Lab 5. gRPC Programming

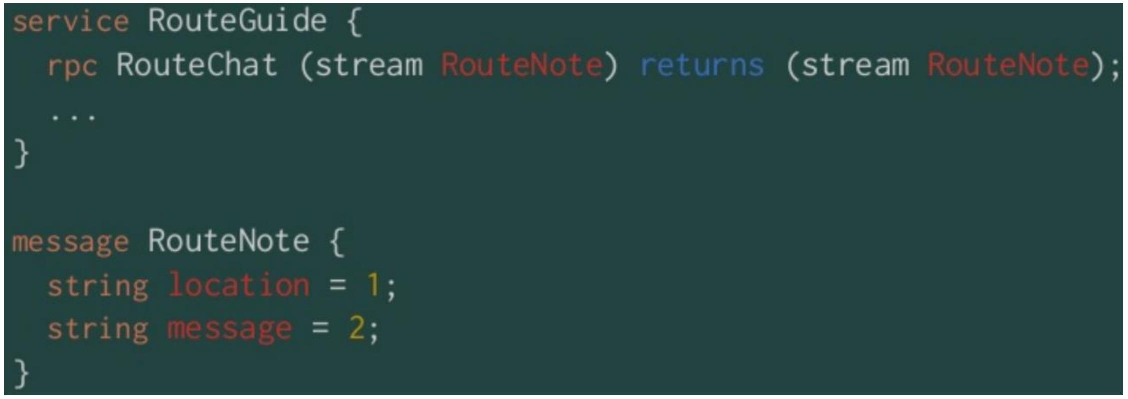
Korea Advanced Institute of Science and Technology

C.H. Youn (Sept. 11, 2018)

gRPC and Interface Definition

- gRPC uses Protocol Buffers as the **Interface Definition Language (IDL)** for describing both the **service interface** and the **structure of the payload messages**.
- gRPC provides Protocol Compiler plugins that generate Client- and Server-side APIs with an interface definition in a .proto file.

Example) Bidirectional Streaming RPC Definition

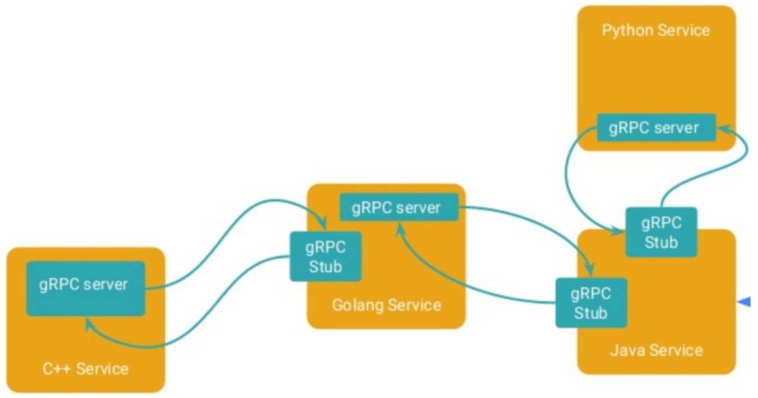


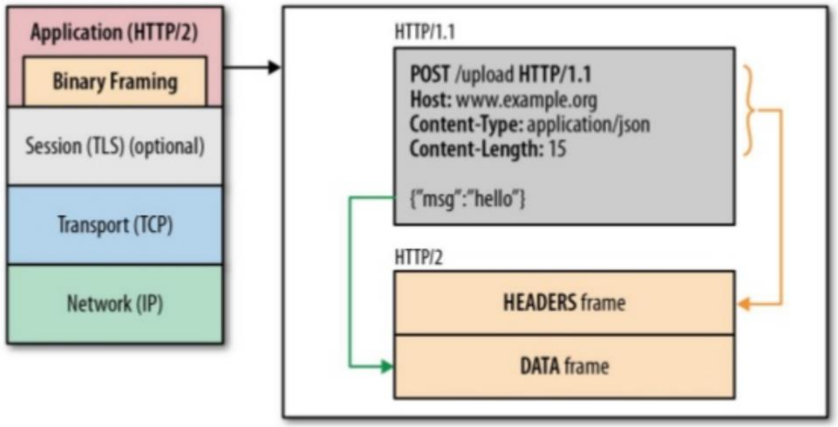
```
service RouteGuide {
  rpc RouteChat (stream RouteNote) returns (stream RouteNote);
  ...
}

message RouteNote {
  string location = 1;
  string message = 2;
}
```

