Extracted from:

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

edited by Dawn Schanafelt and Katharine Dvorak

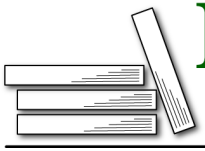
Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Dawn Schanafelt and Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-760-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2021

CHAPTER 4

Serve Requests with gRPC

We've set up our project and protocol buffers and written our log library. Currently, our library can only be used on a single computer by a single person at a time. Plus that person has to learn our library's API, run our code, and store the log on their disk—none of which most people will do, which limits our user base. We can solve these problems and appeal to a larger audience by turning our library into a web service. Compared to a program that runs on a single computer, networked services provide three major advantages:

- You can run them across multiple computers for availability and scalability.
- They allow multiple people to interact with the same data.
- They provide accessible interfaces that are easy for people to use.

Some situations where you'll want to write services to reap these advantages include providing a public API for your front end to hit, building internal infrastructure tools, and making a service to build your own business on (people rarely pay to use libraries).

In this chapter, we'll build on our library and make a service that allows multiple people to interact with the same data and runs across multiple computers. We won't add support for clusters right now; we'll do that in Chapter 8, Coordinate Your Services with Consensus, on page ?. The best tool I've found for serving requests across distributed services is Google's gRPC.

What Is gRPC?

When I was building distributed services in the past, the two common problems that drove me batty were maintaining compatibility and maintaining performance between clients and the server.

I wanted to ensure that clients and the server were always compatible—that the client was sending requests that the server understood, and vice versa with the server's responses. When I made breaking changes to the server, I wanted to ensure that old clients continued to work, and I accomplished this by versioning my API.

For maintaining good performance, your main priorities are optimizing your database queries and optimizing the algorithms you've used to implement your business logic. Once you've optimized those though, performance will often come down to how fast your service unmarshals requests and marshals responses, and down to reducing the overhead each time clients and the server communicate—like using a single, long-lasting connection rather than a new connection for each request.

So I was happy when Google released gRPC, an open source, high-performance RPC (remote procedure call) framework. gRPC has been a great help in solving these problems when building distributed systems, and I think you'll find that it simplifies your work. How does gRPC help you build services?

Goals When Building a Service

Here are the most important goals to aim for when you're building a networked service—and some info about how gRPC helps you achieve them:

Simplicity

Networked communication is technical and complex. When building our service, we want to focus on the problem it solves rather than the technical minutiae of request-response serialization, and so on. You want to work with APIs that abstract these details away. However, when you need to work at lower levels of abstraction, then you need those levels to be accessible.

On the spectrum of low- to high-level frameworks, in terms of the abstractions you're working with, gRPC is mid-to-high level. It's above a framework like Express since gRPC decides how to serialize and structure your endpoints and provides features like bidirectional streaming, but below a framework like Rails since Rails handles everything from handling requests to storing your data and structuring your application. gRPC is extendable via middleware, and its active community[1] has written middleware[2] to

1. https://github.com/grpc-ecosystem
2. https://github.com/grpc-ecosystem/go-grpc-middleware

solve a lot of the problems you'll face when building services—for example, logging, authentication, rate limiting, and tracing.

Maintainability

Writing the *first* version of a service is a brief period of the total time you'll spend working on the service. Once your service is live and people depend on it, you must maintain backward compatibility. With request-response type APIs, the simplest way to maintain backward compatibility is to version and run multiple instances of your API.

With gRPC, you can easily write and run separate versions of your services when you have major API changes, while still taking advantage of protobuf's field versioning for small changes. Having all your requests and responses type checked helps prevent accidentally introducing backward-*incompatible* changes as you and your peers build your service.

Security

When you expose a service on a network, you expose the service to whoever is on that network—potentially the whole Internet. It's important that you control who has access to your service and what they can do.

gRPC supports Secure Sockets Layer/Transport Layer Security (SSL/TLS) to encrypt all data exchanged between the client and server and lets you attach credentials to requests so you know which user is making each request. We'll discuss security in the next chapter.

