gRPC + C++

A way to
go generic …

/WELCOME!

Speaker Info

Irakleia Karyoti
Sr. Software Engineer

Contact details

www.altair.com

karyoti@altair.com

ikaryoti@gmail.com

Thessaloniki, Greece

C++ gRPC

/INTRODUCTION

Embed the power of gRPC into existing applications

- ✓ Keep client and server applications as gRPC agnostic
- ✓ Use the client and server existing data model

/gRPC ... in action!

/01 /THEORY CONCEPTS

- What & Why gRPC
- What & Why protobuf
- Build & Install

/02 /FEATURES

- Rpc Types
- Service Api Types

/03 /THE STANDARD APPROACH

- The Restaurant example
- The auto-generated APIs
- Why not the auto-generated APIs

/04 /THE GENERIC APPROACH

- Wrap it up!
- A generic service with no direct gRPC dependencies
- Benefits & Cost

/01 /THEORY CONCEPTS

What & Why gRPC ⟶

- a modern open-source high performance Remote Procedure Call (RPC) framework
- connects services in and across data centers
- connects devices, mobile applications and browsers to backend services
- provides support for load balancing, tracing, health checking and authentication
- multi-language and cross-platform

https://grpc.io/
https://grpc.io/docs/languages/cpp/

/01 /THEORY CONCEPTS

What & Why protobuf

https://protobuf.dev/

- language-neutral, platform-neutral extensible mechanism for serializing structured data
- generates native language bindings
- smaller and faster than other formats (e.g., JSON, XML)
- a combination of the definition language (created in .proto files), the code that the proto compiler generates and the serialization format for data

/01 /THEORY CONCEPTS

Build and Install →

https://github.com/grpc/grpc/tree/master/src/cpp#to-start-using-grpc-c
https://github.com/grpc/grpc/blob/master/BUILDING.md

- Clone repository

git clone -b RELEASE_TAG_HERE https://github.com/grpc/grpc

- Building with Bazel (recommended)
- Building with CMake
- protoc compiler & other dependencies

/02
/FEATURES

Rpc Types

/Rpc types

/01 /Unary

The client sends a single request to the server and gets a single response back

/02 /Server Streaming

The client sends a single request and gets a stream to read a sequence of messages back

/03 /Client Streaming

The client writes a sequence of messages and sends them to the server, using a provided stream

/04 /Bidirectional Streaming

Both the client and the server send a sequence of messages using a read-write stream

/02
/FEATURES

Service Api
Types

/Service Api types (1/2)

/01 /The Sync Api (Blocking)

- Blocks client to completion

  Immediately returns status upon completion

/02 /The Async Api

- Binds a completion queue to an RPC call

  The server handles multiple requests concurrently

/03 /The Callback Api

- Library directly calls user-specified code at the completion of RPC actions through the usage of reactors

/Service Api types (2/2)

/04 /The Raw Api (Async)

- Async Api using a byte buffer as a request and response message type

/05 /The Raw Api (Callback)

- Callback Api using a byte buffer as a request and response message type

/06 /The Generic Api (Async)

- Raw Async Api
  Accepts all RPCs and hosts
  Can initiate an RPC by name

/07 /The Generic Api (Callback)

- Raw Callback Api
  Accepts all RPCs and hosts
  Can initiate an RPC by name

/Async VS /Callback

/01 Completion-queue-based

/02 Application must provide and manage its own threads

/03 Application must poll the completion queue to determine which asynchronously-initiated actions have completed

/04 Application is responsible for executing the actions for an RPC once it is notified of an incoming RPC via the completion queue

/01 Reactor based

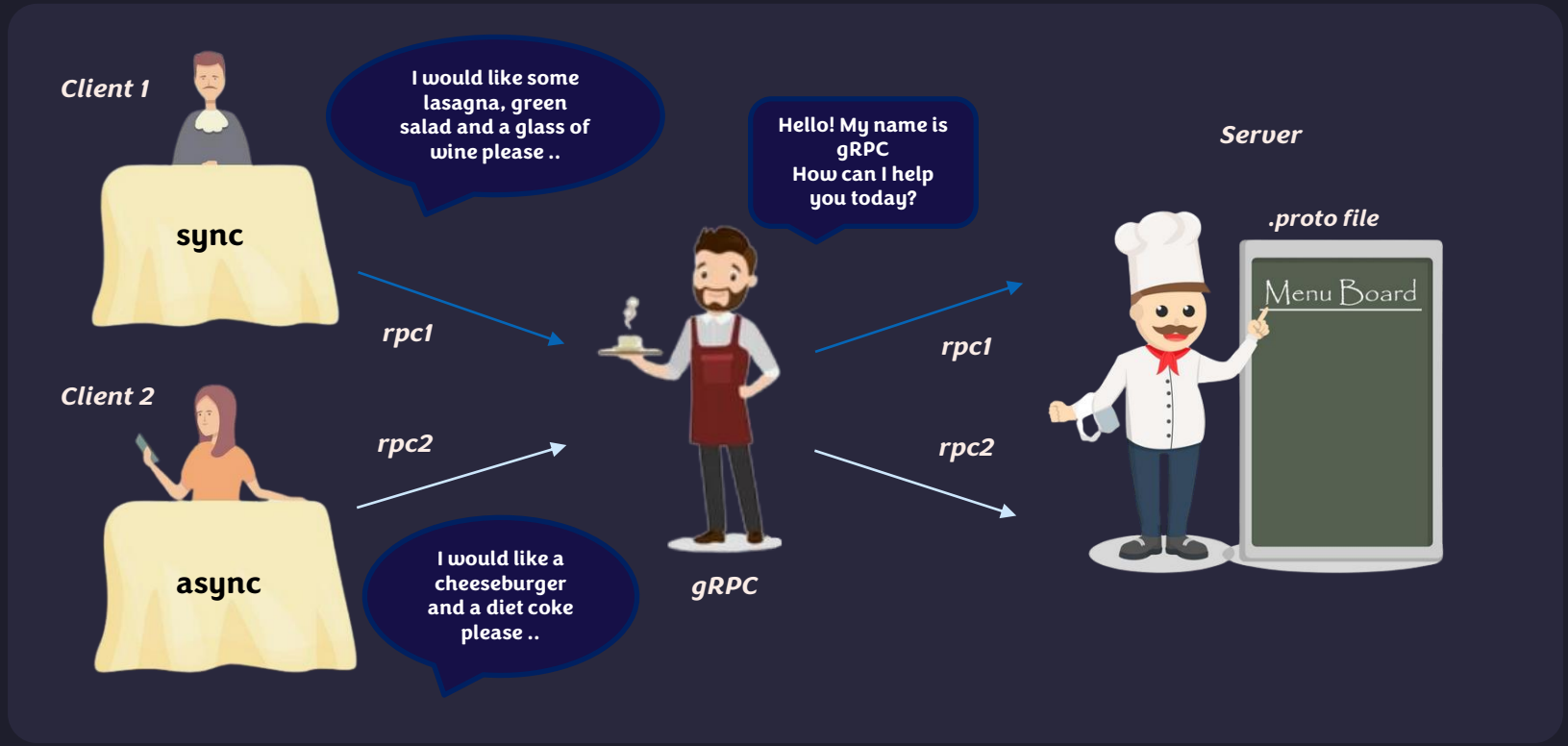
/02 User code is run from the library's own threads

