# Protocol Buffers (Protobuf)

## Polyglot

- *Schema First* design requires a format or protocol to define our *Schema*.
- Ideally, teams will have the *Autonomy* to choose which language to write their *Services* in.
- This means interacting with the API in a language-agnostic format.
- Akka Serverless uses *Protocol Buffers* to define a language-agnostic *Schema*.

# Protocol Buffers (Protobuf)

- Protocol Buffers are an efficient binary protocol supporting many languages.
    - JavaScript, C#, Java, Go, Python, etc.
- Mature, open-source mechanism for serializing data.
- Allows *Schema First* development of your messages.
- Compiles the *Schema* into code to handle the messages.
- 3-10x smaller and 20-100x faster than XML

# Protocol Buffer Messages

```
syntax = "proto3";

package com.example.helloworld;

message Hello {
    string contents = 1;
}
```

- *Message Schemas* are defined in .proto files.
- The message keyword declares a *Message*.
- They contain a series of fields mapped to an index.
- The index is a unique key and allows the field name to change without breaking compatibility.

## Syntax and Package

```
syntax = "proto3";

package com.example.helloworld;
...
```

- The syntax defines the version of the protobuf specification.
  - In this course we will use "proto3".
- A package provides a namespace so collisions won't occur.
  - Enables greater team *Autonomy*.
- Both of these should be in *every* .proto file.

## Custom Extensions

```
syntax = "proto3";

import "cloudstate/entity_key.proto";

package com.example.helloworld;

message SayHelloMessage {
    string user_id = 1 [(.cloudstate.entity_key) = true];
}
```

- We can import additional .proto definitions to extend functionality.
- The entity_key.proto above allows us to annotate our user_id field as an *entity_key* for use in Akka Serverless.
  - More on this later.

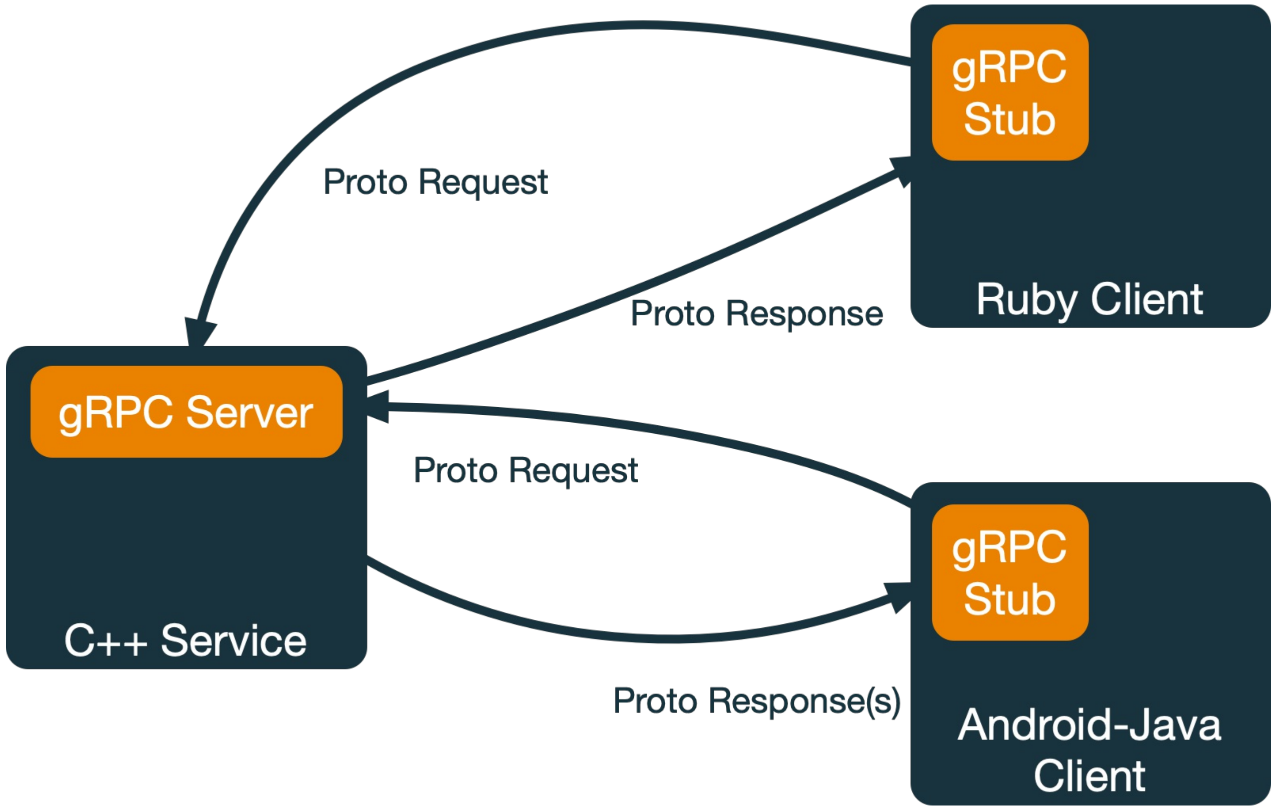
## gRPC

# Schema First APIs

gRPC

- Protocol Buffers define a message format.
- Akka Serverless systems are built around *Services*.
- We need a way to describe *Services* using Protobuf messages.
- Enter *gRPC*, a high-performance, open-source, Remote Procedure Call (RPC) framework.

# gRPC Overview



- *gRPC* applications can talk to each other in a variety of environments.
- They can be written in any of *gRPC's* many supported languages.
- Clients can directly call methods on the server as though they were local.
- We can define *gRPC Services* in our .proto files.

# gRPC Service Definitions

```
service HelloWorld {
    rpc SayHello (SayHelloMessage) returns (HelloMessage) {};
}
```

- The service keyword enables us to describe *gRPC Services*.
- Our *Service* will include a series of rpc calls.
- We can use the messages we define in the same .proto file to describe the input and output of our *RPC* calls.

## Generate code from proto files

- The protoc compiler for Protobuf will compile and generate code from your .proto file.
- A special plugin for protoc enables *gRPC* support.
- It can generate:
  - Client code
  - Server code
  - Message serialization/deserialization code
- For Akka Serverless, this is handled by the compile-descriptor tool (see package.json).

```
compile-descriptor ./helloworld.proto
```