Lab 4

Distributed Software Systems

UNIBO - 2024/25

Anouk Wuyts - 1900124167

Davide De Rosa - 1186536

RPC - Remote Procedure Call

Communication model in distributed systems that allows a program to execute a **procedure** (function or method) on a remote server as if it were a *local procedure call*.

The **RPC mechanism** abstracts the complexity of remote communication, making it look like a regular function call, even though it may involve network communication, data serialization, and error handling.

It's commonly used in client-server models and microservices architectures, where components of a system may need to interact over a network. By abstracting the complexity of network communication, RPC allows developers to write distributed applications more easily.

gRPC

Open-source framework developed by **Google** that facilitates communication between different services in a distributed system. It allows you to define and call functions (also known as *procedures*) that can be executed on different machines, languages, or environments, making it ideal for microservice architectures.

gRPC - Stub

A **stub** in gRPC is an abstraction that provides the client with an interface to invoke remote procedures on the server. Think of the stub as a client-side proxy or helper that mimics the behavior of a local method call while actually communicating with a remote server. The stub is used on both the client and the server:

- **Client Stub**: this is a generated class based on the **.proto file**, which contains methods that correspond to the service's RPC methods. When a client calls one of these methods, the stub handles sending the request to the server, waiting for the response, and then returning the result to the client. The client doesn't need to worry about network communication, serialization, or handling low-level details — the stub does this for them.

- **Server Stub**: on the server side, the stub receives incoming RPC requests, unpacks the arguments, and invokes the actual service implementation (the business logic). After execution, the server stub packages the response and sends it back to the client.

The stub essentially hides the complexity of the network communication, allowing both the client and server to interact as if they were calling methods locally.

gRPC - Protocol Buffers

gRPC uses **Protocol Buffers (protobuf)** as its **Interface Definition Language** (**IDL**) and serialization format.

Protobuf allows developers to define services and their methods, as well as the data structures (messages) used in those methods. It is a key part of how gRPC defines and enforces the communication contract between the client and server.

The *.proto file* is used to define the structure of the messages and the RPC services.

gRPC - Services

A **gRPC service** is a collection of RPC methods that the server provides to clients. Each service is defined in the *.proto file*, and it consists of:

- **Methods**: the RPCs available in the service.
- **Input types**: the request message type that the method accepts.
- **Output types**: the response message type that the method returns.

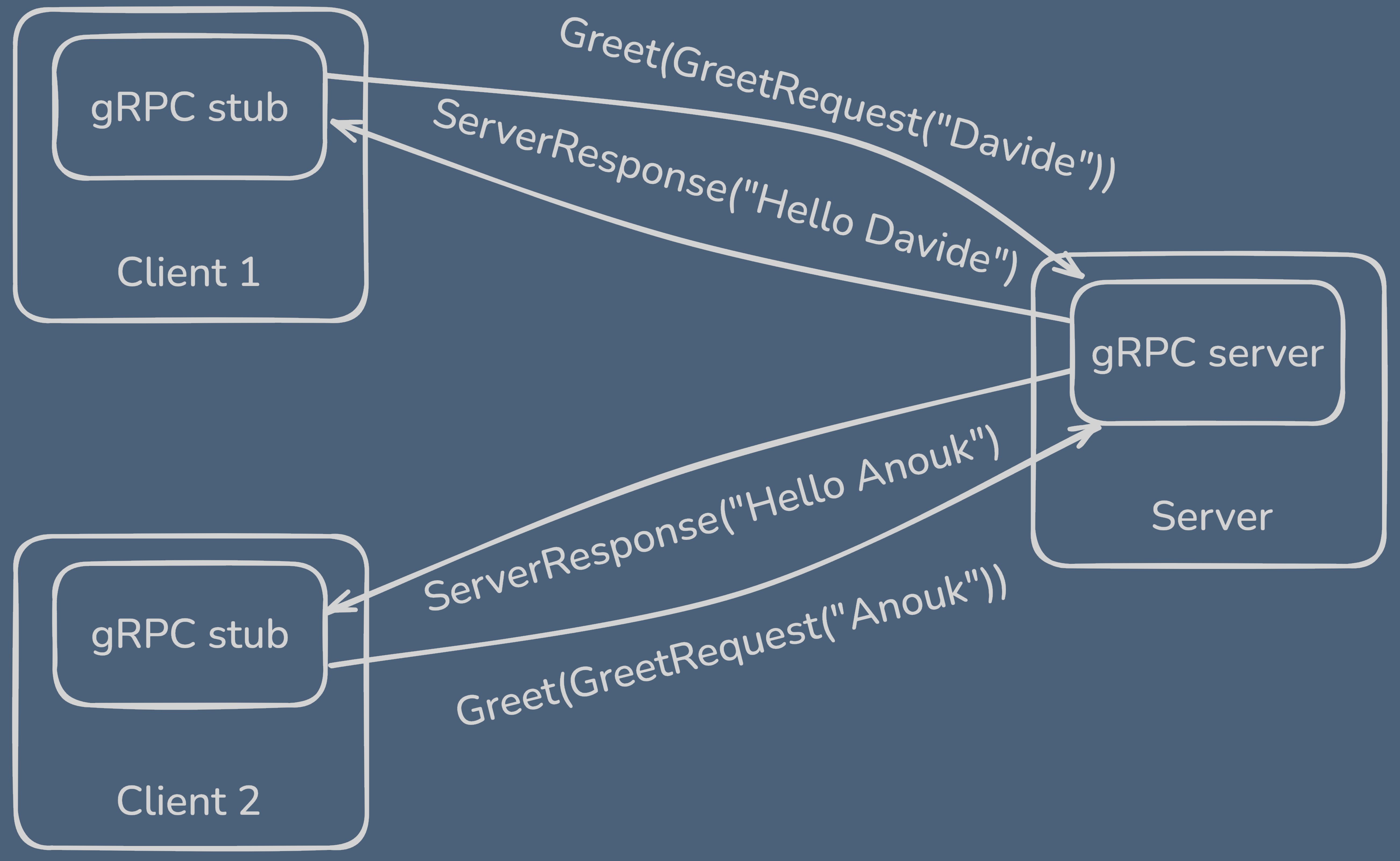
An example will be shown in the *implementation* part.

Greeting Service Implementation

We implemented a greeting gRPC client-server architecture.

When the client wants to interact with the gRPC server it calls the Greet method and gives its name along.

