Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Microservices communication: A case study comparing the communication of microservices with different communication tools

---

Author: Svava Hildur Bjarnadóttir (2640598)

1st supervisor:	Rob van der Mei	
daily supervisor:	Lorena Salamanca	(Picnic Technologies)
2nd reader:	Patricia Lago	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 24, 2020

Abstract

Microservices have gained interest in the last years, often as an alternative to monolithic applications. Microservice architecture revolves around independent services communicating over network interfaces. For them to be a viable alternative to an application in which all components can communicate locally (as is the case in monoliths), it is important that the network communication is efficient and has good performance. REST (REpresentational State Transfer) over HTTP is the most commonly used method of communication in microservice systems, but in recent years GraphQL, gRPC and RSocket have all been introduced as alternatives to REST. This thesis aims to evaluate the suitability of these four technologies for microservices, both through a comparative analysis of their architectures and through performance experiments based on a case study for Picnic Technologies. The results of this evaluation indicate that RSocket and gRPC are well suited for inter-service communication, both showing good performance in multiple cases. Their differences lie mainly in their architecture, with gRPC being an RPC client-server framework built over HTTP/2 while RSocket is a binary protocol with symmetric interaction models intended for byte stream transport such as TCP. On the other hand, GraphQL and REST are both better suited for client-facing APIs. Performance-wise, GraphQL does especially well for complex use cases, while REST performs the worst overall out of the four. GraphQL's schema, graph-based approach and client defined queries make it a good choice for more complex microservice architectures.

Contents

CONTENTS

Chapter 1

Introduction

Microservice architecture has been one of the most popular software architecture patterns in the last years, likely bolstered by the fact that many of the largest tech companies in the world have adopted it (among those are Google (1) and Netflix (2)). Like with any system that relies on distributed components working together to form a cohesive whole, how the microservices communicate is of utmost importance. Any communication over the network adds extra overhead that does not exist in a monolithic architecture, which means that the methods, protocols and frameworks used for the communication need to be highly performant and convenient to minimize this effect. In the 2018 survey *Challenges of Microservices Architecture*, 2 out of every 5 developers noted that performance and response time were very important, both of which can be greatly affected by the communication technology employed (3). Similarly, Viggiato et al. found that 48.3% out of 122 respondents rated "Expensive remote calls" as important or very important challenges of microservices (4). In addition, Zimmerman identified 9 potential research topics related to microservices architecture, the fourth of which included the question "... is RESTful HTTP only one of several valid remote communication options in the microservices architect's toolbox, and, if so, what are the decision drivers when choosing an option?" (5). Thus, the performance of microservices communication, the choice of technology for this purpose and the design of interfaces are all highly relevant and necessary topics of interest for microservices-focused literature.

1.1 Problem statement and research questions

This thesis was written as part of an internship at Picnic, Amsterdam. Picnic is a Dutch online grocery service which handles all the logistics of procuring groceries, taking orders,

fulfilling them and delivering them to the customer's door, with most of the technology used being developed in-house (including the app used by customers for ordering groceries). Like many start-ups, Picnic started out with a monolithic architecture, but has in recent years been extracting functionality out of the monolith into smaller services in an attempt to transfer to a more modular, microservices-based ecosystem. Most of these microservices expose a RESTful HTTP-based API (see section 2.2 for more details on REST) which is used for both between-services and client-server communications. As the Picnic ecosystem grows and more microservices are extracted, it is imperative to keep the system highly performant. It is in Picnic's best interest to investigate whether their current approach is really the best one, or whether performance gains could be made by switching to a new mode of communication. In addition, performance is not the only metric that matters when choosing a technology for remote communications. The chosen style, framework or protocol should ideally also complement the main architecture of the system as well as be convenient in use, development and maintenance. Thus, the following two research questions are posed in this thesis:

1. Which communication patterns are suitable for microservices, in terms of architecture, interface design, features offered and efforts required to develop and maintain such a system?

2. How much can the performance of communication between a web application and a system of microservices improve with these new techniques, compared to a RESTful API served over HTTP requests?

The REST architectural style has been the main choice for developers building HTTP web services and APIs in the last 10 years. According to the 2020 developer survey by Rapid API, 62.5% of respondents currently use REST in production (6). For microservices specifically, a similar trend can be observed. In the 2020 survey of IT experts *State of Microservices* conducted by The Software House, 76.8% of respondents said their microservices communicate with each other over HTTP (7). However, other modes of communication have been slowly gaining traction in the last years. The three technologies chosen for comparison with REST over HTTP for this thesis were GraphQL, gRPC and RSocket.

GraphQL is a query language developed at Facebook and officially introduced in 2015 (8). It was not designed explicitly with microservices in mind, but it has seen growing interest as an alternative to REST for traditional HTTP-based APIs. According to the 2019 State of JS survey, 38.7% of web developers have used GraphQL and would use it again, compared

to 5% in 2016. Additionally, 50.6% of respondents had heard of it and would like to learn it (9). On the other hand, Google developed gRPC specifically for its internal microservices. According to the aforementioned Rapid API survey, 13.5% of respondents use gRPC (6). gRPC is, as the name suggests, an RPC framework built over HTTP/2 which aims to offer better performance than traditional HTTP-based protocols, frameworks and styles (10). The last communication mode chosen for this thesis was RSocket. RSocket is a messaging protocol designed for byte stream transport and intended to support reactive microservices, originating from a Netflix project but currently supported by a number of industry leaders such as Facebook, Netifi and Pivotal (11). The reactive focus makes RSocket an interesting candidate for Picnic and any other company embracing the reactive philosophy.

1.2 Methodology

First, a literature study will be conducted to gain insight into the current landscape surrounding this problem area. This study will aim both to provide background information into the technologies and concepts covered in this thesis, as well as identify related literature and approaches.

The first research question will be answered with a comparative analysis of the four communication technologies, supported by the specifications of each as well as a literature review. First, each technologies' usage of transfer protocols will be discussed. Next, the contracting (or lack thereof) of service interfaces will be compared between the four technologies. In chapter 5, each communication method's suitability for the microservices architecture will then be assessed based on the analysis in the preceding section. The second research question will be answered with a case study within Picnic and experimentation performed to assess the performance of each technology in a number of test cases mimicking real-world use cases, focused solely on fetching data (rather than updating or inserting it). The thesis will then conclude by combining the conclusions resulting from each research question and an attempt will be made to provide a recommendation for which communication technologies are suited to Picnic's use case.

Chapter 2

Background

In this chapter, the most relevant concepts, architectures and technologies will be covered, in order to provide detail for the proceeding chapters. First, microservices architecture will be explained and the most relevant definition of it identified. Next, REST, GraphQL, gRPC and RSocket will be covered. For each one, the motivation and basic principles will be elaborated on, as well as the basic concepts and functionality.

2.1 Microservices Architecture

In the last five years (since 2015), interest in microservices has exploded. When the keyword "microservices" is used to search on Google Scholar[1], 716 results are returned for the period 1980-2015. In contrast, the results from 2016-2020 are around 12,100. The general consensus for the origins of the term is a workshop of software architects in Venice, 2011 (12) (5). However, a few earlier papers have used the term to describe similar concepts, although these definitions do not completely match up with the current generally accepted definitions. Interestingly, it can be seen that over time, the definitions have evolved into what they are now, rather than the term being redefined for a completely different purpose. For reference, the current definitions describe microservices as small, narrow purpose, independently deployed web services which can be combined to form a larger system (12).

The first mention of microservices as an architectural term within the computer science field can be found in a dissertation by Viswanathan from 1994 (13). Viswanathan classifies wireless information services by their geographical scope, and defines microservices as services that cover an area in the range of a few miles. This definition is quite different

[1] https://scholar.google.com/

from the current one, as "micro" refers not to the size of the service itself, but rather as a description of the physical domain it covers. In 2000, Andre and Segarra (14) used the term to describe subtasks of a larger service, which approaches the modern definition while still being distinct from it. The 2002 student research paper "Vergleich Microsoft .NET mit Sun ONE" by Hockmann (15) (only available in German) contains a definition that is remarkably close to the current ones: microservices are defined as "discrete services which can be written in different languages and distributed on different platforms, and can be compiled together to form a complete web service". A similar definition can be found in the 2004 dissertation by Kim (16), where microservices are presented as "independently developable components that can be integrated together".

The unifying factors throughout the different definitions are that a microservice should be a single (or narrow) purpose independently deployable service which can be combined with other microservices to form a larger system. Different requirements are then added, e.g. that the microservice should interact via messages (17) or be developed and maintained by a dedicated, cross-functional team which handles the entire stack and life cycle of the microservice (data management, code, deployment, maintenance) (12). Some view microservices as simply the "right" way to do SOA (Software Oriented Architecture) (5) while others claim that it is a new style distinct from SOA, in part due to the wide range of architectures that have been labelled SOA, many of which differ significantly from microservices architecture (12).

The definition used within Picnic fits best with the definition offered by Martin Fowler in his 2014 blog post simply titled "Microservices":

> /textit In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. (12)

This is the definition that will be used throughout this paper. Most importantly, no requirement is put on the mode of communication, neither internally (between microservices) nor externally (between the microservice ecosystem and its clients), aside from the requirement that the communication be "lightweight".

2.2 REST

The REST (REpresentational State Transfer) architectural style was developed by Roy Fielding and published in his 2000 doctoral dissertation (18). At the time, the Web had been growing exponentially in size for a few years, beyond what Tim Berners-Lee and other early developers of the Web had designed it for. Fielding originally set out to describe the HTTP object model, but this effort quickly turned out to rather be an architectural style which, while designed with HTTP in mind, could be applied to any technology (19). As Fielding phrased it:

> *The goal has always been to maintain a consistent and correct model of how I intend the Web architecture to behave, so that it could be used to guide the protocol standards that define appropriate behavior, rather than to create an artificial model that would be limited to the constraints originally imagined when the work began. (18)*

As an architectural style, REST has several constraints that dictate what the roles and features of, and relationships between, elements should be. However, the choice whether to enforce these constraints is up to the developer of the API. The six constraints of REST are:

1. **Client-Server**: The system should be divided into client and server, with the client handling the user interface and the server handling the data storage. This constraint is driven by the principle of *separation of concerns*.

