(Part 4)

API Design

Software System Design

Spring 2024 @ Ali Madooei

Remote Procedure Call (RPC) APIs

- RPC predates REST, GraphQL, and the modern web!
- RPC became common in the 1970s and 1980 in computer networks and distributed systems.

RPC is a Request–Response Protocol

- An RPC is initiated by the client, sending a request message to a known remote server to execute a specific procedure with supplied parameters.
- Afterwards, the remote server sends a response back to the client, allowing the application to continue its process.
- Typically, the client is blocked while the server processes the call, meaning it waits until the server finishes before resuming execution.

Local Call vs. Remote Call

- A local call refers to a function call within the same process as the caller. This means they share the same machine and memory space.
- In contrast, a remote call refers to a function call in a different process than the caller. They are typically not on the same machine and communicate over a network instead.
- RPC enables you to make a remote call as if it were a local call.

Interface Definition Language (IDL)

- Begin by defining your API in an Interface Definition Language (IDL) file.
- IDL is a generic term for a language that allows a program or object written in one language to communicate with another program written in a different language.
- Typically, IDLs are used to describe data types and interfaces in a language-independent manner.

Example:

```
service OrderService {
  rpc PlaceOrder(OrderRequest) returns (OrderResponse) {}
}
```

Client and Server Stubs

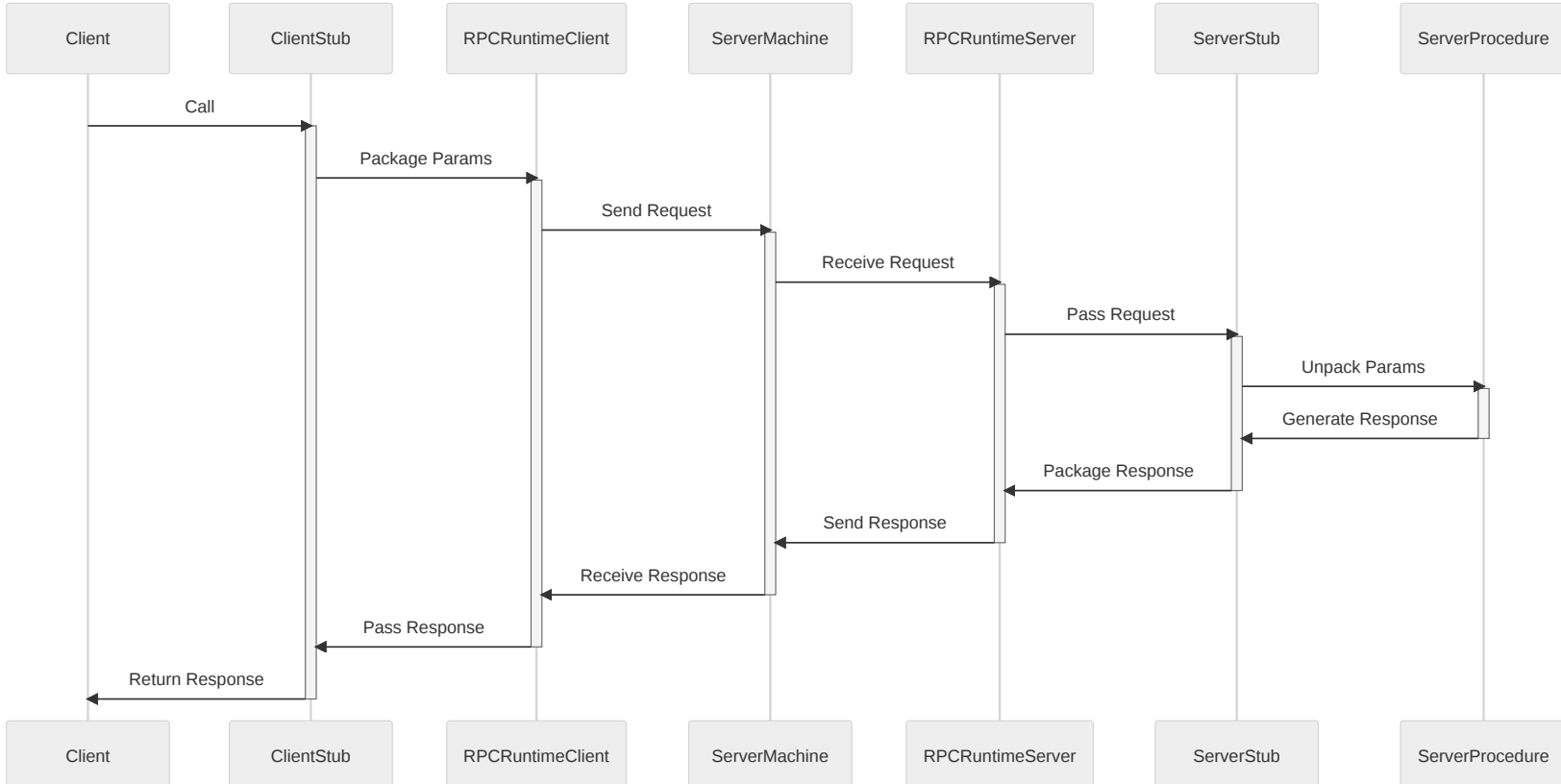
- First, utilize a code generator to create client and server code from the IDL file.

This generator will produce client and server stubs.

- These stubs, which are the client-side and server-side representations of the service, know how to transmit requests to the server and receive responses.

How does RPC work?

- The client starts by calling the client stub. This call is a local procedure and includes the necessary parameters or data.
- The client stub then packages these parameters into a message. Using the RPC runtime, it issues a system call to send the request to the server.
- Upon receiving the request, the server machine uses the RPC runtime to pass the request to the server stub.
- The server stub unpacks the parameters or data from the message and calls the server procedure to process the request and generate a response.
- Finally, the response retraces the steps in reverse to reach the client.


