gRPC vs. REST - Which is faster?

In a world with an ever-growing amount of applications utilizing Microservices, gRPC claims to be faster and more stable than REST. Microservices can be heavily dependent on each other, which means speed and stability is key. When gRPC claims to be faster than REST, why isn't it the de facto standard? In this blog we will put gRPC and REST head to head, to see which is actually faster.

gRPC is a superior technology to REST! At least that is what this[1], this[2], this[3] and this blog[4] claims. According to all the mentioned blogs, gRPC performs better and faster than a REST on several metrics. In this blog we will test specifically, how ***fast*** a REST client can handle different request and responses, and compare it to how fast a similar gRPC client handles the same requests and responses. This begs the question...

Is gRPC faster than REST?

We hypothesize that gRPC is able to send and receive requests faster than a traditional REST. To test this, the following experiments have been developed.

[1] Battle of the API's: https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711
[2] gRPC/REST perfromance simplified: https://medium.com/@bimeshde/grpc-vs-rest-performance-simplified-fd35d01bbd4
[3] Why milliseconds matter: https://www.yonego.com/nl/why-milliseconds-matter/#gref
[4] gRPC/REST evaluating performance: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da

Contents

The experiment

To test the Hypothesis two experiments, one utilizing gRPC and one utilizing REST, is set up. These experiments have to adhere to the following:

Rules

- To ensure accurate measurements, the results must be obtained from the same computer.
- Multiple data structures will be tested.
- The setup for both APIs has to be as similar as possible.
- The time used for measuring should be obtained from the client.

Set up

- to adhere to the multiple data structures rule, a local database has been created, this database will provide a single instance of an object, as well as multiple instances of objects that will be stored in a list.
- Each API will have 12 methods to call.
    - three for a single instance which takes a parameter of Id.
        - each one with a larger payload
    - three for a single instance which takes a parameter of Id.
        - each one with a deeper payload
- three for a collection of 100 instances, which takes no parameters.
    - each one with a larger payload
- three for a collection of 100 instances, which takes no parameters.
    - each one with a deeper payload
- Each API will be tested with a client written in C#, as a console app.
- Time will be measured with .NET Stopwatch[5].
    - the stopwatch will begin when the method is called and end when the API returns the full data.
- To minimize anomalies and outliers, each operation will be executed 100 times, and the average call speed will be evaluated.

Specs

- I7-9700k, 8 Core, *4.6GHz*
- Samsung SSD 840 EVO 250GB
- NVIDIA GeForce GTX 1070

[5] .NET Stopwatch: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netframework-4.8

- 2x8 GB HyperX Fury 2666MHz DDR4 Memory

REST

What is REST?

We have decided to work with the common implementation of REST and not the full implementation of a RESTful API[6].

The key features to take note of when using rest:

Separation of client and server

- Server and client can be implemented independently without knowing each other
- Server code can be changed without affecting the client.
- Client code can be changed without affecting the server.
- Both server and client are aware of methods available.

Statelessness

- Stateless[7] means that the server is not required to know the current state of the client and vice versa.
- Either end can understand any method calls, without knowing the previous called methods.

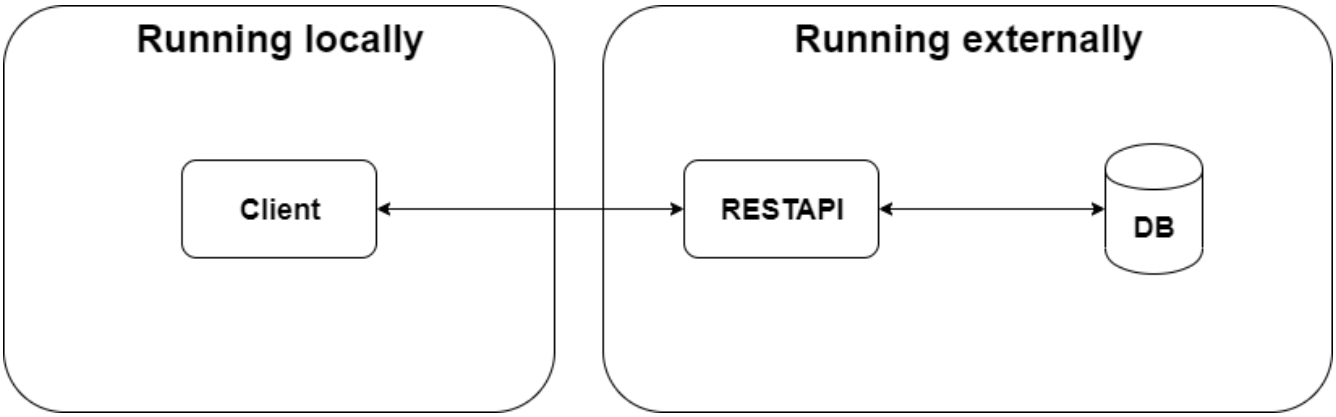
Invocation

- We invoke a method on the server via HTTP operations[8]
    - GET
    - POST
    - PUT
    - DELETE

Setting up the experiment for the REST API

The architecture for this experiment is a simple one:

[6] REST vs RESTful: https://blog.ndepend.com/rest-vs-restful/
[7] Statelessness: https://restfulapi.net/statelessness/
[8] Http Methods: https://www.restapitutorial.com/lessons/httpmethods.html



