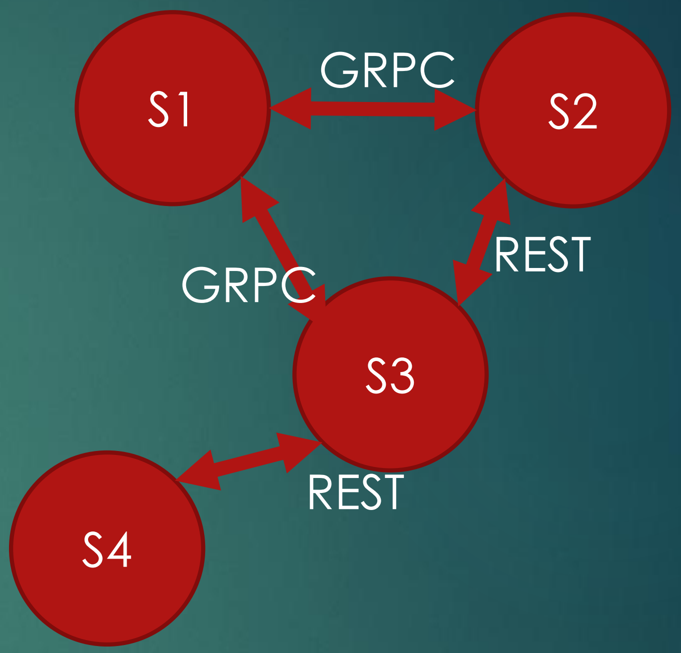
RPC – Remote Procedure Call

K V SUBRAMANIAM

Recall: SOA principle

- Build everything as a service
- Define an interface
- Services call each other using the interface
- Can run on any machine
- Typically use either
  - REST
  - GRPC (RPC)



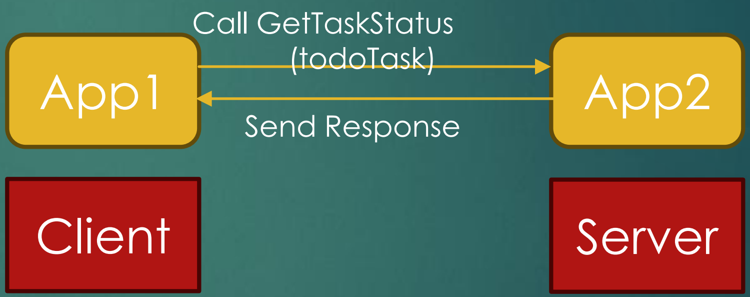
Limitations of REST

- Built on HTTP 1.1
  - Lack of multiplexing
    - Head of line blocking
      - Pipelined, but still has to wait for older request
- No streaming support
- API Versioning
- Using JSON has performance problems
- Lack of typing

- Let us see how GRPC solves these problems

What is RPC?

