CPP 에서 gRPC 사용하기

gRPC 설치 및 사용법 에 이어 C++ 에서 gRPC 를 사용하는 예제를 직접 작성 하면서 정리한 내용을 기술 한다. 예제(HelloWorld) 는 google 에서 작성한 code 이므로 설치만 잘되면 오류 없이 잘 실행이 된다. 하지만 실제 project 에 적용 하려면 sample program 을 작성 해 보아야 개념이 정립된다. 따라서 실제 project 에 적용 하기 이전에 각 언어별로 sample 을 작성 하기로 하였다. 첫번째는 주 개발 언어인 C/C++ 을 이용하기로 하였다. 그리고 server(백엔드) 는 C++ 로 개발 하여 타언어(JAVA, Go, Python) Client Sample 테스트시 사용 할 계획이다.

1. proto 파일 작성

각 서비스간 interface 규약을 `xdb_grpc.proto` 을 이용하여 정의 한다.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.hancomsecure.xecuredb.grpc";
option java_outer_classname = "XecuredbProto";
option objc_class_prefix = "XDB";

package xdbproto;

// The Xecuredb service definition.
service Xecuredb {
  rpc Encrypt (EncRequest) returns (EncReply) {}
  rpc Decrypt (DecRequest) returns (DecReply) {}
  rpc Hash (HashRequest) returns (HashReply) {}
}

// The request message containing the alias and plain text.
message EncRequest {
  string alias = 1;
  string plain = 2;
}

// The response message containing the err code and cipher text.
message EncReply {
  int32 err = 1;
  string cipher = 2;
}

// The request message containing the alias and cipher text.
message DecRequest {
  string alias = 1;
  string cipher = 2;
}

// The response message containing the err code and plain text.
message DecReply {
  int32 err = 1;
  string plain = 2;
}
```

```
// The request message containing the alogorithm id and plain text.
message HashRequest {
  int32 id = 1;
  string plain = 2;
}

// The response message containing the err code and hash string.
message HashReply {
  int32 err = 1;
  string hash = 2;
}
```

- HelloWorld 는 1개의 service 함수(parameter 1개) 만 사용하는 예제 이므로 서비스 개 수 를 늘리고 parameter 도 복수로 사용 할 수 있도록 작성 하였다.

위의 .proto 파일을 이용해 C++ 에서 server 및 client 를 개발 하려면 다음과 같은 명령을 이용하여 source code 및 header 파일을 생성 하여야 한다. 다음은 `protoc` 를 이용하여 source 를 generation 하는 명령어 이다.

```
protoc --cpp_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin`
xdb_grpc.proto
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin`
xdb_grpc.proto
```

gRPC 를 잘 모르는 상태 에서 예제만 보고 test 용 .proto 파일만 만들어 사용 하려다, 이 부분 에서 약간의 시간을 소비 하였다.

우선 첫번째 문장인 `protoc --cpp_out=. --plugin=protoc-gen-grpc=` which grpc_cpp_plugin `xdb_grpc.proto` 를 실행 하면

```
xdb_grpc.pb.h
xdb_grpc.pb.cc
```

이렇게 2개의 파일이 생성 된다. 해당 파일을 gRPC 를 사용 하지 않고 `protocol buffer` 만 사용 할 경우 반드시 필요한 파일들 이다. 그렇다고 위의 2개 파일이 gRPC 를 사용 할 경우 필요 없는것은 아니다.

다음으로 `protoc --grpc_out=. --plugin=protoc-gen-grpc=` which grpc_cpp_plugin `xdb_grpc.proto` 를 실행 하면

```
xdb_grpc.grpc.pb.h
xdb_grpc.grpc.pb.cc
```

두개의 파일이 생성 된다. 즉 gRPC 를 사용 하기 위해서는 위의 4개 파일이 생성 되어야 한다.

처음에는 test 를 위해 shell_script.sh 파일을 만들어서 수행 했지만 확인이 완료된 상태 에서는 Makefile 에 포함시켜서 사용 하면 된다.

`Helloworld` 예제의 Makefile 을 보면 4개의 파일을 생성 할 수 있도록 되어 있다.

2. 컴파일을 위한 Makefile