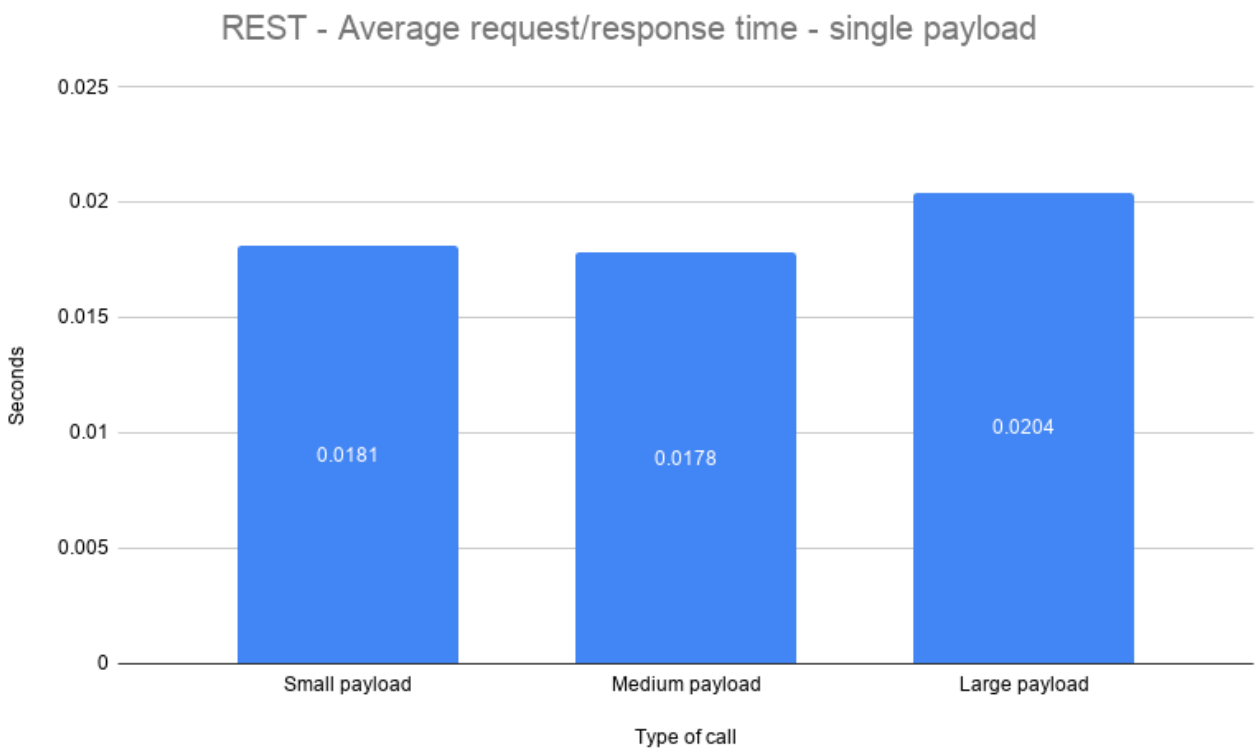
Sample project and metrics

If you want to replicate this experiment yourself, database setup can be found here and source code for the rest-api can be found here

Running our setup yielded the following results:

Single payload

The difference between a single small payload and a single large payload is small in the context of a daily task. A single small payload has a mean response time of 0.0181 whilst a single large payload has a mean response time of 0.0204 seconds. But in relation to each other it's a 12.7% increase in response time.



To put this into perspective a small payload contains 10 values of data.

A large payload contains $(4 + (6 * 9)) * 6 + 4$ or 352 values. This means that we have requested 3420% more data and it only took 12.7% longer.

To test different scenarios, we also created a "deep" payload which contains a different number of nested objects. The deepest payload contains a total of eight nested objects, however the total amount of values is far less in comparison to the previous payload. The previously mentioned payload peaked at 352 values whereas the deepest payload peaks at $(4 + (6 * 4)) + (4 + (7 * 4)) + (4 + (8 * 4))$ values, or 96 values in total. In other words, the deep payloads are much smaller in size, but different in structure.

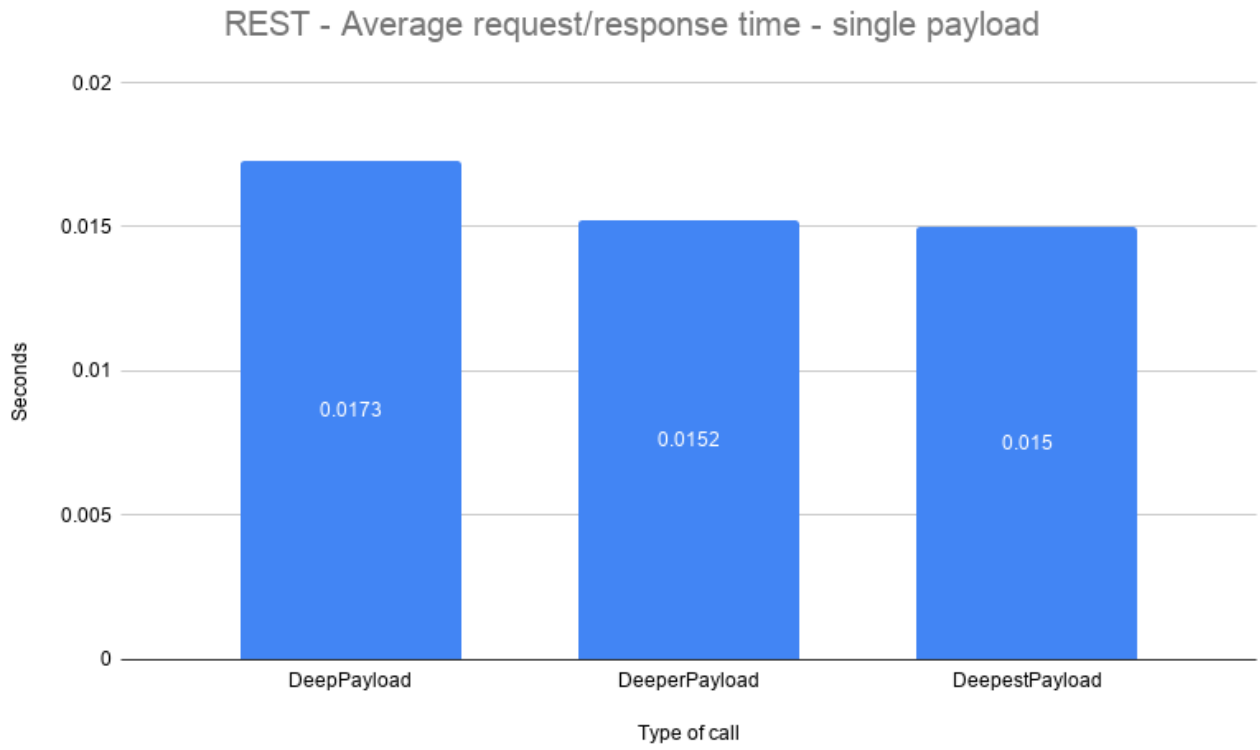
To give a concrete example, a large payload is structured like so:

```
large_payload {
   id,
   string_Value,
   int_value,
   double_value,
   medium_payload {
      id,
      string_value,
      int_value,
      double value,
      small_payload {
            id,
            string_value,
            int_value,
            double_value
         },
      small_payload {
            ...
         },
      ...
      },
   medium_payload {
         ...
      },
   ...
}
```

