HW3 Hint

- In this assignment we are not passing in an IP address, rather we are passing in a hostname
- As a result, you will need to apply what you learned in Lab 6 in order to get an address that you can use to connect() to the server
- Keep in mind that connect() expects an address in network order so you should not be calling inet_ntoa() or inet_ntop() when populating a struct sockaddr.

gRPC

Network Programming

RPC Definition

- Remote Procedure Calls (RPCs)
  - Should appear like a local function call
  - Code executed by call may be on another host
  - Remote code should return when call is complete, possibly pass back data
- RPCs use message passing as the underlying mechanism
- The goal is to make calls instead of requesting resources like in Representational State Transfer (REST)

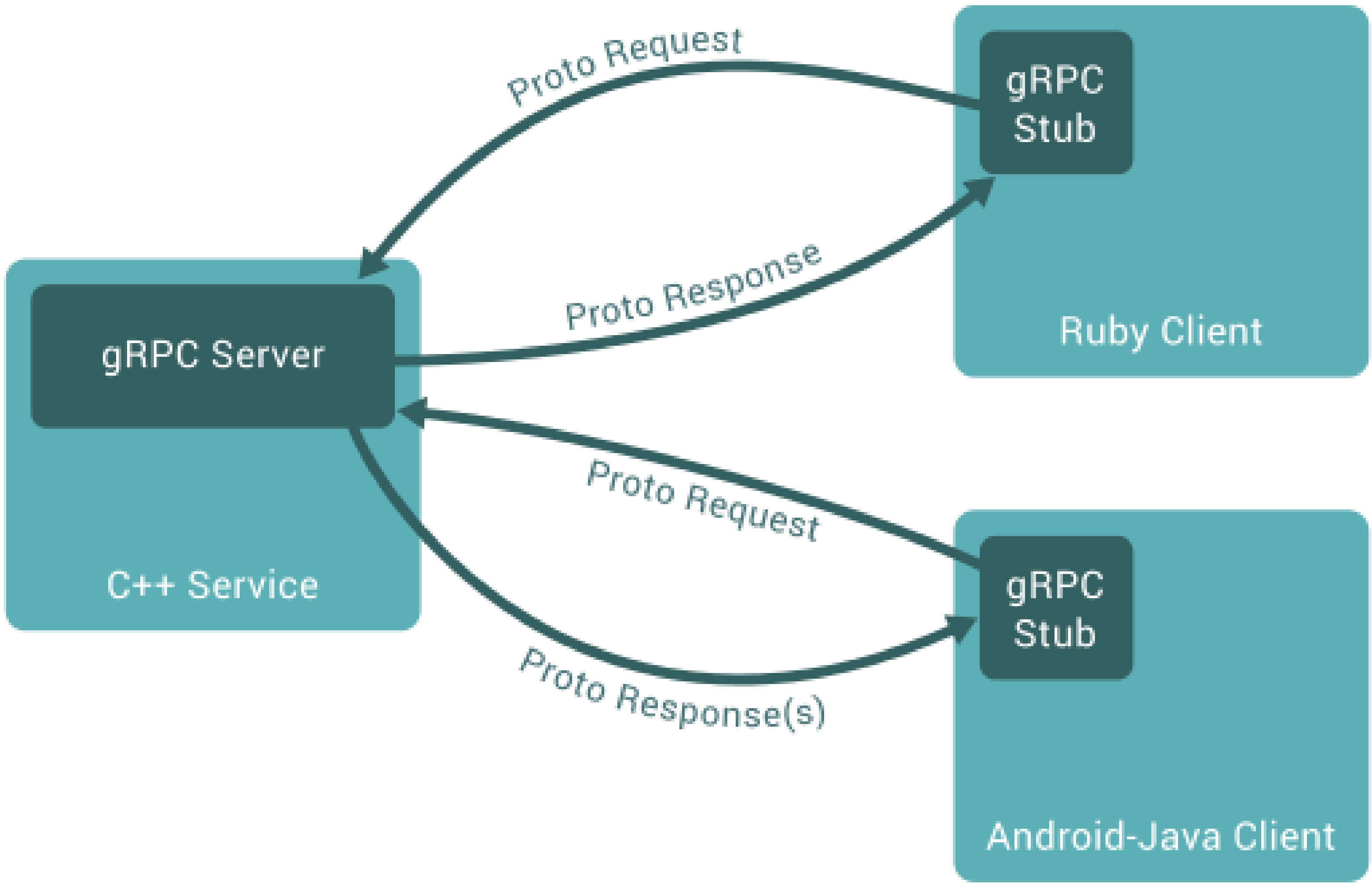
Why Use RPCs?

