A gentle introduction to gRPC in C++

Marco Arena – marco@italiancpp.org

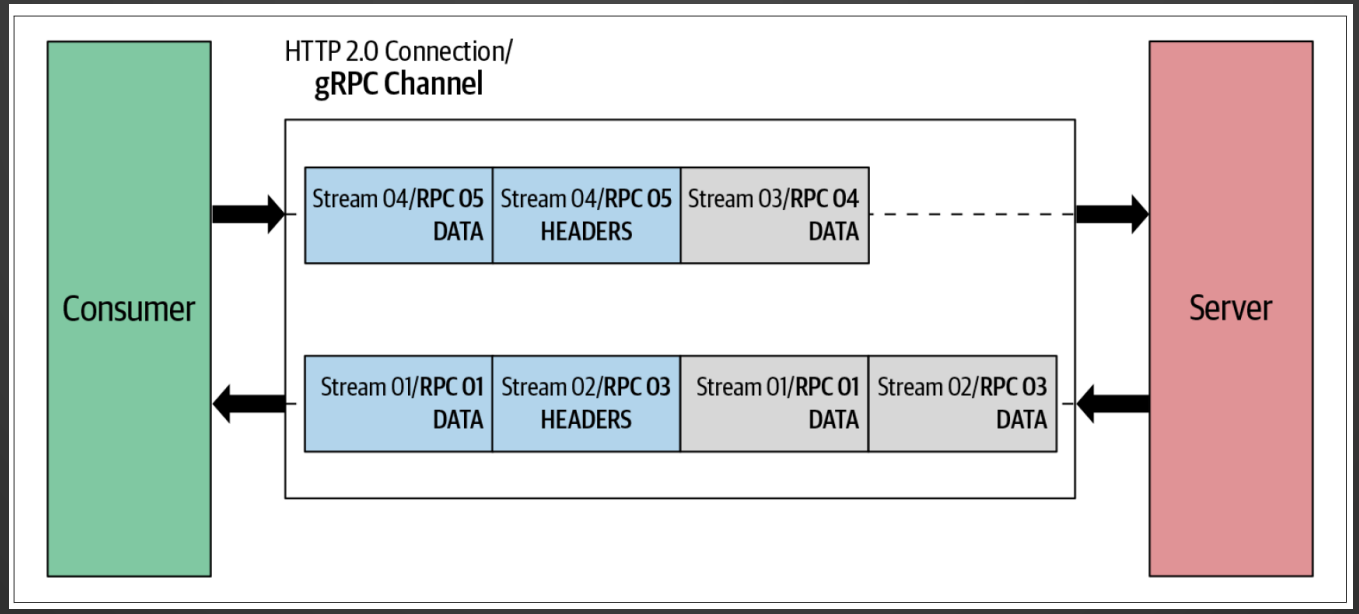
What is gRPC

- Modern inter-process communication framework for connecting over networks applications and micro-services
- As easy as making a function call
- Can operate in synchronous or asynchronous* mode
- Supported by a variety of languages
- **g** stands for something different in every release

*) we won't see the C++ **async** API today, since this is an introductionary talk

gRPC internals in a nutshell

- Built on top of:
  - Protocol Buffers, for data serialization
  - HTTP/2, for data transfer



Courtesy of *"gRPC: Up and Running"*

How to develop a gRPC++ application

To obtain, build, and use gRPC for C++, a simple option is to use `vcpkg` (either you are on Windows, Linux, Mac, etc)

```
vcpkg install grpc --triplet x64-windows
```

