buffers is a language-agnostic binary serialization format also developed by Google, which provides a compact and efficient way to serialize structured data.
* One of the key advantages of gRPC is its performance. It utilizes HTTP/2 as the underlying transport protocol, enabling features like multiplexing, header compression, and server-side streaming, which lead to efficient communication and reduced network latency.
* Additionally, gRPC supports bi-directional streaming, enabling both the client and server to send multiple messages asynchronously.
* gRPC supports various programming languages, including C++, Java, Python, Go, Ruby

**Installing the Protocol Buffer Compiler (protoc)**

1. Download the protoc binary
   * Go to the Protocol Buffers GitHub repository's releases page: <https://github.com/protocolbuffers/protobuf/releases>
   * Scroll down to the "Assets" section of the latest release and find the package suitable for your Windows version. For example, if you have a 64-bit system, download the package with win64 in its name.
   * Click on the downloaded package to start the download.
2. Extract the protoc binary
   * Once the download is complete, extract the contents.
   * You should see several files, including the protoc.exe binary.
3. Add protoc to the system PATH
   * To use protoc from any directory in the command prompt, you need to add the location of the protoc binary to your system's PATH environment variable.
   * Open the Start menu and search for "Environment Variables" and select "Edit the system environment variables".
   * In the "System Properties" window, click on the "Environment Variables" button.
   * In the "Environment Variables" window, under the "System variables" section, scroll down to find the "Path" variable and select it.
   * Click on the "Edit" button, and a new window named "Edit Environment Variable" will appear.
   * Click on the "New" button and enter the path to the ‘bin’ directory where you extracted the protoc binary (e.g., C:\protobuf\bin).
   * Click "OK" to close all the windows.
4. Verify the installation
   * To verify that the installation was successful, open a new command prompt window and run the following command:
   * 
   * If the installation was successful, you should see the version number of the Protocol Buffers Compiler displayed in the console.

**Creating a .proto file**

* Suppose we have the following folder:
* A screenshot of a computer

  Description automatically generated with medium confidence
* A screen shot of a computer program

  Description automatically generated with low confidence
* We have a greeting.prot file
* 
* The "syntax" here represents the version of Protobuf we are using. So, we are using the latest version 3 and the schema thus can use all the syntax which is valid for version 3.
* 
* The “package” here is used for conflict resolution if, say, we have multiple classes/members with the same name.
* A picture containing text, font, screenshot, graphics

  Description automatically generated
* This block represents the name of the service "Greeter" and the function name "greet" which can be called. The "greet" function takes in the input of type "ClientInput" and returns the output of type "ServerOutput". Now let us look at these types.
* A picture containing text, font, screenshot, line

  Description automatically generated
* In the above block, we have defined the ClientInput which contains two attributes, "greeting" and the "name" both of them being strings. The client is supposed to send the object of type of "ClientInput" to the server.
* A picture containing text, font, screenshot, line

  Description automatically generated
* Here, we have also defined that, given a "ClientInput", the server would return the "ServerOutput" with a single attribute "message". The server is supposed to send the object of type "ServerOutput" to the client.
* Each field within a message is identified by a unique number. This number is used for serialization and deserialization of data, as well as for identifying fields within the message. The field numbers should be positive integers.

**Compiling a .proto file**

* We need to make use of the previously installed protocol buffer compiler (protoc) to compile the .proto file to a programming language of our choice.
* If we are using python, we also have to make install the following two packages by running in cmd: python -m pip install grpcio grpcio-tools
* If we want to compile to python, we run the following command:
* python -m grpc\_tools.protoc -I . --python\_out=. --grpc\_python\_out=. greeting.proto
* 
* python -m grpc\_tools.protoc # Invokes the Protocol Buffers Compiler with grpc\_tools package
* -I . # Specifies the directory to search for imported .proto files (in this case, the current folder)
* --python\_out=. # Sets the output directory for generated Python code (in this case, the current folder)
* --grpc\_python\_out=. # Sets the output directory for generated gRPC-related Python code(in this case, the current folder)
* greeting.proto # The path to the .proto file to be compiled
* The command is used to invoke the Protocol Buffers Compiler (protoc) with the grpc\_tools Python package to compile the proto file with gRPC support.
* After compiling, we should see the following two new files:
* A screenshot of a computer