- RPC protocols hide network details
- Simplifies code, lets us write code as though we were using local functions
- Offload responsibilities to specialized hosts
- Maybe…
  - Only certain hosts have a database engine
  - We don’t want to replicate data
  - Some hosts are better at doing fast arithmetic
  - Etc.

Brief History

- 1980s – Practical network implementations
- 1991 – Common Object Request Broker Architecture (CORBA)
- 1998 - XML-RPC
- 2000 - Simple Object Access Protocol (SOAP)
- 1999 - XMLHttpRequest developed
- 2002-2007 – Various browsers officially add XMLHttpRequest
- 2015 – gRPC released by Google

gRPC Flowchart



gRPC proto3 (protocol buffers)

- Note that *pb2* which will show up in some code is for Python Protocol Buffers API v2. This works with proto2 or proto3 (protocol buffer description languages)
- Describes *service*, *rpc*, and *message* types
  - Full documentation online but .proto file is human readable
  - Real data is compressed/serialized and not human readable. **Can** be much more efficient than XML
- *messages* typically correspond to OOP objects

gRPC Channels

- Communication between a server and a client happens over a *channel*
- `with grpc.insecure_channel('localhost:50051') as channel:`
  - The *with* is a context manager in Python – we'll close the channel when we go out of scope
  - We will need to give our *stub* a channel to communicate over (more on this soon)
  - This example uses an unauthenticated (insecure) channel, easy and works for us

gRPC Authentication

- Much like HTTP vs HTTPS, insecure_channel isn't always sufficient
- SSL/TLS support
  - Need a certificate
- Google tokens via OAuth2
- Can have security over a channel
- Can also do per-call security
- Examples: https://grpc.io/docs/guides/auth.html#python

gRPC stubs

- Clients use stubs to actually invoke the RPC calls
- A stub looks like the RPC prototype but only sends one or more messages, and then receives one or more messages back.
- As far as the client can tell, we made a call and got the expected output back
  - Assuming no errors

gRPC Errors

- Calls can raise errors, in Python the client may print runtime errors that are from the server side
- General errors such as **`GRPC_STATUS_CANCELLED`**, **`GRPC_STATUS_UNIMPLEMENTED`**
- Network errors such as **`GRPC_DEADLINE_EXCEEDED`**
- Protocol errors such as **`GRPC_STATUS_UNAUTHENTICATED`**

gRPC Timeout/Deadline

- Depending on language, you may have a timeout (number of seconds) or a deadline (absolute timestamp)
- By default there is no timeout/deadline and calls can run forever
  - We'll use this default in lab, but in general you should always have timeouts to be safe
- In Python:
  - helloworld_pb2.HelloReply(message='Hello, %s!' % request.name, timeout=2)

gRPC Method Types

- Unary
  `rpc y(InMessage) returns (OutMessage) {}`

- Client Streaming
  `rpc y(stream InMessage) returns (OutMessage) {}`

- Server Streaming
  `rpc y(InMessage) returns (stream OutMessage) {}`

- Bidirectional Streaming
  `rpc y(stream InMessage) returns (stream OutMessage) {}`

