PROPOSAL ARCHITECTURE FOR MICROSERVICES

By Dr. Vishwanath Rao

REST

The REST API has been a pillar of web programming for a long time.

But recently gRPC has started encroaching on its territory.

It turns out there are some very good reasons for that. In this tutorial, you'll learn about the ins and outs of gRPC and how it compares to REST.

Protobuf vs. JSON

One of the biggest differences between REST and gRPC is the format of the payload.

REST messages typically contain JSON.

This is not a strict requirement, and in theory you can send anything as a response, but in practice the whole REST ecosystem—including tooling, best practices, and tutorials—is focused on JSON.

It is safe to say that, with very few exceptions, REST APIs accept and return JSON.

gRPC

gRPC, on the other hand, accepts and returns Protobuf messages.

Protobuf is a very efficient and packed format.

JSON, on the other hand, is a textual format.

You can compress JSON, but then you lose the benefit of a textual format that you can easily expect.

HTTP/2 vs. HTTP 1.1

Comparision the transfer protocols that REST and gRPC use.

REST, as mentioned earlier, depends heavily on HTTP (usually HTTP 1.1) and the request-response model.

On the other hand, gRPC uses the newer HTTP/2 protocol.

There are several problems that plague HTTP 1.1 that HTTP/2 fixes.

HTTP 1.1 Is Too Big and Complicated

HTTP 1.0 RFC 1945 is a 60-page RFC.

HTTP 1.1 was originally described in RFC 2616, which ballooned up to 176 pages.

However, later the IETF split it up into six different documents—RFC 7230, 7231, 7232, 7233, 7234, and 7235—with an even higher combined page count. HTTP 1.1 allows for many optional parts that contribute to its size and complexity.

The Growth of Page Size and Number of Objects

The trend of web pages is to increase both the total size of the page (1.9MB on average) and the number of objects on the page that require individual requests.

Since each object requires a separate HTTP request, this multiplication of separate objects increases the load on web servers significantly and slows down page load times for users.

Latency Issues

HTTP 1.1 is sensitive to latency.

A TCP handshake is required for each individual request, and larger numbers of requests take a significant toll on the time needed to load a page.

The ongoing improvement in available bandwidth doesn't solve these latency issues in most cases.

Head of Line Blocking

The restriction on the number of connections to the same domain (used to be just 2, today 6-8) significantly reduces the ability to send multiple requests in parallel.

With HTTP pipelining, you can send a request while waiting for the response to a previous request, effectively creating a queue.

But that introduces other problems. If your request gets stuck behind a slow request then your response time will suffer.

There are other concerns like performance and resource penalties when switching lines. At the moment, HTTP pipelining is not widely enabled.

How HTTP/2 Addresses the Problems

HTTP/2, which came out of Google's SPDY, maintains the basic premises and paradigms of HTTP:

- request-response model over TCP
- resources and verbs
- `https://` and `https://` URL schemas

The optional parts of HTTP 1.1 were removed.

To address the negotiating protocol due to the shared URL schema, there is an upgrade header.

The HTTP/2 protocol is binary!

If you've been around internet protocols then you know that textual protocols are considered king because they are easier for humans to troubleshoot and construct requests manually.

But, in practice, most servers today use encryption and compression anyway.

The binary framing goes a long way towards reducing the complexity of handling frames in HTTP 1.1.

Major improvement of HTTP/2 is that it uses multiplexed streams.

A single HTTP/2 TCP connection can support many bidirectional streams.

These streams can be interleaved (no queuing), and multiple requests can be sent at the same time without a need to establish new TCP connections for each one.

In addition, servers can now push notifications to clients via the established connection (HTTP/2 push).

Messages vs. Resources and Verbs

REST is an interesting API.

It is built very tightly on top of HTTP.

It doesn't just use HTTP as a transport, but embraces all its features and builds a consistent conceptual framework on top of it.

In theory, it sounds great. In practice, it's been very difficult to implement REST properly.

REST has been and is very successful, but most implementations don't fully adhere to the REST philosophy and use only a subset of its principles.

The reason is that it's actually quite challenging to map business logic and operations into the strict REST world.

The conceptual model used by gRPC is to have services with clear interfaces and structured messages for requests and responses.

This model translates directly from programming language concepts like interfaces, functions, methods, and data structures.

It also allows gRPC to automatically generate client libraries for you.

Streaming vs. Request-Response

REST supports only the request-response model available in HTTP 1.x.

But gRPC takes full advantage of the capabilities of HTTP/2 and lets you stream information constantly. There are several types of streaming.

Server-Side Streaming

The server sends back a stream of responses after getting a client request message.

After sending back all its responses, the server’s status details and optional trailing metadata are sent back to complete on the server side.

The client completes once it has all the server’s responses.

Client-Side Streaming

The client sends a stream of multiple requests to the server.