grpc | Version: 1.41.0

An RPC library and framework

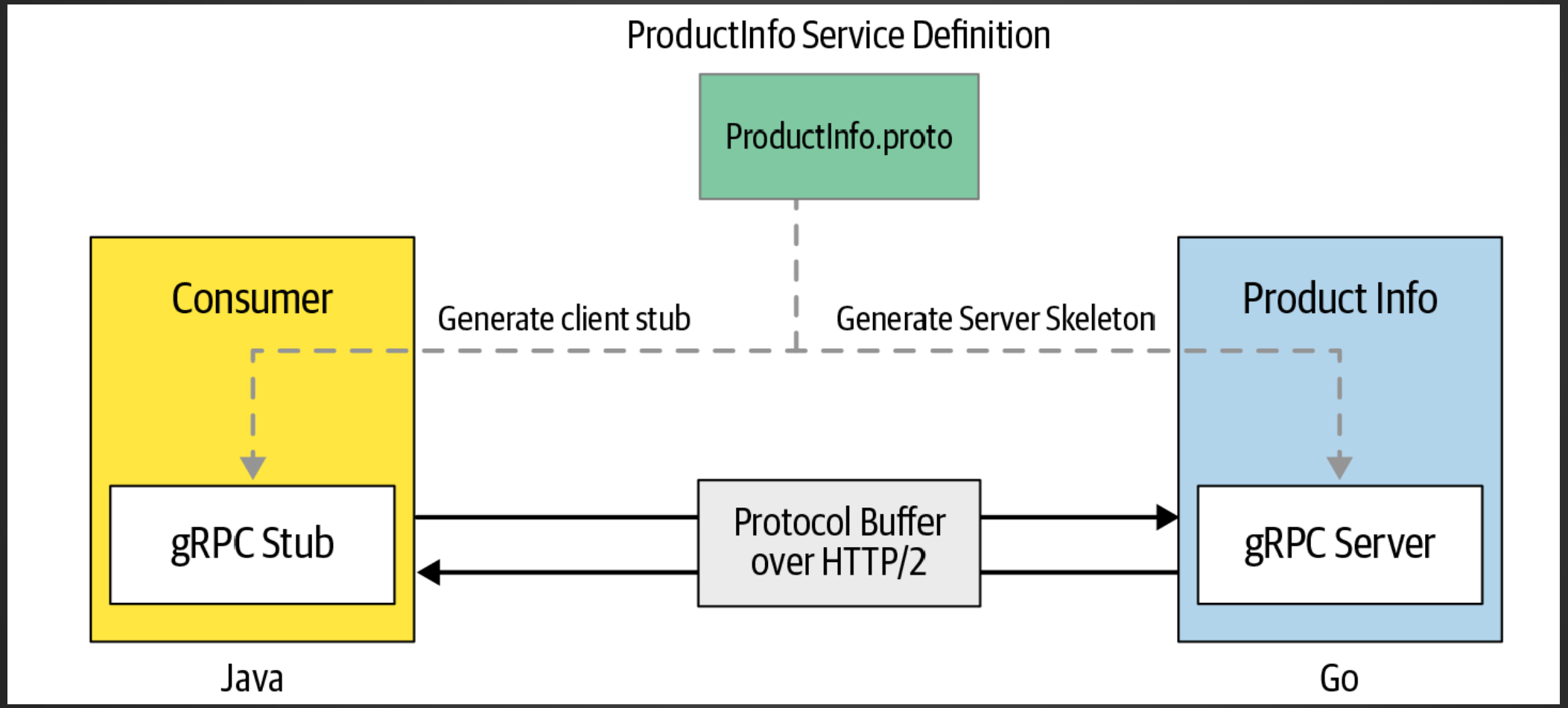
Compatibility: ✓ arm-uwp ✓ arm64-windows ✓ x64-linux ✓ x64-osx ✓ x64-uwp ✓ x64-windows ✓ x64-windows-static ✓ x86-windows

Hide Details

How to develop a gRPC++ application

1. Define a service interface
   - Aka: how your service can be consumed by clients
   - Declare methods that can be called (names, parameters, formats, etc)
   - *Interface Definition Language* (IDL) is protocol buffers by default (but it can be different)
2. Generate *server skeleton* or *client stub*
   - You can generate code for any of the supported programming languages
3. Implement the actual logic

How to develop a gRPC++ application



Courtesy of *"gRPC: Up and Running"*

Hello gRPC++

1

```
syntax = "proto3";

service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string name = 1;
}

message HelloResponse {
  string message = 1;
}
```

hello.proto

2

```
protoc -I . --grpc_out=. --plugin=protoc-gen-grpc=grpc_cpp_plugin.exe hello.proto

protoc -I . --cpp_out=. hello.proto
```

