C++ Client Libraries

Anna Levenberg

Aug 2026

Disclaimer

- I'm not speaking on behalf of Google
- All of the thoughts and ideas expressed in this presentation are my own
- This is my first time giving this talk

Agenda

01 Introduction

02 C++ at Google (and outside of Google)

03 Open Telemetry

04 Distributed Messaging System (Pub/Sub)

05 Conclusion & Questions

01

Introduction

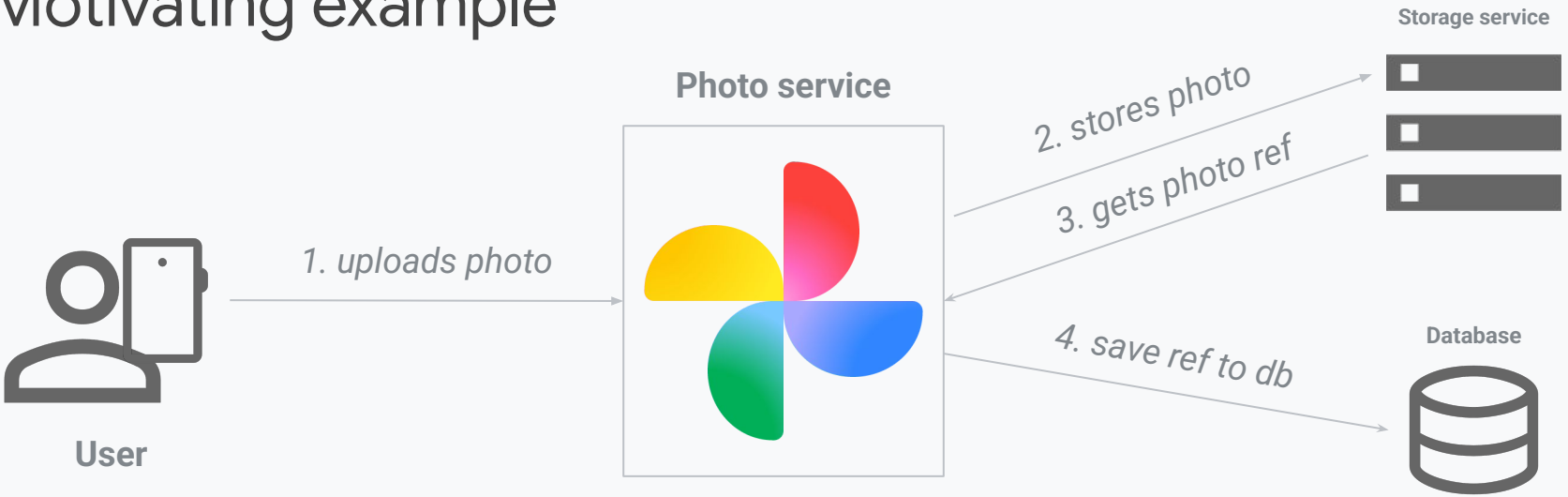
About me

- Software Engineer at Google for 5+ years
  - Currently on the Google Maps Guidance Client team
  - Worked on GPU Virtualization VM team, Cloud C++ Client Libraries, Datacenter software
- Graduated from Binghamton University in 2020
- Interests outside of work include puzzles, swimming, and yoga

Motivating example

- Imagine you are working on a photo sharing application
  - like Google Photos or Flickr
- What's the goal of the application?
- Where do you store the photos?
  - Database
  - Object storage (blob storage)
    - Store the image in the Cloud
- Lots of cloud providers offer this service
  - Google Cloud Storage
  - Amazon S3
  - Azure Blob Storage

Motivating example

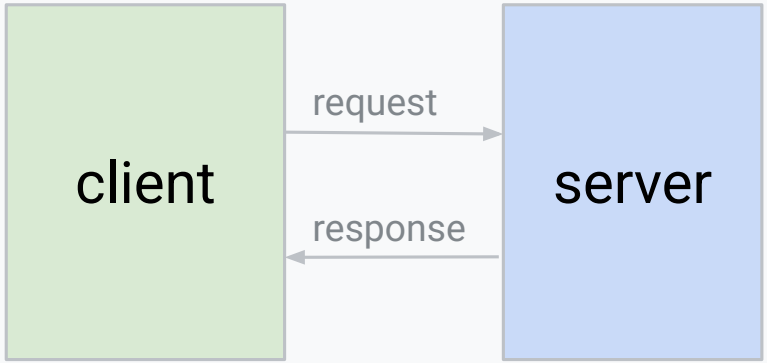


Database
- Users table
- Photos table

Column Name	Column Datatype	Example Value
photo_id (PK)	UUID	12345
user_id (FK)	INT	6789
gcs_url	VARCHAR (512)	https://storage.googleapis.com/google-photos/photo.jpg

What are client libraries?