gRPC over HTTP/2

- The abstract protocol defined above is implemented over HTTP/2.
  - gRPC bidirectional streams are mapped to HTTP/2 streams.
  - The contents of Call Header and Initial Metadata are sent as HTTP/2 headers and subject to HPACK compression.
  - Payload Messages are serialized into a byte stream of length prefixed gRPC frames which are then fragmented into HTTP/2 frames at the sender and reassembled at the receiver.
  - Status and Trailing-Metadata are sent as HTTP/2 trailing headers.
- gRPC inherits the flow control mechanisms in HTTP/2 and uses them to enable fine-grained control of the amount of memory used for buffering in-flight messages.



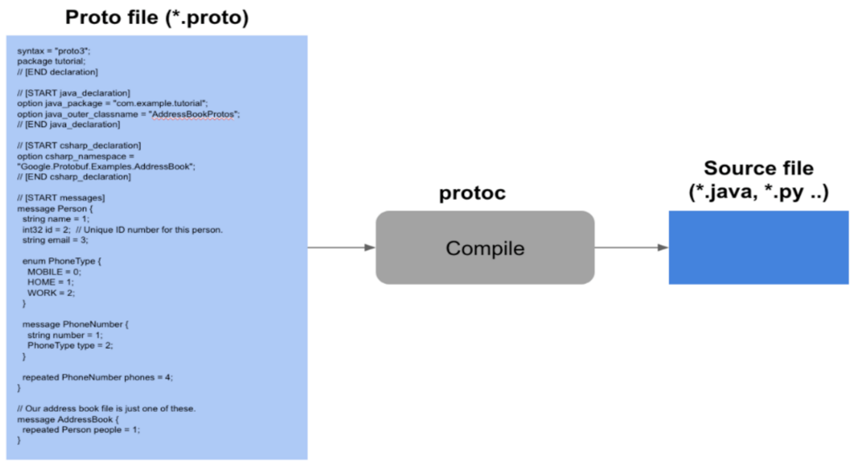


Protocol Buffer

- The Prototype Buffer is a serialized data structure developed by Google and released as open source.
- Serialization is the act of storing data in a binary stream for storage in a file or for transmission over a network.
- It supports various languages such as C ++, C #, Go, Java, Python, Object C, Javascript, Ruby, etc. Serialization speed is fast and serialized file size is small.

Protobuf Compile

- To use the protocol buffer, define the data type to be stored as proto file. Define a datatype that has no dependency on a particular language.
- When you compile the proto file with the protoc compiler, you create a data class file of the appropriate type for each language.



```
syntax = "proto3";
package tutorial;
// [END declaration]

// [START java_declaration]
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
// [END java_declaration]

// [START csharp_declaration]
option csharp_namespace =
"Google.Protobuf.Examples.AddressBook";
// [END csharp_declaration]

// [START messages]
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

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

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person people = 1;
}
```

<An example of compiling with Python>

```
protoc -I=./ --python_out=./ ./address.proto
```

```
import address_pb2

person = address_pb2.Person()

person.name = 'Terry'
person.age = 42
person.email = 'terry@mycompany.com'

f = open('myaddress','wb')
f.write(person.SerializeToString())
f.close()
```