2. **Stateless**: Communication must be stateless, and any request cannot take advantage of stored context on the server.

3. **Cache**: Data within a response should be labeled as cacheable to allow clients to reuse the data later, to minimize the amount of requests made.

4. **Uniform interface**: Interfaces should be generalized, and implementations decoupled from the services they provide. The trade-off is that the information transferred is not in a form specific to the requesting application's needs.

5. **Layered system**: The system should be divided into layers, where each component in a layer is only aware of the layer it is communicating with.

6. **Code-On-Demand**: Client functionality can be extended by downloading and executing code in the form of applets or scripts, simplifying clients. This constraint is optional within REST.

The main architectural element of REST is the *resource*. A resource can represent a person, document, image, a collection of other resources or any other information that can be given a name. Resource identifiers are then used to refer to the resource, and this identifier is chosen by the author of the API. In Fielding's original dissertation, the only rule provided for how this should be done, is that the authority responsible for the assigned identifier should also ensure that the semantic validity of the identifier mapping should not change over time (18). However, over the last two decades some general guidelines have been developed, which have been understood by many to be the "right" way to do REST. As an example, the book *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces* includes extensive rules such as "Underscores should not be used in URIs" and "A singular noun should be used for document names" (20). When implemented over HTTP, the resource identifier is expressed as a URI, allowing for a tree-like representation of resources and sub-resources. Another key principle of REST is that methods to access and transform resources are the same for any kind of resource, e.g. if a REST API has a resource with the URI "/resource/resourceId" and a client invokes that endpoint with the HTTP GET method, the understanding is that the response will be a complete representation of the instance of that resource identified by the resource ID, no matter which resource it is (19).

2.3 GraphQL

GraphQL is a graph query language designed by Facebook around 2012, and officially presented at a React.js conference in early 2015. It was developed during Facebook's move from HTML5-driven applications to native ones, and still to this day powers most interactions in the Android and iOS apps (8). The main motivation behind GraphQL was to serve as a communication protocol for Facebook's client applications, or as it is worded in the official specification for the language:

> *GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions (21).*

The language and accompanying framework is now maintained by the GraphQL foundation, a neutral entity hosted by the Linux Foundation. It is still used by Facebook, and has also been adopted by prominent members of the software industry such as Atlassian, GitHub, Shopify, Pinterest, Twitter and Yelp (22).

GraphQL is a query-based language for declarative data-fetching. GraphQL has five design principles:

1. **Hierarchical**: Queries are structured hierarchically, and are shaped just like the data they return.

2. **Product-centric**: GraphQL is driven by the requirements of the engineers of the front-end clients.

3. **Strong-typing**: Each server defines a type system specific to its own structure. This type system is used to validate queries and make guarantees about the shape and format of the response.

4. **Client-specified queries**: Each server, through a schema, publishes the different functions available to clients. Clients specify how they want to use these functions and which fields and objects they want returned. This ensures that clients never receive data that they didn't specifically ask for.

5. **Introspective**: The type system of each server must be queryable by the GraphQL language itself, which in essence allows clients to request the schema itself. This helps clients know exactly which fields and types are available to them, and can even allow them to validate their queries before sending them to the server.

```
type Query {
  getUser(id: Int): User
  getUsers: [User]
}

type User {
  id: Int
  name: String
  friends: [User]
}
```

Listing 2.1: An example GraphQL schema

A minimal example of a GraphQL schema can be seen in Listing 2.1. The schema defines a single type, a User with two primitively typed fields, and one which refers to a collection of other User typed objects. It is then up to the server's specific implementation how this is stored in a database and retrieved before being returned to the client. The schema also defines a single query type, which defines the queries available for clients to use. In this case, two queries are available. The first accepts a single `id` parameter and returns the user identified by that ID. The second accepts no parameters and returns a collection of (presumably) all the users stored by the server. Listing 2.2 shows an example usage of the first query. Noticeably, the query actually allows the client to fetch much more than could be expected: since each friend is a User type, the client can recursively query the friends of each of those friends.

```
{
  getUser(id:12345) {
    name
    friends {
      name
      friends {
        name
      }
    }
  }
```

Listing 2.2: An example GraphQL query

2.4 RSocket

RSocket is a reactive message-based OSI layer 7 protocol announced in September 2018 at the SpringOne Platform conference in Washington DC. It is developed and supported by Netifi (founded by members of the team that created the RSocket project at Netflix), Facebook and Pivotal (11) and is now maintained by the Reactive Foundation, hosted by the Linux foundation (23). The foundation's members include large tech companies such as Facebook, VMWare and AliBaba (24).

The development of RSocket was driven largely by the need for a communication protocol that embraces the principles of the Reactive Manifesto (25). The following list summarizes the motivations and key design principles of RSocket:

Message-driven: All communication is modeled as non-blocking streams of messages over a single network connection.

Interaction models: Beyond the typical behavior of client-server architectures, RSocket offers the following four interaction models:

- *fire-and-forget*: The client sends a one-off message to the server without expecting or receiving a response.
- *request-response*: The client sends a request and receives a single message.
- *request-stream*: The client sends a request and receives a stream of messages.
- The client and server set up a channel where the client can send multiple messages, expecting one or more response messages per message sent.

Behavior: RSocket supports bi-directional requests, allowing both client and server to act as requester and responder. In addition, any stream can be cancelled by the requester allowing the responder to terminate early.

Resumability: Sessions can easily be resumed after network failures or disruptions with a simple handshake.

Flow control: Two forms of flow control are implemented. Firstly, the requester can dictate exactly how many messages it wants to receive at a time to prevent it from being overloaded. Secondly, a responder can issue a lease to the requester, which means that the requester knows that the server is not at full capacity, and can still receive requests.

Polyglot support: The RSocket protocol provides a contract for the behavior between the client and server. Thus, it does not matter which languages each is written in, as long as they adhere to the contract.

Transport Layer Flexibility: RSocket is not tied to TCP and is expected to work over any protocol with similar features (WebSockets, Aeron, QUIC).

Efficiency and Performance: RSocket aims to improve performance by (among other things) avoiding handshakes as much as possible, using binary encoding and allocating less memory.

2.5 gRPC

gRPC is a remote procedure call (RPC) framework developed at Google in 2015 (26). Besides powering Google's microservices infrastructure, gRPC is currently used by companies such as Cisco, Netflix and Square (10). gRPC was developed as the open-source next

version of Stubby, a non-standardized RPC infrastructure used for Google's internal microservices (27). gRPC was designed to be a open-source, free framework which promotes the microservices design philosophy of message exchange.

gRPC, like other RPC systems, is based around the idea of allowing clients to invoke remote server methods just as they would local methods. The server implements methods described by an interface and the client uses a *stub* which offers the methods described by the interface. gRPC uses *protocol buffers* to serialize structured data, which offers advantages over JSON such as field types, field IDs and rules (repeated, optional and required fields) [1]. gRPC is built on top of HTTP/2 and uses a traditional client/server interaction model, wherein the client can only request data from the server through the methods defined by the protocol buffer interface. gRPC offers both a blocking and non-blocking API, allowing clients to either synchronously or asynchronously request data from the server. gRPC also offers extensive features such as flow control, cancellations, lameducking and streaming (27). Beyond the simple request-response functionality, the three modes of streaming offered in gRPC are the following (28):

- *Server streaming*: The client sends a single request and receives a stream of messages in response.
- *Client streaming*: The client sends a stream of messages and receives a single message in response.
- *Bidirectional streaming*: The client initiates a call and sends a stream of messages to the server, which responds with a stream of messages. The streams are independent and it is up to each to decide how many messages to stream in response to each message received.

[1] It is important to note that gRPC does not require using protocol buffers, and protocol buffers can be used with any transfer protocol. However, they are both developed by Google and gRPC uses protocol buffers by default. Most documentation recommends using protocol buffers for serialization and interface contracting. Thus this thesis will in most cases assume that protocol buffers are used with gRPC

Chapter 3

Related work

In this chapter, literature and papers related to this paper's subject will be reviewed. First, literature relating to microservices, their performance and choice of communication technology will be documented. Next, papers that aim to compare any of the four communication technologies (REST, GraphQL, RSocket and gRPC) or related technologies will be covered, first in general and then specifically in the context of microservices.

3.1 Microservices performance

As mentioned earlier, the interest in microservices has exploded in the last few years. In a Google Scholar search initiated at the time of writing this paper, around 17,500 results are returned when searching for papers containing the word *microservices* in the title. However, few of them are focused explicitly on both communication and performance. As mentioned in chapter 1, the three papers by Ghofrani, Viggiato and Zimmerman (respectively) all serve as overviews of the state of practice and research into microservices (3) (4) (5). Ghofrani and Viggiato both found that performance of remote communications in microservices were rated as important challenges by professionals, while Zimmermann questioned the status of RESTful HTTP as the de facto communication technology of microservices. In Soldani's 2018 review of grey literature identifying pains and gains of microservices, API versioning, communication heterogeneity and service contracts are identified as pains, which are all highly relevant to the choice of communication technology for microservices (29). Finally, in the 2017 article *Processes, Motivations, and Issues for Migrating to Microservices Architectures* by Taibi, Lenarduzzi and Pahl (30), communication among services is named as a drawback to the microservices architecture as it adds complexity to the implementation as well as causing possible latency issues, both of which

could potentially be addressed by using a communication technology better suited to the architecture.