The gRPC server then responds to the client by giving it a personalized greeting message, using the given name of the client.


gRPC stub
Client 1
Greet(GreetRequest("Davide"))
ServerResponse("Hello Davide")
gRPC server
Server
ServerResponse("Hello Anouk")
Greet(GreetRequest("Anouk"))
gRPC stub
Client 2

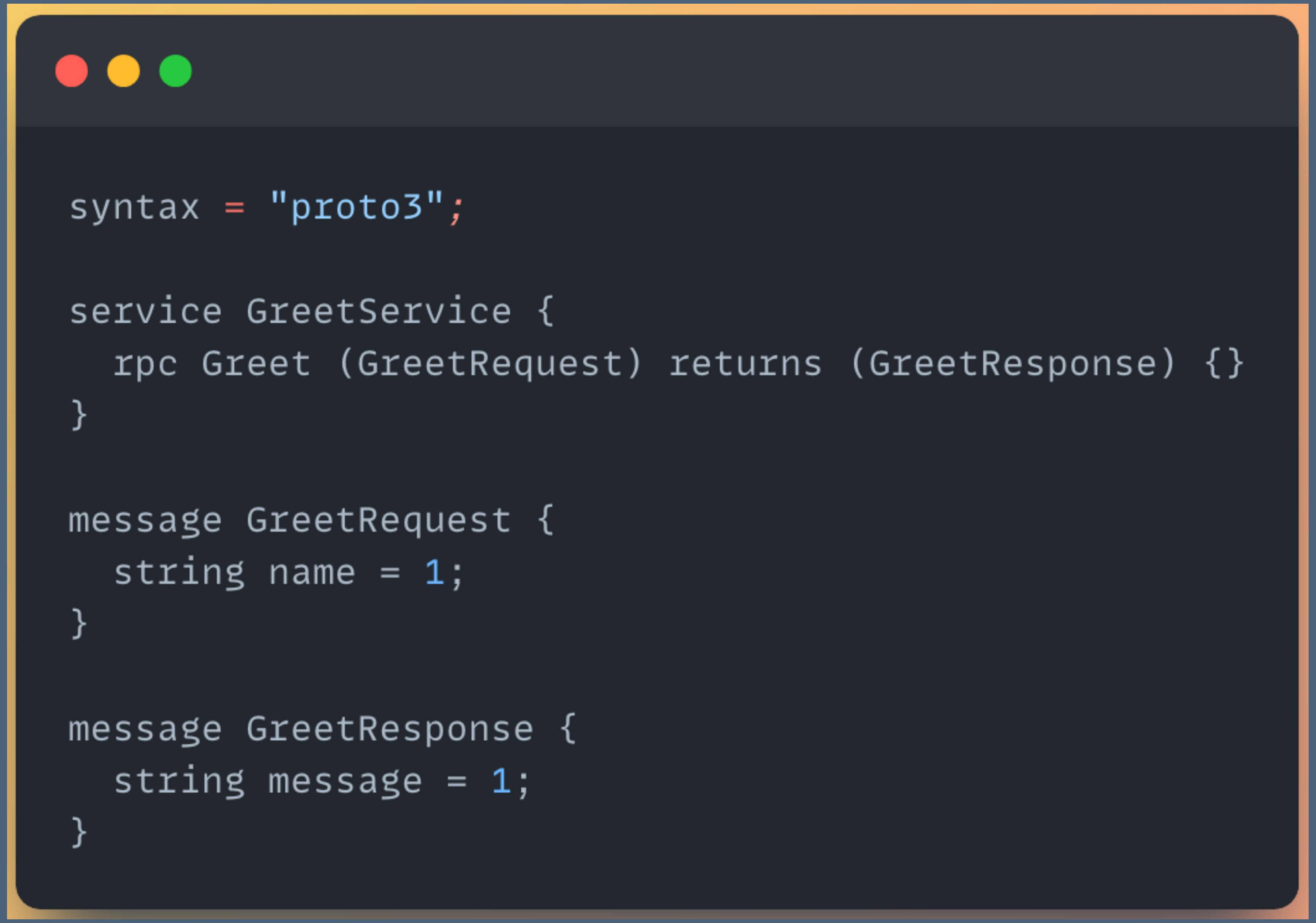
Greeting Service Implementation - .proto file

First of all, we created the *.proto file,* which defines the **GreetService** service. It has only the **Greet** method, which sends a name to the server and receives a message as a response.

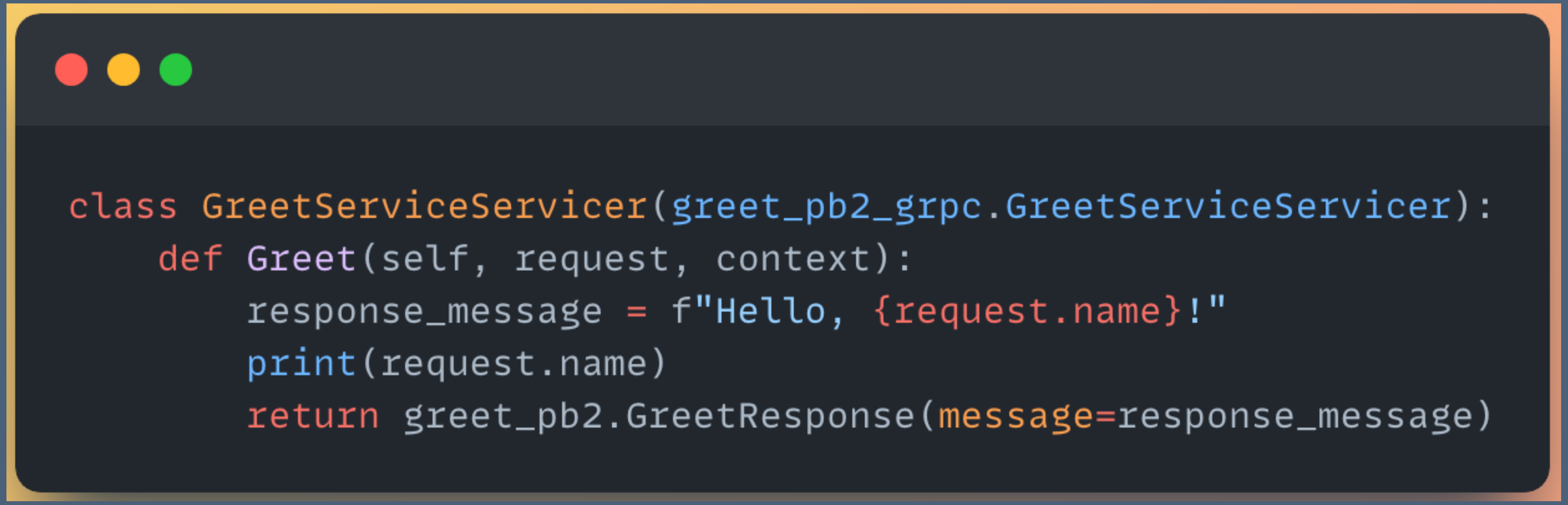
We used the following command to generate the code we needed:

```
python -m grpc_tools.protoc -I=. --python_out=. --grpc_python_out=. greet.proto
```

After generating the code, we used it to define a **GreetServiceServicer** class inside the Server script, where we defined the behaviour of the *Greet* method.



```
syntax = "proto3";

service GreetService {
  rpc Greet (GreetRequest) returns (GreetResponse) {}
}

message GreetRequest {
  string name = 1;
}

message GreetResponse {
  string message = 1;
}
```

