Efficient Communication With Microservices

Petter Johansson

June 14, 2017
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Mikael Rännar
Examiner: Henrik Björklund

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

In this report we present a study of communication between a client application and microservices and also a review of how the performance and scalability of microservices are affected by different hosting choices. The review showed that containers can perform better than virtual machines and that the microservice platforms `AWS Lambda` and `Azure Service Fabric` can be good hosting alternative in some situational cases. `AWS Lambda` performs well with small service backends that does not do any larger process of data while `Service Fabric` provides developers with all the tools necessary to build a microservice system in C# and Java. The focus of the communication study was to find out if the new query language GraphQL could perform as well as a REST architecture with JSON as data serialisation. GraphQL performed considerably worse with a higher response time in all tests.

Contents

Chapter 1

Introduction

With the increasing popularity of software development for the cloud the design pattern microservices has emerged. The concept of microservices is an architectural system design where the systems business capabilities or sub domains are divided into separate services contained in their own process. Microservices should not have any dependencies to platforms, tools or languages and should be developed with the tools that are best suited for the specific cause. Microservices will most likely be accessed through several different clients running on different platforms. It is therefore crucial to use communication protocols or architectures that both large scale desktop applications as well as mobile applications can use. In this report we will investigate which communication methods are suitable to use based on performance and their cross platform functionality.

1.1 Dohi

This project was planned and carried out in cooperation with Dohi Agency which is a software development company and a part of the Dohi corporate group. Dohi Agency develops customised software systems for a large number of customers around Sweden. They are currently working on a project where a monolithic application is broken down into microservices. It is in their interest to investigate if there is a more suitable alternative for communication between their client and server architecture instead of REST/JSON which is currently used.

1.2 Outline

1. **Introduction**
   An introduction to the subject of this thesis and a presentation of Dohi, the company that this project was done in collaboration with.

2. **Background**
   Introduces subjects that are crucial to understand before further reading.

3. **Problem description**
   Describes the problems that this thesis aimed to solve, and the methods that were used to do so.

4. **Hosting microservices**
   A review of the hosting alternatives for a microservice system.

5. **External communication**
   A review of common communication alternatives between web services.

6. **Implementation**
   Presents an implemented test environment, why it was built and how it can help answering the scientific questions.

7. **Results**
   The results from test runs in the implemented microservice system are presented here.

8. **Conclusions**
   Contains thoughts about the results and project in general and the future work that can be done on this subject.

9. **Acknowledgements**
   A chapter dedicated to thank the parties that were involved in this project.

Chapter 2

Background

This chapter will give a further introduction to microservices, how they can be designed and why the concept of microservices should be considered when designing software. It will also include a brief introduction to cloud providers and the services that they offer. Finally it will give a brief introduction to the most common formats when it comes to data serialisation.

2.1 Microservices

Microservices is an architectural style for software systems that focuses on creating highly scalable software for the cloud. Microservices are small independent parts that together fills the purpose of an application. Each service is designed to handle one sub domain or business capability and can be used as a part of any application since they are contained in their own process [28]. Microservices were derived from the Service oriented architecture (**SoA**) with the only real difference being that a microservice should be small and only handle one business capability.

The Figures 2.1 and 2.1 shows how an application, in this case a web shop, can be divided into microservices.

Microservices are not tied to any specific framework, language or software and most importantly does not need to be built with the same technology. To achieve a productive parallel implementation of services using different technologies there needs to be predefined interfaces that define the communication between the services. The internal communication can either be synchronous or asynchronous, but must always be done with network calls to ensure that the services are completely separated.

In a microservice architecture there needs to be predefined interfaces that specify how the services can communicate with each other. There are two ways of handling the internal communication between services, synchronous and asynchronous message passing.

