gRPC

Systems Analysis & Design

Learning Objectives

By the end of this session, you will have acquired the following information:

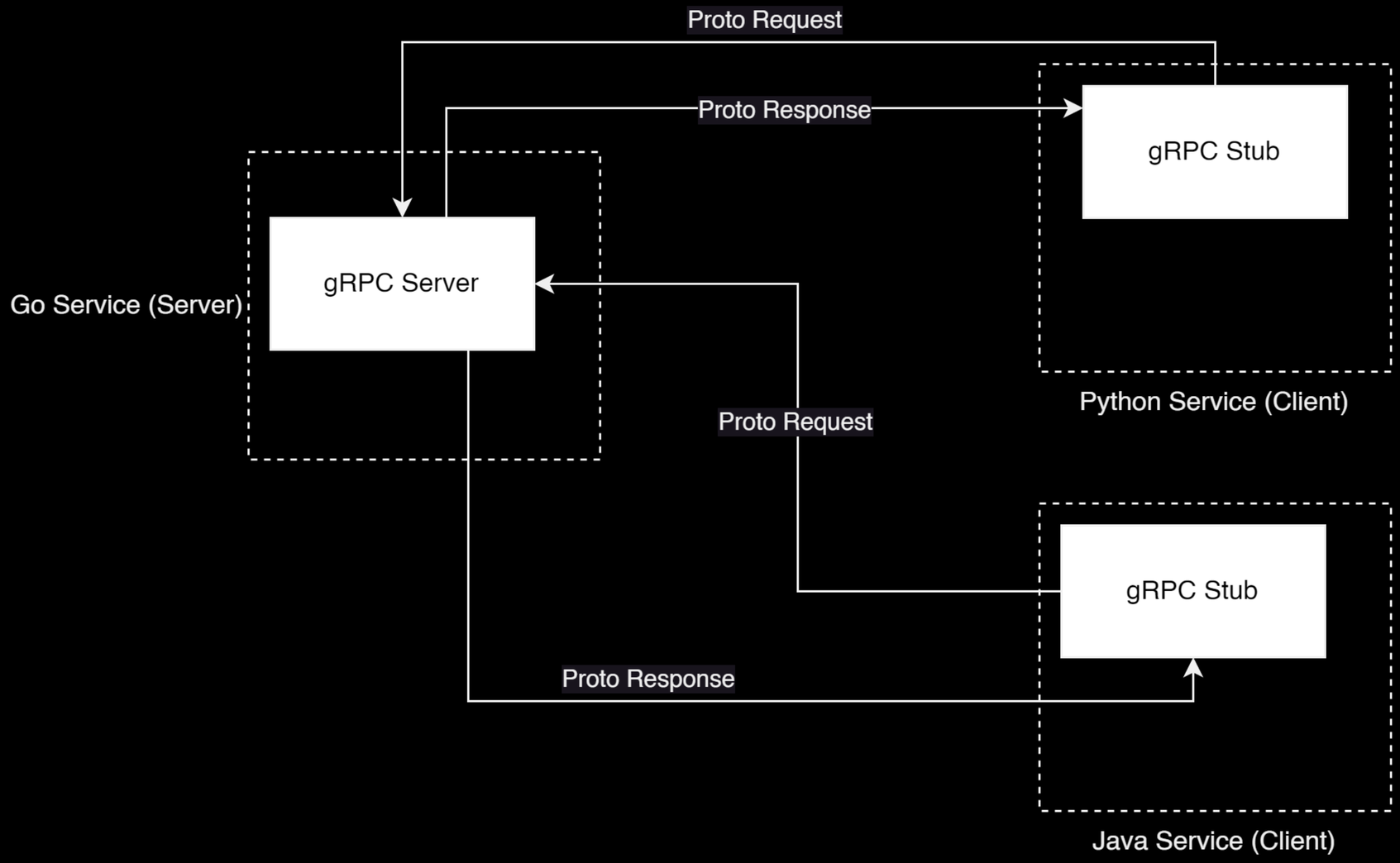
- gRPC
- Protocol Buffers

What is gRPC?