Ease of use

The whole point of writing a service is to have people use it and solve some problem of theirs. The easier your service is to use, the more popular it will be. You go a long way toward making your service easy to use by telling your users when they're doing something wrong, such as calling your API with a bad request.

With gRPC, everything from your service methods to your requests and responses and their bodies are all defined in types. The compiler copies the comments from your protobuf to your code to help users when the type definitions aren't good enough. Your users will know whether they're using the API correctly thanks to their code being type checked. Having everything—requests, responses, models, and serialization—type checked is a big help to people learning how to use your service. gRPC also lets users look up the API's details with godoc. Many frameworks don't offer either of these handy features.

Performance

You want your service to be as fast as possible while using as few resources as possible. For example, if you can run your application on an n1-standard-1 (~$35 per month) instance on Google Cloud Platform rather than on an n1-standard-2 (~$71 per month) instance, that cuts your costs in half.

gRPC is built on solid foundations with protobuf and HTTP/2 because protobuf performs very well at serialization and HTTP/2 provides a means for long-lasting connections, which gRPC takes advantage of. So your service runs efficiently and doesn't cause unnecessarily high server bills.

Scalability

Scalability can refer to scaling up with load balancing to balance the load across multiple computers and to scaling up the number of people developing a project. gRPC helps make both types of scaling easier.

You can use different kinds of load balancing with gRPC[3] based on your needs, including thick client-side load balancing, proxy load balancing, look-aside load balancing, or service mesh.

For scaling up the number of people working on your project, gRPC lets you compile your service into clients and servers in the languages that gRPC supports. This allows people to use their own languages to build services that communicate with each other.

We now know what we want out of building our service, so let's create a gRPC service that fulfills our goals.

Define a gRPC Service

A gRPC service is essentially a group of related RPC endpoints—exactly how they're related is up to you. A common example is a RESTful grouping where the relation is that the endpoints operate on the same resource, but the grouping could be looser than that. In general, it's just a group of endpoints needed to solve some problem. In our case, the goal is to enable people to write to and read from their log.

Creating a gRPC service involves defining it in protobuf and then compiling your protocol buffers into code comprising the client and server stubs that you then implement. To get started, open log.proto, the file where we defined our Record message, and add the following service definition above those messages:

3. https://grpc.io/blog/grpc-load-balancing

ServeRequestsWithgRPC/api/v1/log.proto
```
service Log {
  rpc Produce(ProduceRequest) returns (ProduceResponse) {}
  rpc Consume(ConsumeRequest) returns (ConsumeResponse) {}
  rpc ConsumeStream(ConsumeRequest) returns (stream ConsumeResponse) {}
  rpc ProduceStream(stream ProduceRequest) returns (stream ProduceResponse) {}
}
```

The service keyword says that this is a service for the compiler to generate, and each RPC line is an endpoint in that service, specifying the type of request and response the endpoint accepts. The requests and responses are messages that the compiler turns into Go structs, like the ones we saw in the previous chapter.

We have two streaming endpoints:

- ConsumeStream—a server-side streaming RPC where the client sends a request to the server and gets back a stream to read a sequence of messages.
- ProduceStream—a bidirectional streaming RPC where both the client and server send a sequence of messages using a read-write stream. The two streams operate independently, so the clients and servers can read and write in whatever order they like. For example, the server could wait to receive all of the client's requests before sending back its response. You'd order your calls this way if your server needed to process the requests in batches or aggregate a response over multiple requests. Alternatively, the server could send back a response for each request in lockstep. You'd order your calls this way if each request required its own corresponding response.

Below your service definition, add the following code to define our requests and responses:

ServeRequestsWithgRPC/api/v1/log.proto
```
message ProduceRequest  {
  Record record = 1;
}

message ProduceResponse  {
  uint64 offset = 1;
}

message ConsumeRequest {
  uint64 offset = 1;
}

message ConsumeResponse {
  Record record = 2;
}
```

The request includes the record to produce to the log, and the response sends back the record's offset, which is essentially the record's identifier. Similarly with consuming: the user specifies the offset of the logs they want to consume, and the server responds back with the specified record.

To generate the client- and server-side code with our Log service definition, we need to tell the protobuf compiler to use the gRPC plugin.