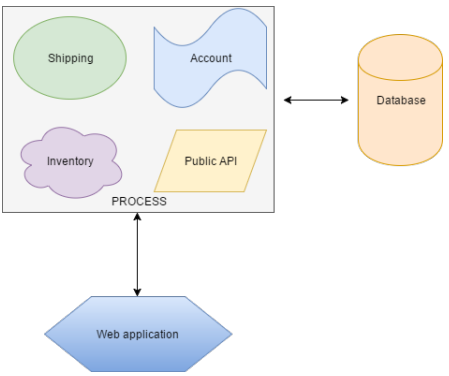

Figure 2.1: A web shop designed as a monolithic application

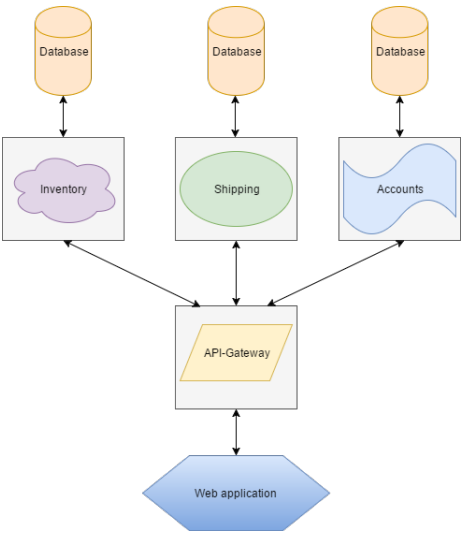

Figure 2.2: A web shop built as microservices

In a monolithic application a small change in the code results in the need for the entire application to be redeployed. With microservices you only need to deploy the service that has been changed. The same principle applies when the system is under high load and needs to be scaled up onto more machines. With several small services you only need to scale up the service that is under high load and not the entire application. It makes it easier to test and faster to deploy but at the cost of an increased latency and a higher complexity.

When a client is to communicate with the services it is often done through an API-Gateway. The API-Gateway works as a single entry point to all the data available across the services. Implementing several gateways where each acts as a specific backend for a frontend application is also a commonly used alternative.[32]

2.1.1 The building blocks in a microservice architecture

Hosts

Every application needs to be deployed onto servers and microservices are no exception. The hosts in a cloud environment is usually in the form of virtual machines instead of physical servers. One virtual machine can host one or several services, depending on the goals and restrictions for the system [32].

Service registry

Since microservices are designed to be deployed in a cloud environment where server addresses are dynamic and can change at any point in time every running service needs to be registered at a storage with a known address. The clients which are to communicate with the services fetches their addresses from the service registry with a known static address.[27]

Load balancer

Microservices are scaled up by increasing the number of instances of a certain service. The load balancer distributes the incoming traffic to services which are under low load to avoid a bottleneck in the system which results in a higher response time. It is assigned the responsibility to constantly do health checks on the services so that it is possible to scale the system after the currently incoming requests. [32]

Internal communication

In the microservice architecture services communicate with each other through message passing over a network. Since the services are designed to be built parallel in time with each other predefined communication methods are required. The chosen communication methods must therefore be supported for all platforms and languages that are used in the system or that might be used in the future. Common alternatives as asynchronous message passing with REST or publish/subscribe patterns where services can subscibe to channels and get a notice when a new message have been added to that channel.[33]

External communication

The external communication must use a protocol or architecture that can support clients built for desktop computers, mobile phones or web applications. The REST architecture have become one of the most common alternative since it is completely platform independent when used with JSON as message serialisation.[32] JSON will be presented in more detail later in to this chapter. REST will be presented along with other communication alternatives in chapter 5.

2.2 Cloud providers

Cloud providers are companies that let users rent servers from their own server halls. The users have remote access to virtual machines hosted on the providers servers. Cloud providers often offers services which helps the development of application on their servers. Features such as load balancers that routes the traffic and scales the applications and interfaces for managing applications and virtual machines can be found in all cloud providers. Two of the major cloud providers are Microsoft Azure [14] and Amazon Web Services [1] (**AWS**).