A number of studies were found that analyzed the performance of microservices, but these were more focused on either comparing microservices to other architectures, or investigating the effects of the infrastructure on the performance. Ueda, Nakaike and Ohara fall into the first camp. They analyzed the performance of microservice-based systems in their 2016 report *Workload Characterization for Microservices* (31). Their goal was to compare microservices to monolithic applications, however they did show that the microservice-based application spent more time in communication libraries than the monolithic one, indicating that improving the communication performance of microservices could minimize the gap in performance between microservices-based architectures and monolithic ones. Similarly, Gan and Delimitrou showed that in a microservice-based system, communication is a more significant bottleneck than computation when compared with the same application written as a monolith (32). On the other hand, Amaral et al. (33) and Kratzke (34) each performed research on the performance of microservices, but both focused on the impact of containers (e.g. Docker) and the underlying infrastructure used to deploy the services, rather than the design or architecture of the microservices themselves.

3.2 Communication technologies in web services

A number of papers have been published which compared GraphQL to REST APIs in terms of either architecture or performance, but not specifically for microservices. Eizinger's 2017 master thesis compared REST and GraphQL in terms of API design, finding that using GraphQL shifts responsibilities from the client to the server (35). Vogel et al. conducted experiments on the same server, one before migrating fro mREST to GraphQL and one after and found that for a single request the performance was similar. However, for cases where multiple resources were required, GraphQL was faster due to a reduction in the number of requests needed (36). Similarly, Vázquez-Ingelmo et al. migrated a system from REST to GraphQL and observed a decrease in average response time due to a reduction of request volume (37). Finally, Cederlund performed experiments on the same system developed with either REST or GraphQL, and showed similar results as Vogel et al. with a similar performance in single-resource test cases but increased performance by GraphQL in multiple-resource test cases (38).

However, as far as the author knows, few research-based literature has been published comparing either REST or GraphQL to gRPC and RSocket. Chamas, Cordeiro and Eler

analyzed the energy cost differences of REST, SOAP, Socket and gRPC, however they did not consider the performance (e.g. speed) of the technologies (39). Multiple papers have been written that seek to compare and contrast REST with RPC frameworks in general, however these are less relevant since even though gRPC is an RPC-based framework, the papers focus mostly on SOAP or other synchronous RPC approaches (while gRPC can be both synchronous and asynchronous) (10, 40, 41, 42, 43). Similarly, Socket communications have been compared with REST (44, 45), however RSocket offers functionalities beyond plain socket communications which these comparisons do not take into account.

3.3 Communication technologies in microservices

For analysis of architecture or performance of communications specifically in microservices, a limited number of papers exist. Hong, Yang and Kim conducted a performance analysis of RESTful APIs versus RabbitMQ for microservices, finding that RabbitMQ (a message broker and queueing server (46)) performed better than REST, with more stability in situations of load (47). However, this thesis focuses more on comparing frameworks and protocols that are centered around direct message transport rather than queueing systems. For this reason, RabbitMQ was not included in the evaluation.

Nguyen performed experiments to compare the performance of three RPC-based frameworks in microservices architecture in her 2016 thesis, concluding that even though gRPC did not have the best performance in all cases, it is a good choice for a microservices architecture based on other advantages, such as extensive documentation, active development and a "modern style" (48).

The paper most directly related to this thesis (and the original inspiration for it) is the 2017 master's thesis *Efficient Communication With Microservices* by Johansson (49). Johansson compared the performance of REST over HTTP with JSON with GraphQL and found that GraphQL was slower in all cases. However, there were issues with his methodology and experiment design. None of his test cases showcased the most prominent benefit of GraphQL: that fewer requests are needed to fetch the same data. Thus, his conclusion that GraphQL has a worse performance than REST is incomplete. In addition, he included a small section on gRPC and Apache Thrift, but did not provide arguments supporting his decision to not include those in the performance comparison.

Chapter 4

Architecture comparison

In this chapter, a comparative analysis of the four technologies will be conducted. First, the choice and usage of transfer protocols for each will be investigated, with a focus on improvements over HTTP/1.1. Then, the approaches each technology uses for interface design and contracting will be compared.

4.1 Transfer protocols

For microservices, communication is usually done via HTTP (5), and before the release of HTT/2 in 2015, that was via HTTP/1.1. From 1999 until 2014, the HTTP/1.1 standard remained unchanged (50). One of the prominent issues with HTTP/1.1 is the fact that, for each transaction, a new connection must be established which increases overall latency when performing multiple requests (51).

Since REST was designed with HTTP in mind, it is rarely used with anything else (although that is possible). That means that RESTful APIs usually suffer from this performance issue of HTTP/1.1. Furthermore, since a URI should only ever refer to a single resource within REST(20), when a client needs to fetch multiple resources, multiple requests need to be issued and in turn, multiple connections need to be established.

GraphQL, while similarly platform independent yet rarely implemented over anything but HTTP/1.1, attempts to mitigate this issue by reducing the amount of requests needed to be made. By allowing clients to refer to any related resources in their queries, the overall request volume needed to fetch data can be decreased, improving performance (8). In the 2017 study on the performance differences between REST and GraphQL mentioned in chapter 3, Vázquez-Ingelmo et al. found that the average response time of loading a

specific dashboard decreased by 2 seconds, largely due to the fact that in REST, the action required 17 requests while with GraphQL, one was sufficient (37).

gRPC however, is designed for and bound to HTTP/2. HTTP/2 is a new version of HTTP which adds multiplexing, allowing clients to send and receive more than one request/response message over the same connection (52). This allows gRPC clients to avoid the latency induced by having to re-establish a connection for each request, improving the overall performance. In addition, gRPC has no constraints or guidelines on the semantics of methods defined, meaning that if desired, developers can create methods that fetch all the data needed in a single request. Finally, HTTP/2 includes the feature of header compression, whereas in HTTP/1.1 headers are uncompressed and sent in plain text. This enables HTTP/2 payloads to be overall smaller, saving on bandwidth (52).

Finally, RSocket can bypass the issues of HTTP entirely. RSocket is a protocol defined at the same level as HTTP and works over binary, duplex protocols such as TCP. RSocket thus can also enjoy the benefit of multiplexing. Further, unlike all versions of HTTP currently in use, RSocket allows both the client and the server to act as sender and receiver. In a system that relies on e.g. push notifications, this can improve performance. With a HTTP-based implementation, the client would periodically need to poll the server to check for new notifications, potentially performing hundreds of requests without any meaningful response. With RSocket, the server can push data to the client only when there is data to push, reducing the overall amount of data sent between the two (25). RSocket also reduces the bandwidth used by utilizing binary encoding, which makes the message less human-readable as opposed to a text-based encoding (like HTTP/1.1 uses) but is also more efficient (53).

4.2 Interface design/contracting

The interface contracting of REST lies mainly in the usage of HTTP verbs (`GET`, `POST`, `DELETE` etc.) as well as the choice of URI for each endpoint. As discussed in section 2.2, the same verb should consistently have the same semantics for each resource type. If a `GET` request is used to implement a method which fetches the entire representation of the resource identified by the URI, the same should go for any other resource. Throughout REST's lifetime, several guidelines and rulebooks have been created by other individuals than Fielding himself, detailing how the URIs should be constructed. However, none of these rules or guidelines are enforced by REST, since it is an architectural style rather

than a protocol or framework. Developers are free to break the rules as they please, which they often do to circumvent issues caused by strict adherence to REST.

The two most common of these issues have been referred to as *over-fetching* and *under-fetching*. *Over-fetching* results mainly from the aforementioned constraint for uniform HTTP method behavior. Since a `GET` request usually returns the resource's entire representation, that also means that in use cases where a client only really needs a subset of the fields, all of them will nevertheless be returned, using up bandwidth for data that is not needed. A common workaround for this is to define new representations of the same resource, tailored for each use case. This can quickly become confusing (especially when documentation is lacking) and cause a bloat in the codebase. *Under-fetching* is caused by the RESTful constraint that a URI should refer to a single resource. When a use case requires multiple resource types, this means that multiple requests need to be issued to the server, one for each resource type. In more complex use cases, this can cause a degradation in performance since each request adds network overhead with an increase in latency. This is often circumvented by developers defining custom, "ad-hoc" endpoints which return two or more resources at a time. These perform better, but do not conform with the REST style specification and should be avoided by anyone striving for a truly RESTful API.

GraphQL queries are pattern matching queries represented by a JSON[1]-like tree structure with a specified type system defined by the server which is queried. The following query is a simple example of a GraphQL query, which locates a user with the ID `4` and returns their name, along with the names of the user's friends.

```
type Query {
  getUser(id:4) {
    name
    friends {
      name
    }
  }
}
```

The type system of the GraphQL server is defined by a schema which specifies the fields belonging to each object type, and which values are allowed by each field (e.g. scalar or even other objects). This schema is also represented with a JSON-like structure. Upon its release the query language offered two different operations, *query* and *mutation*. The query operation is a read-only pattern-matching fetch operation, while the mutation is a write operation followed by a fetch. A single GraphQL request is referred to as a *document*

[1] JavaScript Object Notation (54)

and may contain multiple operations of any type as well as fragments, composition units allowing for query reuse.

The query functionality of GraphQL has several advantages over RESTful interfaces. The client-defined queries address both over- and under-fetching in one fell swoop by allowing clients both to specify exactly which fields they need, as well as utilizing the graph-like representation to allow clients to fetch not just one resource, but also any resource directly related to it. Furthermore, GraphQl offers benefits such as introspection (allowing clients to inspect the schema itself) and strong typing (enabling validation and guarantees about the shape of the data returned).