Hello gRPC++ / Server

3

```
class HelloServiceImpl final : public HelloService::Hello {
  Status SayHello(ServerContext*, const HelloRequest* request, HelloResponse* reply) override {
     reply->set_message(format("Hello {}!", request->name()));
     return Status::OK;
  }
};
// ...
HelloServiceImpl service;
grpc::ServerBuilder builder;
builder.AddListeningPort("localhost:50051", grpc::InsecureServerCredentials());
builder.RegisterService(&service); // yes, you can even register more services...(multiplexing)
auto server = builder.BuildAndStart();
std::cout << "The service is listening! Press Enter to shutdown\n";
std::cin.get();
server->Shutdown();
server->Wait();
```

server.cpp

Service vs Server

Service:

- provides rpc methods declared in the service definition
- has a name

Server:

- exposes one or more services
- can be contacted at one or more endpoints

Hello gRPC++ / Client

4

```
auto channel = grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials());
auto stub = HelloService::NewStub(std::move(channel));
ClientContext ctx;
HelloRequest request;
request.set_name("Marco");
HelloResponse response;
if (const auto status = stub->SayHello(&ctx, request, &response); status.ok())
{
    std::cout << response.message() << "\n";
}
else
{
    std::cout << "error communicating to the server: " << status.error_code() << "\n";
}
```

client.cpp

Sync vs Async in a nutshell

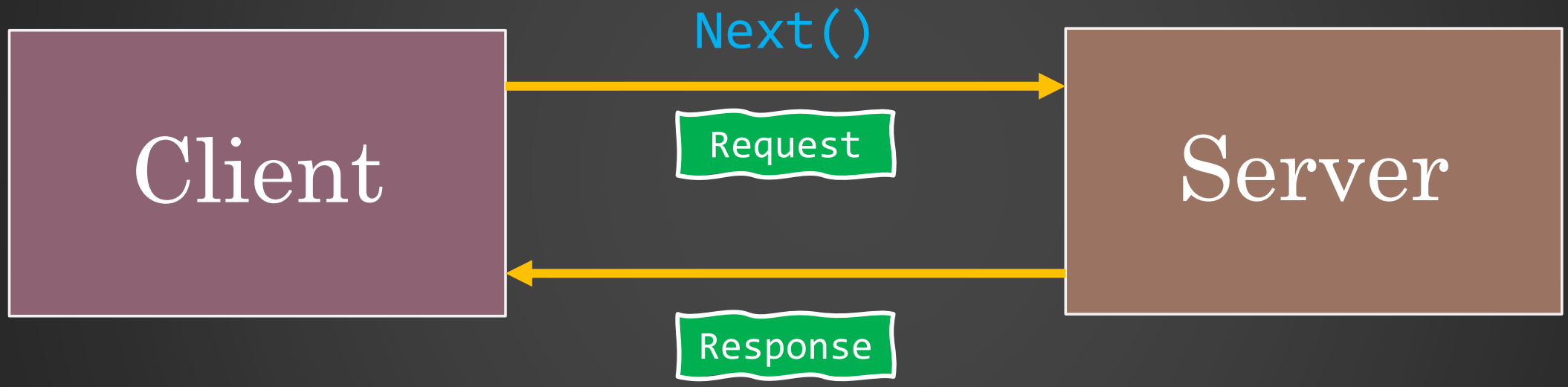
- **Client-side sync API:** every rpc call waits for the server to respond (either with a response or an error)
- **Server-side sync API:** gRPC handles automatically rpc calls employing a worker thread per request (picking one from a customizable thread pool)
- **Async API:** programmers can use their own threading model to manage requests/rpc calls

Advantages of gRPC

- Efficient
- Contract-first
- Strongly typed
- Polyglot
- Supports duplex streaming
- Mature and widely adopted