/03 No explicit polling required for notification of completion of RPC actions

/04 The Library is responsible for executing user-specified code at the completion of RPC actions through the usage of reactors

/03
/THE STANDARD APPROACH

The Restaurant Example →

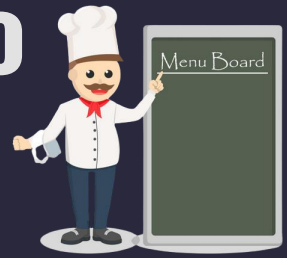


client server architecture using gRPC

OrderService.proto

```proto
syntax = "proto3";

package restaurant;

service OrderService {
  rpc PrepareMeal (Order) returns (Meal) {}
}

message Recipe {
  // ...
}

message Food {
  repeated Recipe recipe = 1;
}

message Order {
  string appetizer = 1;
  string main_course = 2;
  string dessert = 3;
  string drink = 4;
}

message Meal {
  Food food   = 1;
  double cost = 2;
}
```

the .proto file



protoc tool generates

› The data model

› The client library (stub)

used by the client to interact with the server

OrderService.proto

```
// Generated by the protocol buffer compiler.
// source: OrderService.proto

class Recipe final :
    public ::PROTOBUF_NAMESPACE_ID::Message /*
@@protoc_insertion_point(class_definition:restaurant.Recipe) */ {
 // -------- more … ---------------------------------------
};

class Food final :
    public ::PROTOBUF_NAMESPACE_ID::Message /*
@@protoc_insertion_point(class_definition:restaurant.Food) */ {
 // -------- more … ---------------------------------------
};

class Order final :
    public ::PROTOBUF_NAMESPACE_ID::Message /*
@@protoc_insertion_point(class_definition:restaurant.Meal) */ {
 // -------- more … ---------------------------------------
};

class Meal final :
    public ::PROTOBUF_NAMESPACE_ID::Message /*
@@protoc_insertion_point(class_definition:restaurant.Meal) */ {
 // -------- more … ---------------------------------------
};
```

the auto-generated APIs

the data model

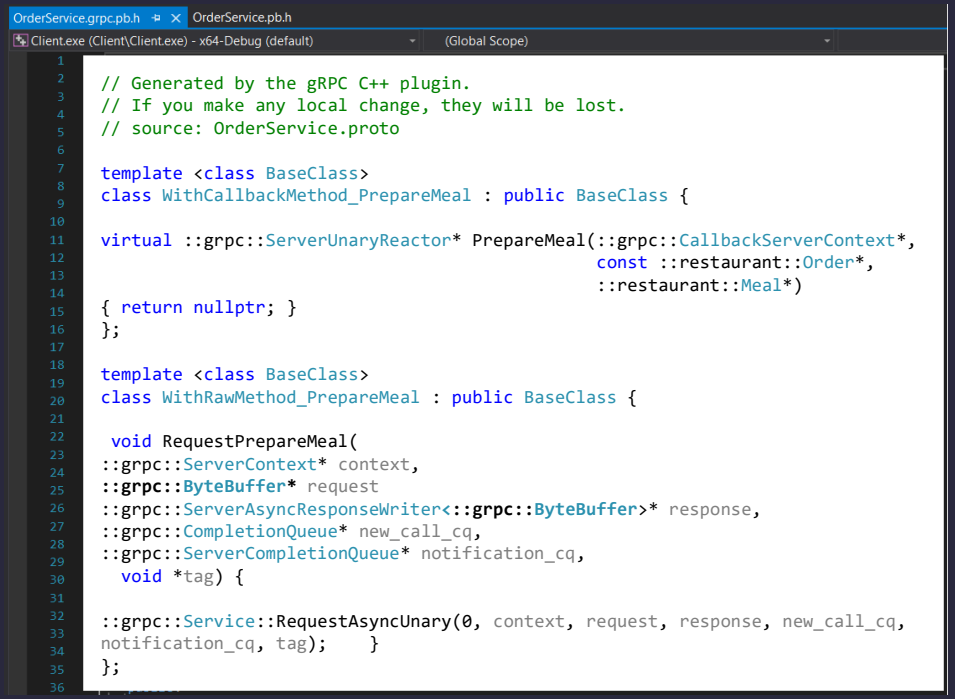
OrderService.grpc.pb.h | OrderService.pb.h