and the deepest_payload is structured like so:

```
deepest_payload {
   id,
   depth_seven {
      ...
   },
   depth_eight {
      ...
   },
   depth_nine {
      id,
      string_value,
      int_value,
      double_value,
      depth_eight {
         id,
         string_value,
         int_value,
         double_value,
         depth_seven {
            ...
            depth_six {
               ...
               dpeth_five {
                  ...
                  depth_four {
                     ...
                     depth_three {
                        ...
                        depth_two {
                           ...
                           depth_one {
                              ...
                           }
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

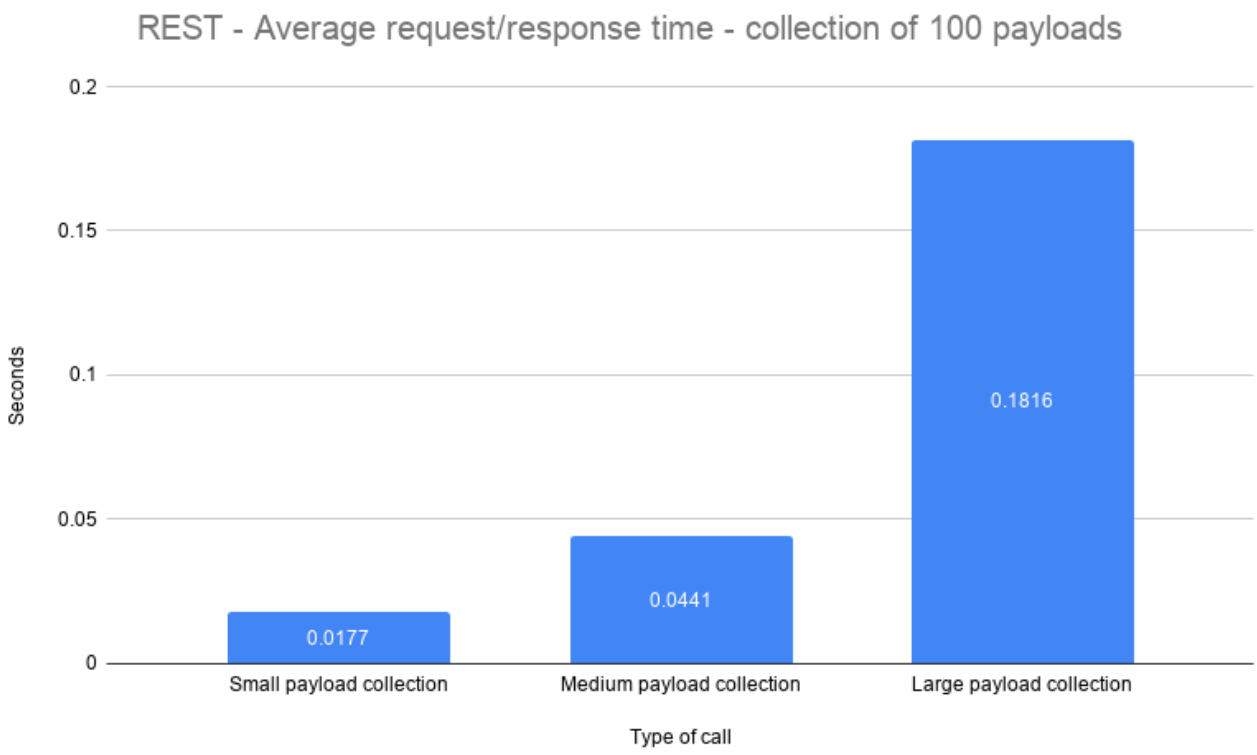
Much to our suprise, the more values, the faster the response, which might indicate possible errors during our tests, such as network outage.



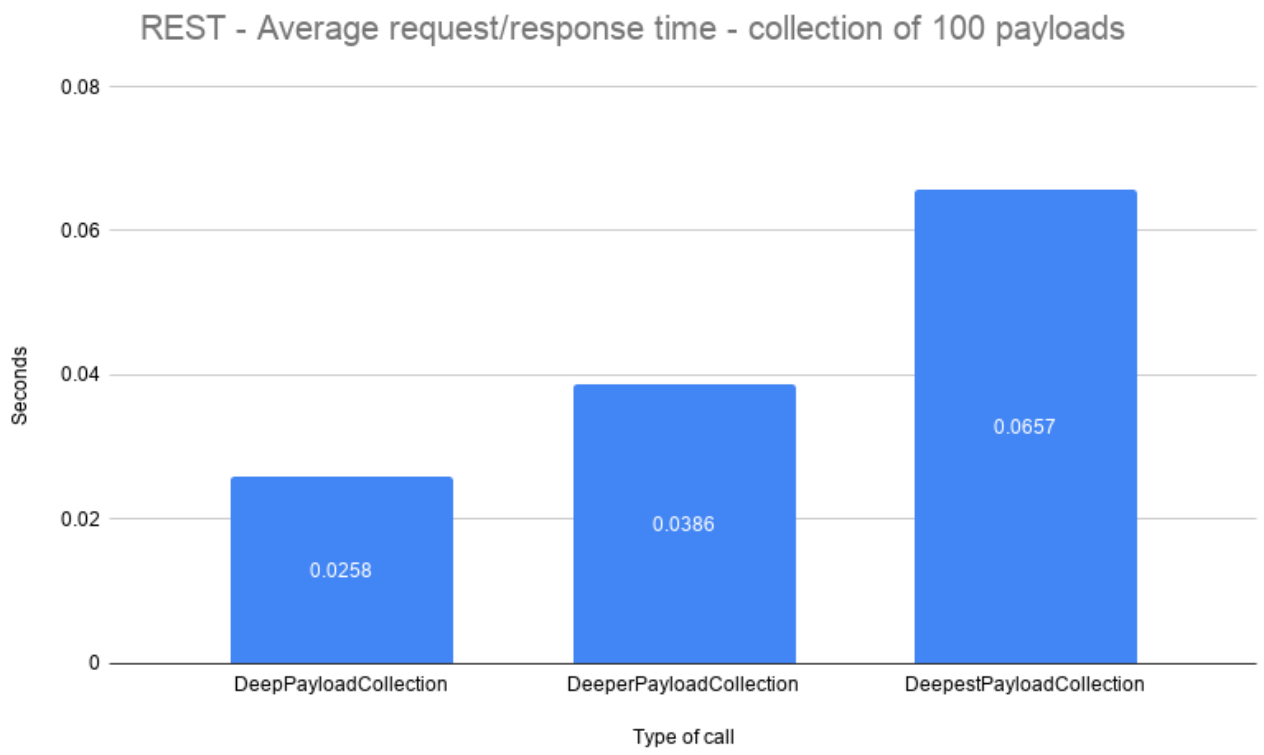
To put things into perspective, a deep payload, which contains a total of $4 + (4 + 4) + (4 + 4 * 2)$ values, or 24 values in total. Averaged at 0.0173 seconds, whereas the deepest payload, which carries a total of 96 values, which means it's 300% larger in size, averaged at 0.015 seconds. In other words, 300% more data was transferred 15.3% faster on average.

Collection of payloads

When we compare collections, the difference becomes very apparent. A small collection payload averaged at 0.0256 seconds and a large averaged at 0.1006 seconds, that is an increase of 293%. It is apparent that when it comes to moving large collections of data over the REST API, it takes a considerate amount of time compared to smaller collections.