  Description automatically generated with medium confidence
* To understand the difference between greeting\_pb2\_grpc.py and greeting\_pb2.py, let's use an analogy:
* Imagine you have a company that manufactures and sells cars. The greeting\_pb2.py file is like the blueprint or specification document for the car. It defines the structure and components of the car, such as its chassis, engine, wheels, and interior.
* On the other hand, the greeting\_pb2\_grpc.py file is like the user manual or guidebook that not only describes the car but also provides instructions on how to interact with it and perform certain actions. It specifies how to start the car, change gears, accelerate, and apply brakes.
* Similarly, in the context of programming and communication between different software components, greeting\_pb2.py is a generated file that contains the definitions of the data structures (or messages) used for communication between different systems. It defines the format of the data being exchanged.
* greeting\_pb2\_grpc.py, on the other hand, is another generated file that contains the necessary code to implement remote procedure call (RPC) functionality using the defined messages. It provides the necessary infrastructure to make remote method calls and handle the communication between the client and the server.
* In summary, greeting\_pb2.py defines the structure of the data being exchanged, while greeting\_pb2\_grpc.py provides the tools and functionality to actually perform remote procedure calls using that data structure.

**Setting up a gRPC server**

* A screen shot of a computer program

  Description automatically generated with low confidence
* The first 4 lines import the necessary modules for creating the gRPC server.
* The lines 6-9 define a class Greeter
  + The Greeter class is an implementation of the gRPC service defined in the .proto file.
  + The GreeterServicer class is a base class generated by the gRPC framework based on the service definition provided in the greeting.proto file. It acts as an interface for implementing the gRPC service. The purpose of the GreeterServicer class is to define the methods that need to be implemented by the derived class in order to handle the RPC methods defined in the gRPC service.
  + When implementing a gRPC service in Python, you create a derived class (such as Greeter) that extends the GreeterServicer base class. This derived class contains the actual logic and implementation for handling the RPC methods.
  + Greeter extends the generated GreeterServicer class. By extending the GreeterServicer class, the Greeter class inherits all the necessary methods and functionality required to implement the gRPC service.
  + In this code, the Greeter class overrides one of the methods provided by the base class, which is the ‘greet’ method. This method is invoked when a client sends a request to the server's greet RPC method. The method takes two parameters:
    - request: This parameter represents the request message sent by the client. The specific structure and fields of the request message are defined in the greeting.proto file and generated in the greeting\_pb2 module.
    - context: This parameter provides information about the RPC, such as metadata associated with the request, authentication details, and cancellation signals.
* Line 12-17 define the server() function which is responsible for setting up and starting the gRPC server. The server() function acts as the entry point for starting the gRPC server. When this function is called, it initializes the server, sets up the necessary configurations, associates the servicer implementation, and then starts the server to handle incoming requests. Once the server is started, it will continue running until explicitly stopped, allowing it to handle multiple requests concurrently as they arrive.
* grpc.server()
  + The grpc.server() method is a function provided by the gRPC library in Python. It is used to create a gRPC server object that can handle incoming RPC (Remote Procedure Call) requests.
  + The grpc.server() method returns an instance of the grpc.Server class, which represents the gRPC server. This server object provides methods to configure and control the behavior of the server.
  + Here is the general syntax of the grpc.server() method: 
  + The executor parameter is an instance of the concurrent.futures.Executor class. It defines the executor to be used by the server for handling incoming requests concurrently. The executor manages a pool of worker threads that execute the RPC method invocations.
  + The options parameter is an optional argument that allows you to provide additional configuration options to the server. These options can be used to customize various aspects of the server's behavior, such as enabling compression, setting maximum message sizes, or configuring authentication.
  + Once the server object is created, you can add service implementations (servicers) to it using the add\_service() or add\_servicer\_to\_server() methods. These methods associate the implementation of your gRPC service with the server, allowing it to handle the incoming requests for that service.
  + Finally, you can start the server by calling the start() method. This will initiate the server's event loop and make it start listening for incoming RPC requests.
  + Thus line 12 creates a gRPC server object using grpc.server and configures it with a ThreadPoolExecutor. The ThreadPoolExecutor allows the server to handle multiple requests concurrently by utilizing a pool of worker threads. In this case, the maximum number of worker threads is set to 2.
* greeting\_pb2\_grpc.add\_GreeterServicer\_to\_server()
  + greeting\_pb2\_grpc.add\_GreeterServicer\_to\_server(...): This is a generated function provided by the gRPC framework. It is automatically generated based on the service definition in greeting.proto and resides in the greeting\_pb2\_grpc module. The add\_GreeterServicer\_to\_server() function adds the Greeter class as a servicer to the gRPC server.
  + Greeter(): This creates an instance of the Greeter class. The Greeter class is the implementation of the gRPC service defined in greeting.proto. It contains the logic and functionality for handling the RPC methods.
  + This function takes two parameters
    - servicer\_instance: This parameter represents an instance of the class that implements the gRPC service. In this case, it expects an instance of the Greeter class, which is the implementation of the Greeter service.
    - server: This parameter represents the gRPC server object to which the servicer\_instance will be added. It expects an instance of the grpc.Server class, which is created using the grpc.server() method.
  + By calling add\_GreeterServicer\_to\_server(Greeter(), server), you are adding an instance of the Greeter class as a servicer to the gRPC server. This allows the server to route incoming requests for the Greeter service to the corresponding methods in the Greeter class for processing.
  + Greeter() (inside the add\_GreeterServicer\_to\_server() function): Here, the instance of the Greeter class is passed as an argument to the add\_GreeterServicer\_to\_server() function. This tells the function which implementation to associate with the server for handling requests.
* server.add\_insecure\_port('[::]:50051')
  + server: This is the gRPC server object created using the grpc.server() method earlier. The add\_insecure\_port() method is called on this server object to configure the network address and port.
  + add\_insecure\_port(): This is a method provided by the gRPC server object. It is used to specify the network address and port on which the server will listen for incoming requests.
  + '[::]:50051': This argument represents the network address and port. In this case, it uses the IPv6 loopback address :: (which represents all IPv6 addresses on the local machine) and port 50051. The :: is enclosed in square brackets as required for IPv6 addresses.
  + By calling server.add\_insecure\_port('[::]:50051'), you are configuring the gRPC server to listen on the specified network address and port. This means that the server will be bound to that address and will be able to receive incoming gRPC requests from clients that connect to that address and port.
* server.start():
  + This line starts the gRPC server, allowing it to begin listening for incoming requests on the specified address and port.
* server.wait\_for\_termination()
  + The server.wait\_for\_termination() line in your code is responsible for blocking the main thread and waiting for the gRPC server to be terminated. It ensures that the server continues running until it is explicitly stopped or interrupted.
  + Here's how server.wait\_for\_termination() works:

1. After the gRPC server is started using the server.start() method, the server.wait\_for\_termination() line is executed.
2. The wait\_for\_termination() method is a blocking call that halts the execution of the main thread and waits for the server to be terminated.
3. The server will continue running, handling incoming requests and executing the corresponding methods in the servicer, until one of the following events occurs
   1. The server is explicitly stopped by calling server.stop(grace) from another thread or process. The grace parameter specifies the grace period before forcefully terminating the server.
   2. An exception or signal interrupts the execution, causing the server to terminate.
4. Once the server is terminated, the wait\_for\_termination() method will return, and the program can proceed to the next line of code, if any.
   * By including server.wait\_for\_termination() in your code, you ensure that the server remains active and continues processing requests until it is intentionally stopped or an external event triggers termination. This allows the server to handle incoming requests for an extended period, waiting for clients to interact with the gRPC service.

**Setting up a gRPC client**

* A screen shot of a computer screen

  Description automatically generated with low confidence
* The run() function is defined, which serves as the entry point for the client application.
* grpc.insecure\_channel
  + Inside the run() function, a gRPC channel is created using grpc.insecure\_channel('localhost:50051'). The insecure\_channel function creates an insecure channel to the specified address and port. In this case, it connects to the gRPC server running on localhost at port 50051.
  + By using with grpc.insecure\_channel('localhost:50051') as channel:, the channel variable is assigned to the result of grpc.insecure\_channel('localhost:50051'), and this channel is used within the scope of the with statement. Once the execution of the code inside the with block is finished, the channel is automatically closed.
  + Closing the channel is important to release the network resources and connections established during the communication. It helps to avoid resource leaks and ensures the proper cleanup of the underlying gRPC infrastructure
* greeting\_pb2\_grpc.GreeterStub
  + In summary, stub = greeting\_pb2\_grpc.GreeterStub(channel) creates a client stub object of the GreeterStub class. A client stub is an API for making RPC requests and handling responses.
  + The GreeterStub class is part of the greeting\_pb2\_grpc module and is generated based on the service definition. It represents the client-side stub for the Greeter service.
  + This GreeterStub class takes a channel parameter which is an object that represents the communication channel between the client and server. It is created using the grpc.insecure\_channel() function, as seen in the code snippet. The channel specifies the network address and port to connect to the gRPC server.
  + Once the stub object is created, it can be used to invoke methods that correspond to the RPCs defined in the Greeter service.
* response = stub.greet(greeting\_pb2.ClientInput(name='John', greeting="Yo"))
  + The line response = stub.greet(greeting\_pb2.ClientInput(name='John', greeting="Yo")) makes an RPC call to the server using the greet method of the stub object. It passes a ClientInput message object as an argument, which is created from the greeting\_pb2 module. The ClientInput message has two fields: name and greeting. In this case, the name is set to 'John' and the greeting is set to 'Yo'.
  + The response from the server is assigned to the ‘response’ variable.
  + We can now do whatever we want with the response. In this case, the received response from the server is printed using print("Greeter client received following from server: " + response.message).