- Google Cloud has a lot of services (150+)
- Google defines these services using protocol buffers and gRPC
- You can access them via UI, CLI, or client libraries
- Client libraries are a layer on top of the generated protobuf code
- Horizontal (languages) vs vertical (services)



Background: Protocol buffers

- Protocol buffers (Protobuf/protos)
  - Schema based language agnostic binary encoding format
  - Like JSON or XML, but smaller and faster

JSON

```
{
"id": 1042,
"name": "Jane Doe",
"email": "jane.doe@example.com",
"isActive": true,
"roles": ["owner", "viewer"]
}
```

Protobuf

```
syntax = "proto3";

package user;

message UserProfile {
  int32 id = 1;
  string name = 2;
  string email = 3;
  bool is_active = 4;
  repeated string roles = 5;
}
```

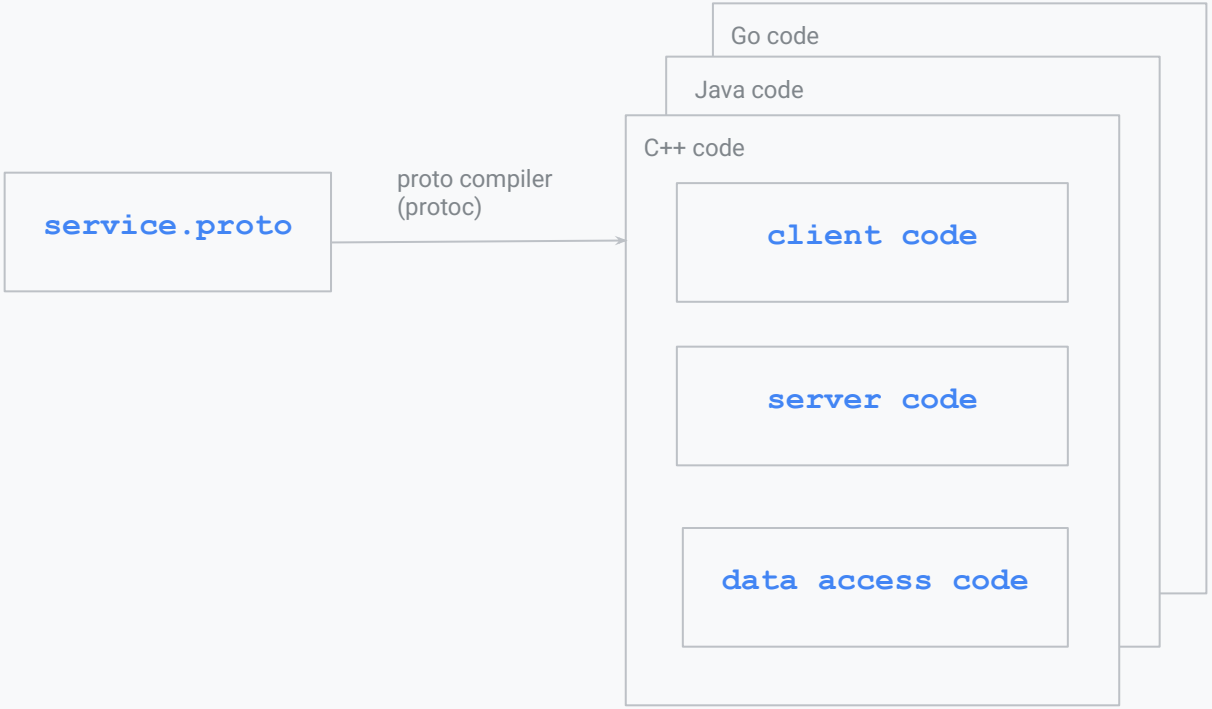
Background: gRPC

- RPC: remote procedure call
- gRPC a high performance RPC framework
  - Like REST
- Uses protobufs to send data
- Code generation via protobuf compiler
- Supports 4 different RPC types
  - Unary, client-side streaming, server-side streaming, bidirectional streaming