The results are very much a like when we tested on collections of deep payloads.



gRPC

What is gRPC?

gRPC[9] is an open source RPC framework, that can run in any environment. gRPC was recently included in the .Net core platform thereby easily accessible by thousands of developers.

Some key features we would like to highlight:

HTTP/2 support

HTTP/2[10] is HTTP/1's successor, which is what most website and frameworks utilize today. In many ways' HTTP/2 is an improved version of HTTP/1, and HTTP/3 is already in the works.

Language independent

[9] gRPC: https://grpc.io/
[10] HTTP/2: https://en.wikipedia.org/wiki/HTTP/2

gRPC is language independent, which means it doesn't matter which language you develop in. The framework supports a handful of popular languages[11]. This is quite an advantage when you're developing microservices, which might have services developed in different languages and frameworks.

Contract First

gRPC is strictly contract firs[12] which is a design approach that works especially well in larger development teams. It also excels when developing microservices, as a contract has exsist, before any actual implementations can be done. The contract is designed in the .proto file[13], which is also where gRPC gains some of its speed from, seeing as .proto files are...

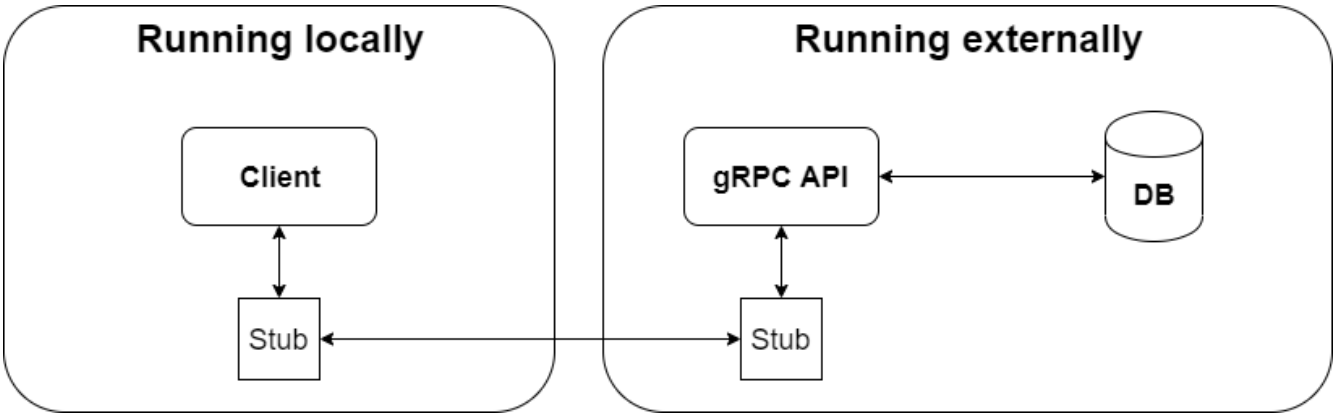
Strongly typed

As a by-product of a strongly typed proto file, which is used as contract between client and server, but also used as an extensible mechanism for serializing[14] structured data.

Setting up the gRPC project