Communication Patterns

Unary RPC

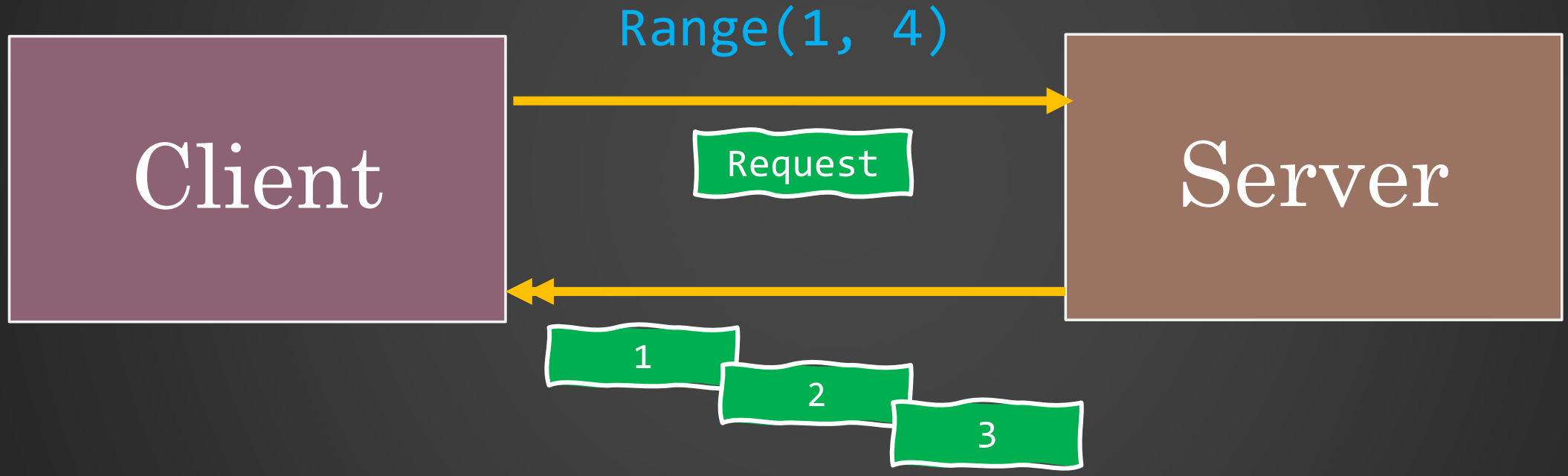


Unary RPC

- It's like the hello world service method we have seen so far

```
service NumberService {
  rpc Next (NumberRequest) returns (NumberResponse);
  rpc Range (RangeRequest) returns (stream RangeResponse);
  rpc Sum (stream SumRequest) returns (SumResponse);
}
```

Server-Streaming RPC



Server-Streaming RPC

- We decorate the method return message with the keyword `stream`

```
service NumberService {
  rpc Next (NumberRequest) returns (NumberResponse);
  rpc Range (RangeRequest) returns (stream RangeResponse);
  rpc Sum (stream SumRequest) returns (SumResponse);
}
```

Server-Streaming Example

- In order to stream data back to the client, we use `ServerWriter`:
  - `Write()` every single data
  - Return a status when done (`OK`) or in case of error

```
Status Range(ServerContext*, const RangeRequest* request, ServerWriter<RangeResponse>* writer) override {
    RangeResponse response;
    for (auto i=request->min(); i<request->max(); ++i) {
        response.set_value(i);
        writer->Write(response);
    }
    return Status::OK;
}
```