Infrastructure as a service (IaaS)

Infrastructure as a service is referring to services offered from cloud providers in forms of raw infrastructure. This includes virtual machines, networking, IP-addresses and load balancing services. IaaS services lets the developers configure every service that they intend to use, such as; operating systems frameworks and tools to name a few.[8]

Platform as a Service (PaaS)

Platform as a Service is referring to services in form of computing platforms that are offered to developers by cloud providers. These types of platforms are designed to automatically handle some aspect of the development and deployment processes to make it easier to develop applications for the cloud. An example of a platform that most providers offer are hosting services which automatically scales the application depending on the current load. When using a platform the developer gets to focus on developing their product while the cloud providers offer hosting solutions for it with installed software and networking.[8]

2.3 Data Serialisation

Serialisation are methods which converts data into a predefined format that can be deconstructed and reconstructed on another machine.

JSON

JavaScript Object Notation (**JSON**) is a text format used to serialise structured data. JSON represents data with structured and and primitive data types. The supported primitive types are string, number, boolean and null. The structured types can either be an object or an array. An object in JSON format contains pairs of names and values, where the name must be a string and the value can be of any supported primitive type or an array. Objects are represented as curly brackets which enclose an arbitrary number of name and value pairs separated by commas. The name that corresponds to a value should be unique to make the parsing of the JSON object more manageable, but it is not a must. Arrays are represented with square brackets, just like in most programming languages. An array can contain zero or more values of any type, the values in an array must not be of the same type.[13] An example of data serialised to a JSON object is shown below in listing 2.1. In the example `Person` is an object which contains the names `ID` with value `007` and `Pets` with the values `Dog` and `Cat`.

```
1  {
2      "Person":{
3          "ID": "007",
4          "Pets":["Cat", "Dog"]
5      }
6  }
```

Listing 2.1: Example data serialised to a JSON-format

XML

Extensible Markup Language (**XML**) is a markup language which can be used to represent structured data in web services. XML uses a tree structure to describe data dependencies and identifies resources with an element name wrapped around the resource value [9]. An example of data structured in XML format is showed below in listing 2.2. Person is the root element with the child elements ID and Pets. Pets has its own child element pet which holds the data resources Cat and Dog.

```
<Person>
      <ID>007</ID>
      <Pets>
           <Pet>Cat</Pet>
           <Pet>Dog</Pet>
      </Pets>
</Person>
```

Listing 2.2: Example data serialised to a XML-format

Binary

Data serialised into a binary format results in smaller objects compared to `XML` or `JSON`. The binary objects are however not in a readable text format and therefore not as easy to present in a meaningful way. Binary serialisation requires special frameworks or tools to serialise and deserialise objects. Two examples of binary serialisation frameworks are `Apache Thrift` [2] and `Google Protocol buffers` [15].

Chapter 3

Problem Description

Microservices makes an application more scalable and easier to deploy compared to a monolithic approach, but having several services instead of a monolithic software leads to an increased complexity of the system. It is therefore crucial that you design the services appropriately for the purpose of the system. Microservices are designed to be reusable in different systems and are supposed to be used by clients built with different languages and tools. It is therefore important to use communication alternatives that can be implemented and consumed in all major languages. We will investigate the commonly used solutions to external communication with clients and review how suitable they are for a microservice architecture. The new query language GraphQL will be implemented and the aspects of efficiency and scalability will be reviewed to see if it is a viable option for external communication between a client and microservices. We have planned to answer the following specific questions:

- Can the communication between a web application and a system of microservices become more efficient with the query language GraphQL, compared to HTTP requests and a REST API?
- How should the public API:s of a microservice system be designed and which communication alternatives are both efficient and suitable for microservices ?
- How does the underlying infrastructure hosting the microservices affect the performance and scalability of the system?

3.1 Methods

The specified questions for this project will be answered using two different methods;

- A prototype implementation of a microservice system where the performance of GraphQL can be measured and compered to a REST.