For the gRPC architecture we use the same as the rest, we have a client and a server running locally. The client calls the methods exposed by the proto file. The method then gets executed on the server and query the database, once the data has been obtained it replies to the client. When the client has received all the data, we stop and log the time elapsed since the call started.

[11] Suported gRPC languages: https://packages.grpc.io/
[12] Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract
[13] Google proto buffers: https://developers.google.com/protocol-buffers
[14] Serialization: https://en.wikipedia.org/wiki/Serialization

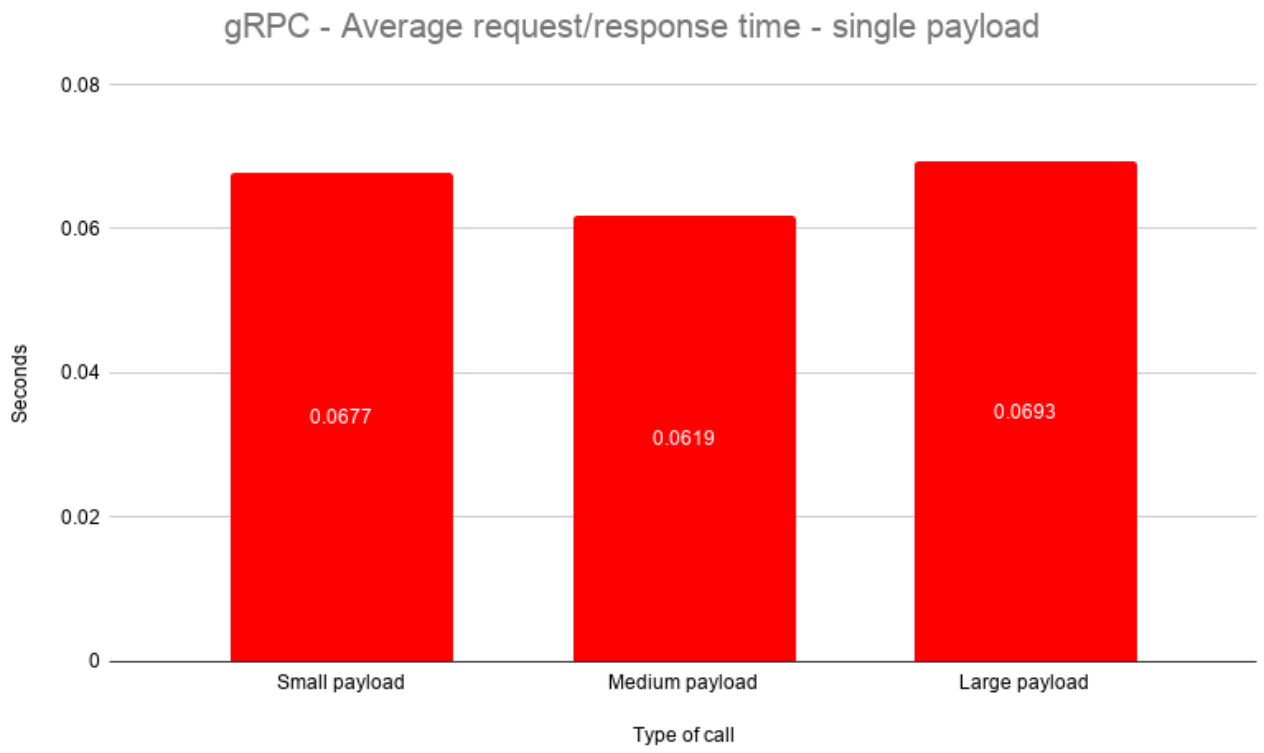


Sample project and metrics

If you want to replicate this experiment yourself database setup can be found here, and source code for the grpc-project can be found here.

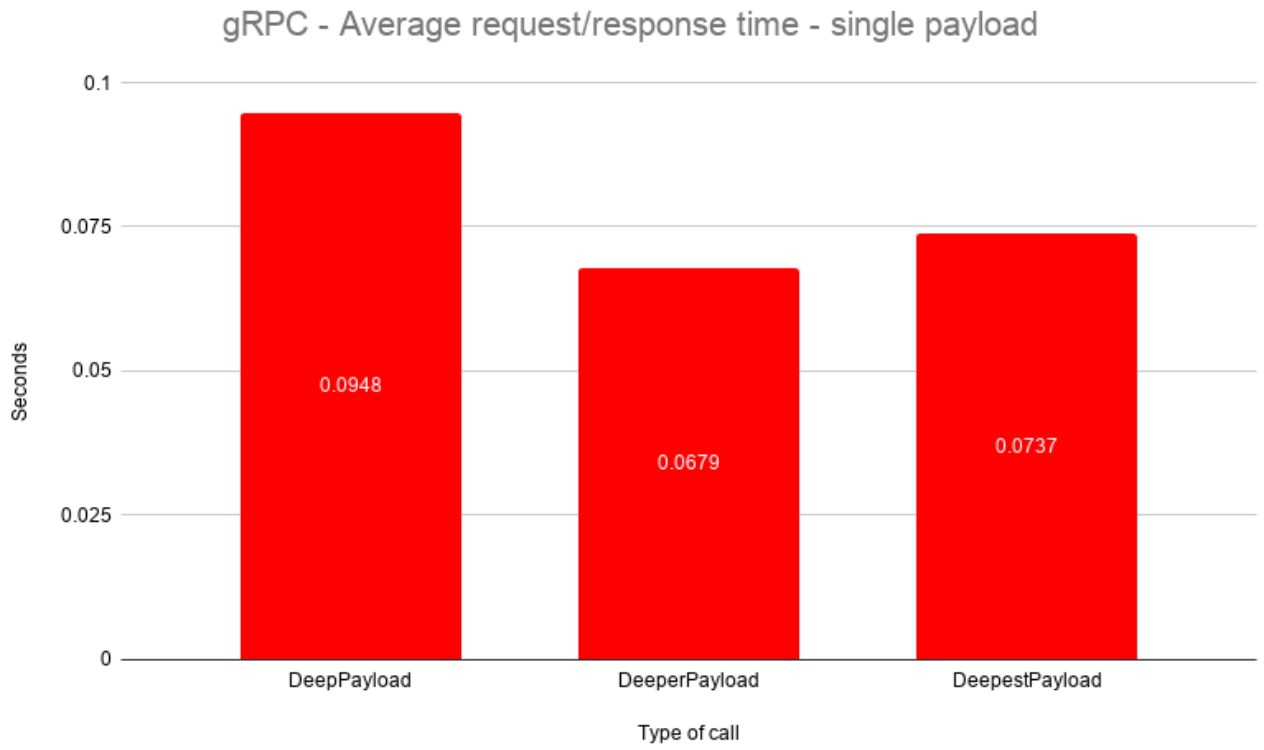
Running our setup yielded the following results results:

Single payloads



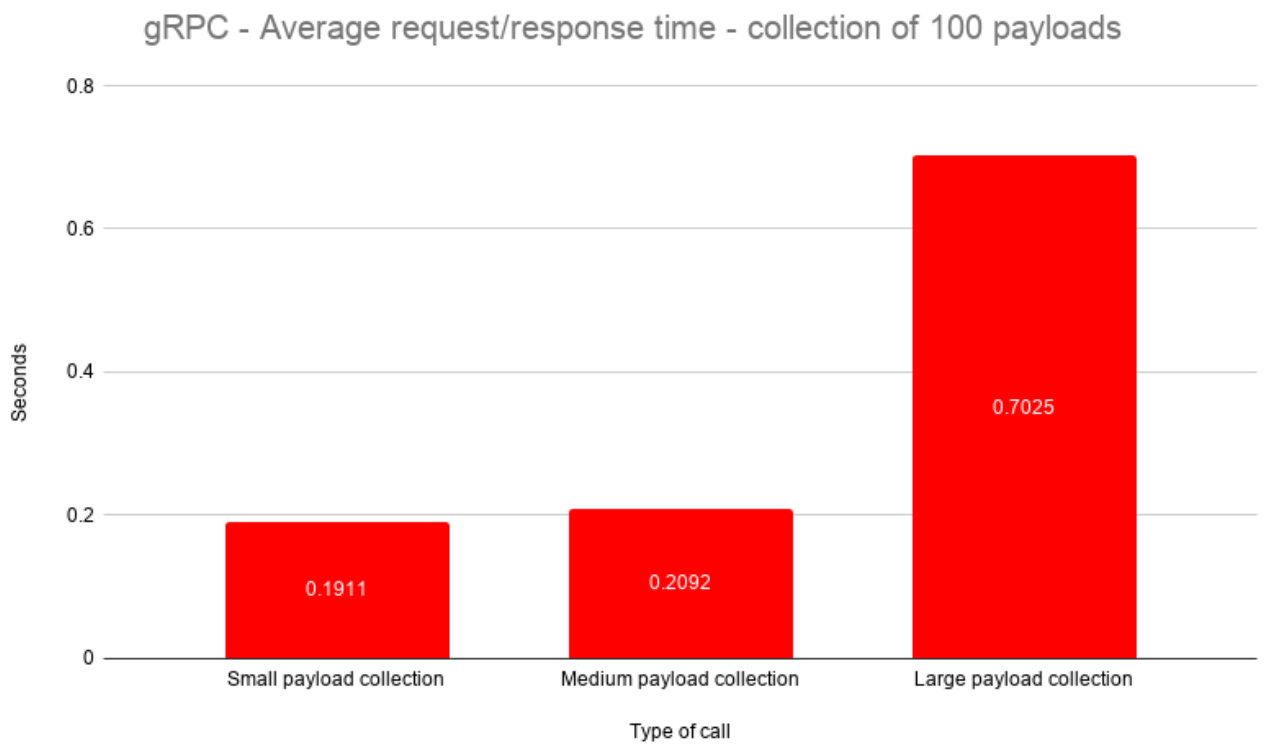
The test on single payloads yielded quite weird results, where the medium payload proved to be the fastest on average, and the largest payload only being slightly

slower than the smallest. Just to recap the numbers; a larger payload contains 3400% more data than a small payload, and yet it only took 2.36% longer to get that data.

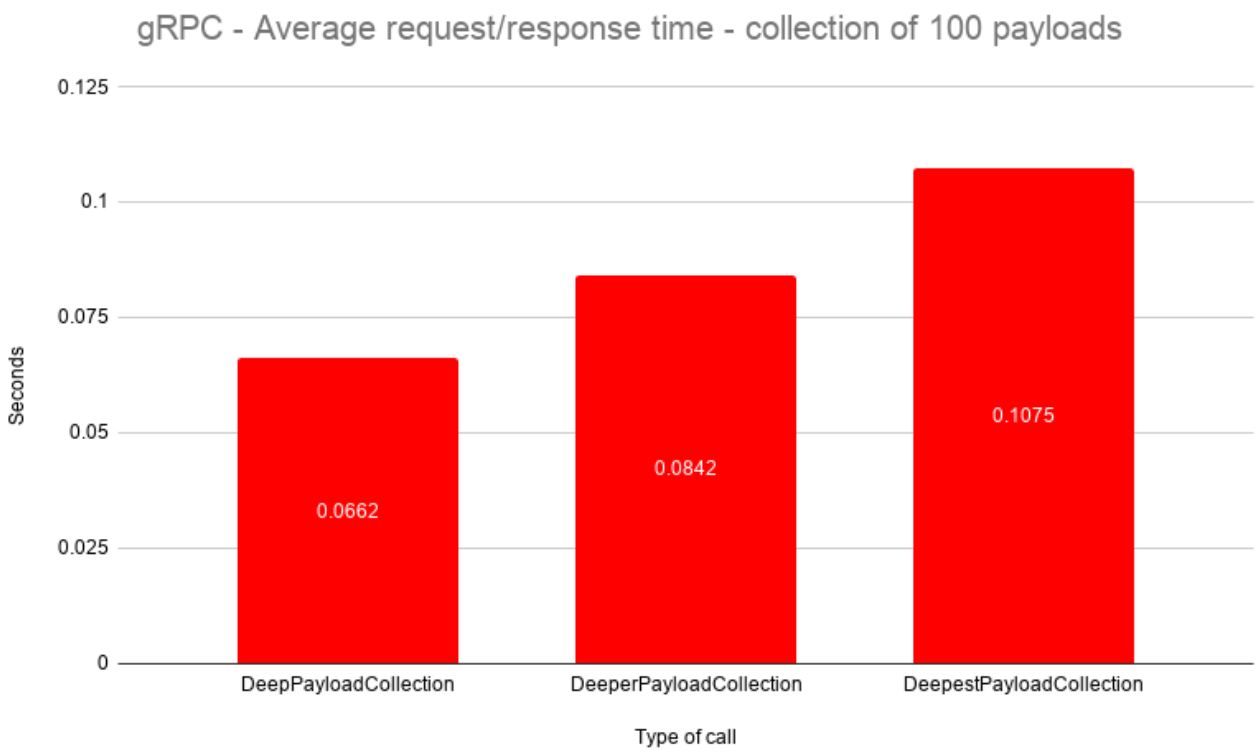


Even more weird were the results of the deep payloads. Once again, the payload containing a "medium" amount of data, was the fastest, just like previously. But unlike previously, the deepest payload was significantly faster than the deep payload, to be precise; the deepest payload, which contains 300% more data than a deep payload was 28.63% faster. As the results are rather unexpected, we have to take a close look at Possible errors that could have occurred.

Collection of payloads