Protobuf to JSON

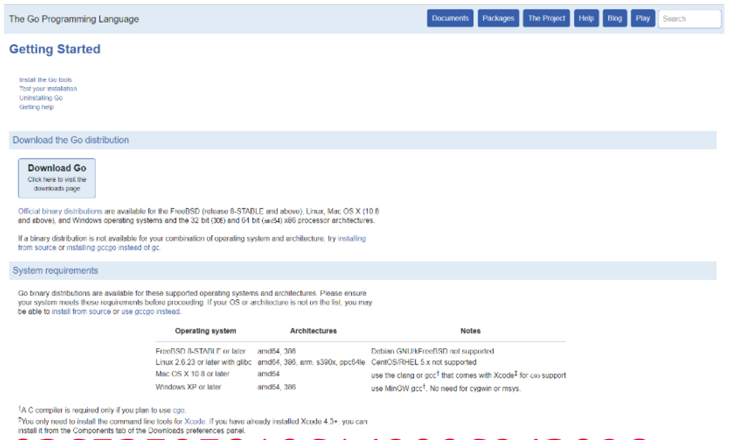
- When implementing a REST API such as HTTP / JSON from a client (mobile) to a server, it is necessary to serialize JSON into protocol buffer format before transmission, to transmit the packet with a reduced overall packet amount.
- API gateway is placed in front of the back-end server, and the message body that comes in the protocol buffer is converted into JSON and passed to the back-end API server.

```
from google.protobuf.json_format import
MessageToJson
jsonObj = MessageToJson(person)
print jsonObj
```

$\longrightarrow$

```
{
 "age": 42,
 "name": "Terry",
 "email": "terry@mycompany.com"
}
```

gRPC Environment Setting (Linux)



The Go Programming Language

Documents Packages The Project Help Blog Play Search

Getting Started

Install the Go tools
Test your installation
Uninstalling Go
Getting help

Download the Go distribution

Download Go
Click here to visit the downloads page

Official binary distributions are available for the FreeBSD (release 8-STABLE and above), Linux, Mac OS X (10.8 and above), and Windows operating systems and the 32 bit (386) and 64 bit (amd64) x86 processor architectures.

If a binary distribution is not available for your combination of operating system and architecture, try installing from source or installing gccgo instead of gc.

System requirements

Go binary distributions are available for these supported operating systems and architectures. Please ensure your system meets these requirements before proceeding. If your OS or architecture is not on the list, you may be able to install from source or use gccgo instead.

Operating system	Architectures	Notes
FreeBSD 8-STABLE or later	amd64, 386	Debian GNU/kFreeBSD not supported
Linux 2.6.23 or later with glibc	amd64, 386, arm, s390x, ppc64le	CentOS/RHEL 5.x not supported
Mac OS X 10.8 or later	amd64	use the clang or gcc† that comes with Xcode‡ for cgo support
Windows XP or later	amd64, 386	use MinGW gcc†. No need for cygwin or msys.

†A C compiler is required only if you plan to use cgo.
‡You only need to install the command line tools for Xcode. If you have already installed Xcode 4.3+, you can install it from the Components tab of the Downloads preferences panel.

- **Golang Installation**
  - Install Golang binaries
  - https://golang.org/doc/install?cm_mc_uid=02652595810614900694309&cm_mc_sid_50200000=1490500863
  - Install the downloaded compressed file through the following command

```
tar -C /usr/local -xzf <다운로드 받은 압축파일>
```

  - Setting Environment Variables in $ HOME / .profile

```
$HOME 아래에 work라는 디렉토리 생성

mkdir $HOME/work
export GOROOT=/usr/local/go
export GOPATH=$HOME/work
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

gRPC Environment Setting (Linux)

- **Check Golang Execution**
  - hello.go

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

  - Build source code

```
cd $GOPATH/src/hello
go build
```

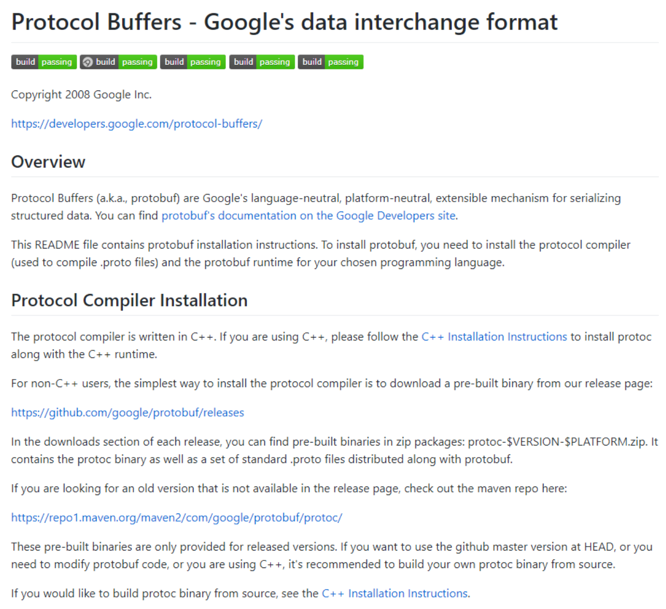
  - Execute