SERVER

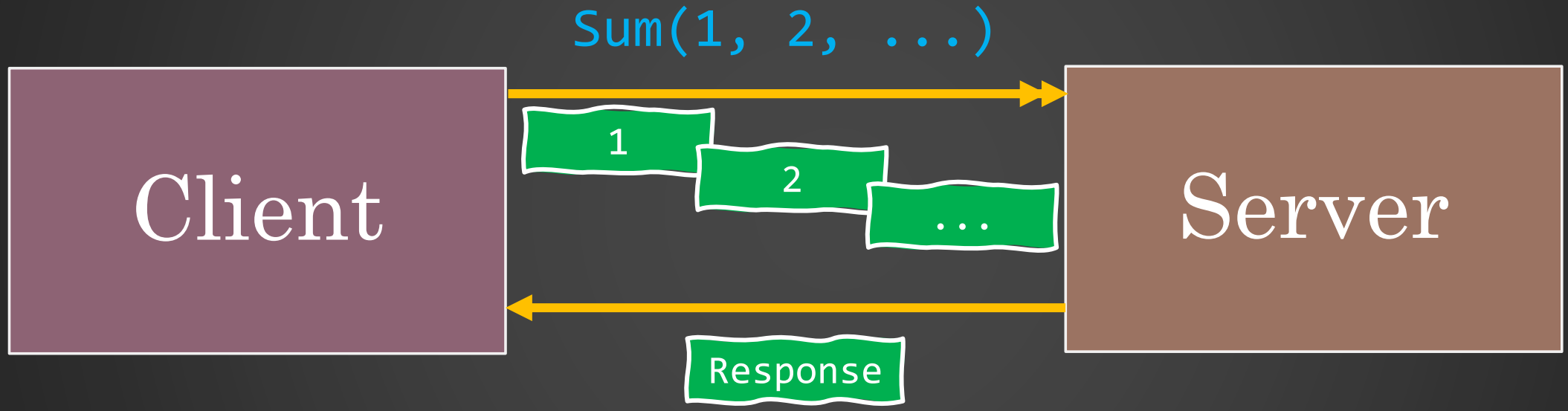
Server-Streaming Example

- In order to get data from to the server, we use a `ClientReaderInterface`:
  - `Read()` until there are no more messages or the stream has ended (aka: returns `false`)
  - Then call `Finish()` to complete the call and get the status

```
RangeRequest request;
request.set_min(min); request.set_max(max);
ClientContext ctx;
auto reader = stub->Range(&ctx, request);
RangeResponse response;
while (reader->Read(&response)) {
    // do something with response.value() ...
}
const auto status = reader->Finish();
```

CLIENT

Client-Streaming RPC



Client-Streaming Example

- We decorate the method parameter with the keyword `stream`

```
service NumberService {
  rpc Next (NumberRequest) returns (NumberResponse);
  rpc Range (RangeRequest) returns (stream RangeResponse);
  rpc Sum (stream SumRequest) returns (SumResponse);
}
```

Client-Streaming Example

- In order to get data from the client, we use `ServerReader`:
  - `Read()` until there are no more messages or the stream has ended (aka: returns `false`)
  - Then fill the response and return the status

```
Status Sum(ServerContext*, ServerReader<SumRequest>* reader, SumResponse* response) override {
    uint64_t sum = 0;
    SumRequest request;
    while (reader->Read(&request))
    {
        sum += request.value();
    }
    response->set_value(sum); // fill the response
    return Status::OK;
}
```