**Unary**

* Unary RPC is a simple request-response pattern in which the client sends a single request to the server and receives a single response in return.
* The client waits for the server's response before proceeding further.
* It is suitable for simple and lightweight operations where a single request is sufficient to complete the task.
* Example use cases: fetching data, executing a command, performing a simple calculation.
* Ex:
  + Proto file:
  + A computer screen with white text

    Description automatically generated
  + Server code:
  + A computer screen with many colorful text

    Description automatically generated
  + Client code:
  + A screen shot of a computer program

    Description automatically generated with low confidence

**Bidirectional Streaming**

* Bidirectional streaming allows both the client and server to send a stream of messages asynchronously.
* The client can send multiple messages to the server, and the server can send multiple messages back to the client.
* The communication is not limited to a fixed number of requests or responses.
* Both the client and server can start sending messages at any time and in any order.
* Example use cases: real-time chat applications, real-time collaboration, data synchronization.
* Ex:
  + Proto file:
  + A computer screen with white text

    Description automatically generated
  + Explanation for the line “rpc StartChat (stream ChatMessage) returns (stream ChatMessage) {}” :
    - rpc: The rpc keyword is used to define an RPC method within a gRPC service.
    - StartChat: StartChat is the name of the RPC method. You can think of it as a function or procedure that can be invoked on the server-side.
    - (stream ChatMessage): This part defines the input parameter for the StartChat method. In this case, it uses the stream keyword, indicating that the input is a continuous stream of ChatMessage objects. The ChatMessage represents the type of message being sent from the client to the server.
    - returns: The returns keyword indicates that the RPC method returns a result.
    - (stream ChatMessage): This part defines the output parameter for the StartChat method. Similarly to the input parameter, it uses the stream keyword, indicating that the output is a continuous stream of ChatMessage objects. In this case, the ChatMessage objects represent the messages sent from the server back to the client.
    - Therefore, the StartChat RPC method allows bidirectional streaming. The client can continuously send ChatMessage objects to the server, and the server can continuously send ChatMessage objects back to the client. This bidirectional streaming allows for real-time and interactive communication between the client and server.
  + Server code:
  + A screen shot of a computer program

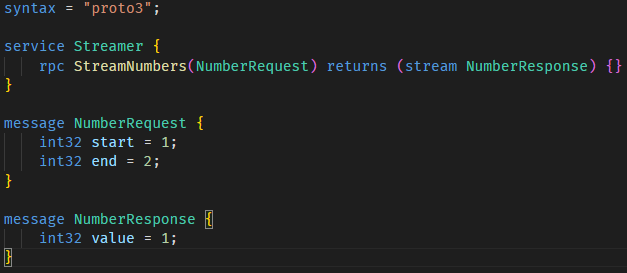
    Description automatically generated
  + The StartChat method is the core logic that handles the bidirectional streaming communication between the client and the server. It receives messages from the client, performs any necessary processing or computations, generates appropriate responses, and sends the responses back to the client.
  + The request\_iterator parameter is an iterator that provides access to the incoming stream of ChatMessage objects sent by the client. It allows you to iterate over the messages sent by the client.
  + The for loop iterates over each request object in the request\_iterator, which represents an incoming ChatMessage sent by the client.
  + The sender and message fields of the request object are extracted and stored in local variables for further processing. A message is printed to the console indicating the received sender and message.
  + A response message is created using mypackage\_pb2.ChatMessage. In this case, the sender field is set to 'Server', indicating that the server is the sender, and the message field is set to 'Hello {sender}! Message received', which is a response message acknowledging the receipt of the client's message.
  + The response message is then yielded using the yield keyword. By yielding the response message using yield response, it is sent back to the client as part of the bidirectional streaming response. Once the response has been yielded, the method continues executing from where it left off. It waits for the next iteration of the loop, where it will process the next incoming message from the client and generate another response to be sent back using yield.
  + The method continues to listen for incoming messages from the client. For each message, it processes the request, generates a response, and yields the response back to the client.
  + Client code:
  + A computer screen shot of text