- Client Server Communication Mechanism

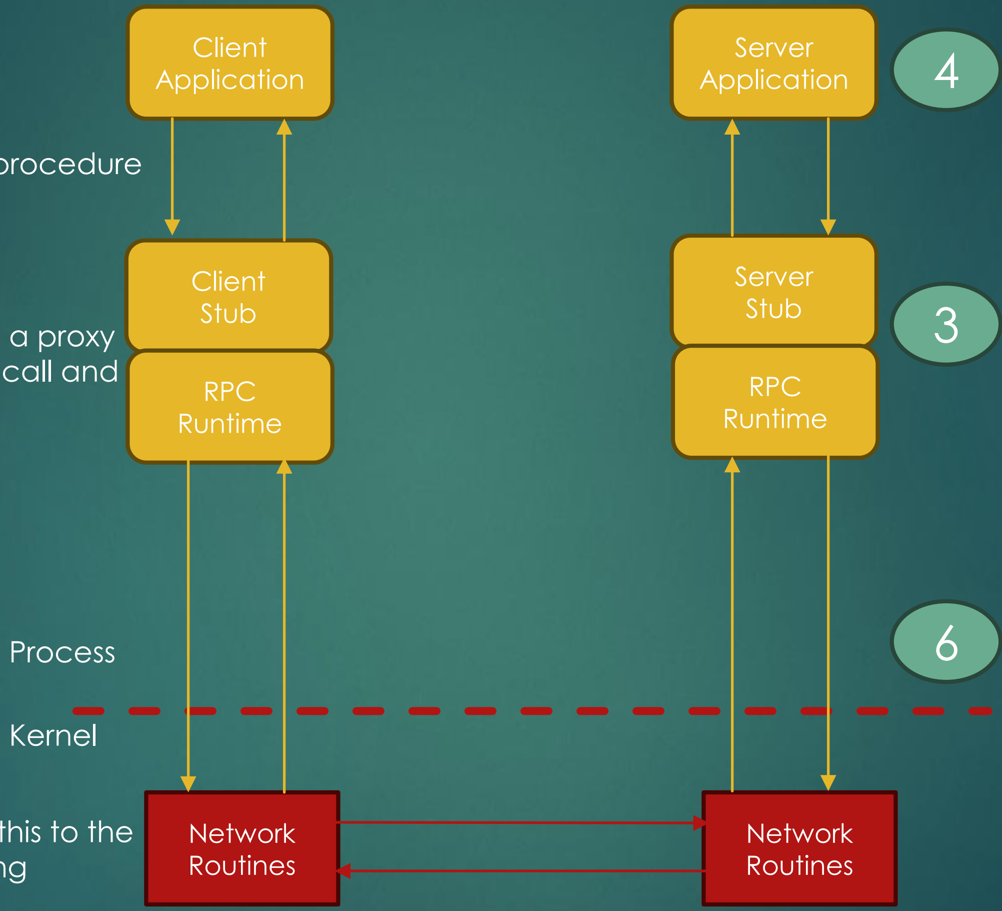


RPC Internals

1 Client calls local procedure on the client stub

2 Client stub acts as a proxy and marshalls the call and the arguments

3 Client stubs sends this to the remote system using TCP/UDP



4 Server stub calls the actual procedure on the server

3 Server stub unmarshalls the call and the arguments from the client

6 Server stub marshalls the response and sends it back to the client

Courtesy: R. Stevens – Unix Network Programming (IPC), 1998

Design Issues

How do we identify the server machine?

Think of a scenario where we have a large number of machines and we need to identify which machine runs the implementation of a particular service.

What is the wire format?

Source and target may have different

- processors – endianness
- OS
- Maybe written in different languages

Fast

Locating the server

Static Configuration

Bind the server to the client statically

Good for starters and small scale

Naming/Directory Service

Server registers end points with a naming service

Clients query the naming service

Load Balancer

When we have more than one server providing the service

Which one to connect to?

Wire Format - requirements

Compactness and efficiency

- Minimal overhead - #bytes used to to represent data must be miminal
- Prefer binary encoding

Platform and Language Neutrality

- Different OS/Machines/Programming languages

Schema definition and Versioning

- Defines structure of fields
- Handle unknown fields

Message Framing

- Where does the message begin and end

Complex Data Types

- Must support simple – int, float char
- Structures and Nested structures
- Arrays/Lists
- Optional fields

Extensibility

- Add new fields

Security Considerations

- Authentication, encryption and integrity

Transport Agnosticism

- Independent of transport protocol

Testability and Debuggability

- Tools for human readable representations for debugging
- Logging and inspection tools

Protocol Buffers

Definition of message format

- Can support multiple data types string, int, arrays.