For gRPC, *protobufs* are typically used to define a typed schema for service interfaces. Data structures are defined as messages in a `.proto` file, which is then used by a specialized compiler to generate code in the language preferred by the developer. This protobuf can then be shared with clients, to allow them to generate their own client-side stubs of the methods. Listing 4.1 shows how an object might be defined in a protocol buffer file.

```
message User {
    string user_id = 1;
    string first_name = 2;
    repeated string phone_numbers = 3;
}
```

Listing 4.1: An example gRPC protocol buffer message definition

The `repeated` keyword indicates that the `phone_numbers` field represents a collection (array, list etc) of strings. The integers assigned to each field represent the field's ID. If the server's *proto* definition changes a field name, clients relying on older versions will still attempt to serialize based on the old specification. If the types do not match, a serialization error occurs. If a field is missing from the response, the client simply skips it in the serialization. This can cause issues if a message type changes often. Either the field is removed, running the risk of someone attempting to reuse the ID (which causes problems if the clients still include the old field) or the proto file slowly accumulates deprecated fields, causing bloat. The proto file is also used to define the services and methods available on the server:

```
service UserService {
    rpc GetUser (UserRequest) returns (User) {}
    rpc getUsers (UsersRequest) returns (stream User) {}
}
```

Listing 4.2: An example gRPC protocol buffer service definition

Each method accepts one message type as parameter, and returns either a single message type, or a stream of one message type. Both the parameter and return type can only be a message, meaning that any primitive parameters need to be enclosed in a message type. Similar to GraphQL, gRPC can avoid over-fetching through the usage of protobuf files. Unlike GraphQL however, this is not done through client-specified queries or methods. Any method and resource payload needs to be separately defined and tailored to each client's needs by the server itself. This can cause some bloat in the same way that multiple representations do for REST, but at least development time is minimized by the fact that the actual code needed to implement these representations is generated. Since gRPC does not set any rules for the semantics of methods, under-fetching can also be avoided by defining methods tailored to each use case, just as one would do in a non-distributed system. gRPC has another disadvantage to GraphQL in regards to introspection. GraphQL's introspection allows clients to always be up-to-date on the construction of the schema, since the actual state of the schema at any point is returned immediately. In contrast, the protobuf files are statically shared with clients, causing the client copies to slowly drift out of sync with the schema as it is updated and changed. When relevant changes to the clients are published, a new version of the file needs to be manually sent to them. Developers can set up a system to make sure clients always have the newest version but this is not implemented by gRPC and depends on external systems or frameworks.

Finally, RSocket's only form of interface contracting is through the usage of *mappings* to identify endpoints. There are no guidelines or rules for the semantics of these mappings and the developer is free to choose any string to represent and identify the methods. Methods can also be differentiated through their interaction models, which is usually implemented at the client-side with a specific method for each while at the server-side, the methods are identified via their accepted arguments and return types. A request-response method accepts a single argument and returns a single object while a fire-and-forget method accepts a single argument and has no return type (void). A request-response method accepts a single parameter but returns a stream of objects and finally, a channel method accepts a stream and returns a stream.

The lack of strict interface contracting and rules gives developers vast control over their API design. A method can easily be tailor-made to any use case required by clients, which would increase run-time performance and efficiency. With this control though, also comes a lack of restraint regarding the quality of the interface. An inexperienced developer could freely design an inefficient and confusing interface, and the lack of introspection or documentation about the interface could cause issues for clients and their developers. In addi-

tion, RSocket does not offer an efficient solution to the over-fetching problem. To prevent it, developers would have to resort to the same workaround as mentioned for REST (multiple representations of the same resource), with the same drawbacks. However, RSocket does offer more efficient encoding of the data than HTTP, minimizing the bandwidth used to transfer the same data.

Chapter 5

Suitability for microservices

In this chapter, the first research question will be answered, assessing the suitability of each of the four technologies for microservices based on the analysis in chapter 4.

5.1 REST

REST has been the main choice for web services for several years now, and not without reason. It was designed for HyperText-driven applications (19) and performs very well for that purpose. However, it was not designed with complex service-to-service communications in mind, where the services typically don't even deal in hypermedia at all. Hypertext and hypermedia are designed to be read and consumed by humans, but in a service-to-service case, no humans are involved on either end. The data is sent from a machine, and received by a machine, and neither end cares that the payload can be interpreted by a human if intercepted along the way. In addition, the first constraint of REST contradicts its usage for inter-service communication, as both actors in the data exchange are in charge of their own data storage. As is often the case, a tool designed for the job at hand usually suits better than a tool designed for a different purpose. REST could however serve well for the client-facing API of a microservices system, for the data which is intended to be consumed by humans. In a system with a simpler data model, where use cases require fewer resources, REST is a good choice as it is familiar to most developers and simple and easy to learn for new developers. For a more complex system where performance is paramount however, there are better options.

5.2 GraphQL

GraphQL was also not designed for microservices and instead intended to support native client applications (8). As such, it is highly suitable for client-facing APIs in a microservices-based system, potentially more so than REST in more complex systems. The advantages of GraphQL over REST regarding the issues of over- and under-fetching have already been discussed in the previous chapter. The issue of under-fetching is even more relevant for microservice architecture specifically. Resource types can often originate from different microservices, which means that when a client only requests a subset of the resources possible in a query type, fewer microservices will be involved in the transaction. This is not communicated to the client and in fact, the schema and single endpoint can essentially abstract away from the client that the system is composed of multiple services, making the whole system appear as one whole. The architectural pattern *API Gateway* has been used by some to achieve this same goal, by routing all requests through a single gateway with a single interface. In Taibi's study of microservices architectural patterns two of the downsides mentioned for the API Gateway pattern are:

- *Implementation complexity*: Due to the implementation of several interfaces for each service.
- *API reuse issues*: Clients that use the same interface all need to be updated when the interface is changed (30).

GraphQL helps address these with the unified schema and client-defined queries, mitigating the issues of the API gateway pattern. Another pattern mentioned in the same study is the *service discovery* pattern, where DNS addresses of services are dynamically resolved through the use of a service registry. The schema of GraphQL can also support this pattern by keeping the interface stable even when a service is changed, to minimize the impact on other services that communicate with it.

5.3 gRPC

gRPC on the other hand was designed explicitly for microservices and its features support the architecture's requirements, especially between services as it is used within the Google microservice ecosystem (27). The multiplexing, flow control and possibilities of both asynchronous and synchronous calls can offer performance and efficiency benefits, as discussed in section 4.1. Each gRPC service exposes a strongly typed interface through

its protocol, providing many of the same advantages as GraphQL offers with its schema, although it does not feature client-defined queries. gRPC, as an RPC framework can be simpler in development than others, due to the fact that methods are designed and used in the same way as local methods, with the specifics of how those methods get translated to network requests hidden from the developers. In addition, the usage of protocol buffers can minimize the amount of time spent developing, since the boilerplate code[1] for each message type is automatically generated. However, the code generation is limited by the libraries and plugins available for each platform. Plenty of libraries exist that generate boilerplate code, unrelated to protocol buffers, and if the developers have a preference for those it can limit the usefulness of protocol buffers since objects now need to be converted between the different types.

5.4 RSocket

RSocket was also designed for microservices and offers features such as multiplexing, flow control and asynchronous communication. What sets RSocket apart from the other three is the non-adherence to the classic client-server model. For a microservices system, the client-server model makes little sense as the services behave more like peers, and should be able to freely communicate with each other no matter which service initiated the interaction (in contrast, gRPC's bidirectional streaming mode can only be initiated by the client). In addition, RSocket does not rely on HTTP, and allows for a lot of freedom in how connections are managed between the services, and in how the interface is designed. With this added freedom though comes a certain complexity in development, since developers need to evaluate their options and make design decisions that may be sub par and could negatively affect the performance or efficiency of the final product.

[1] Such as constructors, builders, getter and setter methods

Chapter 6

System architecture

In this chapter, the system used as basis for the performance experiments will be described. Its data resources, architecture and implementation details will be covered as well as the general deployment setup.

6.1 Case description

The case is based on a proposed service within Picnic which aims to aid in communication between customer service agents and managers at distribution centers. This service is essentially an API gateway that routes requests to and from other microservices. The main data resources of the system are:

- `User` : Represents a Picnic customer who has a user in the Picnic app.
- `Delivery` : Represents a collection of groceries that should be shipped together at a certain point to a customer's address.
- `Trip` : Represents a single trip made by a courier in a specific time window employed by Picnic, in which he/she delivers a number of deliveries to customers.
- `DistributionCenter` : Represents a location where groceries are received from fulfillment centers, and where couriers start and end their trips.
- `DeliveryResult` : Represents the results of a delivery (e.g. whether it has been confirmed/cancelled/completed, which totes (crates containing groceries) were delivered etc).

Figure 6.1: Relations of the different resources.

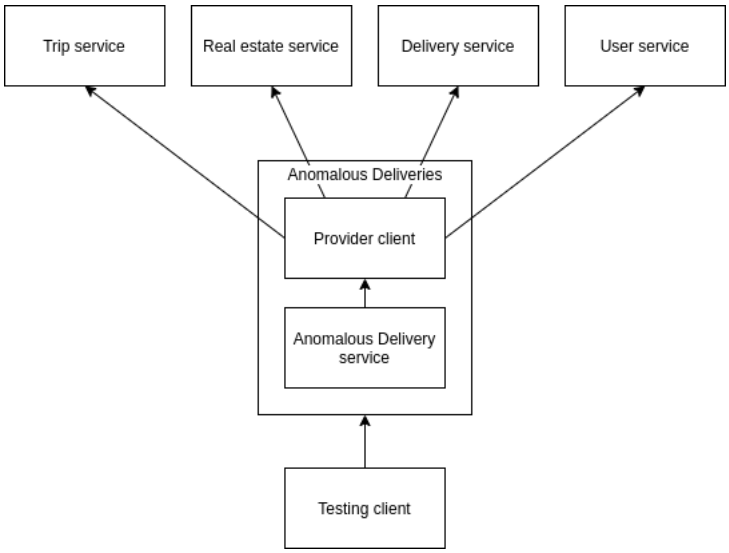

Figure 6.2: The architecture for each service.

The relations between these entities are pictured in Figure 6.1. The general architecture of each service is represented in Figure 6.2. The four services depicted at the top (delivery, real estate, user and trip services) represent real microservices within Picnic.