– A literature study which focuses on communication between web services and the design trade offs that has to be made when designing microservices

3.1.1 Literature study

The literature study had two main focues. The first part of the literature study included a review of communication alternatives between web services. The goal was to find out which methods were suitable to use as a communication interface between microservices and client applications.

The second part of the literature study was a review on different ways to host cloud services. The important aspects of a host was the performance in terms of how well it could utilise the hardware and also how well it can scale.

3.1.2 Implementation

The implementation of microservices and GraphQL were done so that the performance of GraphQL could be reviewed as well as to show Dohi how the technology could work in a microservice system. GraphQL is a new technology that were open sourced by facebook in 2015 [19], and to the best of my knowledge no scientific performance review were available before this project.

Since the only requirement for the test environment were that GraphQL and REST had equal terms for data fetching and serialisation the implementation language or cloud environment used for the test did not matter. Therefore the test system were built using the tools and techniques that are relevant to Dohi and their customer. Therefore the microservices were implemented in .NET and hosted in Microsoft cloud environment Azure.

The REST communication architecture wes implemented with JSON as serialisation method and HTTP as transport protocol since that is the most commonly used alternatives. These choices were also made to make the resulting data easily comparable to results from other papers.

3.2 Related Work

Communication between web services has been researched several times. The performance of messaging in a distributed system is an important topic covered in many papers. Tihomirovs and Grabis presents a performance review of REST and SOAP based web services where REST is shown to have better performance [29].

Maeda resaerches the performance of serialisation and comes to the conclusion that binary serialisation can be much more effective than text serilisation formats such as XML or JSON [31]. Sumaray and Kami Makki presents results of binary and text base serialisation on mobile platforms and confirms that a binary serialisation is faster on mobile platforms as well [18].

The microservice architecture has recently become a popular evolution of the SoA design pattern. The microservice architecture has been studied and reviewed in papers from different angles. Villamizar et al. presents a case study where the performance of different microservice architectures compared to monolithic applications with the same functionality [23].

Messina et al. presents a review of design pattern closely related to the microservice architecture and gives a good overview of the design problems one must face when opting for a microservice architecture [20].

Chapter 4

Hosting microservices

In this chapter the fundamentals of virtual machines, containers and the cloud platforms `Lambda` and `Service Fabric` will be presented. The goal with this chapter is to find out strength and weaknesses from each hosting alternative and ultimately answer the question:

– How does the underlying infrastructure hosting the microservices effect the performance and scalability of the system?

4.1 Choosing the right type of host

Choosing the right infrastructure to build a microservice system on is essential for the overall performance. Since microservices are built to be scaled by increasing the number of service instances, the underlying infrastructure must have tools that supports the ability to host virtual machines, deploying applications and monitoring them. Services can either be deployed directly onto virtual machines, be encapsulated in containers for minimal package size or hosted on platforms supplied by cloud providers. Two platforms built for microservices are Service Fabric provided by Azure and Lambda provided by Amazon Web Services. If microservices are hosted on virtual machines it us up to the developers to set up service registries and load balancers.

4.1.1 Virtual machines

A virtual machine (**VM**) is an emulation of a server that utilises only a part of the real physicals servers computing powers. Virtual machines can use any operating system and tools on any server. This is made possible by a hypervisor, which acts as a translator between the physical servers operating system and the virtual machine.[26] The complete infrastructure for hosting an application on a virtual machine is showed in Figure 4.1.

When a microservice system is hosted on virtual machines a decision has to be made on how many services that should be deployed to each VM. The approaches to deploying microservices on virtual machines are: one or several microservices per virtual machine.

Hosting several microservices on the same VM of course lowers the total cost of the system since fewer machines have to be used. It can also lower the deploy time since new machines don't have to be created for every new service instance. The downside of several service instances per host is that it becomes harder to manage the health state of the system. Since several services are using the same resources, automatic scaling can be troublesome since figuring out how much of the virtual machines resources each service is using. Another problem is that several services on one host cause higher damages if that host crashes or doesn't work properly.