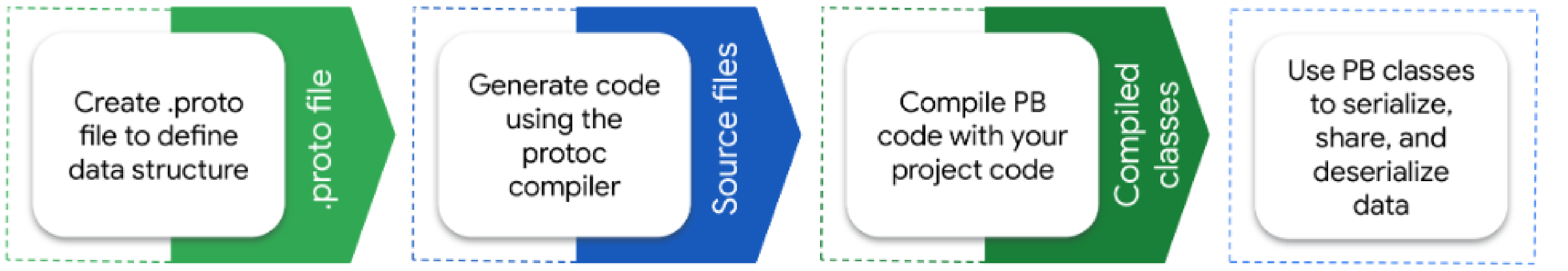
Stub generation

- Both on client and server

Compact wire format

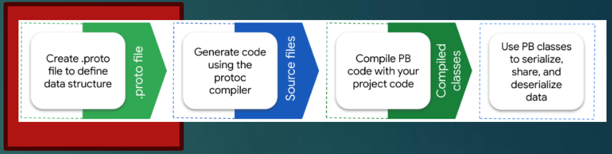
- Uses a binary wire format rather and JSON/XML

Protocol Buffers Workflow


Create .proto file to define data structure
.proto file
Generate code using the protoc compiler
Source files
Compile PB code with your project code
Compiled classes
Use PB classes to serialize, share, and deserialize data

- Define what you want to transfer
- Stubs are generated by the protoc compiler
- Write server and client code and include generated code

Protocol Buffers Working


Create .proto file to define data structure
.proto file
Generate code using the protoc compiler
Source files
Compile PB code with your project code
Compiled classes
Use PB classes to serialize, share, and deserialize data

Define the services

Define the data structures

Define request/response formats

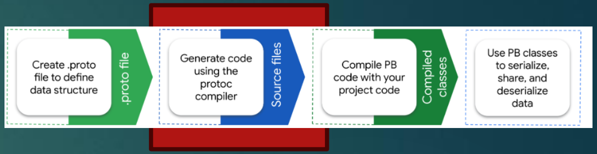
```
syntax = "proto3";
package todo;
service TodoService {
  rpc GetTodoList (Empty) returns (TodoListResponse);
  rpc UpdateTodoStatus (UpdateTodoRequest) returns (UpdateTodoResponse);
}
message Empty {}
message Todo {
  int32 id = 1;
  string title = 2;
  bool completed = 3;
}
message TodoListResponse {
  repeated Todo todos = 1;
}
message UpdateTodoRequest {
  int32 id = 1;
  bool completed = 2;
}
message UpdateTodoResponse {
  string message = 1;
}
```

Encoding in wire format

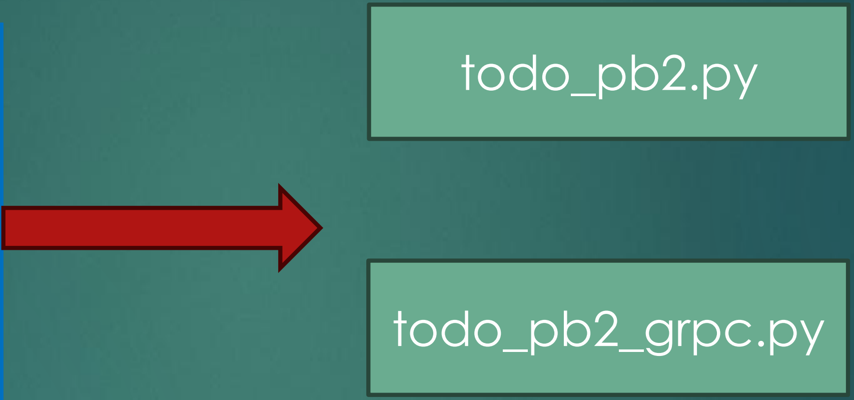
Array – more than one response

- Start with a .proto file – Example: ToDo list on server. Retrieve list and update status

Protocol Buffers Working


Create .proto file to define data structure
.proto file
Generate code using the protoc compiler
Source files
Compile PB code with your project code
Compiled classes
Use PB classes to serialize, share, and deserialize data

```
syntax = "proto3";
package todo;
service TodoService {
  rpc GetTodoList (Empty) returns (TodoListResponse);
  rpc UpdateTodoStatus (UpdateTodoRequest) returns (UpdateTodoRespons
e);
}
message Empty {}
message Todo {
  int32 id = 1;
  string title = 2;
  bool completed = 3;
}
message TodoListResponse {
  repeated Todo todos = 1;
}
message UpdateTodoRequest {
  int32 id = 1;
  bool completed = 2;
}
message UpdateTodoResponse {
  string message = 1;
}
```