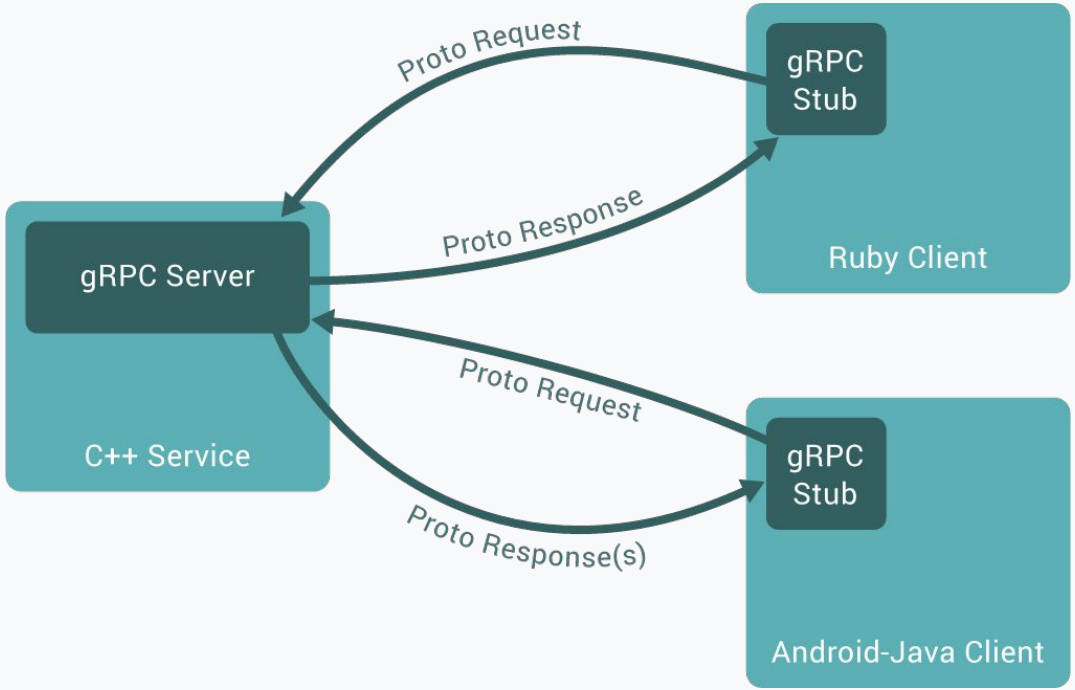
Example gRPC service definition

```
// service.proto
service Storage {
  rpc Upload (UploadRequest) returns (UploadReply) {}
}

message UploadRequest {
  string bucket_name = 1; // i.e. my-photo-bucket
  string object_name = 2; // i.e. sunset.jpg
  bytes data = 3; // bytes of the image
}

message UploadReply {
  string gcs_url = 1; // url where the image is stored
https://storage.googleapis.com/my-photo-bucket/sunset.jpg
}
```

Compiling grpc service proto



Compiling grpc service proto



02

C++ at Google (and outside of Google)

C++ Client Libraries

- Purpose of a library
  - Share reusable code
- If protoc generates the C++ code, then why do we need client libraries?
  - C++ generated gRPC code is not intuitive to use
    - Library uses a protoc plugin to create a generated veneer over the gRPC code
  - Give users out-of-the-box features
    - Retries, backoff, and deadlines
    - Authentication and authorization
- Some clients also have a handwritten layer on top of that to make it easier for developers to use (Bigquery, Pub/Sub)

C++ Client Libraries

- First time working on open source (and C++ outside of Google)
- Client libraries require portable C++ code
  - OS (MacOS, Linux, Windows)
  - Compiler (Clang, gcc, clang, MSVC, Apple Clang)
  - C++ versions (C++17, C++20, etc)
  - Build systems (Bazel and CMake)
  - Protobuf versions, gRPC versions
- Big testing matrix

Example: windows.h min/max

```
#include <limits>
#include <algorithm>

int main(int argc, char* argv) {
  int highest = std::numeric_limits<int>::max();
  int lowest = std::min(5, 10);
}
```

Example: windows.h min/max

```
// windows.h

#ifndef NOMINMAX

#ifndef max
#define max(a,b)  (((a) > (b)) ? (a) : (b))
#endif

#ifndef min
#define min(a,b)  (((a) < (b)) ? (a) : (b))
#endif

#endif /* NOMINMAX */
```

Example: windows.h min/max

- Macros are text substitution

When compiled without NOMINMAX (default)

```
auto const min = std::numeric_limits<int>::min(1,3);
```



```
auto const min = std::numeric_limits<int>::(((1) > (3)) ? (1) : (3));
```

C++

at Google

- Most code at Google is a monorepo
- Built using bazel (blaze)
- Internal teams that manage the toolchain
- Most code relies on internal libraries or dependencies
    - 3rd party dependencies are rare and require a dedicated maintainer

outside of Google

- Each team might have their own repo
- Some use bazel, others use CMake, make, or MSVC
- Compile locally or on smaller scale CI/CD
- Each team has uses a different compiler, platform, etc.
- Dependencies require upstream patches/updates

Background: Build systems

- GNU Make (Makefiles)
  - Shell command script
- CMake
  - Build system generator (i.e. Makefiles)
  - Most popular build system
- Bazel
  - Builds a DAG to track dependencies
  - Incremental builds
  - Hermetic environment
  - Parallel execution
  - Used at Google
- And many others