4.1.2 Containers

The main difference between a container and a virtual machine is that the container is designed to share the operating system of the host instead of having its own. The containers are performing operations and accessing the hardware through a container engine installed on the host. The container itself holds all libraries and binaries that it needs to run the application. Because of this a container is a more lightweight alternative to a virtual machine that needs less memory space.[26] The drawbacks with containers are that the lack of operating system only allows for one process to be run in each container. Sharing an operating system also means that if the host becomes compromised through the fault of one container based application it effects every container on that host. An overview of the container infrastructure is shown in Figure 4.2.

Kominos et al. shows that containers have more efficient use of the CPU than virtual machines. The VMs suffers from the hypervisor requesting computation time from the CPU. Furthermore the containers can also handle more requests per seconds from both UDP and TCP connections compared to a virtual machine [21].

Containers can, because of their compact size and low memory, usage scale faster than virtual machines [30].

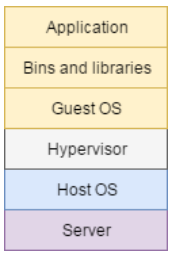

Figure 4.1: Infrastructure stack for an application hosted on a virtual machine

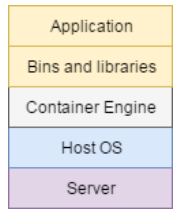

Figure 4.2: Infrastructure stack for an application hosted in a conatiner

4.1.3 Cloud platforms designed for microservices

AWS Lambda

Lambda is a microservice platform that Amazon provides. The Lambda platform hides the underlying infrastructure of servers and calls it a serverless platform. Developers only need to upload the code which they want to run when the service receives a request. Amazon only charges the users of the Lambda platforms in terms of computation hours instead of the server up time which is the standard [6]. The drawbacks of Lambda is a small pool of supported languages(JavaScript, Java, Python and C#) [5], and also the hard task of monitoring an application since it is not continuously running on a server, only invoked when an event happens.

Villamizar et al. shows that microservices hosted on the Lambda platform can be cheaper than both a monolithic application and a microservices application running on regular servers. They also show that the response time of AWS Lambda performed equally or better in terms of response time when compared to other monolithic and microservice systems hosted on virtual machines.[24]

Azure Service Fabric

Service Fabric is a microservice platform that Azure provides.[14] The Service Fabric offers features such as automatic scaling, automatic service registry, an abstract communication platform which automatically distributes internal service messages to all instances of the same service and a redundancy in virtual machines that is always running to ensure minimum downtime. The Service Fabric has full support to host applications on both virtual machines and containers. A huge downside to the Service Fabric is that to access the full features developers are bound to use C# or Java. The platform accepts services coded in any language, but to services which are not programmed in C# or Java the Service Fabric will only work as a platform that automatically scales the application, which could easily be done with any load balancer.

Chapter 5

External communication

In this chapter we will in a more detailed approach explain how the communication between microservices and client application should be done. A review of communication protocols and serialisation formats based on their performance will also be presented. The goal with this chapter is to answer the following questions that were defined in the problem description:

- How should the public API:s of a microservice system be designed and which communication alternatives are both efficient and suitable for microservices ?

5.1 Hosting the public communication interfaces

Applications designed to communicate with the microservices needs one or several APIs that clearly defines the available data and how to access it. You can either have each service expose a public interface or let the traffic go through gateways with public interfaces that internally communicates with the microservices.

Direct communication with services

Maybe the most straight forward approach to designing publicly accessible interfaces to microservices is to implement a public API in every service which can be directly consumed by clients [32]. Having separate APIs in every client guarantees that the system stays decentralised and robust in the sense that the communication with clients is not relying on a single communication channel [23]. The direct communication with does however have its drawbacks. If a clients wants to fetch data from the microservices that are overlapping and is relying on data from another service it will result in several round trips between the client and the services. This would add a lot of latency that results in a poorer user experience.