Client.exe (Client\Client.exe) - x64-Debug (default) | (Global Scope)

```
// Generated by the gRPC C++ plugin.
// If you make any local change, they will be lost.
// source: OrderService.proto

class Service : public ::grpc::Service {
public:

virtual ::grpc::Status PrepareMeal(::grpc::ServerContext* context, const
::restaurant::Order* request, ::restaurant::Meal* response);

};

template <class BaseClass>
class WithAsyncMethod_PrepareMeal : public BaseClass {

void RequestPrepareMeal(
::grpc::ServerContext* context,
::restaurant::Order* request
::grpc::ServerAsyncResponseWriter< ::restaurant::Meal>* response,
::grpc::CompletionQueue* new_call_cq,
::grpc::ServerCompletionQueue* notification_cq,
  void *tag)
{
::grpc::Service::RequestAsyncUnary(0, context, request, response,
new_call_cq, notification_cq, tag);
}
};
```

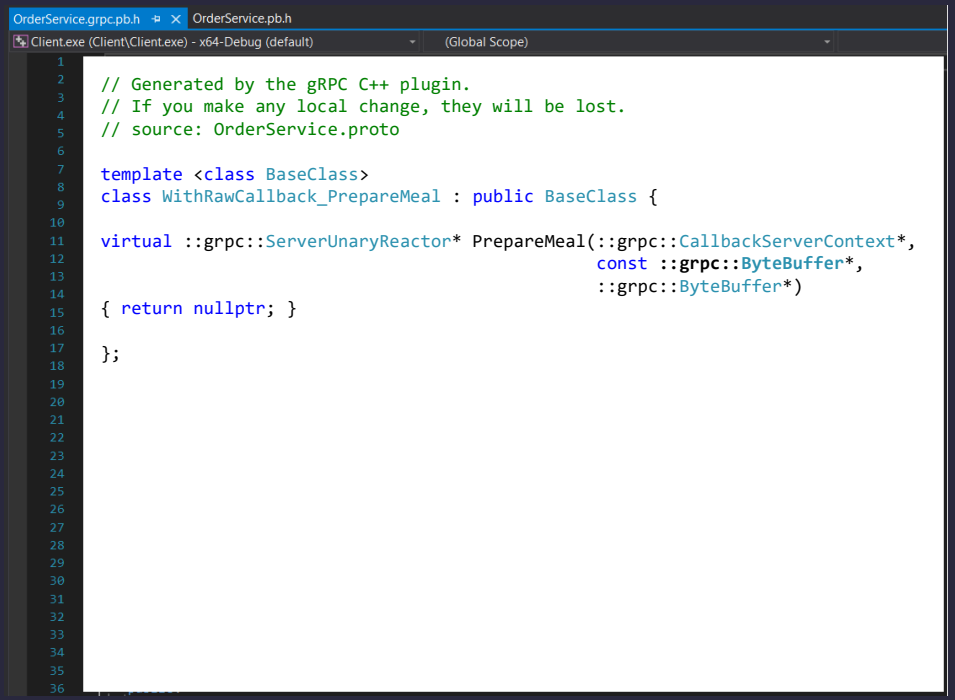
the auto-generated APIs

the service



the auto-generated APIs

the service



the auto-generated APIs

the service

- `grpc::Service`
- `grpc::AsyncService`
- `grpc::CallbackService`
- `grpc::AsyncGenericService`
- `grpc::CallbackGenericService`

the auto-generated APIs

the service

/Your unary sync PrepareMeal method

client

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
class OrderServiceClient
{
public:
OrderServiceClient(std::shared_ptr<grpc::ChannelInterface>
channel) : stub_(OrderService::NewStub(channel)) {}

</*output*/> PrepareMeal(const </*input*/>& input)
{
    Order request;
    Meal reply;
    ClientContext context;

    grpc::Status status =
                stub_->PrepareMeal(&context, request, &reply);
    if (status.ok()) {
        /* … from reply retrieve output*/
        return output;
    } else { /* … do*/ }
}
private:
std::unique_ptr<OrderService::Stub> stub_;
};
```

server

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
#pragma once

using restaurant::Order;
using restaurant::Meal;
using restaurant::OrderService;

using grpc::ServerContext;

class OrderServiceImpl final : public OrderService::Service
{
public:
    grpc::Status PrepareMeal(ServerContext* context,
                             const Order* request,
                             Meal* reply) override
    {
        // ... Create Food ...

        // ... Calculate Cost ...

        // ... Create Meal using Food and Cost ...
    }
};
```

/Your bidi callback PrepareMeal service method

/01 the reactors

client

OrderServiceClientReactor.h

Client.exe (Client\Client.exe) - x64-Debug (default)

client::PrepareMealClientReact

```
class PrepareMealClientReactor : public
grpc::ClientBidiReactor<Order, Meal>
{
public:

explicit PrepareMealClientReactor(OrderService::Stub* stub)
 : order_(new Order), Meal_(new Meal)
{
  // create order

  stub->async()->PrepareMeal(&context_, this);
  NextWrite();
  StartRead(Meal_);
  StartCall();
}

void OnReadDone(bool ok) override { /*…*/ }
void OnWriteDone(bool ok) override { /*…*/ }
void OnDone(const Status& s) override { /*…*/ }

private:
void NextWrite() { /*…*/ }
```

server

PrepareMealReactor.h

Server.exe (Server\Server.exe) - x64-Debug (default)

```
#include "OrderService.grpc.pb.h"

using restaurant::Order;
using restaurant::Meal;
using restaurant::OrderService;
class PrepareMealReactor : public grpc::ServerBidiReactor
                                        <Order, Meal>
{
public:
explicit PrepareMealReactor(): order_(new Order),
                               Meal_(new Meal)
{ StartRead(order_); }

void OnDone() override { /*…*/ }
void OnReadDone(bool ok) override { /*…*/ }
void OnWriteDone(bool ok) override { /*…*/ }

void NextWrite() { /*PrepareMeal Method Implementation*/ }

private:
    Order* order_;
    Meal* Meal_;
};
```

