What is RPC

RPC stands for remote procedure call . As the name suggests, it lets you call a function in on a different process/server/backend and get back a response from it.

Why remote procedure call?

This is how we've made our backends talk to each other until now.

We send out an http request , get back a response.

There are a few flaws in this approach

1. No types. You don't know what is the shape of the data you will get back. You might be able to share types between 2 Node.js backends somehow, but if the other backend is in Rust , then you cant get back the types from it
2. We use JSON to serialize and deserialize data
3. We have to know what axios is , or what fetch is . We need to understand HTTP and how to call it
4. Not language agnostic at all. We have to use a different library in Java, go, rust to send an http request to the server

Let's try doing a quick HTTP request to https://sum-server.100xdevs.com/todos from a Node.js server

Implementing a dump RPC

What if we could auto generate a client that a Node.js server can use and call a function on another service without worrying about the underlying axios / fetch call

rpc.ts (autogenerated)

```
import axios from "axios";
interface Todo {
  id: string;
  title: string;
  description: string;
  completed: boolean;
}

async function getTodos(): Promise<Todo[]> {
  const response = await axios.get(" https://sum-server.100xdevs.com/todos");

  let todos = response.data.todos;
  return todos;
}
```

index.ts

```
import { getTodos } from "./rpc"
const todos = await getTodos();
console.log(todos);
```

Benefits

1. Better types - The `getTodos` function has an associated type of the data being returned.
2. We are still using json, but we will fix that soon (json is slow)
3. We dont need to use axios anymore, all we have to do is call a function
4. If we can get `autogenerated` code for all languages (go, rust), then this becomes language agnostice

Sample clients in other languages

Rust

```rust
use reqwest::Error; // Add reqwest = { version = "0.11", features = ["blocking", "jso

#[derive(Debug)]
struct Todo {
    id: String,
    title: String,
    description: String,
    completed: bool,
}

async fn get_todos() -> Result<Vec<Todo>, Error> {
    let response = reqwest::get("https://sum-server.100xdevs.com/todos").await

    let todos: Vec<Todo> = response.json().await?;
    Ok(todos)
}
```

Go

```go
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
```

```go
    "net/http"
)

type Todo struct {
    ID          string `json:"id"`
    Title       string `json:"title"`
    Description string `json:"description"`
    Completed   bool   `json:"completed"`
}

func getTodos() ([]Todo, error) {
    response, err := http.Get("https://sum-server.100xdevs.com/todos")
    if err != nil {
        return nil, err
    }
    defer response.Body.Close()

    body, err := ioutil.ReadAll(response.Body)
    if err != nil {
        return nil, err
    }

    var todos struct {
        Todos []Todo `json:"todos"`
    }
    if err := json.Unmarshal(body, &todos); err != nil {
        return nil, err
    }

    return todos.Todos, nil
}
```

Some common RPC protocols

JSON RPC

Used by solana/eth when talking to the blockchain.

Creating a JSON RPC Server

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

// Parse JSON bodies
app.use(bodyParser.json());

// Define a sample method
function add(a, b) {
  return a + b;
}

// Handle JSON-RPC requests
app.post('/rpc', (req, res) => {
  const { jsonrpc, method, params, id } = req.body;

  if (jsonrpc !== '2.0' || !method || !Array.isArray(params)) {
    res.status(400).json({ jsonrpc: '2.0', error: { code: -32600, message: 'Invalid
    return;
  }

  // Execute the method
  let result;
```

```
    switch (method) {
      case 'add':
        result = add(params[0], params[1]);
        break;
      default:
        res.status(404).json({ jsonrpc: '2.0', error: { code: -32601, message: 'Metho
        return;
    }

    // Send back the response
    res.json({ jsonrpc: '2.0', result, id });
  });

  // Start the server
  app.listen(port, () => {
    console.log(`JSON-RPC server listening at http://localhost:${port}`);
  });
```

Try hitting the server with the following body

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "add",
  "params": [
    1, 2
  ]
}
```

GRPC

gRPC is an open-source remote procedure call (RPC) framework developed by Google. It allows you to define services and messages using Protocol Buffers, a language-agnostic data serialization format, and then generate client and server code in various programming languages

TRPC

tRPC gives u types on the frontend and backend if u have a full stack app in js.

We have a video from cohort 1 on it. Not used too much

Proto buffs

Ref https://protobuf.dev/

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, JSON.