API Gateway

An API-Gateway is a separate service which aggregates the data resources that the microservice has to offer. The gateway exposes a public communication interface which a client can consume and gathers the requested resources from the services. Adding API-gateways to a microservice architecture solves the problem of multiple round trip times that the direct communication alternative exposes. Adding a gateway to the microservice systems means that the communication suffers with higher response times since the messages need to be sent between two web services instead of one. [20]

A common use of API-gateways is to develop one gateway for every front end application developed to communicate with the microservices. This style is called backend for frontend and it is a good way of dividing the responsibility of external communications in several services and to eliminate the single point of failure.[32] The drawbacks of the backend for frontend pattern is that a lot of code have to be duplicated as well as it creates more services which has to be hosted and monitored.

5.2 Efficient serialisation

In order for the communication between web services to be effective the messages sent over a network must be small and have a low serialisation and deserialisation time. The size of a serialised object vary greatly and depends on the chosen format. The performance of the serialisation formats described in the background have been reviewed and compared by Maeda [31], Popíc [25] and Suamaray and Kami Makki [18]. The results regarding the serialised object size can be seen in Table 5.1. XML produced the largest objects while Protocol Buffers produced the smallest objects. Objects serialised to binary Thrift were slightly larger than the Protocol Buffers objects, but still considerably smaller than JSON or XML objects.

Table 5.2 shows a comparison of the serialisation time for the chosen formats. Both Madea and Sumaray and Kami Makki comes to the conclusion that XML takes the longest time to serialise and places JSON in a clear second position. They do however disagree on the serialisation time for Thrift and Protocol Buffers. What is made clear in both papers is that both binary formats have a lower serialisation time compared to JSON and XML.

The deserialisation time for each format is presented with varying results. There is a consensus on XML having the longest deserialisation time, but the results for Protocol buffers, Thrift and JSON varies. Maeda shows that using the Thrift framework to deserialise JSON performs best. It is however shown that JSON deserialisation is generally slower than deserialising the Thrift binary format or Protocol Buffers.

Size	**Largest**	-	-	**Smallest**
Format	XML	JSON	Thrift Binary	Protocol Buffers

Table 5.1: Object size after serialisation

Time	**Slowest**	-	-	**Fastest**
Format	XML	JSON	Protocol Buffers Thrift Binary	Protocol Buffers Thrift Binary

Table 5.2: Time to serialise object

5.3 Evaluation of communication protocols and REST

The communication technologies presented in this section are chosen because they are widely used in the industry and they all have some degree of cross platform support.

SOAP

Simple Object Access Protocol (**SOAP**) is a cross platform protocol for communication over Internet. SOAP uses XML to encapsulate messages and to define how to transmit and receive them. SOAP is not tied to any application layer for data transmission but HTTP has become the most commonly used [17]. The SOAP architecture can be divided into three parts: Service providers, service registry and service consumer. The service provider creates the SOAP service which maps data from storage into SOAP messages and provides a description of the available service to the service registry. The service registry exposes descriptions of services and how to communicate with them. The service consumer can access the registry to find out which services are available and then communicate with that service using SOAP messages.[29][22]

Remote Procedure Call

A remote procedure call (**RPC**) is a procedure call from one process to another over a network. There are several ways to do a RPC, most modern programming languages has the support for RPC built in, but there are also frameworks built specifically to allow for cross platform RPCs.

These frameworks make cross platform calls possible by describing data resources, procedures, their input parameters and expected outputs in a language independent Interface Definition Language (**IDL**). The procedures in the IDL can then be used to generate server and client code which implements the procedures in different programming languages, on different machines.

Remote calls are made by a client who calls the locally implemented procedure, the message is then packed together with the parameters and sent to

the server which has implemented the same procedure. The server executes the procedure and sends a response with the result [16].

Apache Thrift

Thrift is a framework for cross language application communication. In the previous chapter we evaluated the frameworks serialisation capabilities, but Thrift also has functionality as a communication protocol for remote procedure calls between web services [2].