/Your bidi callback PrepareMeal service method

/02 the service

client

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
#pragma once

// --------- include ..........

class OrderServiceClient
{
public:

OrderServiceClient(std::shared_ptr<grpc::ChannelInterface>
                channel) : stub_(OrderService::NewStub(channel)) {}

void PrepareMeal(const OrderInput& input)
{
   PrepareMealClientReactor reactor(stub_.get(), input);

   std::cout << "Your order is getting ready ..." << std::endl;
}

};
```

server

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
#pragma once

#include "PrepareMealReactor.h"

using restaurant::Order;
using restaurant::Meal;
using restaurant::OrderService;

class OrderServiceImpl final :
                        public OrderService::CallbackService
{

public:
ServerBidiReactor<Order, Meal>* PrepareMeal(
                                CallbackServerContext* context)
{
   return new PrepareMealReactor;
}

};
```

/WHY NOT THE AUTO-GENERATED APIs

Server and client applications have a direct gRPC dependency

Server and Client applications already share a data model and they want to make use of already existing classes

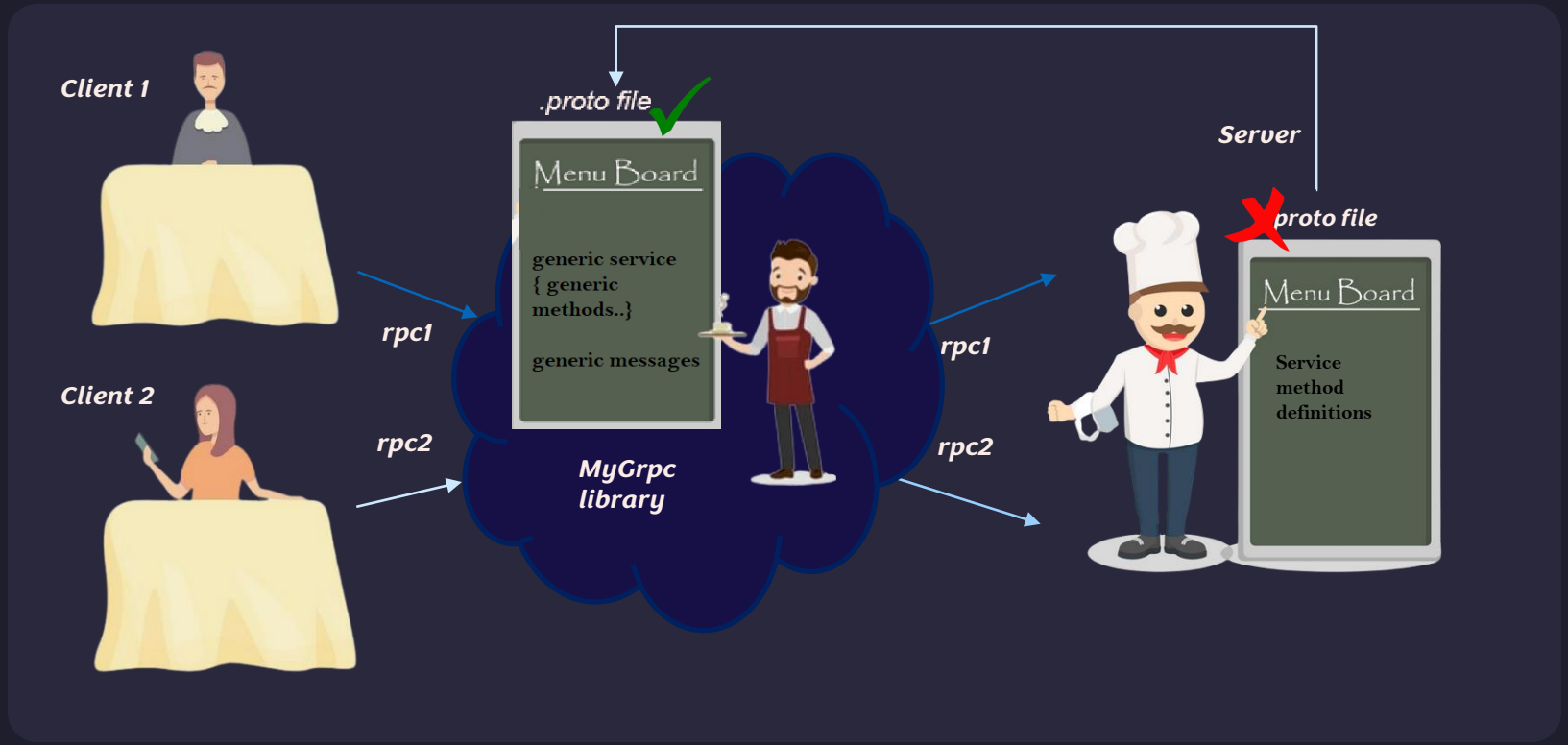
Conversion between the existing and new classes could be complex and inefficient

The existing service APIs may not cover our needs:

Raw/Generic Apis do not cover unary methods
Raw Apis can not be extended (mixing raw with structured data is not feasible)
No clear objective in callback Apis on where the service method definition should be placed