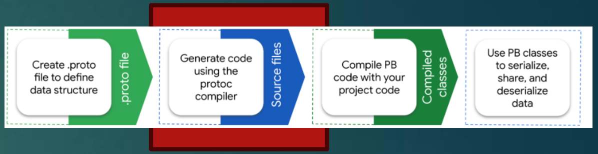
todo_pb2.py

todo_pb2_grpc.py

- Compile using the protoc compiler.
- For python use

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

The todo_pb2.py file



Create .proto file to define data structure

.proto file

Generate code using the protoc compiler

Source files

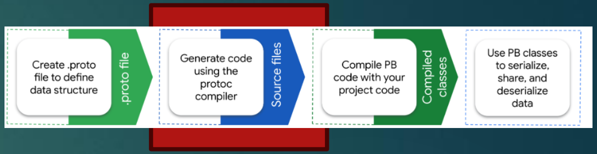
Compile PB code with your project code

Compiled classes

Use PB classes to serialize, share, and deserialize data

- Message Classes
  - Each message in .proto file becomes a class
- Serialization methods
  - Each class has methods like
    - `SerializeToString()` → Converts message to binary format
    - `ParseFromString()` → Converts binary format to message object
    - `CopyFrom()` → copies data to another message
- Field Access
  - Fields are accessed as regular attributes
- Utility Functions
  - Internal helpers for encoding/decoding, field validation etc.

The todo_pb2_grpc.py file



Create .proto file to define data structure

.proto file

Generate code using the protoc compiler

Source files

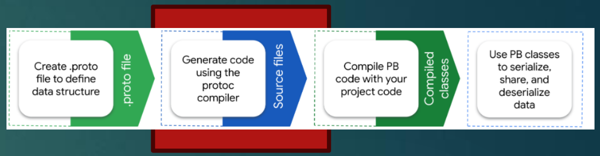
Compile PB code with your project code

Compiled classes

Use PB classes to serialize, share, and deserialize data

- Stub Class for the client
  - Use this to make calls from your client
- Servicer class for the server
  - Extend the stub class to implement your server
- Registration function
  - Registers the implementation with the GRPC server

Debugging



Create .proto file to define data structure

.proto file

Generate code using the protoc compiler

Source files

Compile PB code with your project code

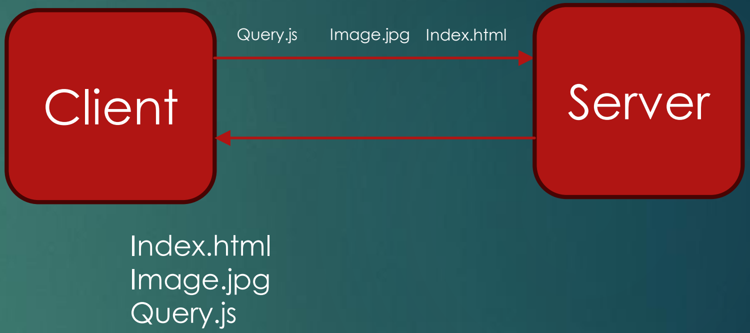
Compiled classes

Use PB classes to serialize, share, and deserialize data

- Check if you compiled correctly
  - Use verbose mode to check for compilation issues
- Inspect the generated classes manually to check if something is missing
- Use Logging at Server and Client (import logging and add logging into the code)
- Check if serialization is working properly
- Use grpcurl or postman without writing a client program
- Check for port conflicts (use netstat/lsof)
- Are both server and client using the same .proto file

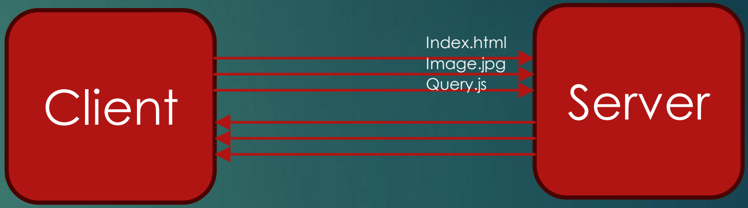
Problems with HTTP 1.1

- Get each component one after the other
  - First get index.html, then image.jpg and then query.js.
  - If index.html is large and delayed, it adds latency to the following requests
  - Leads to head of line blocking



How does HTTP 2 solve the problem?