```
./hello
hello, world
```

Install Protocol Buffer

- Download Protocol Buffer Archive
  - https://github.com/google/protobuf/releases
- Unarchive and register to $PATH

```
#### ProtoBuf ###
export PATH=$PATH:/Users/mjkong/Dev/sdk/protoc-3.2.0rc2-osx-x86_64/bin
```



Protocol Buffers - Google's data interchange format

build passing | build passing | build passing | build passing | build passing

Copyright 2008 Google Inc.

https://developers.google.com/protocol-buffers/

Overview

Protocol Buffers (a.k.a., protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. You can find protobuf's documentation on the Google Developers site.

This README file contains protobuf installation instructions. To install protobuf, you need to install the protocol compiler (used to compile .proto files) and the protobuf runtime for your chosen programming language.

Protocol Compiler Installation

The protocol compiler is written in C++. If you are using C++, please follow the C++ Installation Instructions to install protoc along with the C++ runtime.

For non-C++ users, the simplest way to install the protocol compiler is to download a pre-built binary from our release page:

https://github.com/google/protobuf/releases

In the downloads section of each release, you can find pre-built binaries in zip packages: protoc-$VERSION-$PLATFORM.zip. It contains the protoc binary as well as a set of standard .proto files distributed along with protobuf.

If you are looking for an old version that is not available in the release page, check out the maven repo here:

https://repo1.maven.org/maven2/com/google/protobuf/protoc/

These pre-built binaries are only provided for released versions. If you want to use the github master version at HEAD, or you need to modify protobuf code, or you are using C++, it's recommended to build your own protoc binary from source.

If you would like to build protoc binary from source, see the C++ Installation Instructions.

- Install Protobuf Golang plugin

```
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
go get google.golang.org/grpc
```

gRPC Example

- **Hello world example**

```
cd
$GOPATH/src/google.golang.org/grpc/helloworld
```

  - [dir] greeter_client – Client code
  - [dir] greeter_server – Server code
  - [dir] helloworld - protobuf file specify the gRPC service. With command ‘protoc’ .pb.go file will be created and it specify the message type of server-client communication
  - Run Server Code

```
go run greeter_server/main.go
```

  - Run Client Code

```
go run greeter_client/main.go
```

  - Check out message “Greeting: Hello world”

gRPC Example

- **Service Update**

```
$GOPATH/src/google.golang.org/grpc/examples/helloworld/helloworld/helloworld.proto
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (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;
}
```

  – Build gRPC Code

```
protoc -I helloworld/ helloworld/helloworld.proto --go_out=plugins=grpc:helloworld
```

gRPC Example

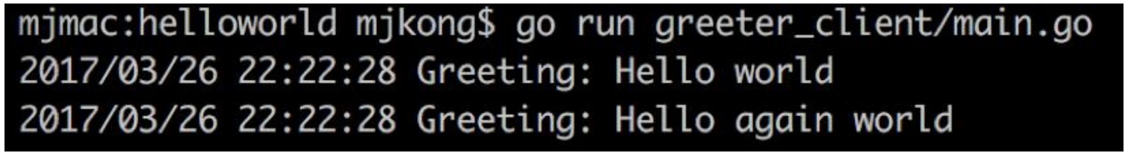
- **Service Update**
  - Server code updated