gRPC Wire Representation

- Uses HTTP2
- Not a casual read: [docs]
- You can look at a trace in Wireshark
  - Strings tend to be left intact
  - Field names and other fields may not be human readable due to compression / serialization
  - Wireshark won't be able to dissect the application layer data

gRPC Generating Files

- See examples/python/route_guide/run_codegen.py
- If we have x.proto which describes a service **`A`** with **`rpc y (Z)`** where **`Z`** is a message type containing just a single string **`b`**
- We will need a stub, for example
  `stub = x_pb2_grpc.AStub(channel)`
- Then we can call
  `stub.y(x_pb2.Z(b="somevalue"))`
- Creates **`Z`** according to **`x.proto`**, uses it in RPC call **`y`** over channel associated with **`stub`**

gRPC Servicer Registration

- Server has to implement the RPC calls
  - "compilation" builds a "Servicer" for each **`class`** defined in the .proto, so in our example AServicer
    - Always just "Servicer" after the class name
  - Derives from **`x_pb2_grpc.AServicer`**
  - **`class MyClass (x_pb2_grpc.AServicer):`**
- A Servicer needs a server, see examples for grpc.server()
- **`x_pb2_grpc.add_AServicer_toserver(MyClass(), server)`**

Hello World Example - .proto

- examples/protos/helloworld.proto
- Ignore all the option java/objc stuff, we don't need it in Python
- Greeter service has SayHello method
  - Expects a HelloRequest
  - Returns a HelloReply
- **`syntax = "proto3";`**
  - Important, says we're using v3 of Protocol Buffers

Hello World Example - .proto

- Each of these types is a message
  - Note that using "message" as a variable name is fine (terrible style though)!
  - The assignment shows a unique “tag”, we must enumerate each variable and they cannot share tags
    - You can nest messages in messages
    - Frequently used variables should have tags < 16

Hello World Example - Client

- examples/python/helloworld/greeter_client.py
- Have to import grpc, x_pb2, x_pb2_grpc (in this case x is helloworld, since we used helloworld.proto)
- Insecure channel on TCP port 50051 used
  - When the `with` context manager is closed, connection is closed automatically
  - Stub takes channel, follows naming rules
  - Use the stub to call RPC SayHello

Hello World Example - Server

- examples/python/helloworld/greeter_server.py
- Note that Greeter extends helloworld_pb2_grpc.GreeterServicer
  - Didn't need to name our class Greeter
- Server is allotted 10 threads in this example
- server.start() does not block, busy wait loop with 1 second sleep, CTRL+C to quit
  - Without some sort of busy wait, after .start() we would immediately terminate the server

Synchronous / Asynchronous Calls

- Most of the examples are synchronous
- Sometimes you might want asynchronous calls
- This can be done using Python's native features
- Just use futures (concurrent.futures or asyncio.Future)

Synchronous / Asynchronous Calls

- Example:

```
call_future =
stub.SayHello.future(helloworld_pb2.HelloRequest(n
ame='you'))

call_future.add_done_callback(process_response)
```

- Source: https://github.com/grpc/grpc/issues/16329

Route Guide Example Dataset

- examples/python/route_guide/route_guide_db.json
- Contains JSON notation (standard format, easy to parse)
- Array of associative arrays (Features)
- Each Feature associates a "name" with a "location" (Point)
- Each Point is an associative array with an integer "latitude" and "longitude"

Route Guide Example Helpers

- examples/python/route_guide/route_guide_resources.py
- Uses the Python json parsing library
- Single function, read_route_guide_database()
  - Creates a list which is returned
  - Every entry is a Feature (name (string) + location (Point)), uses the route_guide_pb2 classes
    - These are based on the message types described in the .proto
    - examples/proto/route_guide.proto