/04
/THE GENERIC APPROACH

The Restaurant example



client server architecture using MyGrpc

OrderService.proto

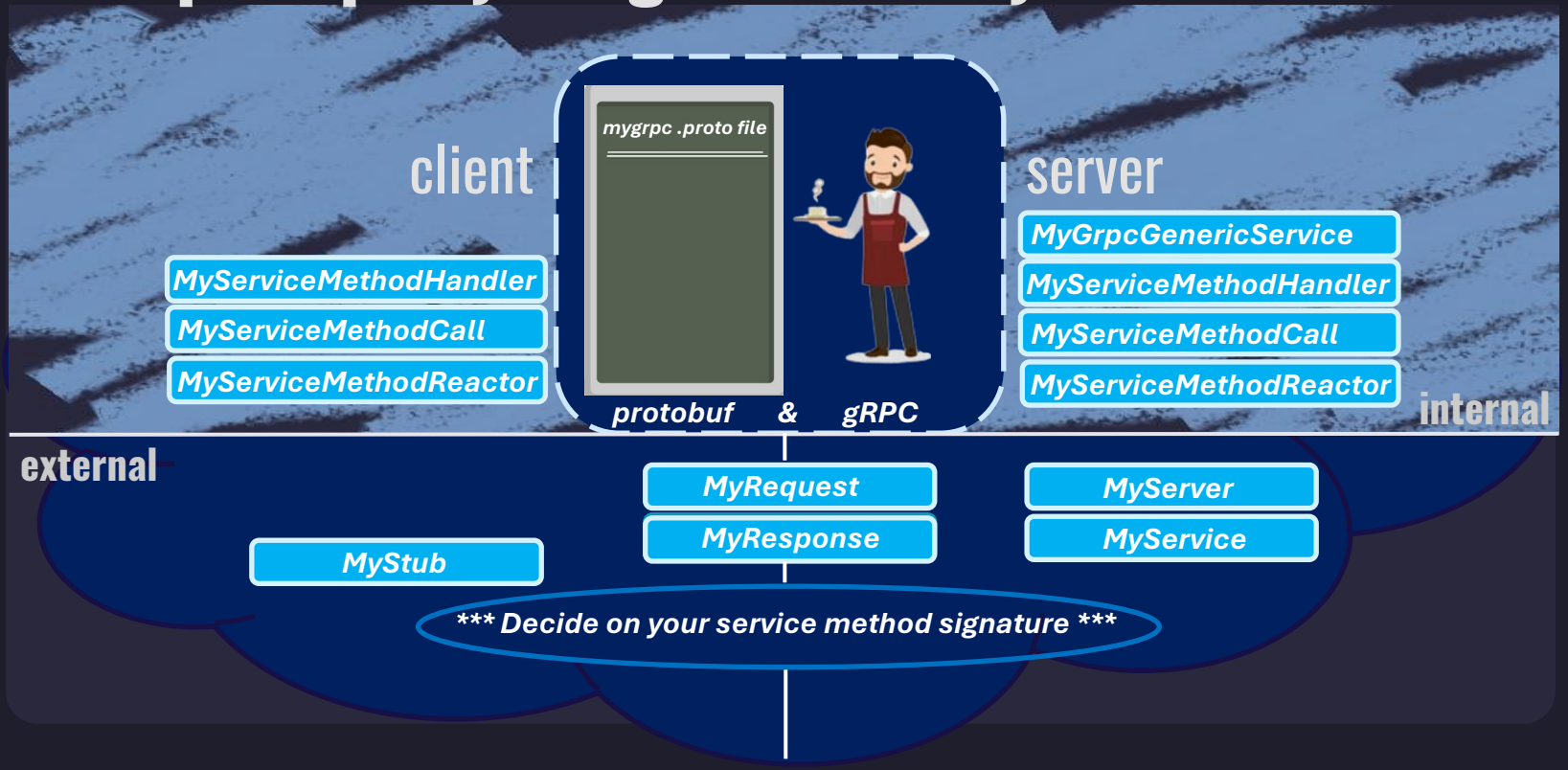
```
1  syntax = "proto3";
2
3  package mygrpc;
4
5  service MyGrpcGenericService
6  {
7    rpc CallFor (stream request) returns (stream response)
8  }
9
10 message data {
11   bytes raw_data = 1;
12 }
13
14 message request {
15   data data    = 1;
16   optional uint32 end  = 2;
17 }
18
19 message response {
20   data data    = 1;
21   optional uint32 end  = 2;
22 }
23
24
25
26
27
```

stream here is used for message chunking

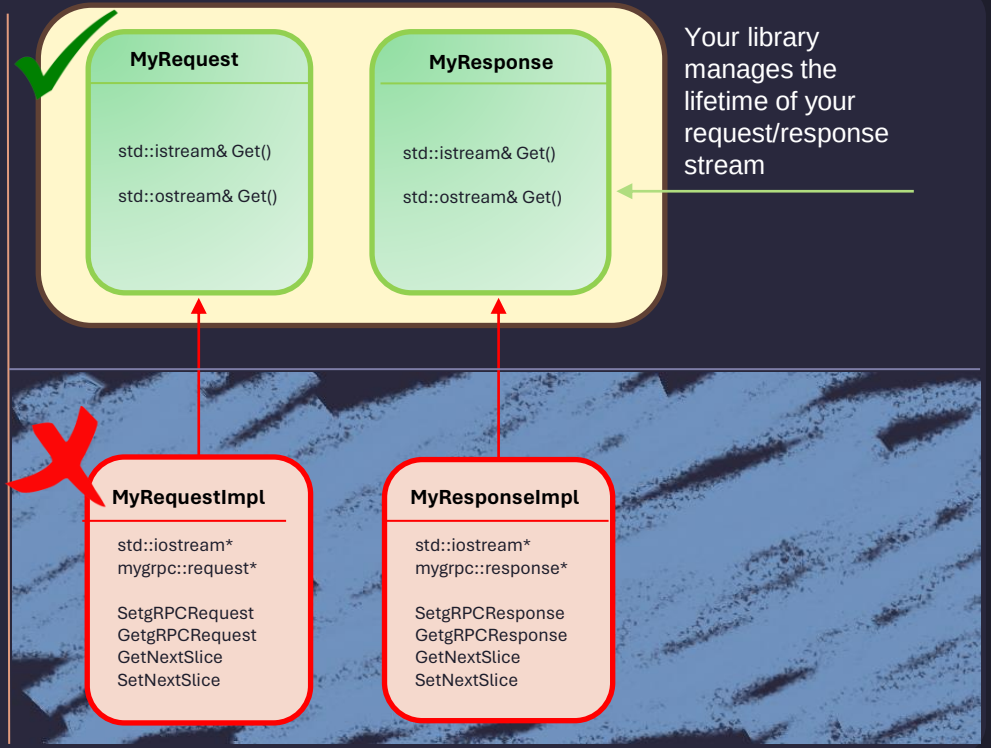
data model providing a buffer and an extra field to denote the end of chunked messages (if any)