"I want to use library X with Bazel, at Google"

- Either the library is internal or external
  - If it's internal, we own it
  - If it's external, it has been imported and has dedicated maintainers
- There's only one version of the library in the codebase (everything builds at HEAD)
- Toolchain is identical for everyone
- Add one line to the BUILD file

"I want to use library X with Bazel, at Google"

```
// BUILD

cc_binary(
  name = "my_tool",
  srcs = ["my_tool.cc"],
  deps = [
    "//base",
  ],
)
```



```
// BUILD

cc_binary(
  name = "my_tool",
  srcs = ["my_tool.cc"],
  deps = [
    "//base",
    "//random_dep:dep",
  ],
)
```

"I want to use library X with Bazel, outside Google"

- Need to fetch the library
- Depending on the libraries support matrix
  - Might need to write and maintain a BUILD file
- Have to handle the dependency resolution
- Have to ensure it works with your toolchain
- Multiple files, configuring, testing to verify it builds

What's the alternative?

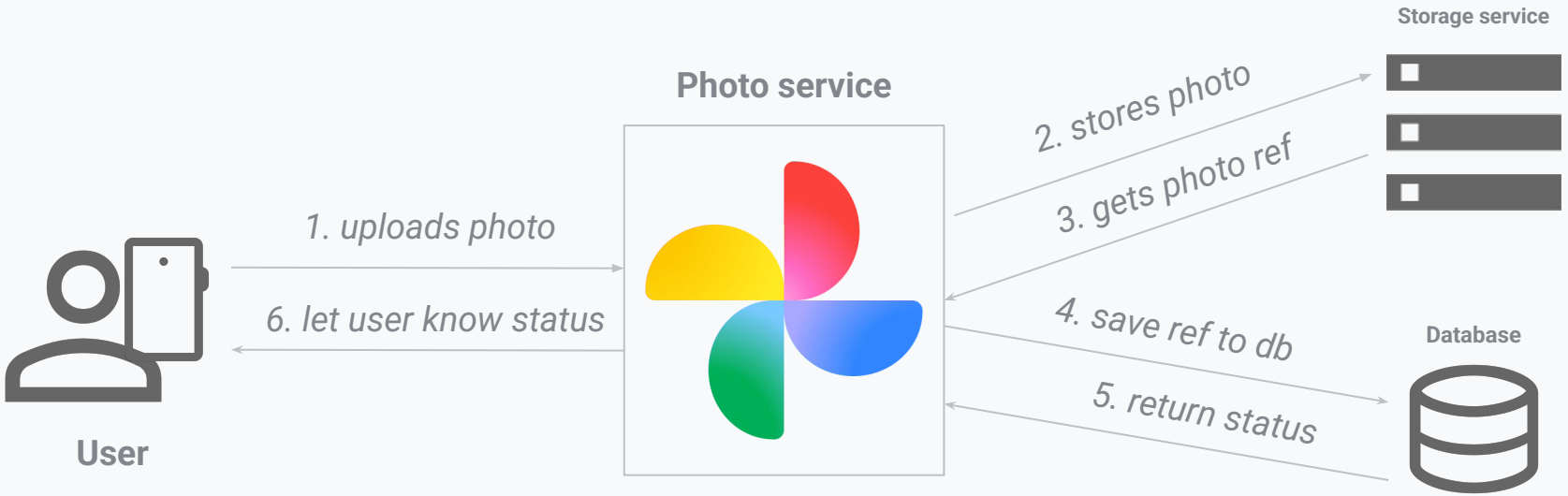
- In python, you have `pip install`
  - This is a package manager
- C++ ecosystem does **not** have a built in package manager
- There are 3rd party tools
  - vcpkg
  - conan
  - CMake features