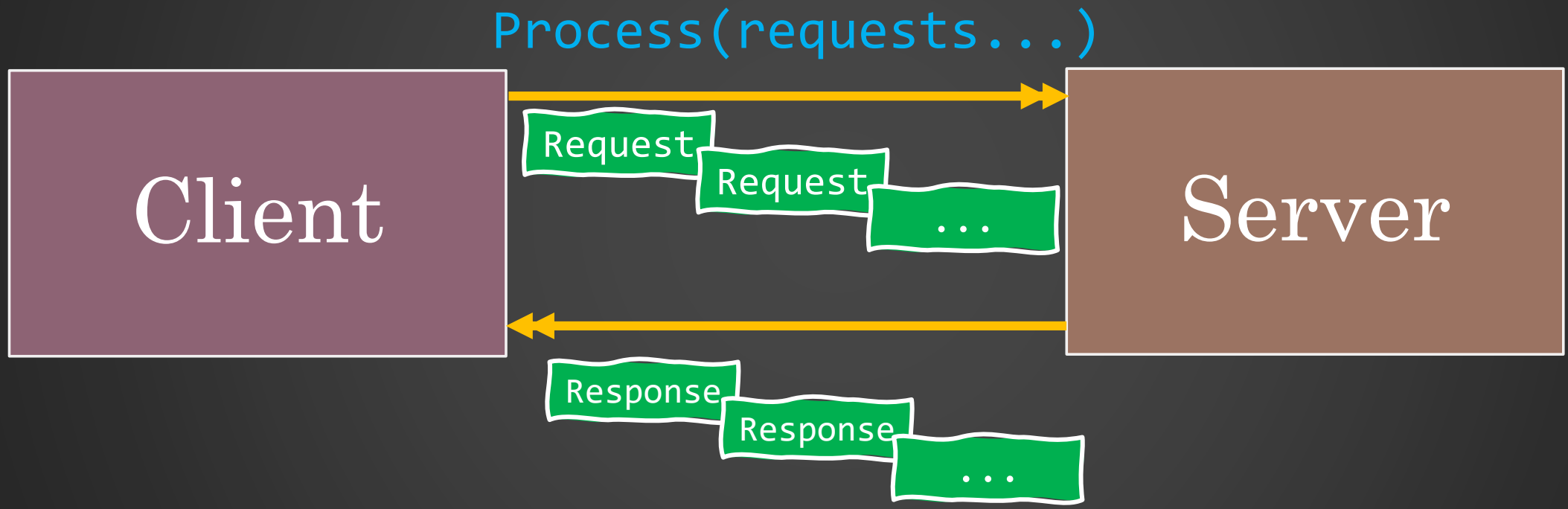
SERVER

Client-Streaming Example

- In order to push data to the server, we use a `ClientWriterInterface`:
  - `Write()` every single data (when returns `false`, the stream has been closed)
  - call `WritesDone()` to let gRPC know we have done
  - finally, call `Finish()` to complete the call and get the status

```
ClientContext ctx; SumResponse response; SumRequest request;
auto writer = stub->Sum(&ctx, &response);
for (auto i : values) {
     request.set_value(i);
     writer->Write(request);
}
writer->WritesDone();
const auto status = writer->Finish();
const auto value = response.value();
```

Bidirectional-Streaming RPC



Bidirectional-Streaming RPC

```
syntax = "proto3";

service SnowflakeServer {
  rpc NextId (stream NextIdRequest) returns (stream NextIdResponse);
}

message NextIdRequest {
}

message NextIdResponse {
  int64 value = 1;
}
```

Tips & Tricks

Unimplemented calls

```
class HelloServiceImpl final : public HelloService::Service
{
    // this is not implementing SayHello at all...
};


// What if the client calls SayHello?
```

It gets an `UNIMPLEMENTED` error!

Deadlines

```
auto channel = grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials());
auto stub = HelloService::NewStub(std::move(channel));
ClientContext ctx;
ctx.set_deadline(std::chrono::system_clock::now() + 1s);
HelloRequest request;
request->set_name("Marco");
HelloResponse response;
if (auto status = stub->SayHello(&ctx, request, &response); !status.ok()) // at most 1 second
{
     cout << "error communicating to the server. Error: " << status.error_code(); // DEADLINE_EXCEEDED
}
else
{
     cout << response->message();
}
```

Deadlines on channel connection

```
shared_ptr<grpc::Channel> CreateChannelOrThrow(const string& target)
{
    auto channel = CreateChannel(target, grpc::InsecureChannelCredentials());
    static constexpr auto deadline = 3s;
    if (!channel->WaitForConnected(chrono::system_clock::now() + 3s))
    {
        throw runtime_error(format("Cannot connect to {} within {}", target, deadline));
    }
    return channel;
}
```

Channels

```
// suppose the server is down here
auto channel = CreateChannel(target, grpc::InsecureChannelCredentials());
auto stub = hello::NewStub(std::move(channel));
// ...
// ... after some time the server wakes up ...
// ...
stub->SayHello(&ctx, request, &response); // what happens here?
```

Connections (and reconnections) are handled automatically and hidden to the user!

Channels in details

- A **gRPC channel** is an interface for interacting with one or more **gRPC services** on a specified host and port, encapsulating a range of functionality (e.g. **name resolution**, **TCP backoff**, **TLS handshakes, flow control**)
- **Can be shared** across multiple **client stubs**
- **Automatically handle errors on established connections and possibly reconnect**
- Roughly speaking, they are modelled as a **state machine** with 5 states:

  `CONNECTING`, `READY`, `TRANSIENT_FAILURE`, `IDLE`, `SHUTDOWN`

- gRPC provides an API to **query the channel state** / **subscribe on changes**
  (in C++, both sync and async versions are provided)

Cancellation cheatsheet

- TCP connection lifecycle is not exposed to the programmer (gRPC handles all the details)
- On the other hand, gRPC exposes the **stream** lifecycle (e.g. an RPC call)
- To find out if a stream has been **terminated**, we can explicitly check:
  - `{Server|Client}Context::IsCancelled()`
  - On server-streaming, the result of `Read()` (or `Write()`)
  - On client-streaming, the result of `Read()` + `Finish()` (or `Write()` + `WritesDone()` + `Finish()`)
- To explicitly terminate a stream, we can use `{Server|Client}Context::TryCancel()`
- Also, someone adds **graceful cancellation** mechanism directly into the service protocol