```
HOST_SYSTEM = $(shell uname | cut -f 1 -d_)
SYSTEM ?= $(HOST_SYSTEM)
CXX = g++
CPPFLAGS += `pkg-config --cflags protobuf grpc`
CXXFLAGS += -std=c++11
ifeq ($(SYSTEM),Darwin)
LDFLAGS += -L/usr/local/lib `pkg-config --libs protobuf grpc++ grpc`\
           -lgrpc++_reflection\
           -ldl
else
LDFLAGS += -L/usr/local/lib `pkg-config --libs protobuf grpc++ grpc`\
           -Wl,--no-as-needed -lgrpc++_reflection -Wl,--as-needed\
           -ldl
endif
PROTOC = protoc
GRPC_CPP_PLUGIN = grpc_cpp_plugin
GRPC_CPP_PLUGIN_PATH ?= `which $(GRPC_CPP_PLUGIN)`

PROTOS_PATH = .

vpath %.proto $(PROTOS_PATH)

all: xdb_server xdb_client

xdb_server: xdb_grpc.pb.o xdb_grpc.grpc.pb.o xdb_server.o
    $(CXX) $^ $(LDFLAGS) -o $@

xdb_client: xdb_grpc.pb.o xdb_grpc.grpc.pb.o xdb_client.o
    $(CXX) $^ $(LDFLAGS) -o $@

.PRECIOUS: %.grpc.pb.cc
%.grpc.pb.cc: %.proto
    $(PROTOC) -I $(PROTOS_PATH) --grpc_out=. --plugin=protoc-gen-
grpc=$(GRPC_CPP_PLUGIN_PATH) $<

.PRECIOUS: %.pb.cc
%.pb.cc: %.proto
    $(PROTOC) -I $(PROTOS_PATH) --cpp_out=. $<

clean:
    rm -f *.o *.pb.cc *.pb.h xdb_server xdb_client
```

- 예제의 복잡한 Makefile 에서 반드시 필요한 부분만 남기고 삭제 하였다.

3. Server Source 작성

`xdb_server.cc`

```
#include <iostream>
#include <memory>
#include <string>
```

```
#include <grpcpp/grpcpp.h>

#include "xdb_grpc.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;

using xdbproto::DecReply;
using xdbproto::DecRequest;
using xdbproto::EncReply;
using xdbproto::EncRequest;
using xdbproto::HashReply;
using xdbproto::HashRequest;
using xdbproto::Xecuredb;

class XecuredbImpl final : public Xecuredb::Service {
  Status Encrypt(ServerContext* context,
                 const EncRequest* request,
                 EncReply* reply) override {
    std::string prefix("enc ");
    reply->set_err(0);
    reply->set_cipher(prefix + std::string(" ") + request->alias() +
std::string(" ") + request->plain());

    return Status::OK;
  }

  Status Decrypt(ServerContext* context,
                 const DecRequest* request,
                 DecReply* reply) override {
    std::string prefix("dec ");
    reply->set_err(0);
    reply->set_plain(prefix + std::string(" ") + request->alias() +
std::string(" ") + request->cipher());

    return Status::OK;
  }

  Status Hash(ServerContext* context, const HashRequest* request,
                  HashReply* reply) override {
    reply->set_err(0);
    reply->set_hash( std::string("hash ") + request->plain());

    return Status::OK;
  }
};


void RunServer() {
  std::string server_address("0.0.0.0:50052");
  XecuredbImpl service;

  ServerBuilder builder;

  builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
```

```
  builder.RegisterService(&service);

  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << std::endl;

  server->Wait();
}

int main(int argc, char** argv) {
  RunServer();

  return 0;
}
```

4. Client Source 작성

```
#include <iostream>
#include <memory>
#include <string>

#include <grpcpp/grpcpp.h>

#include "xdb_grpc.grpc.pb.h"