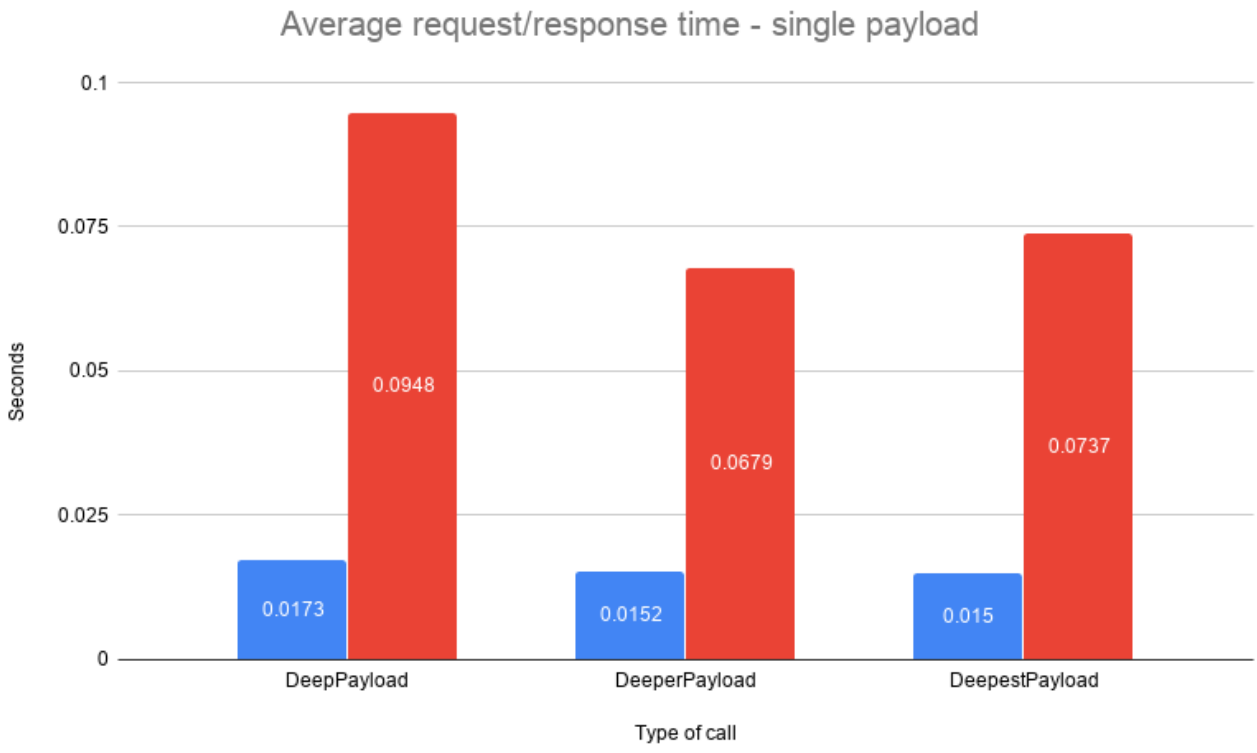
Collections paint a different picture, a small payload collection averaged at 0.01911 seconds, while a collection of large payloads took 0.7025. Which means a large payload on average took 267.6% longer to get. These results are much closer to what we would expect. We did the same test with a collection of deep payloads.

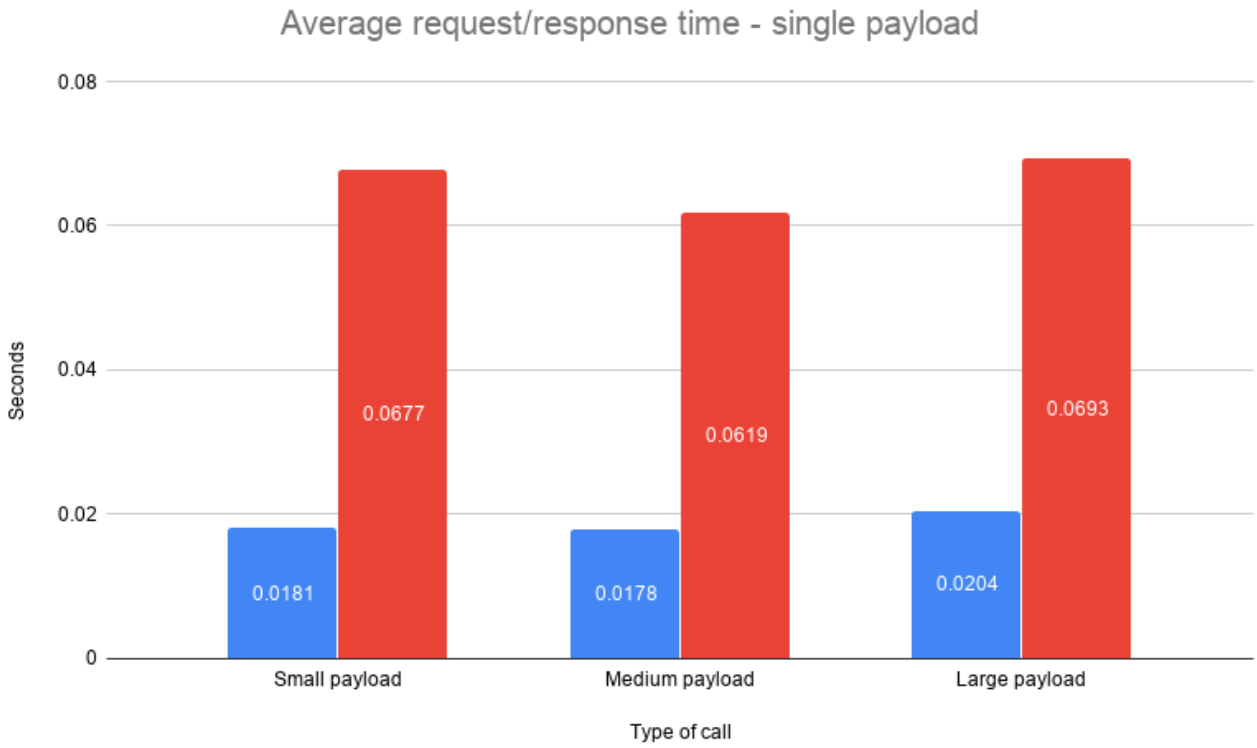


The results of this test, were as you would expect, as the payloads incrementally increase in size, they also increase incrementally in response time.

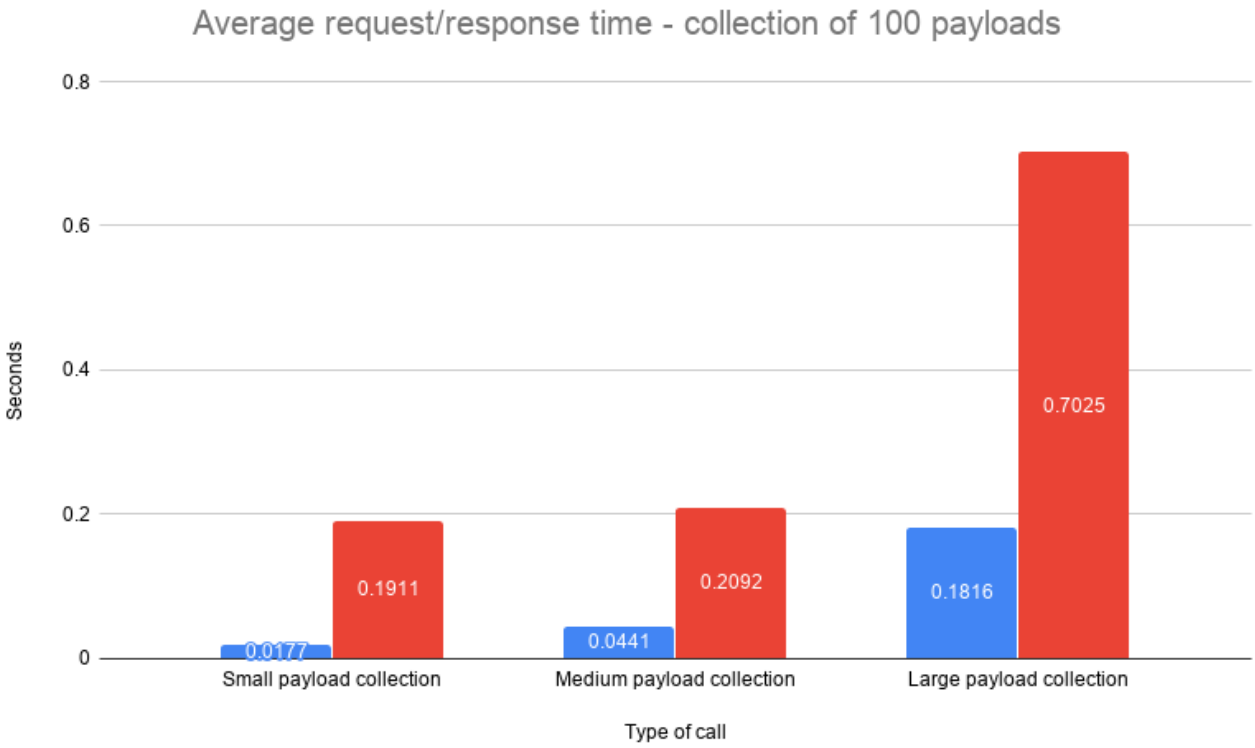
Conclusion

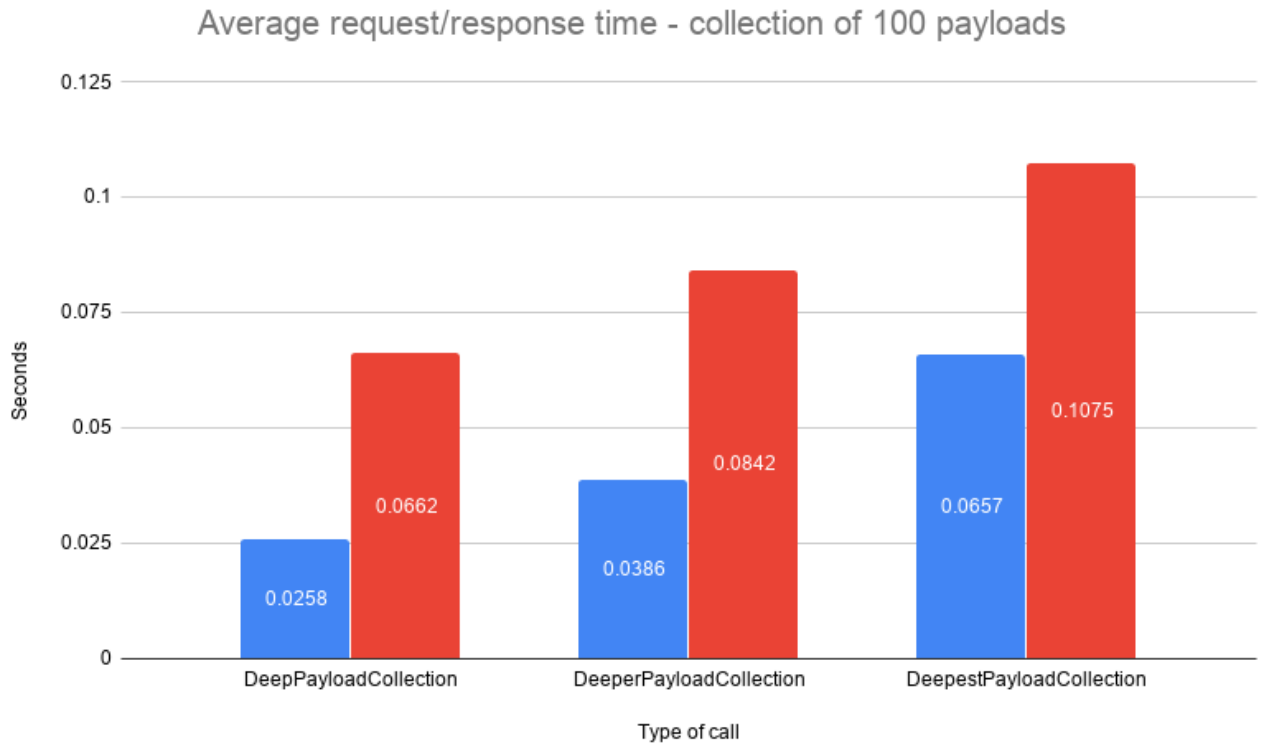
When we put the two charts next to each other, it's easy to see which one has an edge. The REST API is represented by the blue blocks, whilst gRPC is represented by the red blocks, just like previously.





This is the case for both single instances of objects as well as collections of objects.


Average request/response time - collection of 100 payloads
0.8
0.6
0.4
0.2
0
Seconds
0.1911
0.0177
0.2092
0.0441
0.1816
0.7025
Small payload collection
Medium payload collection
Large payload collection
Type of call



We hypothesized that gRPC would be faster than rest, based on the numerous blogs claiming this to be true, with their own tests. Our tests suggest the opposite being true.

These results might not seem as much, but it has been proven[15] that people on average don't wait around for data to load and will abandon a web page or program if loading times are too long. When moving large amounts of data, a small amount of time can be the difference between keeping or losing a customer.

this prompts the question: ***When to use gRPC and when to use REST?***