Client
ClientStub
RPCRuntimeClient
ServerMachine
RPCRuntimeServer
ServerStub
ServerProcedure
Call
Package Params
Send Request
Receive Request
Pass Request
Unpack Params
Generate Response
Package Response
Send Response
Receive Response
Pass Response
Return Response
Client
ClientStub
RPCRuntimeClient
ServerMachine
RPCRuntimeServer
ServerStub
ServerProcedure

RPC Shortcomings

- **Tight coupling to the underlying system.** The client and server are closely linked, making RPC more suitable for internal APIs rather than external ones.
- **Low discoverability.** There’s no method to introspect the API or to understand which function to call based on its requests.
- **Difficult to debug.** The multiple layers of abstraction can make it challenging to comprehend what’s happening under the hood.

RPC vs REST

- **Communication:** RPC is procedure-oriented and invokes specific functions, while REST is resource-based and manipulates resources using HTTP methods.
- **Statelessness:** REST is stateless, with each request containing all necessary information, while RPC can maintain state between requests.
- **Coupling:** RPC often results in tighter coupling due to the necessity to know the available procedures, while REST offers looser coupling.

RPC vs GraphQL

- **Data Retrieval:** GraphQL allows clients to specify the data structure, while RPC invokes specific procedures.
- **Flexibility:** GraphQL offers more flexibility in fetching data, while RPC is more rigid and procedure-oriented.
- **Coupling:** RPC tends to have tighter coupling due to the need to know available procedures, while GraphQL has looser coupling.

When to Use RPC

- **Procedure-Oriented Applications:** When the interaction is more about invoking functions or procedures rather than manipulating resources.
- **Strict Contract:** When a strict contract between client and server is beneficial, as the client needs to know the available procedures.
- **Efficiency & Performance:** When the overhead of HTTP is undesirable and alternative transport protocols can optimize performance.
- **Network Constraints:** In environments with network constraints where a lightweight protocol is essential.

What is gRPC?

- gRPC is an open-source remote procedure call framework that Google released in 2016.
- It is a rewrite of Google’s internal RPC infrastructure, which they used for many years.
- gRPC can be thought of as an implementation of RPC.
- Numerous organizations, including Google, Netflix, Slack, Square, CoreOS, Docker, CockroachDB, and Cisco, use gRPC for their RPC needs.

Why use gRPC?

gRPC is the preferred API design for connecting a large number of microservices in a distributed system, both within and across data centers.

- A thriving ecosystem of tools, libraries, and plugins. It makes it easy to develop and maintain production quality and type-safe APIs that scale well.
- By supporting multiple programming languages, the client and server (and any other components) can be written in different languages.
- gRPC is known for high-performance and low-latency.

Implementing gRPC: Example (Server Side)

```
const grpc = require('grpc');
const booksProto = grpc.load('books.proto');

const server = new grpc.Server();
const books = [{ id: 1, title: 'Book One', author: 'Author One' }];

server.addService(booksProto.books.BookService.service, {
  list: (_, callback) => callback(null, books),
});

server.bind('127.0.0.1:50051', grpc.ServerCredentials.createInsecure());
server.start();
```

Using gRPC: Example (Client Side)

```
const grpc = require('grpc');
const booksProto = grpc.load('books.proto');

const client = new grpc.Client('localhost:50051', grpc.credentials.createInsecure());

client.list({}, (error, books) => {
  if (!error) console.log('Book List', books);
  else console.error(error);
});
```

When to Use gRPC?

- For internal microservices communication where efficiency and low latency are critical.
- When strong type safety and contract-first development are required (gRPC uses Protocol Buffers for serialization).
- For scenarios requiring streaming capabilities (e.g., real-time data transfer, bidirectional communication).
- In systems where network bandwidth is a concern, as gRPC is more efficient in data serialization compared to JSON.

gRPC API: Activity

- Which of the three use cases (University Student Hub App, Boom Virtual Meeting Service, To-Do App) would be best suited to employ a gRPC API?

gRPC API: Activity Solution

- **Boom Virtual Meeting Service:** Best suited for gRPC due to its requirements for high-performance, low-latency communication between clients and servers, which is critical for real-time audio and video streaming.
- **Student Hub App:** Not the best fit for gRPC as it does not require the high-performance, low-latency features that gRPC offers. GraphQL would be more suitable for its data querying needs.
- **To-Do App:** Not the best fit for gRPC as it is a simple CRUD application without the need for real-time communication or high-performance requirements. A RESTful API would be more appropriate for its simplicity and ease of use.

Other Notable Implementations of RPC

- **Apache Thrift:** An open-source RPC framework developed at Facebook, which provides a complete stack for creating clients and servers.
- **SOAP:** Simple Object Access Protocol (SOAP) developed by Microsoft that uses XML to encode its calls and HTTP/SMTP as a transport mechanism. It operates on strict standards and specifications, defining an extensive set of rules for structuring messages. It is common in enterprise solutions, financial services, and telecommunication services where data integrity and confidentiality are paramount.
- **tRPC:** A modern RPC framework for TypeScript and Node.js, offering enhanced type safety and developer experience.

Conclusion

- **RESTful APIs:** Stateful, scalable interactions using standard HTTP methods, URIs, and status codes, focusing on web resources.
- **GraphQL APIs:** Flexible, efficient data interactions, allowing precise data retrieval, relying on a strongly typed schema, considering domain, performance, and security in design.
- **RPC-based APIs:** Intuitive for non-CRUD actions, leading to tighter coupling, with modern iterations like tRPC offering enhanced type safety, and the choice among them aligns with project needs and context.