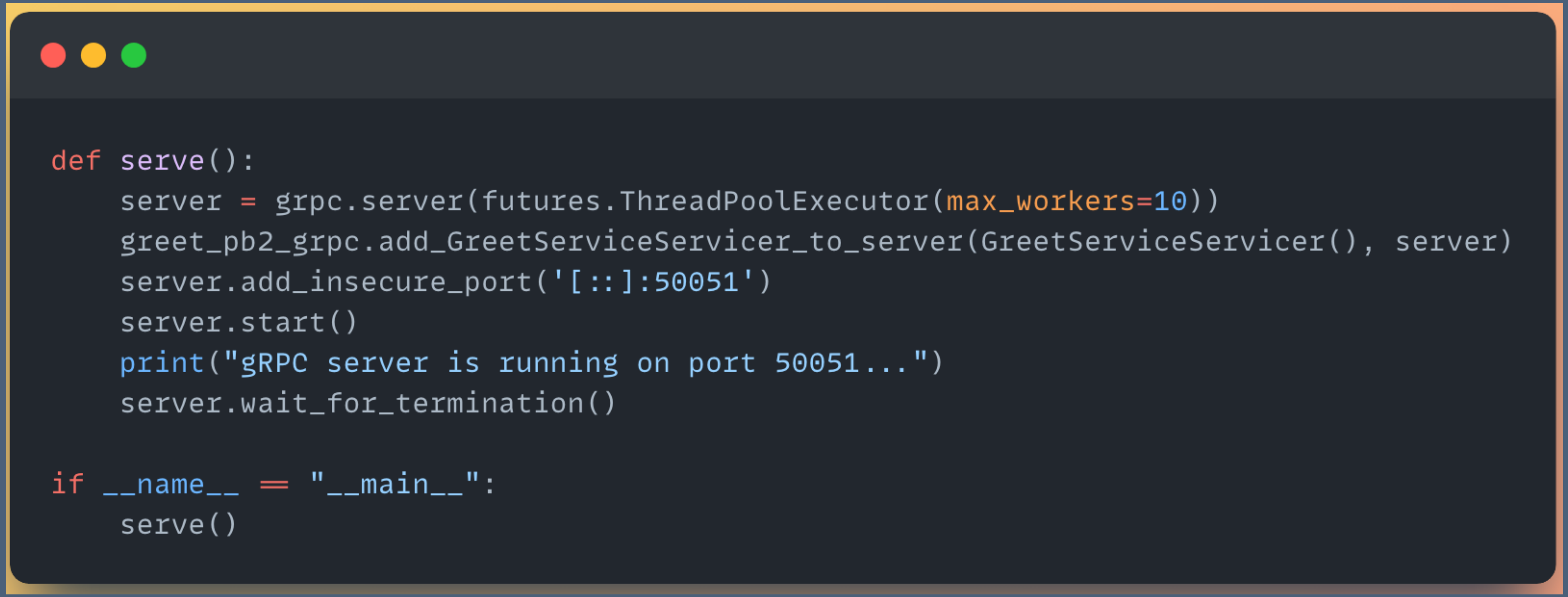


```
class GreetServiceServicer(greet_pb2_grpc.GreetServiceServicer):
    def Greet(self, request, context):
        response_message = f"Hello, {request.name}!"
        print(request.name)
        return greet_pb2.GreetResponse(message=response_message)
```

Greeting Service Implementation - Server

Inside the server main, we execute the *serve()* function, that creates and starts the gRPC Server, adding the GreetService to the server.

The server will wait and listen to the port assigned for a client to connect.

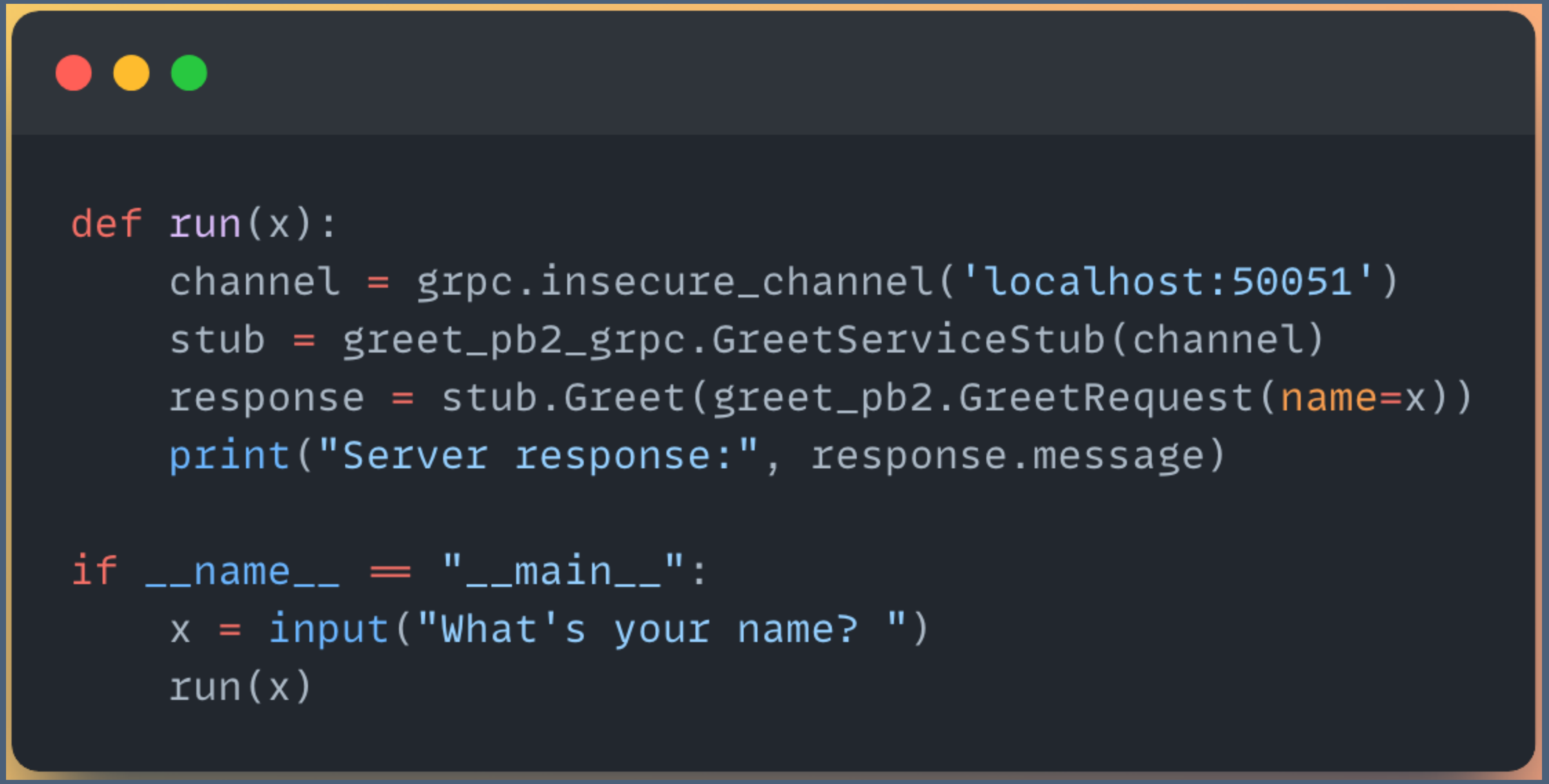


```python
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    greet_pb2_grpc.add_GreetServiceServicer_to_server(GreetServiceServicer(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    print("gRPC server is running on port 50051...")
    server.wait_for_termination()

if __name__ == "__main__":
    serve()
```

Greeting Service Implementation - Client

Inside the client main, we ask for the user to write his name. After it we execute the *run(name)* function, which tries to connect to the server and run the **Greet** method.