the .proto file

/Wrap it up ... your gRPC Library



the data


MyRequest
std::istream& Get()
std::ostream& Get()
MyResponse
std::istream& Get()
std::ostream& Get()
Your library manages the lifetime of your request/response stream
MyRequestImpl
std::iostream*
mygrpc::request*
SetgRPCRequest
GetgRPCRequest
GetNextSlice
SetNextSlice
MyResponseImpl
std::iostream*
mygrpc::response*
SetgRPCResponse
GetgRPCResponse
GetNextSlice
SetNextSlice

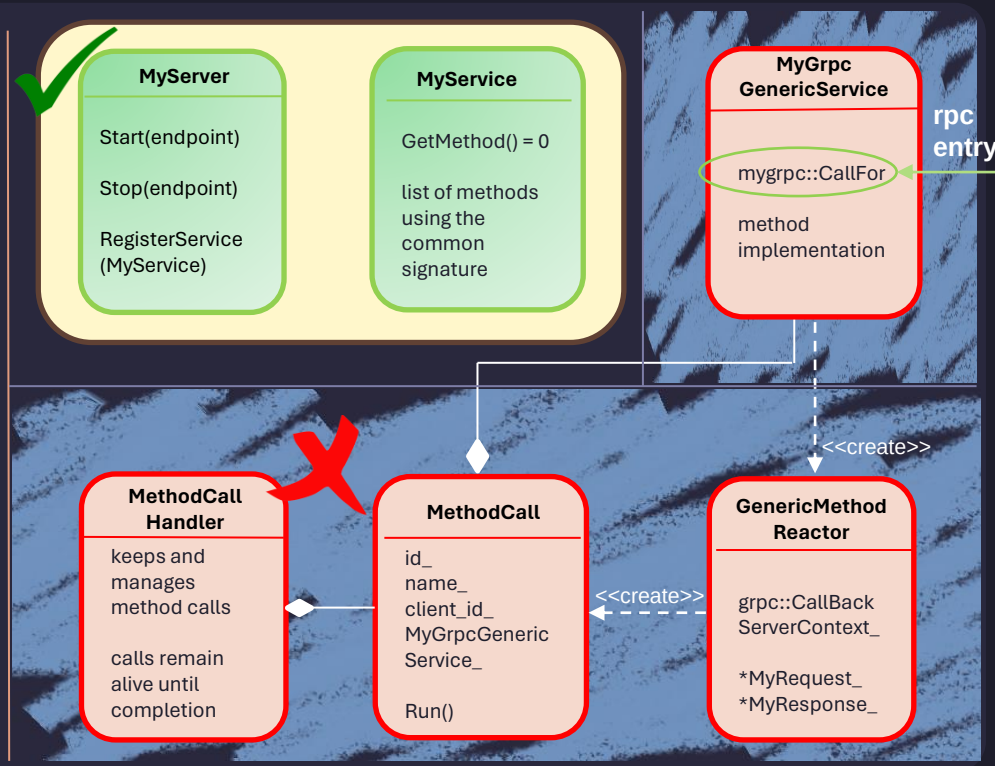
/MyRequest MyReponse

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
class MyRequestImpl
{
public:

  mygrpc::request* GetGrpcRequest()
  void SetGrpcRequest(const mygrpc::request* rq)

  mygrpc::request* GetNextSlice();
  void SetNextSlice(const mygrpc::request* rq);

private:
  std::iostream*   rq_stream_;
  mygrpc::request* rq_message_;
};

class MyReponseImpl
{
public:
// --------- same as above ---------

};
```

the server


MyServer
Start(endpoint)
Stop(endpoint)
RegisterService (MyService)
MyService
GetMethod() = 0
list of methods using the common signature
MyGrpc GenericService
mygrpc::CallFor
method implementation
rpc entry point
<<create>>
MethodCall Handler
keeps and manages method calls
calls remain alive until completion
MethodCall
id_
name_
client_id_
MyGrpcGeneric Service_
Run()
<<create>>
GenericMethod Reactor
grpc::CallBack ServerContext_
*MyRequest_
*MyResponse_

/MyGrpcGenericService

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
class MyGrpcGenericService : public grpc::Service
{
public:

  // --------- Unary Sync ---------
  virtual grpc::Status CallFor(grpc::ServerContext* context,
                               const ::mygrpc::request* request,
                               ::mygrpc::response* response);

  // --------- Bidi Sync ---------
  virtual grpc::Status CallFor(::grpc::ServerContext*,
                               ::grpc::ServerReaderWriter< ::mygrpc::response, ::mygrpc::request>*)};


 // --------- Bidi Callback ---------
  virtual grpc::ServerBidiReactor< ::mygrpc::request, ::mygrpc::response>*
                       CallFor(grpc::CallbackServerContext* context)

  {   return new MyServerMethodReactor(context, this);  }
```

/MyServerMethodReactor

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
class MyServerMethodReactor : public grpc::ServerBidiReactor< ::mygrpc::request, ::mygrpc::response >
{

public:

  explicit MyServerMethodReactor(grpc::CallbackServerContext* context, MyGenericGrpcService* service);

  void OnDone() override;

  void OnReadDone(bool ok) override;

  void OnWriteDone(bool ok) override;

  void OnCancel() override;

private:
  void NextWrite();

  bool Run()