03

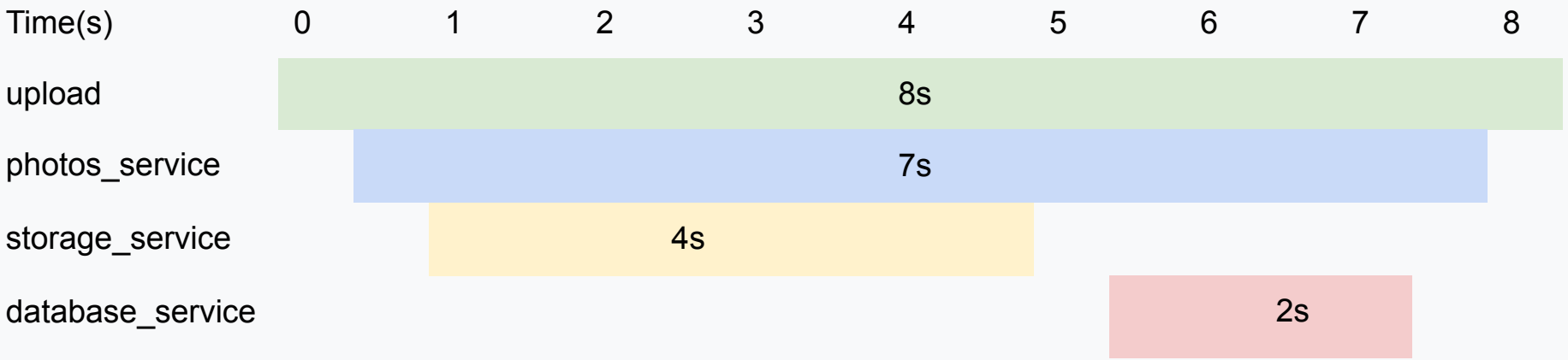
Open Telemetry

Motivating example

- What if this takes 8s to upload the photo?



Motivating example

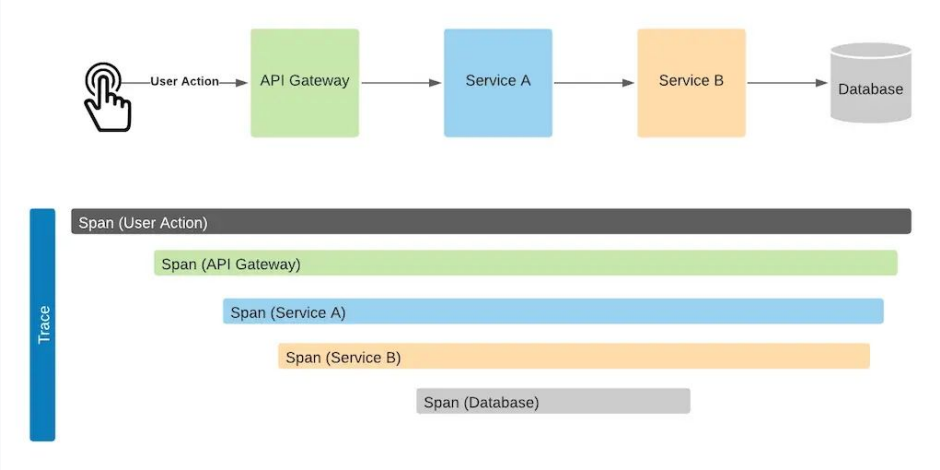


Open Telemetry

- Observability
- Instrumentation
- Core concepts
  - Logs
  - Metrics
  - Traces

Traces

- Trace is a set of spans
- A span is
  - A start/end time
  - Operation name
  - Status
  - Attributes (key value metadata)
- Semantic conventions



Adding Open Telemetry as a dependency

- Libraries use semantic versioning
  - XX.XX.XX (Major.Minor.Patch)
  - **Major** (X.0.0): backwards-incompatible (breaking) changes
  - **Minor** (1.X.0): new, backwards-compatible functionality
  - **Patch/Bugfix** (1.0.X): backwards-compatible internal bug fixes
- Taking on a new dependency is a breaking change (requires a major version change)
  - It requires work for a consumer of the library to upgrade
- Support policies around how long to support after a breaking change
  - Typically release them ~ 1-2 years

Adding Open Telemetry as a dependency

- At the time, we were still one year away from our next major version bump
- What do we do?
  - Defer the change
  - Do the major release early
  - Add a backwards compatible change (optional dependency)
- We wanted to deliver to customers as soon as possible
- Customers can opt-in to open telemetry if they want

Adding Open Telemetry as a dependency

- How do we do this?
  - Conditionally compile the code with IFDEF blocks

Adding Open Telemetry as a dependency

```
#ifdef GOOGLE_CLOUD_CPP_HAVE_OPENTELEMETRY
#include <opentelemetry/trace/provider.h>
#include <opentelemetry/trace/scope.h>
#endif

class MyCloudServiceTracingStub {
  public:
    MyResponse ExecuteCall(MyRequest const& request) {
#ifdef GOOGLE_CLOUD_CPP_HAVE_OPENTELEMETRY
      auto span = internal::MakeSpan("MyCloudService::ExecuteCall");
      auto scope = opentelemetry::trace::Scope(span);
#endif
      return child_->ExecuteCall(request);
    }
};
```