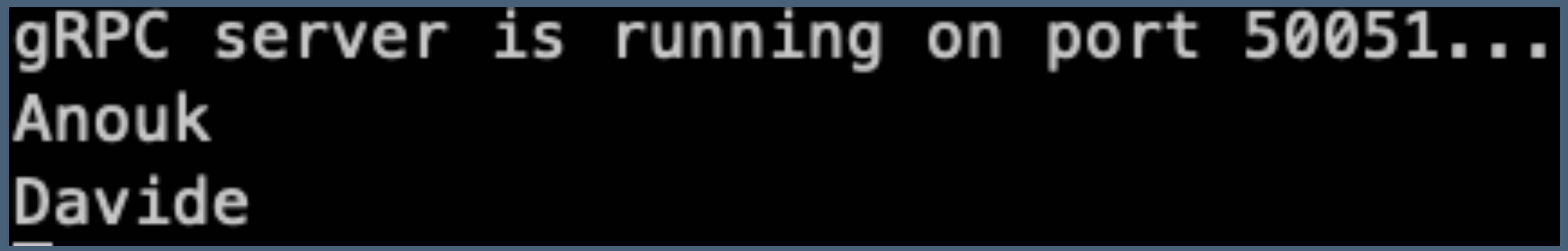
When the server responds, the message is printed on the console.



```python
def run(x):
    channel = grpc.insecure_channel('localhost:50051')
    stub = greet_pb2_grpc.GreetServiceStub(channel)
    response = stub.Greet(greet_pb2.GreetRequest(name=x))
    print("Server response:", response.message)

if __name__ == "__main__":
    x = input("What's your name? ")
    run(x)
```

Greeting Service Implementation - Usage

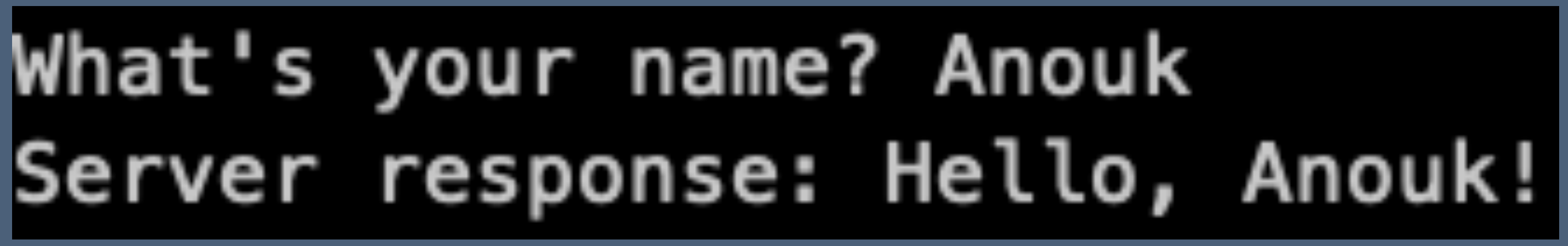
After running the server script, the server waits for clients to connect on the given port.



```
gRPC server is running on port 50051...
Anouk
Davide
```

Whenever a client connects and runs the **Greet** function, the name sent by the client gets printed on the server console.

On the client side, the user is asked to write a name. The server response is then printed on the console.



```
What's your name? Anouk
Server response: Hello, Anouk!
```

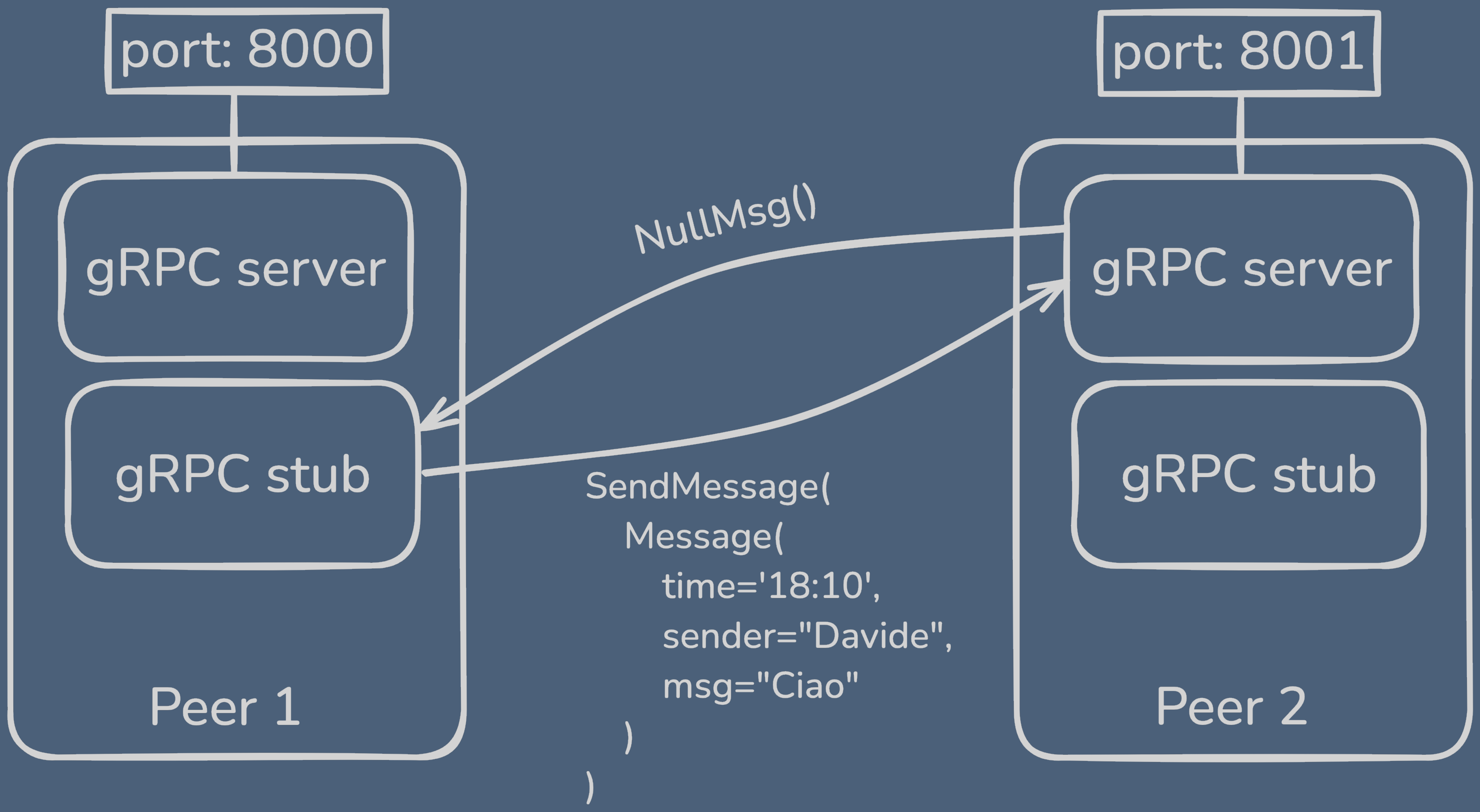
Challenge: Peer-to-Peer Chatting

We implemented a **Peer-to-Peer Chatting** service where the peer can act both as a *client* and a *server*.

When *Peer 1* wants to chat with *Peer 2*, a connection is established between the two.

When *Peer 1* wants to send a message, his **stub** sends it to the *Peer 2*'s **gRPC server**, and viceversa.