We would argue that gRPC fit into a setting, where you need to have multiple programs or services talking to each other across different languages, especially when the tasks that needs to connected to an endpoint is an action that needs to be executed; one such action could be TurnOnTheWater().

[15] Website should load in 4 seconds: https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds

This argument is based on the research made into gRPC, rather than the results of these particular tests.

A rest on the other hand operates on the four aforementioned HTTP operations, these operations indicated data transfers of one sort or the other. While a rest can execute the same actions as gRPC, the action TurnOnTheWater() doesn't fit into what a REST API was designed for. We would instead use REST where we required data transfers and other typical CRUD mechanics.

Possible errors

- Network outage during some of the tests
- gRPC serverside logging was set to critical, tweaking this option might yield different results.

What's next?

This blog has been only been about the differences in speed between REST and gRPC, but in reality, many other factors are present, if we were to truly compare the two frameworks. gRPC has claimed to not only be faster, but also more reliable, stable and secure, and all of these metrics, as well as other metrics, would be interesting to cover, they are however out of scope in this particular blog.

Technologies used

gRPC[9]

Net Web Api[16]

Net console app[17]

MySql Database[18]

[16] .NET Web API: https://dotnet.microsoft.com/apps/aspnet/apis
[17] .NET Console App: https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2019
[18] MySQL: https://www.mysql.com/

References