The server sends back a single response, typically but not necessarily after it has received all the client’s requests, along with its status details and optional trailing metadata.

Bidirectional Streaming

In this scenario, the client and the server send information to each other in pretty much free form (except the client initiates the sequence). Eventually, the client closes the connection.

Strong Typing vs. Serialization

The REST paradigm doesn't mandate any structure for the exchanged payload.

It is typically JSON.

Consumers don't have a formal mechanism to coordinate the format of requests and responses.

The JSON must be serialized and converted into the target programming language both on the server side and client side.

The serialization is another step in the chain that introduces the possibility of errors as well as performance overhead.

The gRPC service contract has strongly typed messages that are converted automatically from their Protobuf representation to your programming language of choice both on the server and on the client.

JSON, on the other hand, is theoretically more flexible because you can send dynamic data and don't have to adhere to a rigid structure.

The gRPC Gateway

Support for gRPC in the browser is not as mature.

gRPC is used primarily for internal services which are not exposed directly to the world.

If you want to consume a gRPC service from a web application or from a language not supported by gRPC then gRPC offers a REST API gateway to expose your service.

The gRPC gateway plugin generates a full-fledged REST API server with a reverse proxy and Swagger documentation.

With this approach, you do lose most of the benefits of gRPC, but if you need to provide access to an existing service, you can do so without implementing your service twice.

Conclusion

In the world of microservices, gRPC will become dominant very soon.

The performance benefits and ease of development are just too good to pass up.

However, make no mistake, REST will still be around for a long time.

It still excels for publicly exposed APIs and for backward compatibility reasons.

Introduction to HTTP/2

HTTP/2 will make our applications faster, simpler, and more robust

A rare combination — by allowing us to undo many of the HTTP/1.1 workarounds previously done within our applications and address these concerns within the transport layer itself.

Even better, it also opens up a number of entirely new opportunities to optimize our applications and improve performance!

The primary goals for HTTP/2 are to reduce latency by enabling full request and response multiplexing, minimize protocol overhead via efficient compression of HTTP header fields, and add support for request prioritization and server push.

There is a large supporting cast of other protocol enhancements, such as new flow control, error handling, and upgrade mechanisms, but these are the most important features that every web developer should understand and leverage in their applications.

HTTP/2 does not modify the application semantics of HTTP in any way.

All the core concepts, such as HTTP methods, status codes, URIs, and header fields, remain in place.

Instead, HTTP/2 modifies how the data is formatted (framed) and transported between the client and server, both of which manage the entire process, and hides all the complexity from our applications within the new framing layer.

As a result, all existing applications can be delivered without modification.

Why not HTTP/1.2?

To achieve the performance goals set by the HTTP Working Group, HTTP/2 introduces a new binary framing layer that is not backward compatible with previous HTTP/1.x servers and clients—hence the major protocol version increment to HTTP/2.

Unless you are implementing a web server (or a custom client) by working with raw TCP sockets, then you won't see any difference: all the new, low-level framing is performed by the client and server on your behalf.

The only observable differences will be improved performance and availability of new capabilities like request prioritization, flow control, and server push.

A brief history of SPDY and HTTP/2

SPDY was an experimental protocol, developed at Google and announced in mid 2009, whose primary goal was to try to reduce the load latency of web pages by addressing some of the well-known performance limitations of HTTP/1.1.

Specifically, the outlined project goals were set as follows:

- Target a 50% reduction in page load time (PLT).
- Avoid the need for any changes to content by website authors.
- Minimize deployment complexity, and avoid changes in network infrastructure.
- Develop this new protocol in partnership with the open-source community.
- Gather real performance data to (in)validate the experimental protocol.

Note: To achieve the 50% PLT improvement, SPDY aimed to make more efficient use of the underlying TCP connection by introducing a new binary framing layer to enable request and response multiplexing, prioritization, and header compression; see Latency as a Performance Bottleneck.

As of now we have only tested SPDY in lab conditions.

Experiments

The initial results are very encouraging: when we download the top 25 websites over simulated home network connections, we see a significant improvement in performance—pages loaded up to 55% faster. *(Chromium Blog)*

Fast-forward to 2012 and the new experimental protocol was supported in Chrome, Firefox, and Opera, and a rapidly growing number of sites, both large (for example, Google, Twitter, Facebook) and small, were deploying SPDY within their infrastructure.

In effect, SPDY was on track to become a de facto standard through growing industry adoption.

Observing this trend, the HTTP Working Group (HTTP-WG) kicked off a new effort to take the lessons learned from SPDY, build and improve on them, and deliver an official "HTTP/2" standard.

A new charter was drafted, an open call for HTTP/2 proposals was made, and after a lot of discussion within the working group, the SPDY specification was adopted as a starting point for the new HTTP/2 protocol.