For that to work, both servers have an unique port number, which is needed to communicate.



```
SendMessage(
   Message(
      time='18:10',
      sender="Davide",
      msg="Ciao"
   )
)
```

Challenge: Peer-to-Peer Chatting - .proto file

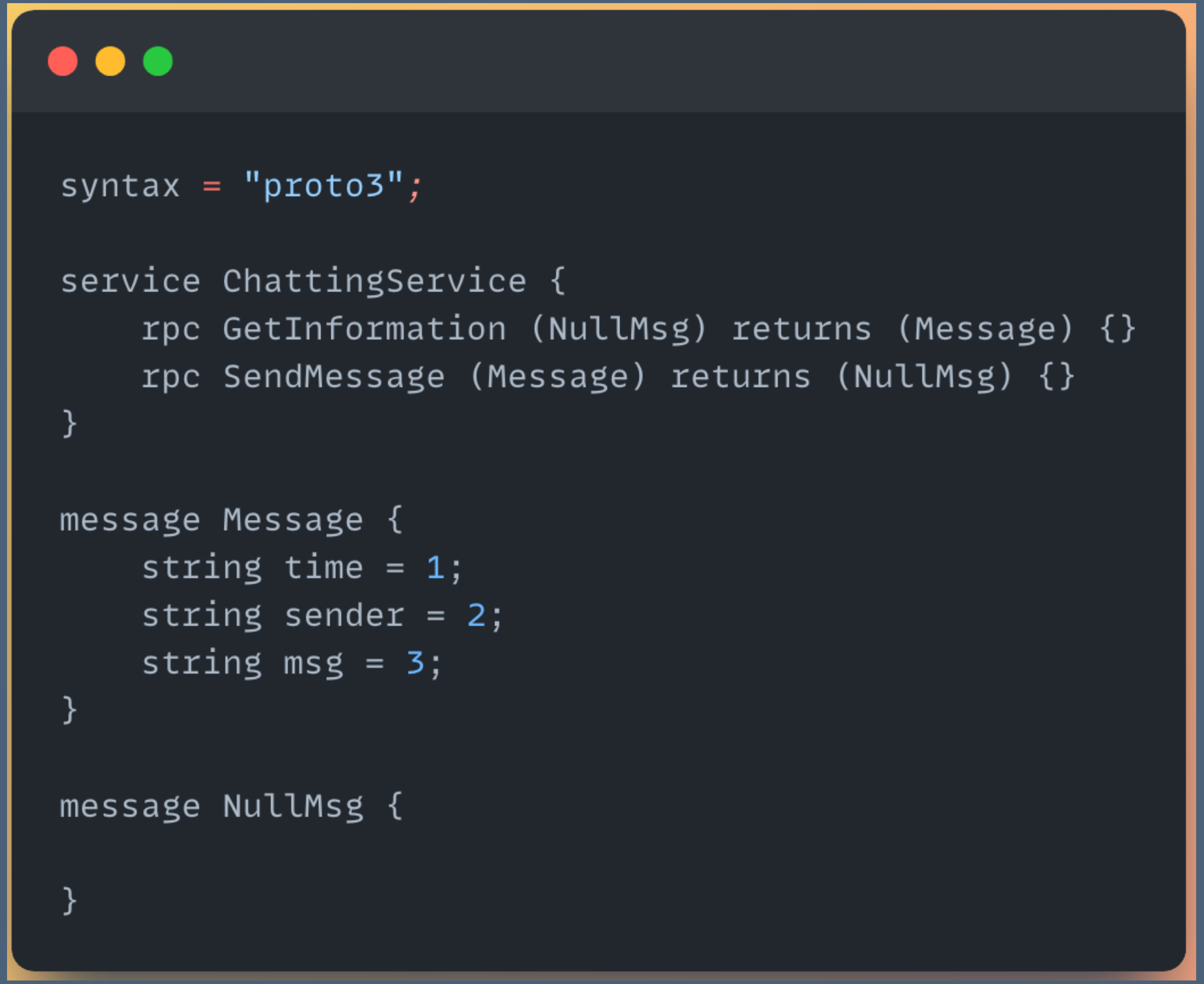
First of all, we created the *.proto file,* which defines the **ChattingService** service. It has two methods:

- **GetInformation**: used to get the other user information and establish the connection
- **SendMessage**: used to send messages

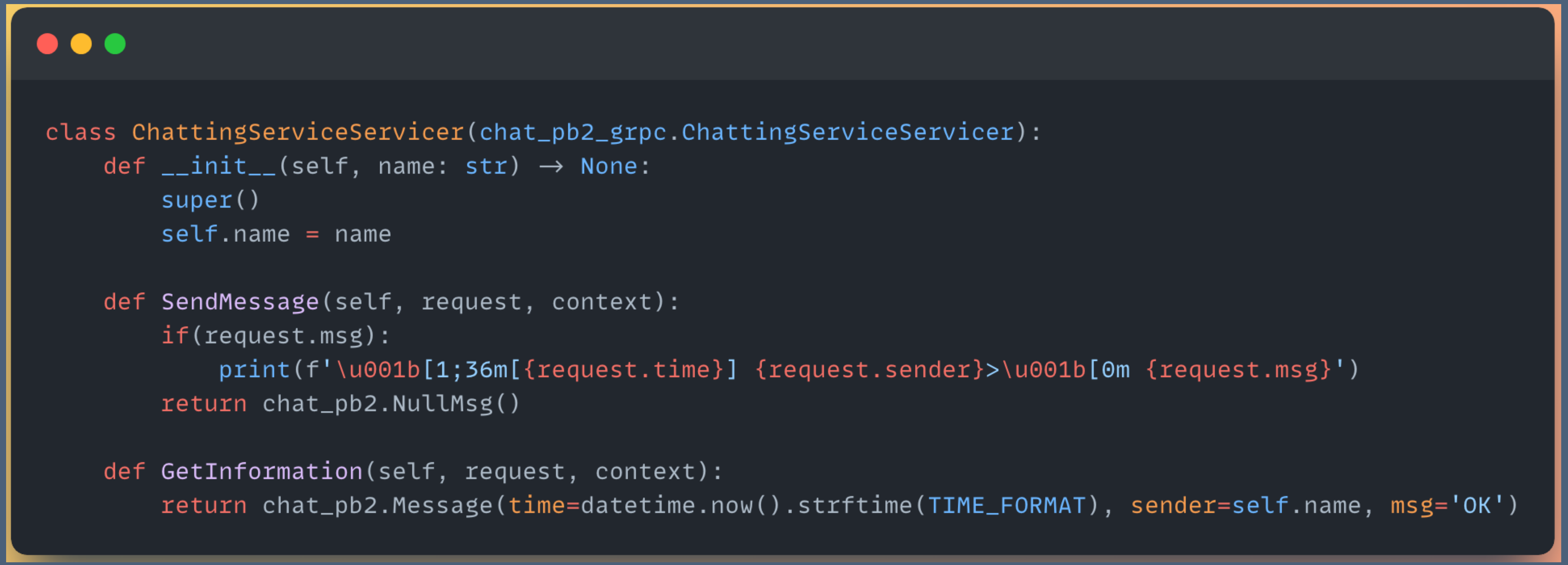
We used the following command to generate the code we needed:

```
python -m grpc_tools.protoc -I=. --python_out=. --grpc_python_out=. chat.proto
```

After generating the code, we used it to define a **ChattingServiceServicer** class inside the Chatter script, where we defined the behaviour of both methods.



```
syntax = "proto3";

service ChattingService {
    rpc GetInformation (NullMsg) returns (Message) {}
    rpc SendMessage (Message) returns (NullMsg) {}
}

message Message {
    string time = 1;
    string sender = 2;
    string msg = 3;
}

message NullMsg {

}
```