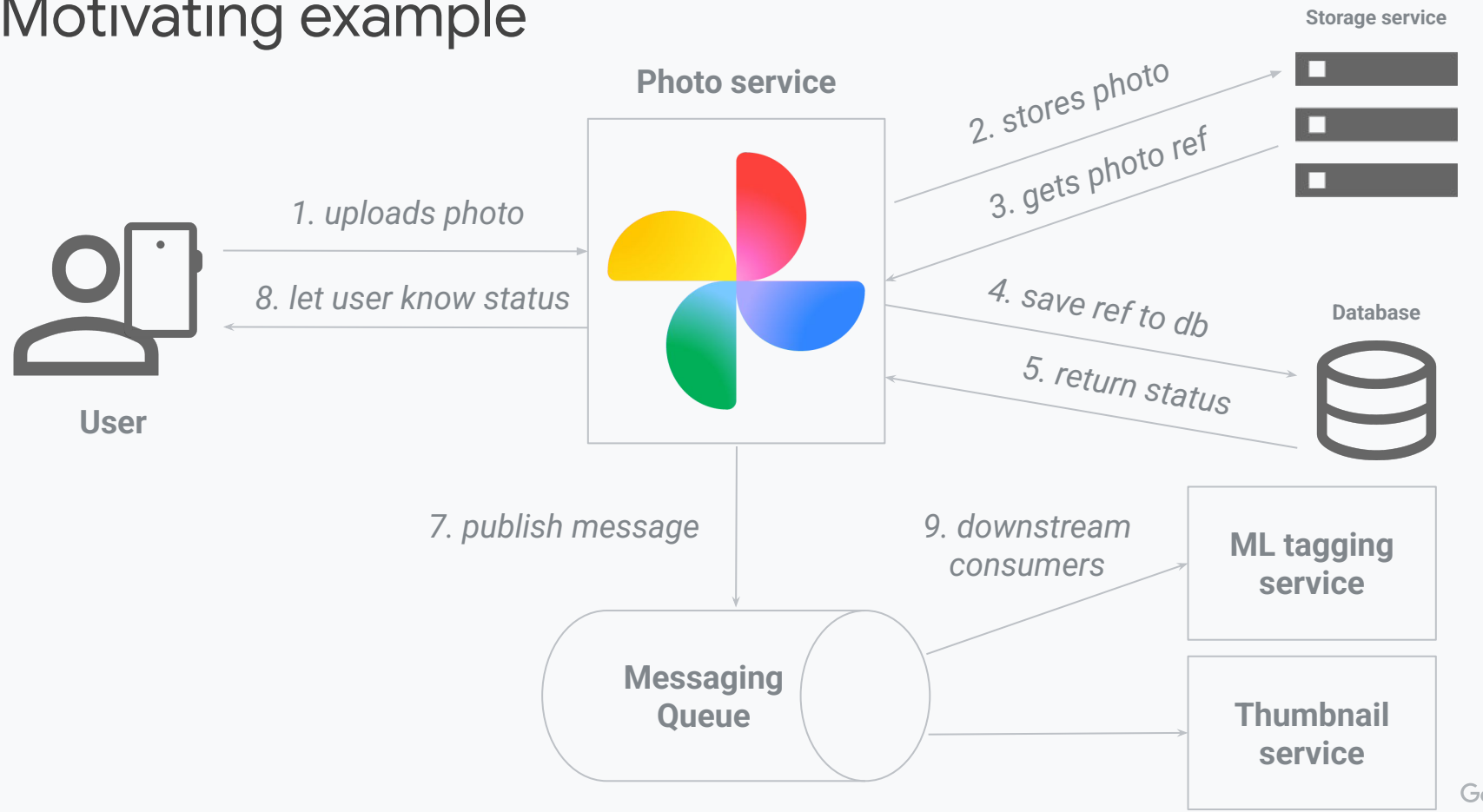
04

Distributed Message Queue

Motivating example

- Let's say we want to add a way to view the photos
  - A thumbnail generator
  - ML photo tagger

Motivating example



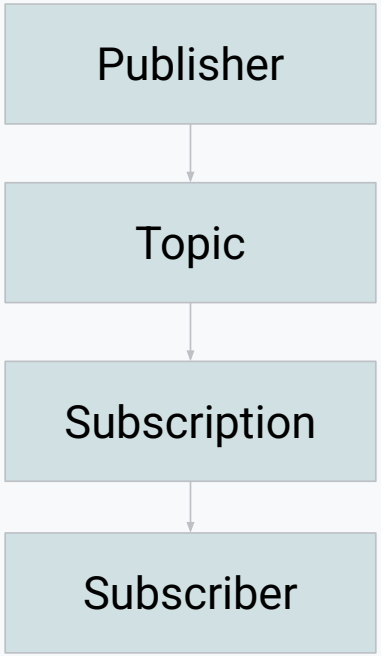
Pub/Sub

- A distributed messaging system
  - Message broker
  - Distributed messaging queue
- Kafka, AWS Simple Queue Service, Azure Service Bus, RabbitMQ
- Asynchronously sending messages
  - Decouple services