Cancellation – Stop a blocking `Read`

```
ClientContext context;
auto reader = jthread{ [&] {
          ReadRequest request;
          auto clientReader = Stub->StreamingOperation(&context, request);
          StreamingResponse response;
          while (reader->Read(&response))
          {
               std::cout << format("got data={}\n", response.message());
          }
          const auto status = reader->Finish();
} };

// ...
context.TryCancel(); // warning: it's best-effort and may throw an exception
```

Cancellation – Explicit check

```
void ClientHandler::timeout()
{
     if (context->IsCancelled())
     {
          // the client has gone
     }
}
```

Cancellation – Explicit `Write` check

```
void Service::StreamMessage()
{
     if (!writer->Write(response)) // writer is ServerWriter<SomeMessage>*
     {
          // the client has gone...
     }
}
```

Health Checking Protocol

- gRPC defines a health checking protocol that allows services to expose their status in a standard way
- Also, you can easily add a default `HealthCheckingService` to your own server (however, shutting down does not stop `Watch` streaming)

```
grpc::EnableDefaultHealthCheckService(true);
HelloServiceImpl service;
grpc::ServerBuilder builder;
builder.AddListeningPort("localhost:50051", grpc::InsecureServerCredentials());
builder.RegisterService(&service);
auto server = builder.BuildAndStart();
server->GetHealthCheckService()->SetServingStatus("HelloService", true);
std::cout << "The service is listening! Press Enter to shutdown\n";
...
```

Error handling

- gRPC calls return an instance of type `grpc::Status` that contains:
  - `error_code` (gRPC provides 17 standard error codes)
  - `error_message`
  - `error_details` (this is available **only if IDL is protobuf**)

```
Status SayHello(ServerContext*, const HelloRequest* request, HelloResponse* reply) override
{
    return Status(StatusCode::INVALID_ARGUMENT, "Some error...");
}
```

- Other approaches:
  - Metadata
  - Additional data into response messages

Testing – Server

A few options available here:

1. Craft ad-hoc clients (not necessarily written in C++) that connect to the running service and interact with it (functional tests, performance tests, etc)
2. Use tools like BloomRPC or gRPCurl
3. Organize the service code in order to easily **unit test the business logic underneath**

Unit Testing – Client

- You can **mock** the service stub yourself, or
- let `protoc` generate it for you (as **google mock** class):

```
protoc -I . --grpc_out=generate_mock_code=true:. \
            --plugin=protoc-gen-grpc=grpc_cpp_plugin.exe \
            hello.proto

protoc -I . --cpp_out=. hello.proto
```

Server Reflection Protocol

- gRPC provides a way (aka: a protocol) to publicly expose service definitions
- Enabling this feature, we don't need to precompile service definitions to communicate with the service (e.g. useful for building CLI tools)
- In C++, we can just enable a plugin for this purpose (already shipped with the library):

```
#include <grpcpp/ext/proto_server_reflection_plugin.h>
...
grpc::reflection::InitProtoReflectionServerBuilderPlugin();
```

- And, magically, we can use tools like `grpcurl` to list and interact with available services:

```
grpcurl.exe -plaintext localhost:50051 list

grpcurl --plaintext -d "{\"name\":\"Hello\"}" localhost:50051 HelloService/SayHello
```

Some useful tools

- BloomRPC (GUI client for interacting with gRPC services)
- gRPCurl (like Like cURL, but for gRPC)
- Postman (recently added support for gRPC)
- OpenCensus (set of libraries for collecting metrics)
- ghz (load-testing tool implemented in Go)
- grpcbin (httpbin like for gRPC)