The protocol buffers are where we define our service definitions and messages. This will be like a contract or common interface between the client and server on what to expect from each other; the methods, types, and returns of what each operation would bear.

1. **Schema Definition Language:** Protocol Buffers use a schema definition language (`.proto` files) to define the structure of data. This language allows you to specify message types, fields, enums, and services.
2. **Binary Serialization:** Protocol Buffers serialize data into a binary format, which is more compact and efficient compared to text-based formats like XML and JSON.
3. **Language Support and Code Generation:** Protocol Buffers support code generation for a wide range of programming languages, including C++, Java, Go, Python, JavaScript, Ruby, and more. Protocol Buffers come with tools (e.g., `protoc`) that generate code in various programming languages based on your `.proto` files.

Let's create a simple Proto file

message.proto

```
syntax = "proto3";

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
```

```
}

service PersonService {
  // Add a person to the address book.
  rpc AddPerson(Person) returns (Person);

  // Get a person from their name
  rpc GetPersonByName(GetPersonByNameRequest) returns (Person);
}

message GetPersonByNameRequest {
  string name = 1;
}
```

There are a few things to unpack here -

- `message` - A message that can be encoded/decoded/transferred
- types
  - string
  - int32
- service - Describes what all `rpc` methods you support
- **Field numbers**

  In Protocol Buffers, each field within a message type is assigned a unique numerical identifier called a tag or field number. These field numbers serve several purposes:

  1. **Efficient Encoding:** Field numbers are used during serialization and deserialization to efficiently encode and decode the data. Instead of including field names in the serialized data, Protocol Buffers use field numbers, which are typically more compact and faster to process.
  2. **Backward Compatibility:** Field numbers are stable identifiers that remain consistent even if you add, remove, or reorder fields within a message type. This means that old serialized data can still be

decoded correctly by newer versions of your software, even if the message type has changed.

3. **Language Independence:** Field numbers provide a language-independent way to refer to fields within a message type. Regardless of the programming language used to generate the code, the field numbers remain the same, ensuring interoperability between different implementations.

Seializing and deserializing data (easy)

https://www.protobufpal.com/

Serializing and deserializing data (hard)

- Create a.proto
- npm init -y
- npm i protobufjs

```
const protobuf = require('protobufjs');

// Load the Protocol Buffers schema
protobuf.load('a.proto')
  .then(root => {
    // Obtain the Person message type
    const Person = root.lookupType('Person');

    // Create a new Person instance
    const person = { name: "Alice", age: 30 };

    // Serialize Person to a buffer
    const buffer = Person.encode(person).finish();

    // Write buffer to a file
```

```
  require('fs').writeFileSync('person.bin', buffer);

  console.log('Person serialized and saved to person.bin');

  // Read the buffer from file
  const data = require('fs').readFileSync('person.bin');

  // Deserialize buffer back to a Person object
  const deserializedPerson = Person.decode(data);

  console.log('Person deserialized from person.bin:', deserializedPerson);
 })
 .catch(console.error);
```

- Check the size of person.bin
- Create a `person.json` file and check it's size

```
{
  name: "Alice",
  age: 31
}
```

Types in pbf

Protobuffs give you access to a lot of types/enums/message types

1. **Scalar Types:**
   - **int32, int64, uint32, uint64:** Signed and unsigned integers of various sizes.
   - **float, double:** Floating-point numbers.
   - **bool:** Boolean values (true or false).
   - **string:** Unicode text strings.
   - **bytes:** Arbitrary binary data.

```
syntax = "proto3";

// Define a message type representing an address.
message Address {
  string street = 1;
  string city = 2;
  string state = 3;
  string zip = 4;
}

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
  Address address = 3;
}
```

2. **Message Types:**

- Message types allow you to define structured data with nested fields. They can contain scalar types, other message types, or repeated fields (arrays).
- You define message types using the **message** keyword followed by the name of the message type and its fields.

```
message Person {
  string name = 1;
  int32 age = 2;
  repeated string phone_numbers = 3;
}
```

1. **Enum Types:**
    - Enum types define a set of named constant values.
    - You define enum types using the **enum** keyword followed by the name of the enum type and its values.

```
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}
```