Pub/Sub

- Message
- Topic
- Subscription
- Publisher
- Subscriber

Message




```
message PubsubMessage {
  bytes data = 1;
  map<string, string> attributes = 2;
  string message_id = 3;
  google.protobuf.Timestamp publish_time = 4;
}
```

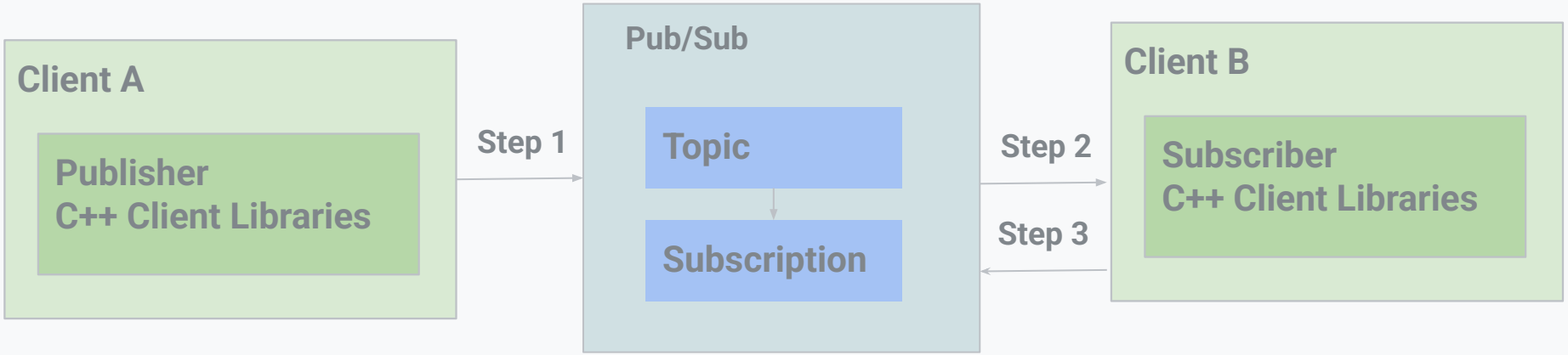
Pub/Sub

- Admin API
  - CRUDL for topics/subscriptions
- Data API
  - Publisher
    - Async publish
    - Sync publish
  - Subscriber
    - Unary pull
    - Streaming pull

Pub/Sub: Publisher

- Asynchronous or synchronous
- Batching
- Flow control

To use Pub/Sub:
Step 1: Publish message to Topic
Step 2: Pull message from Subscription
Step 3: Acknowledge that message has been processed



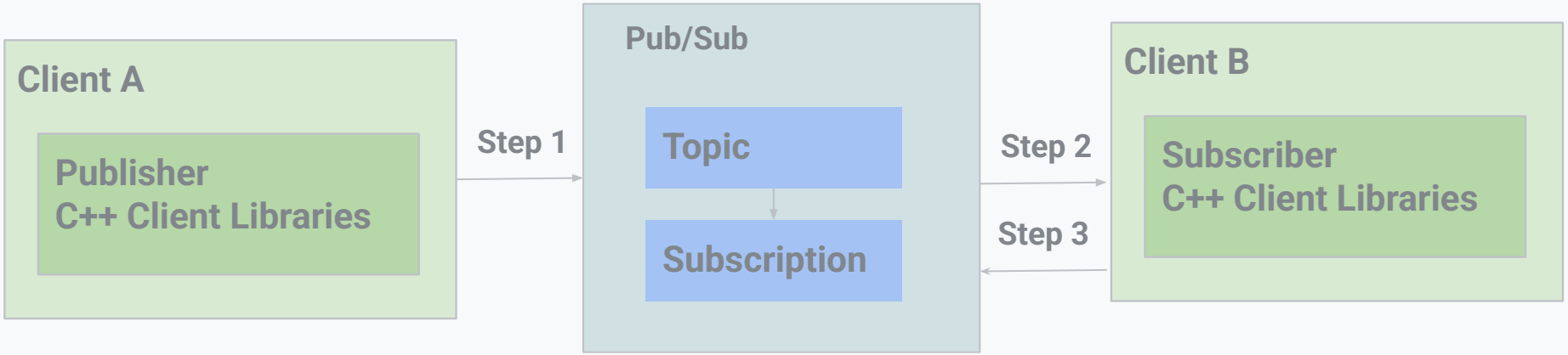
Pub/Sub: Publisher

```
service Publisher {
  rpc Publish(PublishRequest) returns (PublishResponse);
}

message PublishRequest {
  string topic = 1;
  repeated PubsubMessage messages = 2;
}

message PublishResponse {
  repeated string message_ids = 1;
}
```

Pub/Sub: Subscriber

- Unary or bidirectional streaming pull
- Bidirectional pull
  - Flow control
  - Concurrency control
  - Lease management