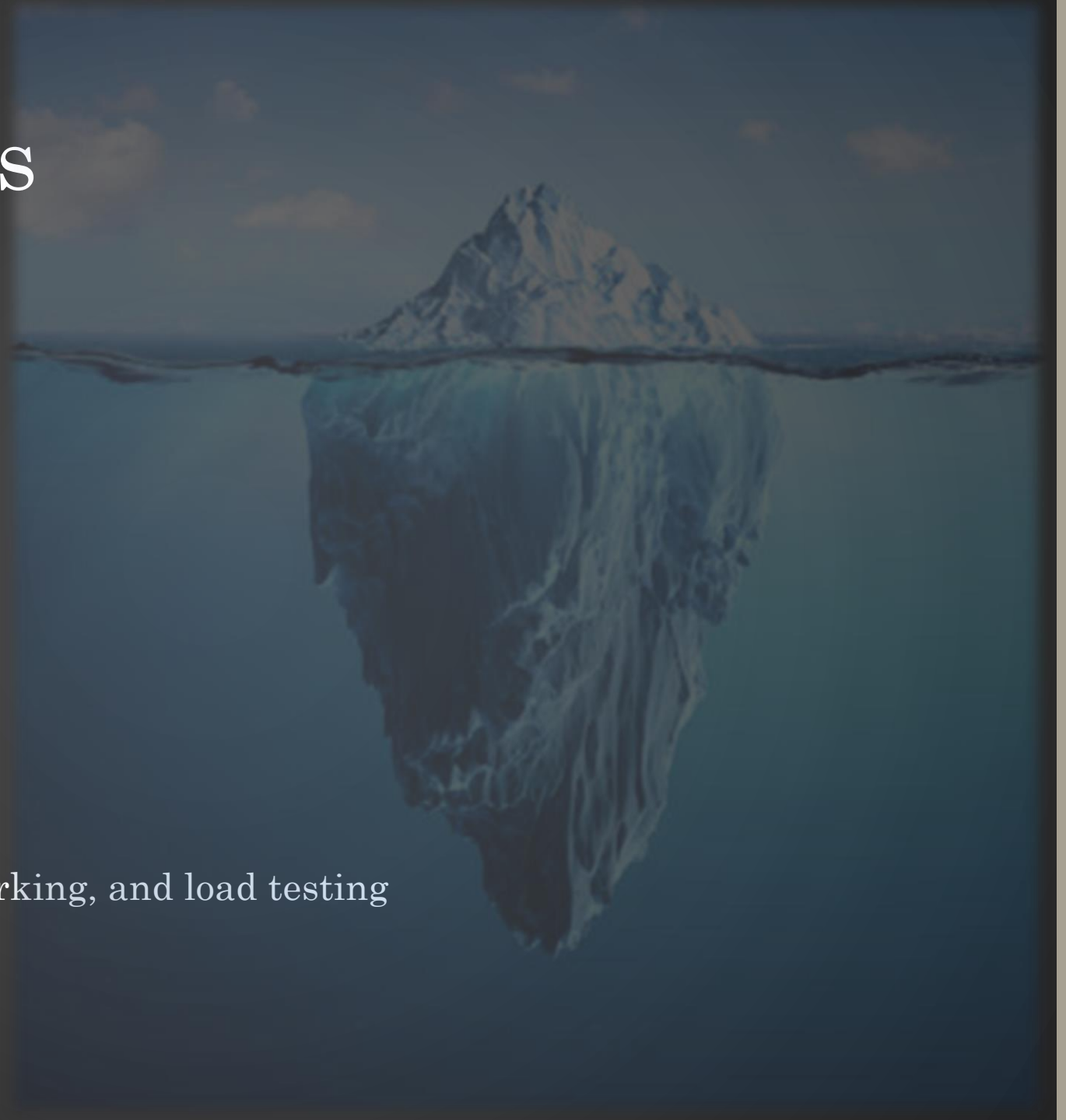
https://github.com/ilpropheta/hello-grpc

Hands-on

Epilogue

Uncovered topics

- Asynchronous API
- Advanced features such as:
  - Metadata
  - Authentication
  - Compression
  - Interceptors
  - Load Balancing
- Fine tuning/optimizations, benchmarking, and load testing
- Observability and metrics



Resources

- gRPC: Up and Running (awesome book)
- Awesome gRPC (huge curated list of resources, tools, etc)
- Official C++ Tutorials & Raw documentation
- Performance best practices and here
- gRPC Official blog
- Periodic benchmark results

```
syntax = "proto3";

service MarcoService {
  rpc Ask (Question) returns (Answer);
}

message Question {
  string content = 1;
}

message Answer {
  string content = 1;
}
```