1. Maps

```
message MapMessage {
  map<string, int32> id_to_age = 1;
}
```

Trying a more complicated proto

```
syntax = "proto3";

// Define an enum representing the type of phone numbers.
```

```
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

// Define a message type representing a phone number.
message PhoneNumber {
  string number = 1;
  PhoneType type = 2;
}

// Define a message type representing an address.
message Address {
  string street = 1;
  string city = 2;
  string state = 3;
  string zip = 4;
}

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
  repeated PhoneNumber phone_numbers = 3;
  Address address = 4;
}
```

Try using it in

Implementing services

This is what our proto file looks like

```
syntax = "proto3";

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
}

service AddressBookService {
  // Add a person to the address book.
  rpc AddPerson(Person) returns (Person);

  // Get a person from their name
  rpc GetPersonByName(string) returns (Person);
}
```

There is a `service` section which describes all the services our server would support. But `protobufs` are not used for service creation.

While the concept of a service exists in Protocol Buffers, it's up to the developer to choose how to implement the RPC communication. gRPC is one such RPC framework that uses Protocol Buffers for defining services and messages, but other frameworks or custom implementations can also be used.

Implementing services using grpc

Ref - https://grpc.io/docs/languages/node/basics/

- Initialize node.js project

```
npm init -y
```

- Initialize typescript

```
npx tsc --init
```

- Add dependencies

```
npm i @grpc/grpc-js @grpc/proto-loader
```

- Create a.proto file

```
syntax = "proto3";

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
}

service AddressBookService {
  // Add a person to the address book.
  rpc AddPerson(Person) returns (Person);

  // Get a person from their name
  rpc GetPersonByName(GetPersonByNameRequest) returns (Person);
}

message GetPersonByNameRequest {
```

```
  string name = 1;
}
```

- Create index.ts

```
import path from 'path';
import * as grpc from '@grpc/grpc-js';
import  { GrpcObject, ServiceClientConstructor } from "@grpc/grpc-js"
import * as protoLoader from '@grpc/proto-loader';

const packageDefinition = protoLoader.loadSync(path.join(__dirname, './a.prc

const personProto = grpc.loadPackageDefinition(packageDefinition);

const PERSONS = [
   {
      name: "harkirat",
      age: 45
   },
   {
     name: "raman",
     age: 45
   },
];

//@ts-ignore
function addPerson(call, callback) {
  console.log(call)
   let person = {
     name: call.request.name,
     age: call.request.age
   }
   PERSONS.push(person);
   callback(null, person)
}

const server = new grpc.Server();

server.addService((personProto.AddressBookService as ServiceClientConstruc
server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () =>
```

```
    server.start();
});
```

- Run it

```
tsc -b
node index.js
```

- Test it in postman
  - File ⇒ New ⇒ GRPC
  - Import the proto file in GRPC
  - Send a request (select URL as grpc://localhost:50051)

Adding types

Ref https://github.com/grpc/proposal/blob/master/L70-node-proto-loader-type-generator.md

Ref https://www.npmjs.com/package/@grpc/proto-loader

Generate types

```
./node_modules/@grpc/proto-loader/build/bin/proto-loader-gen-types.js --
```

Update the code

```
import path from 'path';
import * as grpc from '@grpc/grpc-js';
import  { GrpcObject, ServiceClientConstructor } from "@grpc/grpc-js"
import * as protoLoader from '@grpc/proto-loader';
import { ProtoGrpcType } from './proto/a';
import { AddressBookServiceHandlers } from './proto/AddressBookService';
import { Status } from '@grpc/grpc-js/build/src/constants';

const packageDefinition = protoLoader.loadSync(path.join(__dirname, './a.prc

const personProto = (grpc.loadPackageDefinition(packageDefinition) as unkno

const PERSONS = [
  {
    name: "harkirat",
    age: 45
```

```
    },
    {
      name: "raman",
      age: 45
    },
];

const handler: AddressBookServiceHandlers =  {
  AddPerson: (call, callback) => {
    let person = {
      name: call.request.name,
      age: call.request.age
    }
    PERSONS.push(person);
    callback(null, person)
  },
  GetPersonByName: (call, callback) => {
    let person = PERSONS.find(x => x.name === call.request.name);
    if (person) {
      callback(null, person)
    } else {
      callback({
        code: Status.NOT_FOUND,
        details: "not found"
      }, null);
    }
  }
}


const server = new grpc.Server();

server.addService((personProto.AddressBookService).service, handler);
server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () =>
    server.start();
});
```