1. Battle of the APIs: https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711
2. gRPC/REST perfromance simplified: https://medium.com/@bimeshde/grpc-vs-rest-performance-simplified-fd35d01bbd4
3. Why milliseconds matter: https://www.yonego.com/nl/why-milliseconds-matter/#gref
4. gRPC/REST evaluating performance: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da
5. .NET Stopwatch: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netframework-4.8
6. REST vs RESTful: https://blog.ndepend.com/rest-vs-restful/
7. Statelessness: https://restfulapi.net/statelessness
8. Http Methods: https://www.restapitutorial.com/lessons/httpmethods.htm
9. gRPC: https://grpc.io/
10. HTTP/2: https://en.wikipedia.org/wiki/HTTP/2
11. Suported gRPC languages: https://packages.grpc.io/
12. Design by Contract: Https://en.wikipedia.org/wiki/Design_by_contract
13. Google proto buffers: https://developers.google.com/protocol-buffers
14. Serialization: https://en.wikipedia.org/wiki/Serialization
15. Website should load in 4 seconds: https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/
16. .NET Web API: Https://dotnet.microsoft.com/apps/aspnet/apis
17. .NET Console App: https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2019
18. MySQL: https://www.mysql.com/

About the authors

Mikkel Wexøe Ertbjerg

AP computer science grad.

Currently pursuing a bachelor in software development.

Can be contacted on Mikkelertbjerg@gmail.com

Nikolai Dyring Jensen

AP computer science grad.

Currently pursuing a bachelor in software development.

Can be contacted on Nikodyring@gmail.com

Nikolai Sjøholm Christiansen

AP computer science grad.

Currently pursuing a bachelor in software development.

Can be contacted on Nikolaiviby3@gmail.com