6.2 Architecture

Each service implemented 5-6 methods/endpoints for fetching the data. These methods were based on the existing methods in the four Picnic microservices:

- `getDeliveryById`: Accepts a single string parameter, returns a Delivery.
- `getDeliveryIdsByTripId`: Accepts a single string parameter, returns a list of strings.
- `getDistributionCenterById`: Accepts a single string parameter, returns a DistributionCenter.
- `getUserById`: Accepts a single string parameter, returns a User.

- `getTrips`: Accepts two date-time objects and a string, returns a list of Trips with a matching distributionCenterId which have a start time which falls between the two time instances supplied.

In addition, REST, RSocket and gRPC also implemented a sixth method:

- `getTripOverviews`: Same parameters as `getTrips`, but returns a list of `TripOverviews`. Each `TripOverview` has all the same fields as a Trip, as well as a list of `DeliveryOverviews`. Similarly, each `DeliveryOverview` has the same fields as a `Delivery`, as well as a field for the respective User.

This method allows the other three services to fetch a collection of trips, the deliveries associated with them and the users associated with the deliveries, in a single request. The same was not needed for GraphQL, since GraphQL allows clients to specify which fields are needed, so `Deliveries` and `Users` can be included or omitted without any change in which method is called.

GraphQL and gRPC both offer the option of defining the set of fields in a resource deemed relevant for clients; GraphQL through its schema and gRPC through the proto file. Thus, for all resources the expected size of returned resources should be much smaller than for RSocket and REST. As an example, the `User` Java class imported from the Picnic ecosystem and returned in full by REST and RSocket contains 24 fields of types ranging from primitive (string), to complex, to sets of either type. In contrast, GraphQL and gRPC each only return 4 fields of a user, the `userId`, `firstname`, `lastname` and `contactEmail`.

6.3 Implementation and deployment

The services were all developed in Java Spring. That is both the technology of choice within Picnic, as well as convenient since a Spring Boot configuration already existed for each of the four communication methods. Efforts were made to keep the development of each service as homogeneous as possible, to ensure the performance was only affected by the specifics of each communication protocol/style. This included using Reactor for any reactive components, Jackson for serializing JSON (except for gRPC, which uses protocol buffers) and (where possible) using the same Java objects for the same resource (usually the ones defined in the existing microservices of Picnic). GraphQL and REST both use the Spring Webflux framework for HTTP communication and all four services use Netty as the server framework.

Each service was wrapped in a Docker container and deployed to a Kubernetes cluster, to mimic the setup of a typical service within Picnic. The same configuration was used for each one to make sure no external variables affected their performance.

Chapter 7

Experiment design

In this chapter, the design and implementation of the test cases used as the basis for the experiment will be discussed, as well as the metric used for the performance measurements.

7.1 Test cases

The primary goal of the experiments was to measure the overall time it took to execute a test case, thus providing insight into the latency of the different communication methods and the differences in performance. All the testing code was developed with Node.js and using the core module `perf_hooks`, a time measurement was taken right before each test case iteration was set off. Another measurement was taken as soon as all data had arrived from the server, and the difference of these two measurements (in milliseconds) written to a file to register the total time taken to request and receive all the data required by the test case. Any work needed to set up connections before a request was made (as is needed for RSocket and gRPC) was included in the performance measurement to ensure a fair comparison. Furthermore, measures were taken to ensure the network conditions were also fair, by running all tests at the same place, with the same network setup and at around the same time of day.

For each service, five test cases were defined, with the exception of GraphQL which had four (further explained below). These five test cases were designed to test a variety of scenarios representing real-world use cases, with a mix of single and multiple resources and sequential and parallel execution. The five test cases are as follows:

1. Single resource: A single request-response for a single resource

2. Multiple related resources

   (a) *In one request*: Fetch all resources in one ad-hoc request. The logic for connecting the resources together is on the server side.

   (b) *In multiple requests* : Sequentially fetching each resource type in a separate request, connecting the resources together client-side.

3. Multiple unrelated resources

   (a) *Parallel*: Initiate the requests asynchronously and wait until all have returned a response.

   (b) *Sequential*: Initiate the requests in sequence and wait until the last one has returned a response.

As mentioned before, only four of these test cases were performed on GraphQL. Test case 2.B. was excluded for GraphQL. GraphQL's query structure and query resolving functionality is based on the idea that such requests can always be done in one request and thus, artificially fetching the data in multiple requests when no extra effort is needed to do it in one was deemed pointless, as that would not be done in a real world setting. A similar argument could be made for REST. REST is designed around the principle of fetching a single resource (type) at a time, so creating an ad-hoc endpoint that fetches several at once would go against that principle. However, since developers might be tempted to create such endpoints to circumvent the issue of under-fetching, the decision was made to test the performance of those as well, to see whether these ad-hoc endpoints offer a sufficient performance advantage that could convince developers to stick with HTTP endpoints rather than spend time and money on switching to another protocol (such as GraphQL). For gRPC and RSocket, no principles have been introduced that dictate which method of fetching multiple related resources should be used. Thus, to ensure a complete comparison, both test cases were applied.

7.2 Test implementation

The five test cases were implemented based on fetching the following resources:

1. A single user.

2. All trips beginning in a certain time period, for a certain distribution center, then all delivery IDs associated with each trip. Next, each delivery associated with these delivery IDs and finally the user associated with each delivery.

3. A single user, a single distribution center and a list of delivery IDs for a single trip.

For test cases 2.A. and 2.B., the process was as follows:

- 2.a: All resources fetched by supplying the initial parameters of the time period and distribution center ID.
- 2.b: Trips fetched by supplying the same initial parameters. The trip IDs were extracted from the results, and used to fetch the list of delivery IDs for each trip. Those were then used to fetch each delivery, after which the user IDs were extracted from each to fetch the respective user.

For test cases 3.A. and 3.B., the three requests were performed respectively in parallel (firing off each in sequence without waiting for the results of the previous) and in sequence (only firing off a request when all data from the previous has been received).

These five test cases (four in the case of GraphQL) were all performed 100 times each [1], in sequence with a 0.5 s delay between runs to ensure the previous run was not affecting the next one. In total, 19 tests were run and 19 sets of measurements collected, resulting in 5 comparisons that are detailed in graphs in chapter 8. The results for test case 2 for GraphQL is represented in both the graphs for test case 2a as well as for test case 2b.

RSocket and gRPC both offer the functionality of reusing the connection established for multiple requests. In gRPC, this is the default, unchangeable mechanism and performed under the hood. RSocket offers more control and allows the developer to decide exactly when and how the connection is reused. To ensure a fair and realistic comparison, both protocols were tested with the setup where each iteration used a single connection, no matter how many requests were performed in that iteration. Thus, for the 100 iterations, 100 connections were established. This was done to mimic the behavior of 100 different clients connecting to the server.

[1] The tests were also performed with 1000 iterations with similar results, which indicates that 100 was high enough to give statistically significant results.

Chapter 8

Results

In this chapter, the results of the experimentation will be presented. For each test case, a box plot is drawn showing the average response times of each of the four services, accompanied by an explanation of the results as well as a small discussion about the possible explanations for the results. The box plots are drawn without outliers [1] to increase the readability of the graphs, but all data points are included in any other calculation and statistical analysis.

8.1 Test case 1

Figure 8.1 shows the results of test case 1, fetching a single delivery. The results are very similar for all four services. GraphQL has a median response time of 121.31 *ms*, gRPC has 127.36 *ms*, REST has 122.77 *ms* and RSocket has 124.00 *ms*. All four required a single request to retrieve the data.

8.2 Test case 2.A.

Figure 8.2 shows the results of test case 2.A., fetching a collection of trips, deliveries and users in a single (ad-hoc) request. GraphQL and gRPC show a similar performance with median response times of 182.59 and 179.64 *ms*, respectively. RSocket is slightly slower with 217.69 *ms* response time, and REST comes in last with a median response time of 315.26 *ms*. All four services required a single request to retrieve all data.

[1] The boxplots were drawn using R Studio, and to exclude the outliers, the parameter `outline=FALSE` was supplied. With this option, any point that lies outside the whiskers of the plot, or further from the median than 2.5 times the distance between the median and the relevant quartile, is considered an outlier.

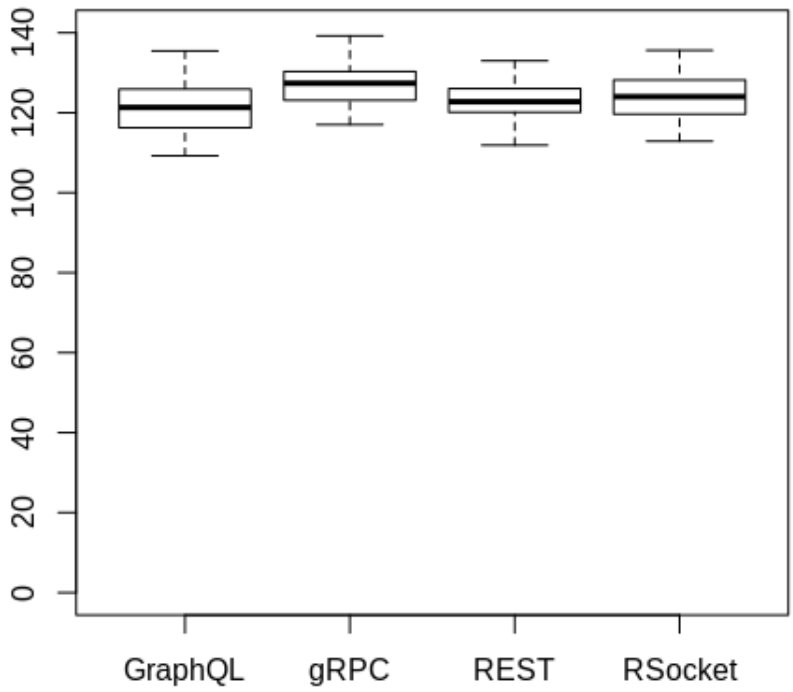

Figure 8.1: Test case 1: Response times (*ms*) of fetching single delivery.