};
```

Here you'll end up calling your method by name through your service

/MyService Interface

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
using MethodFunction = std::function<bool(const MyRequest* request, MyResponse* response)>;

class MyService
{
public:

  virtual MethodFunction GetMethod(const std::string& method_name) = 0;
}
```

/MyOrderService Implementation

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
using MethodFunction = std::function<bool(const MyRequest* request, MyResponse* response)>;

class MyOrderService : public mygrpc:: MyService
{
public:
  virtual MethodFunction GetMethod(const std::string& method_name) override
  {
     return std::bind(&MyOrderService::PrepareMeal, this, std::placeholders::_1, std::placeholders::_2);
  };

  bool PrepareMeal(const MyRequest* request, MyResponse* response)
  {
     // Get stream from request - Deserialize request using your data model deserializer

     // Prepare Meal (calculate response)

     // Get stream from response - Serialize response using your data model serializer
  };
}
```

/MyServer

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

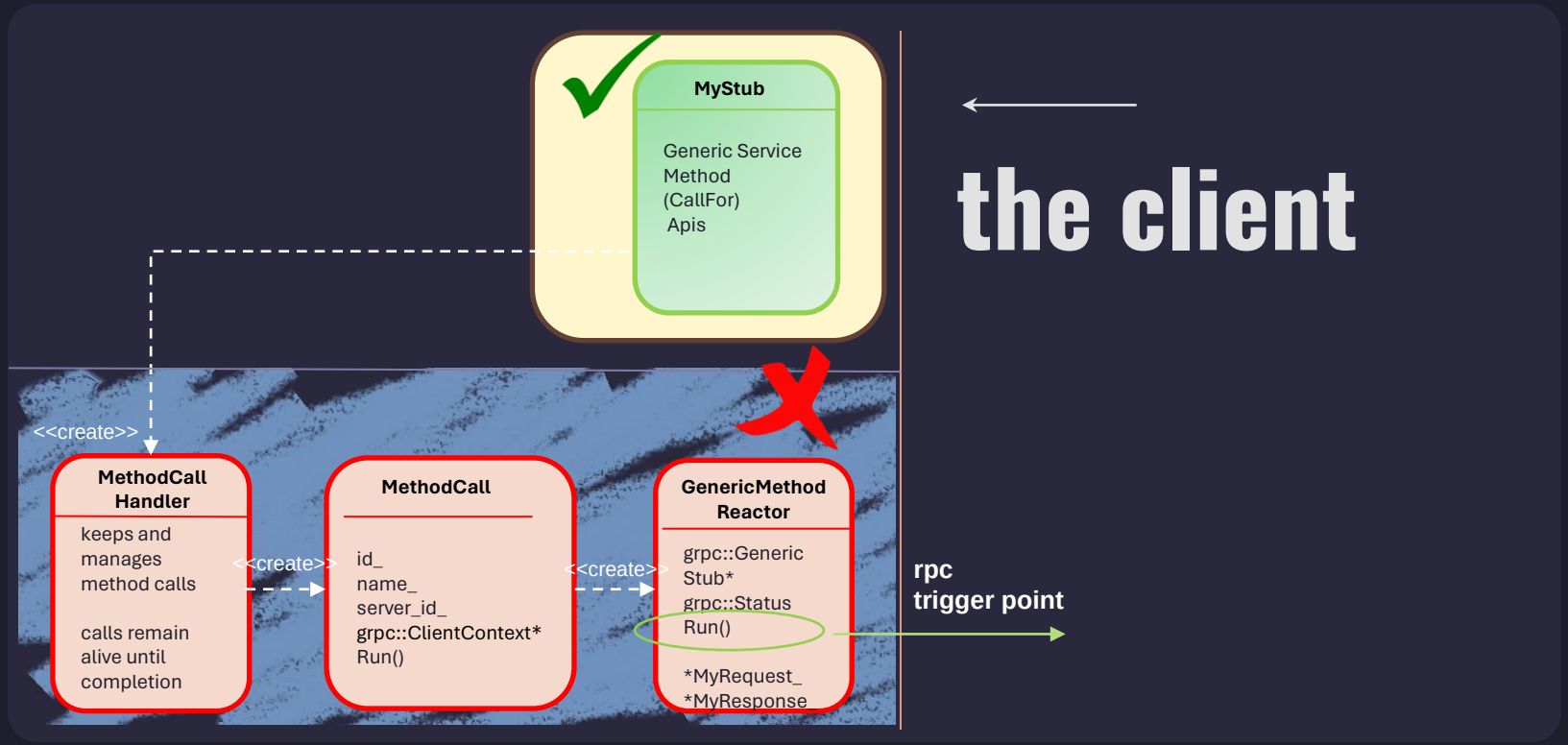
```
class MyServer
{
public:

 void Start() // The start process is similar to an exposed grpc RunServer process
 {
     // 1. Configure mygrpc generic service

     // 2. Register mygrpc generic service

     // 3. Start grpc server
  };



};
```

/Configuring generic method

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
// 1. Configure generic method CallFor as a bidi streaming callback service method

grpc::internal::CallbackBidiHandler<mygrpc::request, mygrpc::response>* handler
    = new grpc::internal::CallbackBidiHandler<mygrpc::request, mygrpc::response>
    ([this](grpc::CallbackServerContext* context) { return service_->CallFor(context); });

grpc::internal::RpcServiceMethod* service_method = new
grpc::internal::RpcServiceMethod("/mygrpc.MyGrpcGenericService/CallFor",
                                 grpc::internal::RpcMethod::RpcType::BIDI_STREAMING, handler);

service_method->SetServerApiType(grpc::internal::RpcServiceMethod::ApiType::CALL_BACK);
service_->AddServiceMethod(service_method);

// 2. Register MyGrpcGenericService

MyGrpcGenericService service;
ServerBuilder builder;
builder.RegisterService(&service);

// 3. Start grpc Server

std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
server->Wait();
```