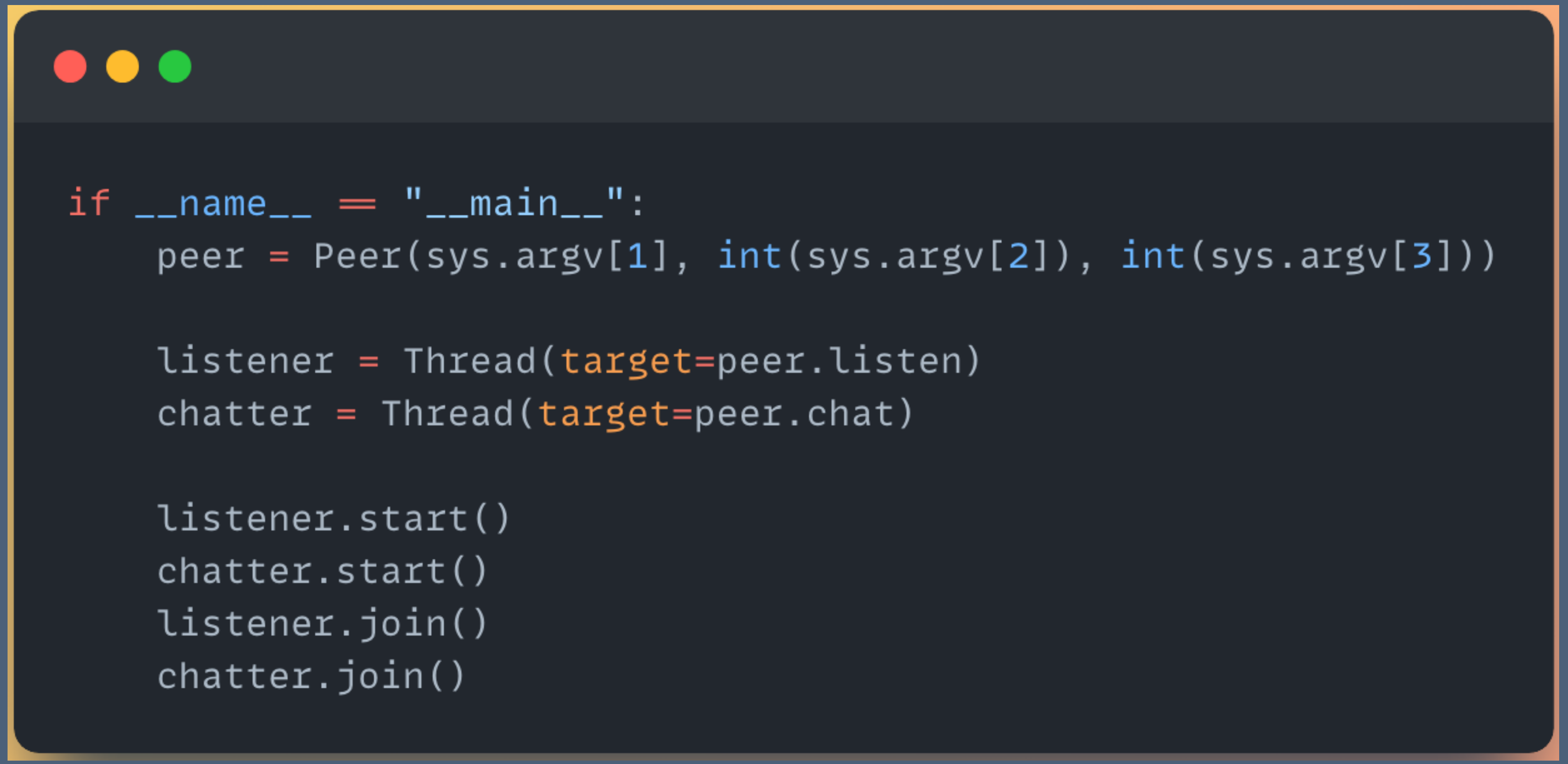
```
class ChattingServiceServicer(chat_pb2_grpc.ChattingServiceServicer):
    def __init__(self, name: str) → None:
        super()
        self.name = name

    def SendMessage(self, request, context):
        if(request.msg):
            print(f'\u001b[1;36m[{request.time}] {request.sender}>\u001b[0m {request.msg}')
        return chat_pb2.NullMsg()

    def GetInformation(self, request, context):
        return chat_pb2.Message(time=datetime.now().strftime(TIME_FORMAT), sender=self.name, msg='OK')
```

Challenge: Peer-to-Peer Chatting - Chatter

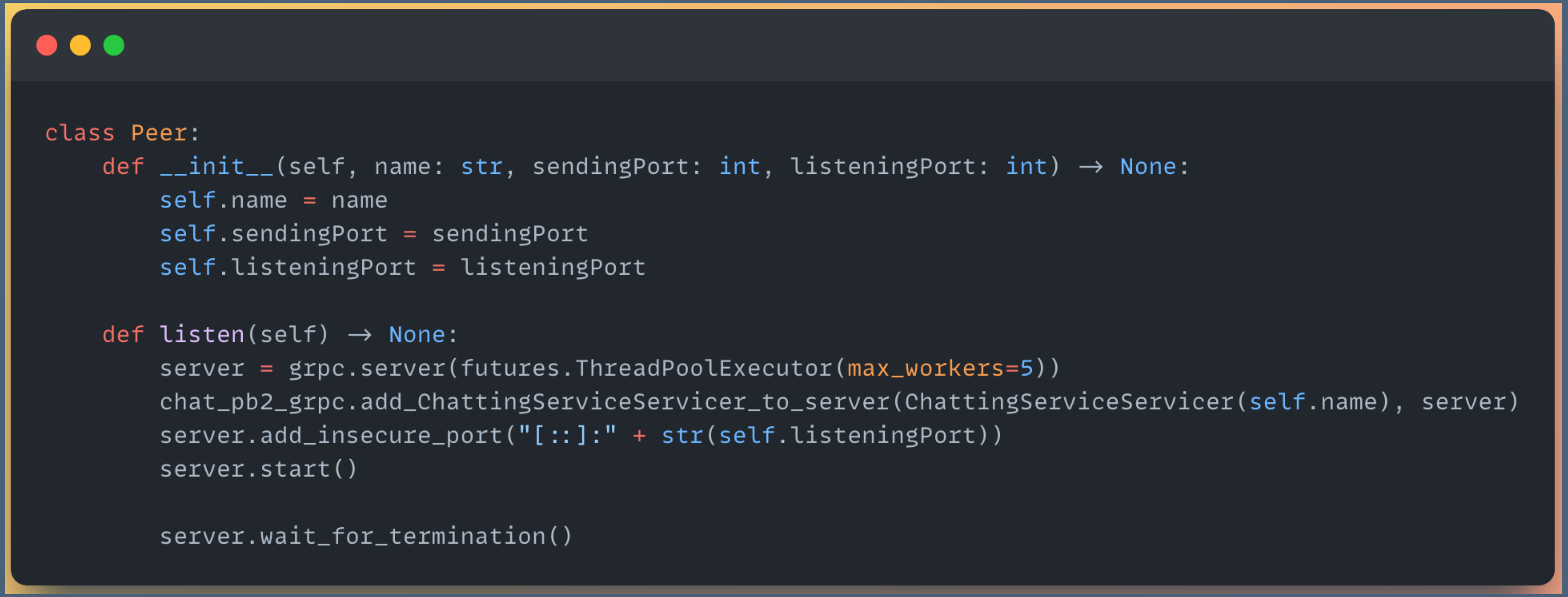
Inside the chatter main, we create a **Peer** object, with a **name**, a **sendingPort** and a **listeningPort** given as an argument by the user.