- Single TCP connection
- Multiple requests in parallel multiplexed on the connection
- **Server Push**
  - Server pushes data before client asks for it



Client

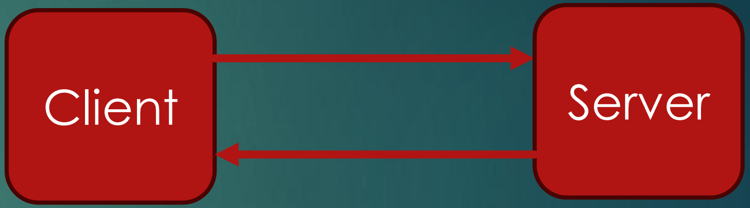
Index.html
Image.jpg
Query.js

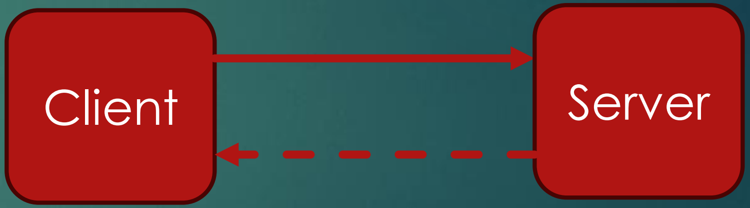
Server

Index.html
Image.jpg
Query.js

Types of GRPC APIs

- Unary
  - Client sends single request
  - Receives single response
- Streaming
  - Client sends a single request
  - Receives a stream of response
  - Where to use
    - Real time data feeds – financial, sports
    - Large data set retrieval
    - ML Inference
    - IOT device monitoring
    - Health monitoring
    - Streaming Search results

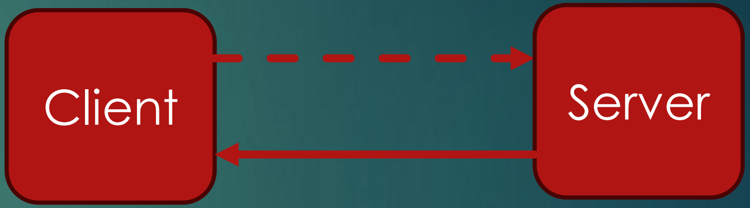




```
rpc StreamTodos (TodoStreamRequest) returns (stream Todo);
```

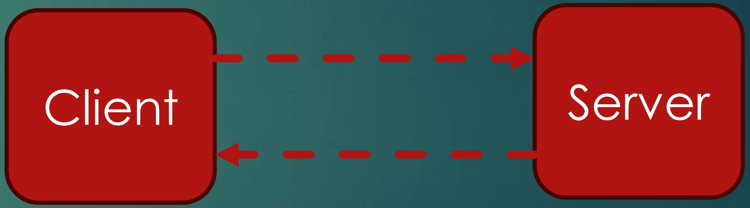
Types of GRPC APIs

- Client Streaming
  - Client sends a stream of request
  - Receives single response
  - Use cases:
    - Upload large data, ML Training Data, Batch data processing, Form submission, Survey responses, Event Logging,



rpc UploadTodos (**stream Todo**) returns (UploadSummary);

- Bidirectional Streaming
  - Client and server send messages in any order
  - Messages independent of each other
  - Client initiates and ends streaming
  - Use cases:
    - Real time messaging, Multiplayer gaming, Live collaborative tools (e.g. Google Docs), Financial trading, ML Interactive inference. Remote device control (e.g drones), wearable health devices



rpc ChatStream (**stream ChatMessage**) returns (**stream ChatMessage**);

GRPC vs REST

	gRPC	REST
Transport	HTTP 2.0	HTTP 1.1
Serialization	Protocol Bufferes	JSON
Performance	Binary transfers in Protobuf, so much faster	Parsing JSON takes CPU cycles
APIs	Bidirectional, Streaming and Asynchronous	Client requests and Server response
Typing	Protobuf ensures typing	Needs checking by server

References

- https://protobuf.dev/
- https://www.youtube.com/watch?v=sN8w-Llp_6g

Coding Exercise

Write a simple chat application that store a list of well known message requests and responses on the server in a dictionary. The chat client send a request message to the server and the server looksup the dictionary for the response and sends a response. If the lookup fails, the server sends "Sorry, I did not understand response". A "Bye" message in request closes the connection.

- Use bidirectional streaming with GRPC to implement the above.
- Due date: 12/9/2025

Video Watch

- The End of Cloud Computing - Peter Levine
  - https://www.youtube.com/watch?v=I9tOd6fHR-U ()