Server to microservice communication Basics of gRPC using gRPC

Mikhail Gaydamakha

University of Strasbourg

Abstract

There are different types of web application architectures. The more classic is one server with a lot of clients connected to. The second popular approach is micro-service architecture, where server functionality is decoupled into several small entities, communicating with each other. Very often we want to pass from the first concept to the second as more reliable and easily maintainable, without changing existing client applications. In this case, the client is not able to communicate to the microservice directly, and the question is which way of communication between server and microservice to choose. For the first time, we would like to replace the weaker places of the server, those which requires a lot of CPU performance. For example, image or large text files processing. Of course, in different cases we would apply different technologies, such as MQTT (pub/sub model), Websockets, or even plain HTTP (version 1.1 or 2). Another good alternative is RPC (remote procedure call) model, which is synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [1]. Its advantage is that the details of network implementation are abstracted from the developer, so it is extremely easy to replace a standard method call in an existing application by RPC call. The one of modern implementations of RPC model is gRPC – an open source multilanguage framework based on HTTP/2 protocol. In this poster, I will estimate a performance of this solution, compare it with others and test its different built-in features, such as load balancing.

Keywords: gRPC, Performance, Microservice, Web application

Basics of gRPC

RPC is an old model that was much used on the past. Famous RPC models are CORBA, RMI and DCOM. Their solutions have much in common with classical request-response model. Despite the name, gRPC takes only the concept of client-server application in common with the old solutions and of course a transport-level abstraction. Unlike the previous RPC solutions, it can be used and interacted with server and clients written in different programming languages (technology-agnosticism). It has become possible by using protocol buffers – gRPC's serialization mechanism. It compresses messages before sending them to the services. That allows not only to use different languages/platforms, but also to process more data with less network load. According to Protocol Buffers documentation [2], their messages gives such advantages relative to the traditional data schemas as XML or JSON, as simplicity, efficiency, they are up to 10 times less in size and they are less ambiguous.

Architecture of gRPC is layered. In transport layer, it relies on HTTP/2 protocol, whose transfer data is always binary encoded. It also influences positively on performance. As HTTP/2 supports streaming, there is 4 different types of gRPC communications:

1. Unary RPCs. The simplest type and closer to classical RPC, where client makes a request and server returns a response.
2. Server streams. In this case, client makes a single request and server returns a stream of messages.
3. Client streams. This is the opposite type to the previous one. The client sends a batch of messages and waits for a single response to be provided by the server.

4. Bidirectional stream. This is the more complex but more dynamic solution, as both streams are independent from each other. It means also that there is no implicit termination and the stream can be closed without waiting another.

Second layer is the channel, that is a thin wrapper over the transport layer. It defines different conventions between RPC and underlying transport. RPC client pass service and method names, request metadata and messages. A call is considered as completed when server respond with the server provides header metadata, messages and response trailer metadata. The last indicates the status of response, was it successful or failure. There is no knowledge about data types, message encoding and format. Message here is just a sequence of bytes, potentially empty. Third layer is the “stub”: this is where different constraints of type and interface are defined. They are IDL-defined (Interface Definition Language), which is Protocol Buffers for gRPC.

Another interesting point to highlight, is that gRPC allows to use both synchronous and asynchronous calls and this is the developer to decide which approach to use. Generally, gRPC and Protocol Buffers lead to developer productivity as the programming model is simple to understand. Proto compiler generates a lot of code, so developer can not only be abstracted of HTTP/2 protocol itself, but also of the gRPC low-level calls. The only thing to do is to establish the connection, send messages and close them. No more need to know correct URI paths and manually setting up query parameters, requests and response bodies.

Experimental setup

To see the more expressive results, the best would be to test the jobs that implies high network load, such as file processing (that could be a text processing, image processing, audio files processing, etc.). I chose a simple job, the extraction of text from a PDF file. It means that we can upload a file by a stream (if an option supports streaming), but we can begin its processing only when the file is uploaded. gRPC supports streaming, so firstly I will try to choose an optimal chunk size to achieve better performance.

List of tasks already performed

1. Write a gRPC client and gRPC server.

List of tasks to do

1. Collect metric of first version.
2. Add ssh execution option (no RPC call).
3. Add http 2 and http 1.1 options (no RPC call).
4. Collect metrics of three last options and compare them with gRPC solution.
5. Add load balancing option to the gRPC communication.
6. Collect metrics and compare with all previous options.