To use Pub/Sub:
Step 1: Publish message to Topic
Step 2: Pull message from Subscription
Step 3: Acknowledge that message has been processed



Pub/Sub: Subscriber

```
service Subscriber {
  rpc Pull(PullRequest) returns (PullResponse) {}
}

message PullRequest {
  string subscription = 1;
  int32 max_messages = 2;
}

message ReceivedMessage {
  string ack_id = 1;
  PubsubMessage message = 2;
  int32 delivery_attempt = 3;
}

message PullResponse {
  repeated ReceivedMessage received_messages = 1;
}
```

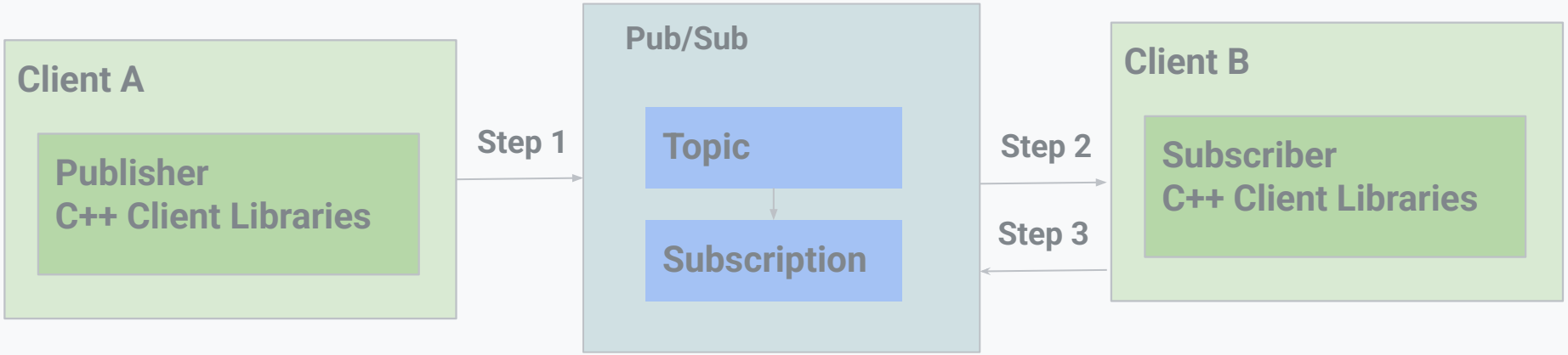
Pub/Sub: Subscriber

```
message AcknowledgeRequest {
  string subscription = 1;
  repeated string ack_ids = 2;
}

service Publisher {
  rpc Acknowledge(AcknowledgeRequest) returns (google.protobuf.Empty);
}
```

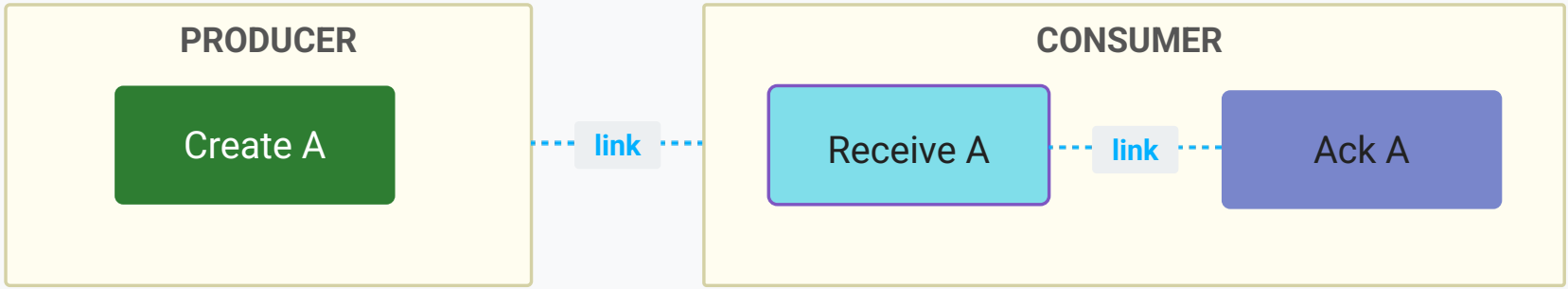
Instrumenting Pub/Sub

- How do you connect spans across different machines?
  - Context propagation

To use Pub/Sub:
Step 1: Publish message to Topic
Step 2: Pull message from Subscription
Step 3: Acknowledge that message has been processed



Instrumenting Pub/Sub



05

Conclusion

Conclusion

- C++ at Google
  - grpc, protobufs
- C++ outside of google
  - C++ client libraries
  - C++ ecosystem
- Open Telemetry
- Pub/Sub
- Questions?