While the performances for RSocket, gRPC and GraphQL are similar here, these results show that even if a custom ad-hoc HTTP endpoint is used in an otherwise RESTful service, the performance can still not keep up with the other protocols.

8.3 Test case 2.B.

Figure 8.3 shows the results of test case 2.B., fetching the same set of objects as 2.A. but in separate requests per resource type for REST, RSocket and gRPC. The result-set for GraphQL here is the same as in Figure 8.2. GraphQL is now fastest with the median response time of 182.59 *ms*. Next is RSocket, with a 341.59 *ms* response time followed closely by gRPC with 406.99 *ms*. Finally, REST is again the slowest with a median response time of 482.42 *ms*. GraphQL again required only one request, while the other three required in total 30 requests each.

This test case showcases especially well how the GraphQL feature of fetching multiple connected resources in one request improves performance. Furthermore, the performance of gRPC and RSocket is better than for REST, most likely due to the connection reuse feature that both offer.

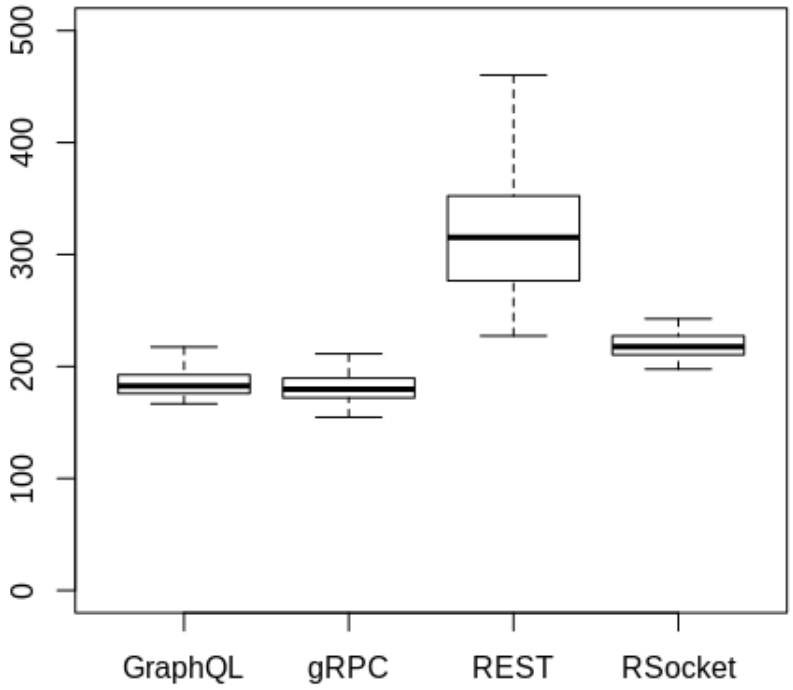

Figure 8.2: Test case 2.A.: Response times (*ms*) of fetching trips, deliveries and users in a single request.

8.4 Test case 3.A.

Figure 8.4 shows the results of fetching three different resources in one request each, in parallel. The results are again similar (although not as close as in test case 1) for all four services. GraphQL has a median response time of 130.46 *ms*, gRPC has 142.62 *ms*, REST has 128.65 *ms* and RSocket has 125.26 *ms*.

8.5 Test case 3.B.

Figure 8.5 shows the results of fetching the same set of objects as 3.A., but in sequence rather than parallel. RSocket now is the fastest with a median response time of 220.19 *ms*. Next is gRPC with a median response time of 294.07 *ms*. Finally, GraphQL and REST have similar results, with median response times of 360.70 and 363.09 *ms*, respectively.

The reason the connection reuse doesn't impact the performance as much in the parallel test case can be explained by the fact that when the three requests are performed in parallel, they are all being handled by the same connection, which could be putting an extra load on that connection. The load thus could be counteracting the benefit provided

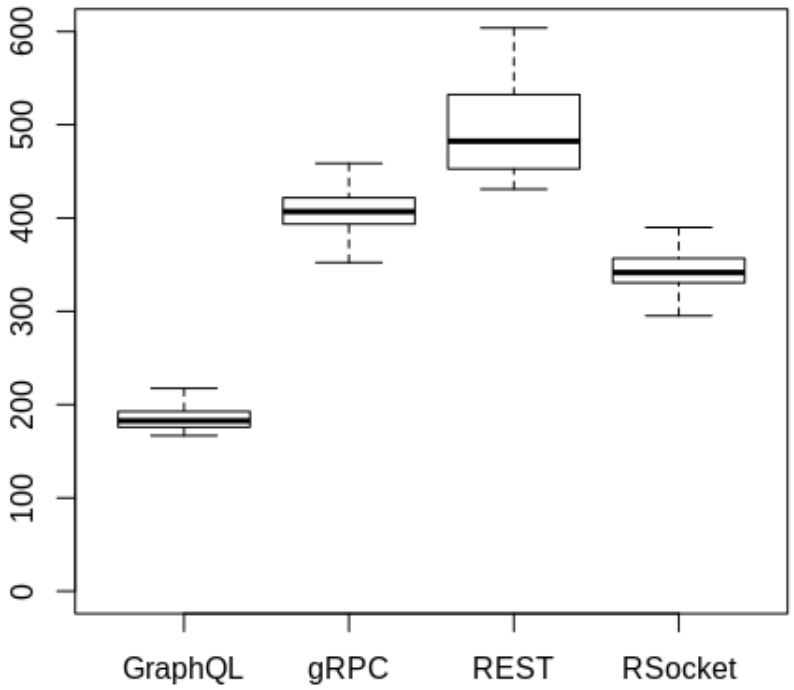

Figure 8.3: Test case 2.B.: Response times (*ms*) of fetching trips, deliveries and users in multiple requests (except GraphQL).

by not having to re-establish the connection. However, when performed in sequence, the connection only handles one request at a time, so the connection reuse benefit can be enjoyed.

The results of these five test cases can be seen in table 8.5

	GraphQL	gRPC	REST	RSocket
Test case 1	121.31	127.36	122.77	124.00
Test case 2.A.	182.59	179.64	315.26	217.69
Test case 2.B.	182.59	406.99	482.42	341.59
Test case 3.A.	130.46	142.62	128.65	125.26
Test case 3.B.	360.70	294.07	363.09	220.19

Table 8.1: Median response time in ms of the five test cases.

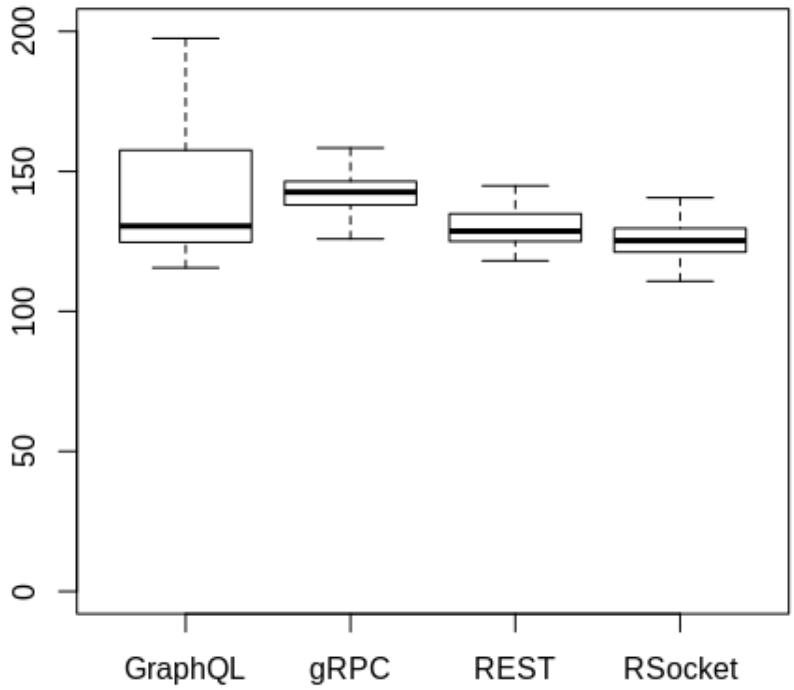

Figure 8.4: Test case 3.A.: Response times (ms) of fetching a user, a distribution center and delivery IDs in multiple, parallel requests.

8.6 Extra test case - connection reuse

To further illustrate the effect discussed in section 8.5, two extra tests were performed on RSocket and gRPC: using the same connection for all 100 iterations. Test case 1 was chosen for this. The results obtained previously were compared with a test where all 100 iterations were set off in parallel. The results for each service can be seen in Figures 8.7 and 8.6. In both plots, "stagger" refers to the sequential test run with a 0.5 s delay between iterations while "load" refers to the test wherein all iterations are run concurrently.

For both services, we can see that in the staggered mode, the performance is better when using one connection for all 100 iterations since the connection has a chance to "cool down" between requests. However, when running all 100 iterations concurrently, the performance is much better when a new connection is established for each iteration. Here, the performance degradation induced by overloading the single connection far outweighs the benefits of not having to re-establish the connection.