MyStub
Generic Service Method (CallFor) Apis
the client
<<create>>
MethodCall Handler
keeps and manages method calls
calls remain alive until completion
<<create>>
MethodCall
id_
name_
server_id_
grpc::ClientContext*
Run()
<<create>>
GenericMethod Reactor
grpc::Generic Stub*
grpc::Status
Run()
*MyRequest_
*MyResponse
rpc
trigger point

/MyStub

MygRPCService.pb.h

Client.exe (Client\Client.exe) - win64d | mygrpc::request | _internal_end() const

```
class MyStub
{
public:

mygrpc::Status CallForBlocking(const std::string& service,
                               const std::string& method,
                               std::shared_ptr<mygrpc::Request> rq,
                               std::shared_ptr<mygrpc::Response> rp,
                               );

unsigned int CallForCallback(const std::string& service,
                             const std::string& method,
                             std::shared_ptr<mygrpc::Request> rq,
                             std::function<bool(const mygrpc::Response* r,
                                                const mygrpc::Status s)>
                             );

};
```

1. choose to return a status or an rpc call id
2. choose among Blocking/Callback Apis or combine the two
3. choose to add a callback even for unary rpcs
4. choose the signature of your callback
5. embed your own signaling mechanism omitting the callback function

P1: use a vector of requests for streaming calls

P2: use a context if you want to exchange metadata at rpc level

/04

/BEFORE

... AND AFTER

/Your bidi callback PrepareMeal service method

/01 the reactors

client

OrderServiceClientReactor.h

Client.exe (Client\Client.exe) - x64-Debug (default)

client::PrepareMealClientReact

```
class PrepareMealClientReactor : public
grpc::ClientBidiReactor<Order, Meal>
{
public:

explicit PrepareMealClientReactor(OrderService::Stub* stub)
 : order_(new Order), Meal_(new Meal)
{
  // create order

  stub->async()->PrepareMeal(&context_, this);
  NextWrite();
  StartRead(Meal_);
  StartCall();
}

void OnReadDone(bool ok) override { /*…*/ }
void OnWriteDone(bool ok) override { /*…*/ }
void OnDone(const Status& s) override { /*…*/ }

private:
void NextWrite() { /*…*/ }
```

server

PrepareMealReactor.h

Server.exe (Server\Server.exe) - x64-Debug (default)

```
#include "OrderService.grpc.pb.h"

using restaurant::Order;
using restaurant::Meal;
using restaurant::OrderService;
class PrepareMealReactor : public grpc::ServerBidiReactor
                                        <Order, Meal>
{
public:
explicit PrepareMealReactor(): order_(new Order),
                               Meal_(new Meal)
{ StartRead(order_); }

void OnDone() override { /*…*/ }
void OnReadDone(bool ok) override { /*…*/ }
void OnWriteDone(bool ok) override { /*…*/ }

void NextWrite() { /*PrepareMeal Method Implementation*/ }

private:
    Order* order_;
    Meal* Meal_;
};
```

/Your bidi callback PrepareMeal service method

/02 the service

client

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
#pragma once

// --------- include ..........

class OrderServiceClient
{
public:

OrderServiceClient(std::shared_ptr<grpc::ChannelInterface>
                channel) : stub_(OrderService::NewStub(channel)) {}

void PrepareMeal(const OrderInput& input)
{
   PrepareMealClientReactor reactor(stub_.get(), input);

   std::cout << "Your order is getting ready ..." << std::endl;
}

};
```

server

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | { } server

```
#pragma once

#include "PrepareMealReactor.h"

using restaurant::Order;
using restaurant::Meal;
using restaurant::OrderService;

class OrderServiceImpl final :
                        public OrderService::CallbackService
{

public:
ServerBidiReactor<Order, Meal>* PrepareMeal(
                                CallbackServerContext* context)
{
   return new PrepareMealReactor;
}

};
```

/Calling for PrepareMeal

client

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | server

```
void PrepareMealCallback(const MyResponse* r,
                         const mygrpc::Status s))
{
   // PrepareMeal Callback
}


// Create request (using your own serializer)

// Create your stub

unsigned int callId = mystub->
                  CallFor("OrderService",
                          "PrepareMeal",
                          request,
                          std::bind(&PrepareMealCallback,
                          std::placeholders::_1,
                          std::placeholders::_2)
```

server

OrderService.h | OrderService.grpc.pb.h | OrderService.pb.h

Server.exe (Server\Server.exe) - x64-Debug (default) | server

```
using MethodFunction = std::function<bool(const MyRequest* rq,
                                          MyResponse* rp)>;

class MyOrderService : public mygrpc:: MyService
{
public:
  virtual MethodFunction GetMethod(const std::string& m)
                                                   override
  {
     return std::bind(&MyOrderService::PrepareMeal,
                      this,
                      std::placeholders::_1,
                      std::placeholders::_2);
  };

  bool PrepareMeal(const MyRequest* request,
                   MyResponse* response)
  {
     // Prepare Meal (calculate response)
  };
}
```

/WHY THE GENERIC APPROACH

No direct gRPC dependency both on the client and server side.

Clients can decide on a sync or async (/callback) mode.

Gives us the ability to group multiple methods with different Api types under the same generic service.

More intuitive client APIs. Cleaner Code

Gives us the ability to customize (enable/disable extra functionalities) for multiple services or service methods (e.g., message chunking, progress callback)

/THE CONS ..

Existing classes need to provide their own serializing mechanism

Increases development cost

This is a C++ only solution. For multi-language support the library needs to provide its own bindings

/05 /SUMMARY

- the standard approach of using gRPC
- an alternative approach to follow up with requirements
- the extra bonus of flexibility added along the way

→ is it worthed?

#include "THANK YOU!"

#pragma once

#include "QUESTIONS?"