    Description automatically generated
  + The start\_chat function is defined to encapsulate the client logic.
  + with grpc.insecure\_channel('localhost:50051') as channel:: The client creates an insecure gRPC channel to connect to the server. It specifies the server address as 'localhost:50051', indicating that the server is running on the local machine and listening on port 50051. The with statement ensures that the channel is properly closed after the client finishes its work.
  + stub = mypackage\_pb2\_grpc.ChatStub(channel): The client creates a stub, which acts as a client-side proxy for the gRPC service. The ChatStub class is generated from the protocol buffer definitions and provides methods corresponding to the RPC methods defined in the Chat service.
  + Here, a request iterator is created using a generator expression. It generates ChatMessage objects with the sender set as "Client" and the message containing a sequential number. In this case, the request iterator will produce 5 ChatMessage objects with messages like "Hello Server! Message 1", "Hello Server! Message 2", and so on.
  + The bidirectional streaming RPC is invoked by calling the StartChat method on the gRPC stub (stub.StartChat(request\_iterator)). This method takes the request iterator as an argument. It sends the chat messages from the client to the server and simultaneously receives the responses from the server. The for loop iterates over the responses received from the server. Each response is stored in the entry\_response variable, and in this example, it is printed to the console using print(entry\_response). It's important to note that the for loop will block until all the responses are received from the server. The loop will continue as long as the server continues to send responses.
  + Overall, the start\_chat method establishes a connection to the server, creates a request iterator, sends the chat messages to the server using bidirectional streaming, and receives and processes the responses from the server.
  + Output from client:
  + A screen shot of a computer

    Description automatically generated
  + Output from server:
  + A screenshot of a computer screen

    Description automatically generated

**Server Streaming**

* In server streaming, the client sends a single request to the server, and the server responds with a stream of messages.
* The server sends multiple responses asynchronously, and the client can read them one by one.
* The client waits for the server to complete sending all the responses or can stop reading at any time.
* Example use cases: sending a large amount of data from the server to the client, real-time data feeds, logging.
* Below is an example:
* First, you need to define the service and messages using protocol buffers. Let's assume we have a simple service called "Streamer" that streams a sequence of numbers:
* 
* Below is the server code:
* A computer screen shot of text

  Description automatically generated
* In the above code, we define a class StreamerServicer that inherits from the generated StreamerServicer class. Inside this class, we implement the StreamNumbers method, which takes a NumberRequest and a context object. We iterate over the range specified in the request and yield a NumberResponse for each number in the range.
* To test the server streaming, you can write a client that calls the StreamNumbers method and receives the stream of numbers.
* Below is the client code:
* A screen shot of a computer code

  Description automatically generated
* This client code connects to the server, sends a request, and receives a stream of responses from the server.
* In the client code, we start by creating a channel using grpc.insecure\_channel with the server's address (localhost:50051 in this case). We then create a stub using streamer\_pb2\_grpc.StreamerStub(channel).
* Next, we create a NumberRequest with the desired range (start=1, end=10 in this example). We call the StreamNumbers method on the stub, passing the request. This will return an iterator of NumberResponse objects.
* We iterate over the responses using a for loop, printing each received value. The client will keep receiving responses until the server completes or encounters an error.

**Client Streaming**

* gRPC client streaming is a type of communication pattern where the client sends a stream of messages to the server. This is useful in scenarios where the client needs to send a large amount of data or multiple messages to the server over a single connection.
* To demonstrate gRPC client streaming in Python, we'll create a simple example using a file transfer service. The client will stream chunks of a file to the server, which will save them to disk.
* Ex:
* .proto file:
* A screen shot of a computer program

  Description automatically generated
* Client.py
* A computer screen shot of text

  Description automatically generated
* Server.py
* A computer screen shot of a program code

  Description automatically generated