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;

using xdbproto::DecReply;
using xdbproto::DecRequest;
using xdbproto::EncReply;
using xdbproto::EncRequest;
using xdbproto::HashReply;
using xdbproto::HashRequest;
using xdbproto::Xecuredb;

class XecuredbClient {
 public:
  XecuredbClient(std::shared_ptr<Channel> channel)
      : stub_(Xecuredb::NewStub(channel)) {}

  std::string Encrypt(const std::string& alias, const std::string& plain) {
    // Data we are sending to the server.
    EncRequest request;
    request.set_alias(alias);
    request.set_plain(plain);

    // Container for the data we expect from the server.
    EncReply reply;

    // Context for the client. It could be used to convey extra information to
    // the server and/or tweak certain RPC behaviors.
```

```
    ClientContext context;

    // The actual RPC.
    Status status = stub_->Encrypt(&context, request, &reply);

    // Act upon its status.
    if (status.ok()) {
      return reply.cipher();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }

  std::string Decrypt(const std::string& alias, const std::string& cipher) {
    DecRequest request;
    request.set_alias(alias);
    request.set_cipher(cipher);

    DecReply reply;

    ClientContext context;

    Status status = stub_->Decrypt(&context, request, &reply);

    if (status.ok()) {
      return reply.plain();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }

  std::string Hash(int algoid, const std::string& plain) {
    HashRequest request;
    request.set_id(algoid);
    request.set_plain(plain);

    HashReply reply;

    ClientContext context;

    Status status = stub_->Hash(&context, request, &reply);

    if (status.ok()) {
      return reply.hash();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }

 private:
  std::unique_ptr<Xecuredb::Stub> stub_;
};
```

```
int main(int argc, char** argv) {

  XecuredbClient xecuredb( grpc::CreateChannel( "localhost:50052",
                        grpc::InsecureChannelCredentials() ) );

  std::string alias("normal");
  std::string plain("1234567890123");
  std::string enc_str = xecuredb.Encrypt(alias, plain);
  std::cout << "Encrypt received: " << enc_str << std::endl;

  std::string dec_str = xecuredb.Decrypt(alias, plain);
  std::cout << "Decrypt received: " << dec_str << std::endl;

  std::string hash_str = xecuredb.Hash(7, plain);
  std::cout << "Hash received: " << hash_str << std::endl;

  return 0;
}
```

5. 컴파일 및 실행

소스 및 Makefile 이 있는 directory 에서 make 를 실행 하면 `xdb_server` 와 `xdb_client` 두개의 program 이 생성 된다.

`서버 실행`

```
./xdb_server
```

`클라이언트 실행`

```
./xdb_client
```

`실행 결과`

```
Encrypt received: enc  normal 1234567890123
Decrypt received: dec  normal 1234567890123
Hash received: hash 1234567890123
```

위 내용을 참조 하여 약간의 작업을 하면 Server, Client, Makefile 을 자동 생성 할 수 있을듯 하다. 물론 내부 코드는 추가로 작업 해야 하겠지만 틀만 이라도 자동으로 만들어 줄 수 있을듯, 시간 되면 작업 예정.

- 서버가 기동 되어 있지 않은 상태 에서 client 를 수행 하면 다음과 같은 error 가 발생 한다.

```
14: Connect Failed
Encrypt received: RPC failed
14: Connect Failed
Decrypt received: RPC failed
14: Connect Failed
Hash received: RPC failed
```

참고 생성된 client 프로그램을 어떤 lib 를 link 하고 있는지 확인 해 보았다.

```
$ ldd xdb_client

linux-vdso.so.1 =>  (0x00007ffeccdeb000)
libgrpc++.so.1 => /usr/local/lib/libgrpc++.so.1 (0x00007fe0ef6d3000)
libgrpc++_reflection.so.1 => /usr/local/lib/libgrpc++_reflection.so.1
(0x00007fe0ef310000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fe0eef8e000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fe0eed78000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fe0eeb5b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe0ee791000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe0ef933000)
libgrpc.so.6 => /usr/local/lib/libgrpc.so.6 (0x00007fe0ee2e3000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fe0edfda000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fe0eddd2000)
```

위 내용은 추가로 보완 예정.