This indicates that for use cases where a single client has to issue a great number of requests to the same server in a short time frame, it may in fact be better to spread the

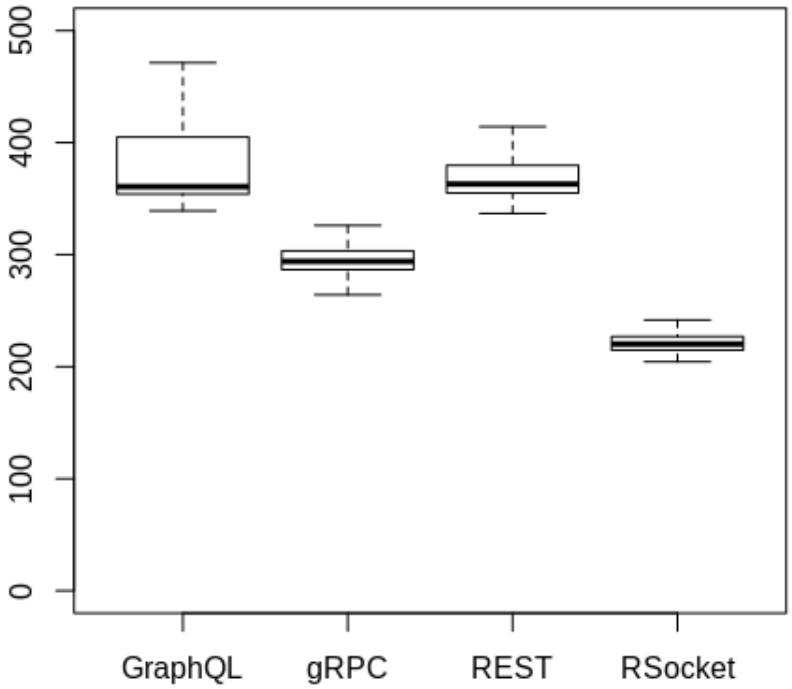

Figure 8.5: Test case 3.B.: Response times (ms) of fetching a user, a distribution center and delivery IDs in multiple, sequential requests.

load over multiple connections rather than to reuse the same one.

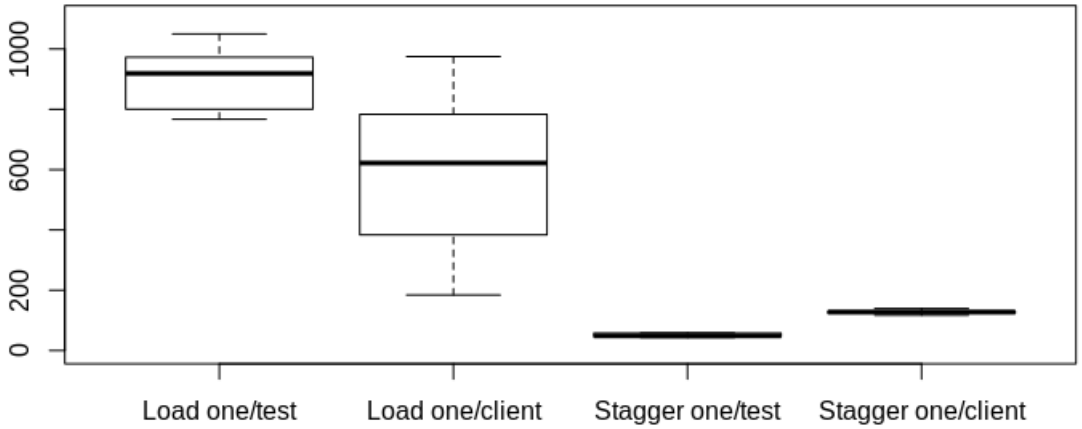

Figure 8.6: Response times (ms) of gRPC fetching a single delivery.

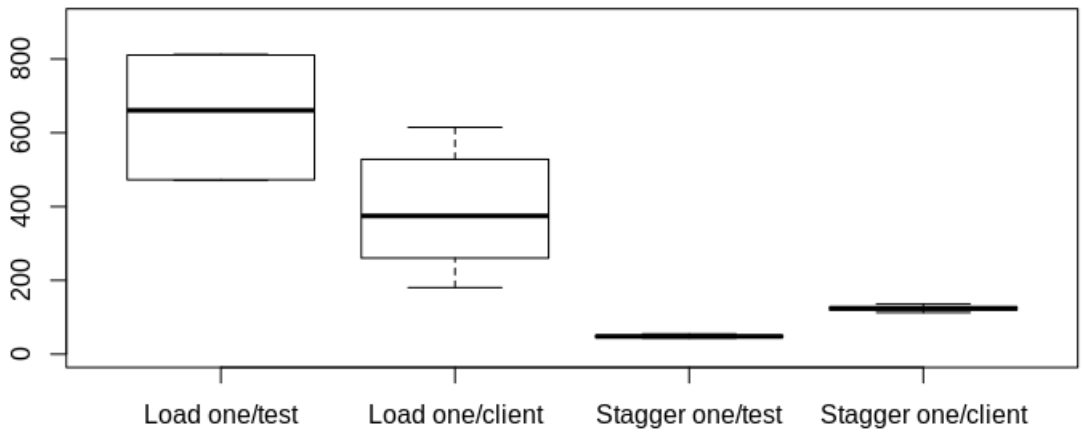

Figure 8.7: Response times (ms) of RSocket fetching a single delivery..

Chapter 9

Discussion

The results of the five test cases have been shown to support the claims of developers of the three communication methods GraphQL, RSocket and gRPC that services can benefit performance-wise from using them over RESTful HTTP APIs.

In the standard case of submitting a single request for a single resource, the four methods all perform similarly. In the case where a single request is made for multiple connected resources, REST has a significantly worse performance. GraphQL and gRPC likely perform better than REST here due to their schema/protobuf which enables the avoidance of sending fields not needed for the use case at hand. RSocket does not offer this advantage, however its binary encoding is possibly contributing to its better performance than REST. As mentioned in the results section, this test case does show that using ad-hoc endpoints that do not fit with the RESTful architectural constraints can not achieve the same performance benefits as the other technologies, even though that is a common workaround for the problem of under-fetching.

However, in cases where other methods require multiple requests to fetch data where GraphQL can do it in one, GraphQL outperforms the others. In contrast, where all four technologies require the same number of requests, GraphQL does not outperform the others. This indicates that GraphQL's main advantage is in minimizing the amount of requests needed overall. For systems with complex underlying data models (especially ones based on graphs), this indicates that GraphQL is a very good choice. In a system with a simpler data model and simpler use cases (e.g. where a majority of use cases only depend on a single data resource), using GraphQL would not offer a major performance advantage.

In the cases where all four methods require multiple requests performed in sequence, RSocket and gRPC both benefit greatly from multiplexing by sending all requests over the

same connection. Avoiding the overhead incurred by constantly needing to re-establish the connection thus offers a great advantage. For parallel requests however, this benefit could be counteracted by the overloading incurred by sending too many requests over the same connection. Further tests might need to be done to investigate at which point this trade-off switches, to provide guidelines to developers for when to spread the requests over multiple connections, and when using the same connection is sufficient. However, since gRPC does not allow for this in most implementations, in this aspect RSocket has the upper hand.

There are a few limitations with these experiments. Firstly, this thesis did not put any explicit focus on the performance of the data serialization (JSON vs Protocol Buffers). Some studies have been done on this topic, with Popić et al. showing that the size of messages can be minimized by 83% by using protocol buffers rather than JSON (55). In addition, Wibowo showed that the time taken to serialize messages went from around 10.5 μs with JSON to around 6.5 μs (56). Given that gRPC was the only one out of the four that did not use Jackson for JSON serialization, it is possible that this gave gRPC an unfair advantage over the others. However, since most documentation and tutorials assume the usage of protocol buffers, the choice was made to use it as:

1. There are few resources available that explain how to use other serialization methods.
2. In practice, protocol buffers would most likely be used due to their status as the default serialization method.

Thus, to emulate a real-world scenario, it was considered more prudent to use gRPC with protocol buffers.

Next, the selection of the 0.5 second interval between requests in the experiments was not properly supported with arguments. Other theses with similar focus opted for either a 1 second interval (49) or a 0.5 second interval (38). However, it would have been better to perform additional tests to see whether the 0.5 second interval was long enough to allow the services to cool down between requests. If any of the services require a longer time to cool down than the others, their performance would have been affected. This was not taken into consideration and could have changed the final conclusion.

Finally, the author originally intended to also do experiments to see whether the four services behaved differently under load, which could provide insight into which of the four is best suited for larger systems with greater performance in high-traffic periods. However, due to time constraints this was not possible.

Chapter 10

Conclusions

Among the four communication technologies discussed in this thesis, GraphQL, gRPC and RSocket all seem to be a better fit for microservice architecture than REST. For inter-service communication, REST lacks many of the features that give the other three increased performance, such as multiplexing, header compression, more efficient encoding or flexible client-defined queries. The experimental results showcase this, attested by the fact that REST never had the best performance out of the four in any test case. REST suffers from the fact that it is a style developed long before microservice architecture was popularized. It is designed for HTTP communications focusing on hypermedia intended for display in browsers. Within the microservices world, REST is most appropriate for client-facing APIs but even then, the issues of under- and over-fetching can cause worse performance as well as lead developers to rely on ill-designed workarounds that break the constraints of REST. However, since REST has been so popular since its inception, most web and back-end developers are very familiar with it. For simpler systems with simpler use cases, REST still serves its purpose.

For similar reasons, GraphQL is also better suited for client-facing APIs than for inter-service communications. It has several advantages over REST, mainly due to its strongly typed schema and client-defined queries. A single GraphQL request can have the same performance as a single REST request but the ability to easily query for multiple resources with a single request without having to define a specific method/endpoint for it gives GraphQL a considerable edge not just on REST, but also on gRPC and RSocket as was shown in figure 8.3. For microservices systems with complex underlying data models (especially ones structured as graphs) and/or serving multiple clients with different needs, GraphQL is a good choice.