Two threads are then started for the methods **listen** and **chat**.

The **listen** method listens for another peer to connect.



```python
if __name__ == "__main__":
    peer = Peer(sys.argv[1], int(sys.argv[2]), int(sys.argv[3]))

    listener = Thread(target=peer.listen)
    chatter = Thread(target=peer.chat)

    listener.start()
    chatter.start()
    listener.join()
    chatter.join()
```



```python
class Peer:
    def __init__(self, name: str, sendingPort: int, listeningPort: int) -> None:
        self.name = name
        self.sendingPort = sendingPort
        self.listeningPort = listeningPort

    def listen(self) -> None:
        server = grpc.server(futures.ThreadPoolExecutor(max_workers=5))
        chat_pb2_grpc.add_ChattingServiceServicer_to_server(ChattingServiceServicer(self.name), server)
        server.add_insecure_port("[::]:" + str(self.listeningPort))
        server.start()

        server.wait_for_termination()
```

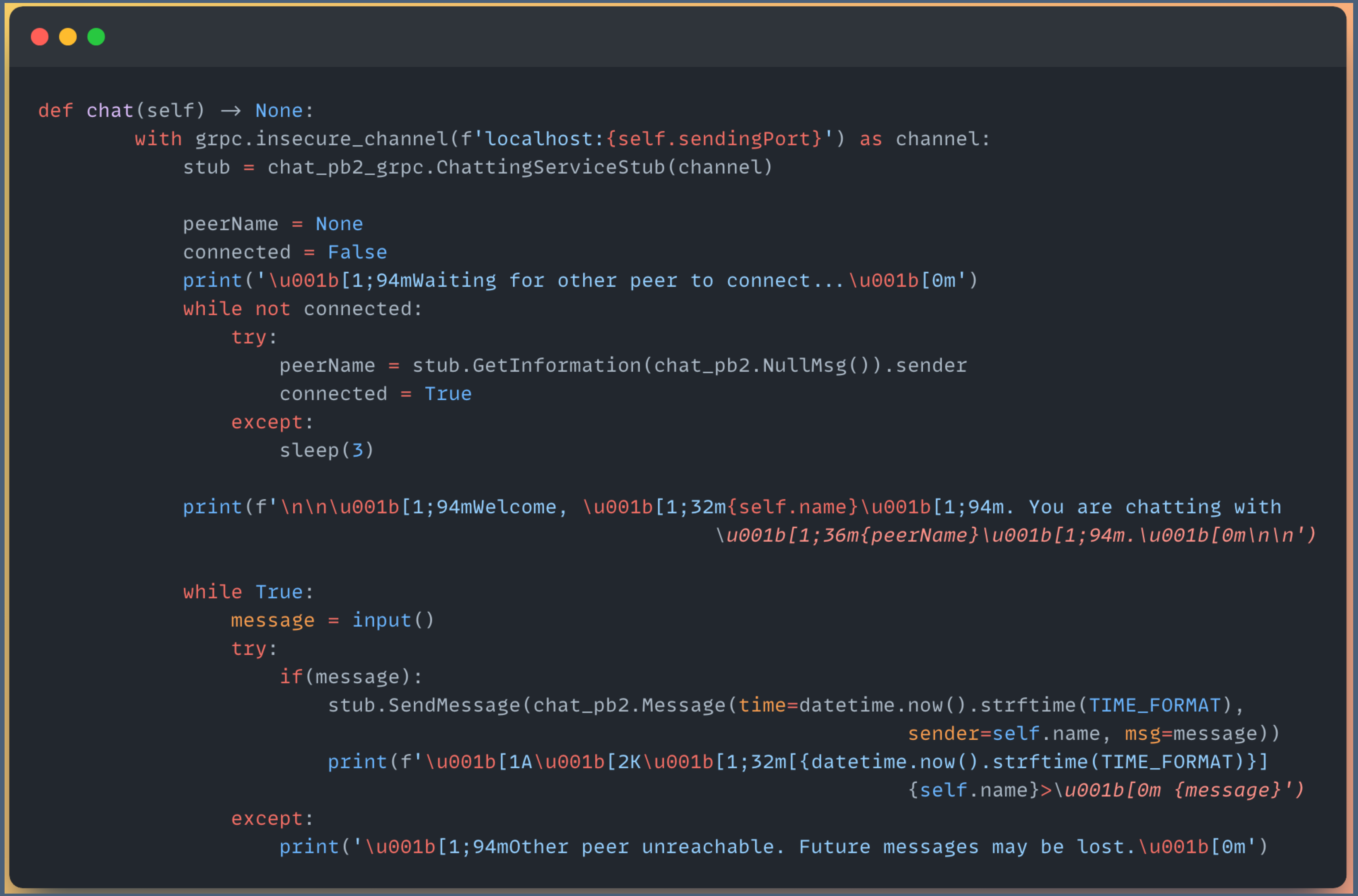
Challenge: Peer-to-Peer Chatting - Chatter

The **chat** method waits for another peer to connect and chat with.

When a connection is established, the name of the other peer is printed on the console and you can start writing messages.

Every message sent and received is printed on the console with the timestamp related to it.

When the other peer is unreachable, an error message is printed.



```python
def chat(self) → None:
        with grpc.insecure_channel(f'localhost:{self.sendingPort}') as channel:
            stub = chat_pb2_grpc.ChattingServiceStub(channel)

            peerName = None
            connected = False
            print('\u001b[1;94mWaiting for other peer to connect...\u001b[0m')
            while not connected:
                try:
                    peerName = stub.GetInformation(chat_pb2.NullMsg()).sender
                    connected = True
                except:
                    sleep(3)

            print(f'\n\n\u001b[1;94mWelcome, \u001b[1;32m{self.name}\u001b[1;94m. You are chatting with
                                                        \u001b[1;36m{peerName}\u001b[1;94m.\u001b[0m\n\n')

            while True:
                message = input()
                try:
                    if(message):
                        stub.SendMessage(chat_pb2.Message(time=datetime.now().strftime(TIME_FORMAT),
                                                                        sender=self.name, msg=message))
                        print(f'\u001b[1A\u001b[2K\u001b[1;32m[{datetime.now().strftime(TIME_FORMAT)}]
                                                                        {self.name}>\u001b[0m {message}')
                except:
                    print('\u001b[1;94mOther peer unreachable. Future messages may be lost.\u001b[0m')
```