Route Guide Example .proto

- Single service, RouteGuide
  - GetFeature(Point), Unary
  - ListFeatures(Rectangle), Server Stream
  - RecordRoute(stream Point), Client Stream
  - RouteChat(stream RouteNote), Bidirectional Stream
- Messages
  - Point (int32, int32) - lat. lon.
  - Rectangle(hi, lo) - two corners to make a diagonal
  - Feature(name, Point)
  - RouteNote(location, message) - used in RouteChat
  - RouteSummary(int32 x5) - RecordRoute() results

Route Guide Example Server

- Main loop looks the same as helloworld example
- get_feature() returns a Feature or `None`
- get_distance() calculates distance between two points
- Everything else in RouteGuideServicer

Route Guide GetFeature() Server

- `RouteGuideServicer.__init__()` loads the json into `self.db`
- GetFeature()
  - `context` contains info like deadlines
  - Input `request` is a Point (see .proto)
  - Returns Feature from our database or None if it can't be found

Route Guide GetFeature() Client

- Two calls to `guide_get_one_feature()`, one using a specific Point, one using the origin (0,0)
- guide_get_one_feature()
  - Make the gRPC call
  - Sanity check for formatting
  - If there's a name, lookup succeeded, print the name and location
  - Otherwise print that lookup didn't find the location, and print location

Route Guide ListFeatures() Server

- ListFeatures()
  - Input `request` is a Rectangle (see .proto)
  - Client will expect a stream of Features
  - Returning all Features at once could be a lot of data, instead, this is a Server Streaming call
  - Treat the server as a generator by using the `yield` keyword
  - Code runs until first time yield is hit, returns yield argument
  - Every time after, resume from yield, run until yield/function end

Route Guide ListFeatures() Client

- guide_list_features() creates a Rectangle and uses the stub to RPC invoke ListFeatures()
- for feature in features:
  - Print each feature
  - Treats the result as an iterable or generator
  - We can start this loop as soon as we get a single Feature back from the server
  - Don't need to wait for all N Features, don't need to store N Features in memory!

Route Guide RecordRoute() Server

- RecordRoute()
  - Input **`request_iterator`** is a stream of Points (see .proto)
    - Just like the last stream we looked at, there's a for loop
    - Server treats this like a generator, can start as soon as one Point is received, keeps going until stream ends.
  - Do a lot of statistic counting, including distance, how many Points were Features, total length of time the call takes (on localhost this will likely be 0 seconds)
  - Package up all our results into one RouteSummary object, return

Route Guide RecordRoute() Client

- Starts with guide_record_route()
- Uses generate_route() to get a generator of random Features
- route_iterator stores an instantiation of the generate_route() generator
- Use the stub to call RPC RecordRoute()
  - It's fine that we directly pass a generator
- Print the RouteSummary we get back

Route Guide RouteChat() Server

- RouteChat()
  - Input **`request_iterator`** is a stream of RouteNote (see .proto)
  - Keep a list of previous notes
  - If the latest RouteNote from the stream matches the location of a note in previous notes, return the match from the past as a RouteNote using yield
    - We are returning a stream of RouteNotes as well
  - If we haven't encountered this location, add the new RouteNote to the end of the previous notes list

Route Guide RouteChat() Client

- guide_route_chat() gets a RouteNote generator generate_messages() and passes it directly to RPC RouteChat
- Prints every message we receive from the RPC, looping over the return value (because it's also a generator)
- generate_messages()
  - Fixed array of RouteNotes each made by passing arguments to make_route_note()
  - `yield` one message at a time, act as generator

Lab 7

- gRPC lab
- Should have to change relatively little code
- Downloads will take some time, need about 300MB of space between packages and source code
- Refresh yourself on Python3 syntax for printing, functions, objects
  - Remember that globals are frowned upon and to refer to a global inside a function you have to use the `global` keyword

Lab 7

- Both server and client will run on localhost for simplicity, but you could stick the server anywhere and your client should work the same (just with a different remote address in the channel)
- grpc/examples/python/route/run_codegen.py will make your life easier. -I specifies directory to find the .proto in, the --out directories being "." (current directory) puts all generated code inside the directory you call run_codegen from.