In contrast, gRPC and RSocket are better suited for inter-service communications than both REST and GraphQL, each being designed explicitly for microservices communications. The two show very similar performance results, with the same result in the first test case, gRPC being slightly faster in the second and RSocket slightly faster in the other three. For both, the impact of a high number of simultaneous requests over a single connection decreased the performance, indicating that spreading the requests over multiple connections could improve the overall performance. However, RSocket allows much more control over this than gRPC, with implementations of gRPC implicitly reusing the connection within the same process even when the developer explicitly attempts to open a separate one. Both technologies offer multiple interaction models, multiplexing and reduced bandwidth either through header compression or binary encoding. RSocket however has an advantage over gRPC in not adhering to the client-server model, allowing communicating services to act more like peers. Finally, the default usage of protocol buffers in gRPC can put it either at an advantage or disadvantage, depending on the requirements of the system at hand. The strong typing offered can be safer and provides better understanding between different teams of the interface behavior and the code generation can save time in development, however it limits flexibility and freedom over what can be sent over the network. gRPC is also tied to HTTP/2, while RSocket can use any byte stream protocol, even different ones in the same service.

Ultimately, the choice of a technology not only depends on the architecture chosen for a system, but also the experience and preference of the developers, the specifics of the use case, complexity of the system and data model, and the performance requirements. It would be near impossible to design one protocol or framework which is perfect for microservices in every way. However, it should always be possible, through careful analysis, to find the approach that works the best for the case at hand. For Picnic's use case, the author's recommendation would be GraphQL for client-facing APIs due to its better performance and flexible querying, and RSocket for inter-service communications due to the freedom and control it provides.

References

[1] Daniel Spoonhower. **Lessons From the Birth of Microservices at Google**. https://dzone.com/articles/lessons-from-the-birth-of-microservices-at-google. Accessed: 2020-07-09. 1

[2] Tony Mauro. **Adopting Microservices at Netflix: Lessons for Architectural Design**. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. Accessed: 2020-07-09. 1

[3] Javad Ghofrani and Daniel Lübke. **Challenges of Microservices Architecture: A Survey on the State of the Practice.** In *ZEUS*, pages 1–8, 2018. 1, 13

[4] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. **Microservices in Practice: A Survey Study**. *arXiv preprint arXiv:1808.04836*, 2018. 1, 13

[5] O Zimmermann. **Microservices tenets: Agile approach to service development and deployment**. In *Proceedings of the Symposium/Summer School on Service-Oriented Computing*, 2016. 1, 5, 6, 13, 17

[6] **RapidAPI Developer Survey Insights 2019 - 2020**. https://rapidapi.com/blog/wp-content/uploads/2020/03/Dev-Survey-Updated.pdf. Accessed: 2020-07-06. 2, 3

[7] Marek Gajda, Peter Cooper, Luca Mezzalira, Richard Rodger, and Sarup Banskota. **State of Microservices 2020**. https://tsh.io/state-of-microservices. Accessed: 2020-07-09. 2

[8] Nick Schrock. **GraphQL Introduction**. https://reactjs.org/blog/2015/05/01/graphql-introduction.html, 2015. 2, 8, 17, 24

[9] Raphaël Benitte, Sacha Greif, and Michael Rambeau. **State of JS 2019: GraphQL**. https://2019.stateofjs.com/data-layer/graphql/. Accessed: 2020-04-09. 3

[10] **About gRPC**. https://grpc.io/about/. Accessed: 2020-07-06. 3, 11, 15

[11] **RSocket**. https://www.netifi.com/rsocket. Accessed: 2020-07-06. 3, 10

[12] Martin Fowler. **martinfowler.com**. *martinfowler.com*, Mar 2014. 5, 6

[13] Subramaniyam R Viswanathan. *Publishing in wireless and wireline environments*. PhD thesis, Rutgers, The State University of New Jersey, 1994. 5

[14] Françoise Andre and Maria-Teresa Segarra. **A generic approach to satisfy adaptability needs in mobile environments**. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 10–pp. IEEE, 2000. 6

[15] Sebastian Hockmann. *Vergleich Microsoft .NET mit Sun ONE*. Master's thesis, der Technischen Universität Ilmenau, 2002. 6

[16] Sung Joong Kim. *Foundation for composable microservices for rapid synthesis of highly reliable software systems*. PhD thesis, The University of Texas at Dallas, 2004. 6

[17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. **Microservices: yesterday, today, and tomorrow**. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017. 6

[18] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, **7**. University of California, Irvine Irvine, 2000. 7, 8

[19] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. **Reflections on the REST architectural style and "principled design of the modern web architecture"(impact paper award)**. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 4–14, 2017. 7, 8, 23

[20] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011. 8, 17

[21] **GraphQL Specification**. https://spec.graphql.org/June2018/, 6 2018. Accessed: 2020-04-17. 8

[22] **Who's using GraphQL?** https://graphql.org/users/. Accessed: 2020-04-09. 9

[23] Mike Melanson. **HTTP Alternative RSocket Gets a Home at The Linux Foundation**. https://thenewstack.io/http-alternative-rsocket-gets-a-home-at-the-linux-foundation/, 2019. Accessed: 2020-07-06. 10

[24] **Reactive foundation members**. https://reactive.foundation/members/. Accessed: 2020-07-06. 10

[25] **Motivations**. https://rsocket.io/docs/Motivations. Accessed: 2020-07-06. 10, 18

[26] Mugur Marculescu. **Introducing gRPC, a new open source HTTP/2 RPC Framework**. https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html?m=1, 2015. Accessed: 2020-07-06. 11

[27] Louis Ryan. **gRPC Motivation and Design Principles**. https://grpc.io/blog/principles/, 9 2015. Accessed: 2020-07-13. 12, 24

[28] **Core concepts, architecture and lifecycle**. https://grpc.io/docs/what-is-grpc/core-concepts/. Accessed: 2020-08-08. 12

[29] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. **The pains and gains of microservices: A systematic grey literature review**. *Journal of Systems and Software*, **146**:215–232, 2018. 13

[30] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. **Architectural patterns for microservices: a systematic mapping study**. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS, 2018. 13, 24

[31] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. **Workload characterization for microservices**. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016. 14

[32] Yu Gan and Christina Delimitrou. **The architectural implications of cloud microservices**. *IEEE Computer Architecture Letters*, **17**(2):155–158, 2018. 14

[33] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. **Performance evaluation of microservices architectures using containers**. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015. 14

[34] Nane Kratzke. **About microservices, containers and their underestimated impact on network performance**. *arXiv preprint arXiv:1710.04049*, 2017. 14

[35] Thomas Eizinger. *API Design in Distributed Systems: A Comparison between GraphQL and REST*. Master's thesis, University of Applied Sciences Technikum Wien, 2017. 14

[36] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. **Experiences on migrating RESTful web services to GraphQL**. In *International Conference on Service-Oriented Computing*, pages 283–295. Springer, 2017. 14

[37] Andrea Vázquez-Ingelmo, Juan Cruz-Benito, and Francisco J García-Peñalvo. **Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL**. In *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality*, pages 1–8, 2017. 14, 18

[38] Mattias Cederlund. *Performance of frameworks for declarative data fetching*. Master's thesis, KTH School of Information and Communication Technology, 2016. 14, 44

[39] C. L. Chamas, D. Cordeiro, and M. M. Eler. **Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis**. In *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2017. 15

[40] Pîrnau Mironela. **The Importance of Web Services Using the RPC and REST Architecture**. In *2009 International Conference on Computer Technology and Development*, **1**, pages 377–379. IEEE, 2009. 15

[41] Xinyang Feng, Jianjing Shen, and Ying Fan. **REST: An alternative to RPC for Web services architecture**. In *2009 First International Conference on Future Information Networks*, pages 7–10. IEEE, 2009. 15

[42] Steve Vinoski. **Rpc and rest: Dilemma, disruption, and displacement**. *IEEE Internet Computing*, **12**(5):92–95, 2008. 15

[43] Sean Kennedy and Owen Molloy. **A framework for transitioning Enterprise Web Services from XML-RPC to REST**. In *proceedings of 3rd International Conference on Information Resources Management, UAE*, 2009. 15

[44] Min Choi and Jang-Eui Hong. **Performance Analysis of Socket and REST Web Service OpenAPI for Mobile-Cloud Applications**. In *Proceedings of the Korean Information Science Society Conference*, pages 97–99. Korean Institute of Information Scientists and Engineers, 2012. 15

[45] Hae-Ryong Cho and Min Choi. **Replacing Socket Communication by REST Open API for Acquisition Tax Analyzer Development**. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 462–468. IEEE, 2014. 15

[46] Maciej Rostanski, Krzysztof Grochla, and Aleksander Seman. **Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ**. In *2014 federated conference on computer science and information systems*, pages 879–884. IEEE, 2014. 15

[47] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. **Performance analysis of RESTful API and RabbitMQ for microservice web application**. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 257–259. IEEE, 2018. 15

[48] Thuy Nguyen. *Benchmarking Performance of Data Serialization and RPC Frameworks in Microservices Architecture: gRPC vs. Apache Thrift vs. Apache Avro.* Master's thesis, Aalto University, 2016. 15

[49] Petter Johansson. *Efficient communication with microservices.* Master's thesis, Umea University, 2017. 15, 44

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T Berners-Lee. **Hypertext Transfer Protocol – HTTP/1.1**. RFC 2616, Network Working Group , June 1999. 17

[51] Simon E. Spero. **Analysis of HTTP Performance Problems**. https://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html, July 1994. 17

[52] M. Belshe, R. Peon, and M. Thomson. **Hypertext Transfer Protocol Version 2 (HTTP/2)**. RFC 7540, Internet Engineering Task Force (IETF), May 2015. 18

[53] Alex Ng, Paul Greenfield, and Shiping Chen. **A study of the impact of compression and binary encoding on SOAP performance**. In *Proceedings of the Sixth Australasian Workshop on Software and System Architectures (AWSA2005)*, pages 46–56, 2005. 18

[54] **Introducing JSON**. https://www.json.org/json-en.html. Accessed: 2020-08-16. 19

[55] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. **Performance evaluation of using Protocol Buffers in the Internet of Things communication**. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 261–265. IEEE, 2016. 44

[56] Canggih Puspo Wibowo. **Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication**. In *International Conference on Informatics for Development, At Yogyakarta, Indonesia*, pages 40–43, 2011. 44