gRPC Programming

References for study:

- gRPC Tutorials (https://grpc.io/docs/tutorials/)

Code Generation

- gRPC supports (static or dynamic) client and server stub generation for several programming languages
  - C# C++ Dart Go Java Kotlin Node Objective-C PHP Python Ruby
- Clients can use (one or more) stub instances and can open gRPC channels (mapped to HTTP/2 connections)
  - Channels are bidirectional
  - One channel can carry several RPC interactions (i.e., HTTP/2 streams)
  - Each channel side can be closed independently (and gracefully)

Example: Code Generation in Java

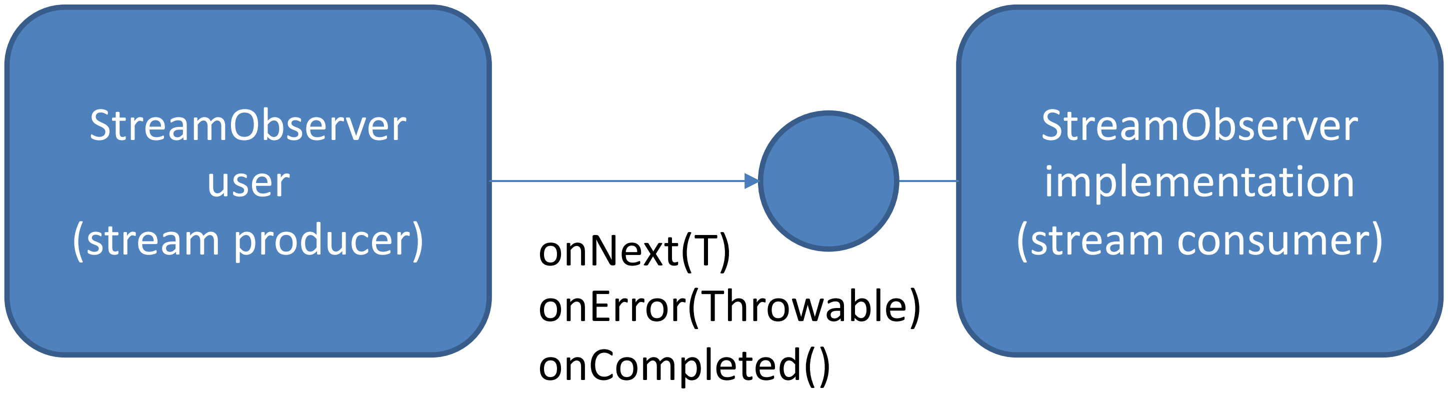
- Static generation of stub classes from proto file
  - One class per service
    - including nested server and client stub classes
  - One class per proto file
    - including message classes and enums
- Generation can be automated within the build process (e.g., by maven or gradle)
- Documentation and tutorial:
  - https://grpc.io/docs/languages/java/basics/

gRPC Programming in Java

- Server-side:
  - Extend server stub
    - implement methods, one per remote procedure
  - Create Server
- Client-side:
  - Create channel (open HTTP/2 connection)
  - Instantiate client stub and associate it with channel
  - call methods on the stub

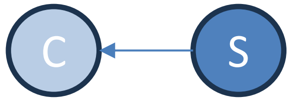
Managing Streaming in Java

- StreamObserver<T> library interface
  - represents an Observer on a stream of RPC messages (of type T)

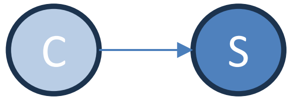

StreamObserver user (stream producer)
onNext(T)
onError(Throwable)
onCompleted()
StreamObserver implementation (stream consumer)

Managing Streaming in Java

- Response streaming in server-side programming
  - programmer develops stream producer (uses observer)
  - library provides consumer (creates observer)
  - used to return response in all methods (even those that do not use server streaming)



- Request streaming in server-side programming
  - programmer develops stream consumer (provides observer)
  - library produces data (uses observer)



- Opposite roles in client-side programming

Example: Stub Generation in Node

- Proto file dynamic parsing and stub constructor generation

```
const PROTO_PATH = __dirname + '/../proto/bank.proto';
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

let packageDefinition = protoLoader.loadSync(
    PROTO_PATH,
    {keepCase: true,
     longs: String,
     enums: String,
     defaults: true,
     oneofs: true
    });
let protoD = grpc.loadPackageDefinition(packageDefinition);
let bank = protoD.it.polito.dsp.bank;
```

proto file descriptor

namespace containing stub constructors

- Documentation: https://grpc.github.io/grpc/node/index.html

gRPC Programming in Node

- Server-side:
  - load proto file (create **proto file descriptor**)
  - create generic **Server instance** (constructor from gRPC library)
  - add **service object** (obtained from stub constructor) and **implementation mapping** to server
  - bind the server to an unused port and start it
- Client-side:
  - load proto file (create **proto file descriptor**)
  - Instantiate **stub** by calling stub constructor (which also creates HTTP/2 channel to server)
  - call methods on the stub

Server Creation

```
function addDeposit(call, callback) {
    ...
}
function addWithdrawal(call, callback) {
    ...
}

if (require.main === module) {
    ...
    var server = new grpc.Server();
    server.addService(bank.BankOperations.service,
      { addDeposit: addDeposit,
        addWithdrawal: addWithdrawal }
    );
    server.bindAsync(...);
}
```

implementations of operations

stub constructor namespace generation (bank)

generic server instance

service object

operation mapping object

server binding

Server Binding

```
if (require.main === module) {
    ...
    var server = new grpc.Server();
    server.addService(bank.BankOperations.service,
      { addDeposit: addDeposit,
        addWithdrawal: addWithdrawal }
    );
    server.bindAsync('localhost:5000',
       grpc.ServerCredentials.createInsecure(),
       function(err,actualPort){
           if(err) {console.log(err); return}
           console.log(`starting server on ${actualPort}`);
           server.start()
           console.log('server ready');
       }
    );
}
```

Managing Streaming in Node (Server-Side)

- No streaming
  - **function** name (call, callback)
    - request available as call.request
    - response returned by calling callback(err,response)
- Request Streaming
  - **function** name (call, callback)
    - call implements a Readable to read request (events: data, error, end)
- Response Streaming
  - **function** name (call)
    - call implements a Writable to write response (methods: write, end)
- Request and Response Streaming
  - **function** name (call)

Managing Streaming in Node (Client-Side)

- No Streaming
  - stub.fname(request, **function**(err, response) { ... });
- Request Streaming
  - var call = stub.fname(**function**(err, response) { ... });
  - call implements a Writable to write request (methods: write, end)
- Response Streaming
  - var call = stub.fname(request);
  - call implements a Readable to read response (events: data, error, end)
- Request and Response Streaming
  - var call = stub.fname();