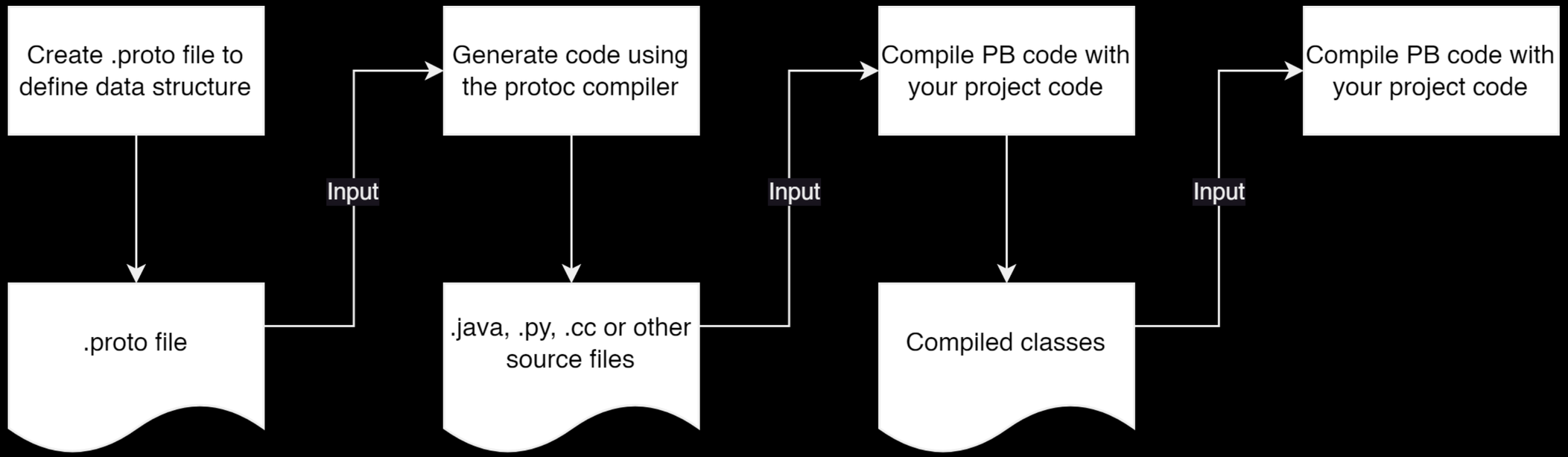
- gRPC is a high-performance Remote Procedure Call (RPC) framework that operates over HTTP/2.
- With gRPC, a client application can call a method on a server application on a different machine as if it were a local object.
- The interfaces that can be called remotely, along with their parameters and return types, are specified.
- The server implements this interface and runs a gRPC server to handle client calls.
- The client, on the other hand, has a stub providing the same methods as the server.

By default, gRPC uses Protocol Buffers:

Google's open source mechanism for serializing structured data.



Protocol Buffers Workflow



- The process begins with defining the data structure for serialization.
- This is done in a .proto file.
- The data is structured into messages.
- Each message contains name-value pairs, referred to as fields.

```
// .proto file
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

- The protoc compiler generates data access classes in the chosen language.
- These classes provide simple accessors for each field.
- They also provide methods to serialize and parse the entire structure to and from raw bytes.

```
import <Java-Package>.Person

Person alireza = Person.newBuilder()
        .setId(1234)
        .setName("Alireza Aghamohammadi")
        .setEmail("al.aghamohammadi@gmail.com")
        .build();
```

- gRPC is designed around the concept of defining a service.
- gRPC services are defined in standard proto files.
- The service outlines the methods that can be invoked remotely.
- The methods include their parameters and return types.

```
// .proto file

// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}
// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Four Kinds of Service Methods

Unary RPCs

- The client sends a single request and receives a single response, similar to a normal function call.

Server streaming RPCs

- The client sends a request and receives a stream of messages until there are no more messages. Message ordering is guaranteed.

Client streaming RPCs

- The client sends a sequence of messages using a stream and waits for the server to read them and return a response. Message ordering is guaranteed.

Bidirectional streaming RPCs

- Both sides send a sequence of messages using a read-write stream. The streams operate independently, allowing clients and servers to read and write in any order. Message ordering is preserved in each stream.

```
//  Four Kinds of Service Method

// Unary RPCs
rpc SayHello(HelloRequest) returns (HelloResponse);

// Server streaming RPCs
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);

// Client streaming RPCs
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);

// Bidirectional streaming RPCs
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

Server Implementation

This line sends the reply to the client. If this were a streaming call, you could call `onNext` multiple times to send multiple responses.

```
private class GreeterImpl extends GreeterGrpc.GreeterImplBase {

  @Override
  public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
    HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.getName()).build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
  }

}
```

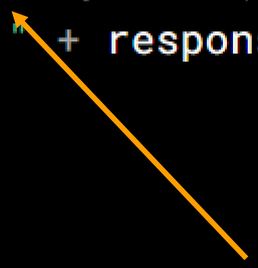
This line signals that the server has finished sending responses. After calling this method, the server can't send any more responses for this call.

- Start a gRPC server for client use.
- Use `forPort()` to set the port.
- Create `GreeterImp` instance and add it to the service with `addService()`.
- Use `start()` to launch the RPC server.

```
this(ServerBuilder.forPort(port), port);
server = serverBuilder.addService(new GreeterImp()).build();
server.start();
```

Client Implementation

```
public void greet(String name) {
  HelloRequest request = HelloRequest.newBuilder().setName(name).build();
  HelloReply response;
  response = blockingStub.sayHello(request);
  logger.info("Greeting: " + response.getMessage());
}
```



This line calls the `sayHello` method on the `blockingStub` object, passing in the request object created earlier.

- A Channel represents a communication line to a gRPC server. You can create a channel using the `ManagedChannelBuilder`.
- Once you have a Channel, you can use it to create a blocking stub. The stub is used to call methods on the server. The class and method used to create the stub depend on your service definition.

```
import <Java-Package>.ManagedChannel;
import <Java-Package>.GreeterGrpc.GreeterBlockingStub

channel = ManagedChannelBuilder.forAddress(HOST, PORT)
            .usePlaintext()
            .build();
blockingStub = GreeterGrpc.newBlockingStub(channel);
```

Further Resources

- Introduction to gRPC in Java