```
func (s *server) SayHelloAgain(ctx context.Context, in
*pb.HelloRequest) (*pb.HelloReply, error) {
    return &pb.HelloReply{Message: "Hello again " +
in.Name}, nil
}
```

  - Client code updated

```
r, err = c.SayHelloAgain(context.Background(),
&pb.HelloRequest{Name: name})
if err != nil {
    log.Fatalf("could not greet: %v", err)
}
log.Printf("Greeting: %s", r.Message)
```

  - Execution Result



```
mjmac:helloworld mjkong$ go run greeter_client/main.go
2017/03/26 22:22:28 Greeting: Hello world
2017/03/26 22:22:28 Greeting: Hello again world
```

Node.js gRPC Example (Protobuf)

```
service BookService {
  rpc List (Empty) returns (BookList) {}
  rpc Insert (Book) returns (Empty) {}
  rpc Get (BookIdRequest) returns (Book) {}
  rpc Delete (BookIdRequest) returns (Empty) {}
  rpc Watch (Empty) returns (stream Book) {}
}

message Empty {}

message Book {
  int32 id = 1;
  string title = 2;
  string author = 3;
}

message BookList {
  repeated Book books = 1;
}
message BookIdRequest {
  int32 id = 1;
}
```

Node.js gRPC Example (Server)

```
var grpc = require('grpc');

var booksProto = grpc.load('books.proto');

var server = new grpc.Server();


server.bind('0.0.0.0:50051',
  grpc.ServerCredentials.createInsecure());
console.log('Server running at
http://0.0.0.0:50051');
server.start();
```

```
server.addService(booksProto.books.BookService.service, {
    list: function(call, callback) {
        callback(null, books);
    },
    insert: function(call, callback) {
        var book = call.request;
        books.push(book);
        callback(null, {});
    },
    get: function(call, callback) {
        for (var i = 0; i < books.length; i++)
            if (books[i].id == call.request.id)
                return callback(null, books[i]);
        callback({
            code: grpc.status.NOT_FOUND,
            details: 'Not found'
        });
    },
    delete: function(call, callback) {
        for (var i = 0; i < books.length; i++) {
            if (books[i].id == call.request.id) {
                books.splice(i, 1);
                return callback(null, {});
            }
        }
        callback({
            code: grpc.status.NOT_FOUND,
            details: 'Not found'
        });
    }
});
```

Node.js gRPC Example (Client)

```
var grpc = require('grpc');

var booksProto = grpc.load('books.proto');

var client = new
booksProto.books.BookService('127.0.0.1:50051',
  grpc.Credentials.createInsecure());

function printResponse(error, response) {
  if (error)
    console.log('Error: ', error);
  else
    console.log(response);
}

function listBooks() {
  client.list({}, function(error, books) {
    printResponse(error, books);
  });
}
```

```
function insertBook(id, title, author) {
  var book = { id: parseInt(id), title: title,
author: author };
  client.insert(book, function(error, empty) {
    printResponse(error, empty);
  });
}

function getBook(id) {
  client.get({ id: parseInt(id) },
function(error, book) {
    printResponse(error, book);
  });
}

function deleteBook(id) {
  client.delete({ id: parseInt(id) },
function(error, empty) {
    printResponse(error, empty);
  });
}
```

Result

```
$ node client.js insert 2 "The Three Musketeers"
"Alexandre Dumas"
{}

$ node client.js list
{ books:
   [ { id: 123,
       title: 'A Tale of Two Cities',
       author: 'Charles Dickens' },
     { id: 2,
       title: 'The Three Musketeers',
       author: 'Alexandre Dumas' } ] }
```

```
$ node client.js delete 123
{}

$ node client.js list
{ books:
   [ { id: 2,
       title: 'The Three Musketeers',
       author: 'Alexandre Dumas' } ] }

$ node client.js get 2
{ id: 2,
  title: 'The Three Musketeers',
  author: 'Alexandre Dumas' }
```

Lab 5. gRPC Programming

- Read text materials and test practices
- Use your notebook PC to examine context and produce the source code when you finish the successful practice.
- Objectives
  - Understand the process of RPC and gRPC
  - Learn the how to use gRPC and protocol buffer