Challenge: Peer-to-Peer Chatting - Usage

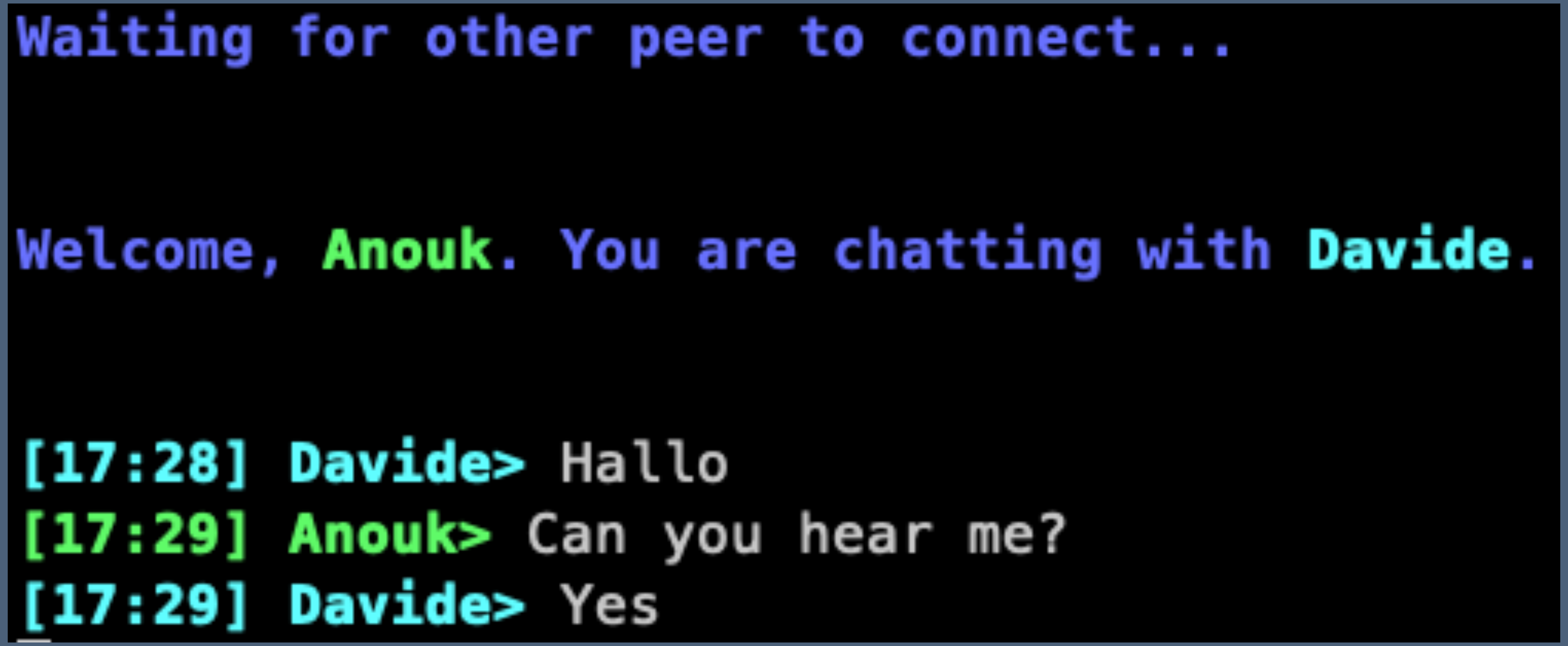
To establish the connection between the peers, you have to run both scripts with matching ports.

In this usage example, the **sendingPort** for *Anouk* was 8000, and the **listeningPort** was 8001.

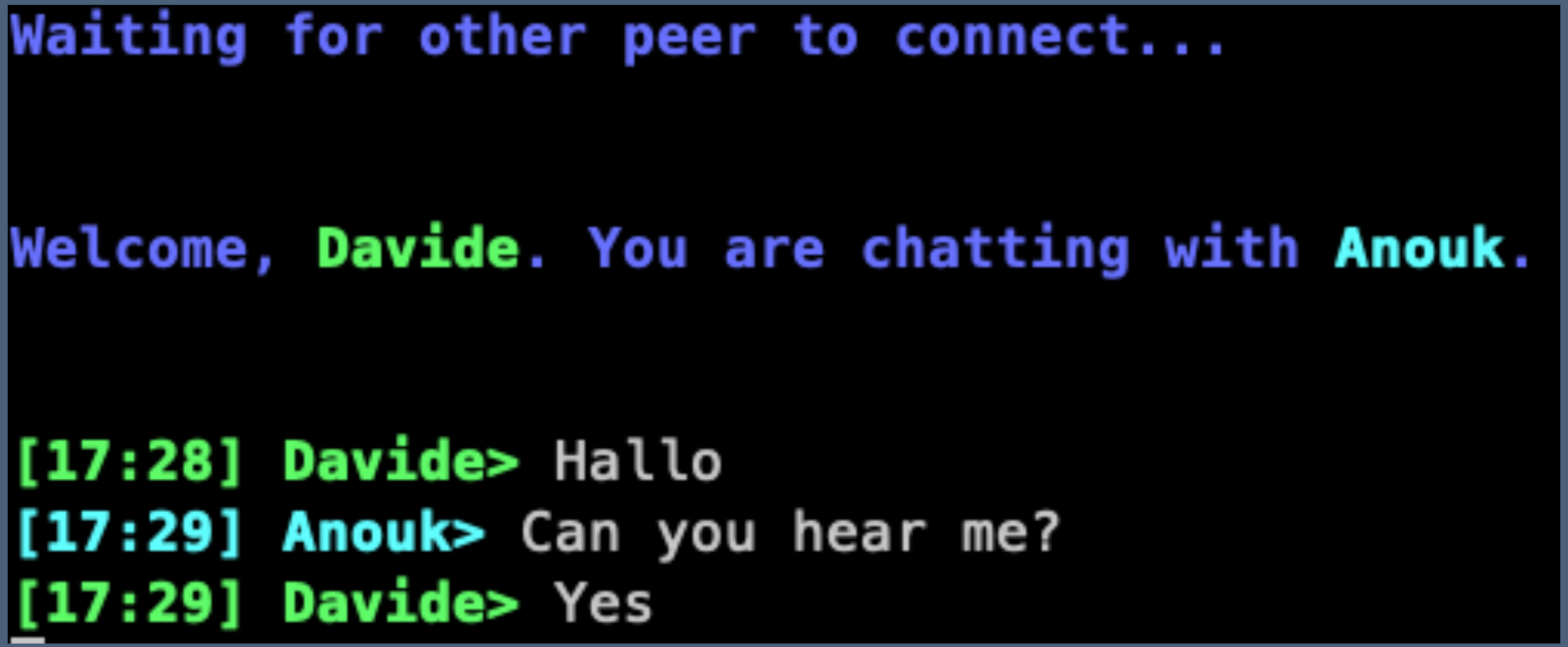
The ports were of course inverted for *Davide,* where the **sendingPort** was 8001, and the **listeningPort** was 8000.

After the connection is established, the peers can chat freely and the messages will be printed on both peers console.

If one peer closes the console, the other will print an error message stating that the other peer is unreachable.



```
Waiting for other peer to connect...


Welcome, Anouk. You are chatting with Davide.


[17:28] Davide> Hallo
[17:29] Anouk> Can you hear me?
[17:29] Davide> Yes
```



```
Waiting for other peer to connect...


Welcome, Davide. You are chatting with Anouk.


[17:28] Davide> Hallo
[17:29] Anouk> Can you hear me?
[17:29] Davide> Yes
```

Challenge: Peer-to-Peer Chatting - Coupling Issues

When we want to establish the connection between two peers, a peer must know which port the other peer is listening on, and viceversa.

This causes **space coupling**, that could be avoided by using a naming server system which would query the port associated with the name only before starting the communication.

Another issue is related to **time coupling**, where if one of the peers is unreachable, all the sent messages by the other peer are lost.

This cannot be avoided, given the fact that we are not storing the messages anywhere else.

Thank you!