Since Thrift is a framework it is not completely platform independent. It has support for 14 different languages, including Java, C#, C++, JavaScript, Python and PHP. The serialisation alternatives which are presents across all languages in the Thrift framework are JSON and binary. Thrift has support for both HTTP and raw TCP as transportation layers.

gRPC

gRPC is also a framework for cross platform remote procedure calls just like Thrift. In gRPC Protocol Buffers are used both as a serialisation format for messages but also as an interface definition language that makes the cross platforms calls feasible.

The gRPC framework is available in 10 different languages, including Java, C#, C++, JavaScript and Python. gRPC has support for the newly realeased HTTP 2 protocol, but can also use TCP as its transportation layer.[12]

REST

Representational state transfer (**REST**) is an architectural style for communication between web services. REST is not tied to any specific transportation protocol or serialisation method, but transporting data over HTTP and serialising it to either JSON or XML has become the most widely used alternatives. Data resources are accessed through an individual uniform resource identifier (**URI**).

When REST is implemented with HTTP as a means for transportation the resources are identified with a unique URL. Resources are managed with the use of the HTTP verbs `GET`, `POST`,`PUT` and `DELETE`.

Data is requested from a REST architecture by sending a GET request to the URL that identifies a specific resource. If an application were to add or change data a POST message is sent to the REST architecture containing information about that data resource. REST has become one of the most popular alternatives to communication between web services due to the fact that it is lightweight and can be adopted in any language.

GraphQL

GraphQL [11] is a query language for application programming interfaces that models data as a graph of JSON-objects. GraphQL is meant to be implemented as an application layer on top of an underlying data storage system. It can therefore be implemented in any system regardless of the underlying data storage model. In GraphQL a data query is modelled as a JSON-object where the name of the data fields in the query specifies which information that is requested. The GraphQL framework parses the request and adds the missing values to the fields in the query. The client will receive the requested data with values added to the specified fields in the query GraphQL is not tied to any transportation protocol but using a HTTP endpoint to transmit and receive data in form of GET and POST requests have become the standard approach [19].

An example of a GraphQL request and response is shown below in listing 5.1 and 5.2. The query is structured as a JSON object but without the values for every name/value pair. The GraphQL frameworks parses the request and adds the values to the requested fields specified in the query. The field friend is in this case defined in the server to be of the type person. The GraphQL server specifies which queries that are acceptable and the fields that they may contain. The client application then has the option to send a query for only the fields that it needs and will never have to cause unnecessary strain to the network due to over fetched data.

```
query{
    person{
        name
        ID
        }
        friends{
            name
        }
}
```

Listing 5.1: Example GraphQL request, a JSON object with only the name of the data fields

```
{
    "data":{
        "person":{
            "name" : "John Smith",
            "ID" : "12345",
            friends : [
                {
                    "name" : "Bill Smith"
                },
                {
                    "name" : "Bob Smith"
                }
            ]
        }
    }
}
```

Listing 5.2: Example GraphQL response, a JSON object with name/value pairs

5.4 Summary

There are no definite answer to which communication method is best for microservices. While the frameworks gRPC and Thrift potentially offers the best performance they add dependencies to the microservice architecture, forcing it to be developed in certain languages. REST, which is completely platform independent since it only is an architectural style, can be implemented in any platform, with any language, using any transportation and any serialisation. REST however lacks the structure that frameworks and protocols offers, and must often use third party tools to keep track of its interfaces. SOAP offers a structured way to send messages between services with strict rules defining the structure of the messages and how to process them. SOAPS major drawback is that it is tied to using XML which clearly has the worst performance of all reviewed serialisation formats.

GraphQL offers a new way of requesting data in a format that is easy to understand. GraphQL offers an abstract view of data resources as nodes in a graph where any nodes can be combined to form a query. GraphQL was built to aggregate data to a single service and fits the API-gateway pattern very well, but it adds unwanted dependencies to the system since it requires a framework to parse requests and create responses. If the performance of the GraphQL framework can parse and serialise data with no major performance loss compared to a REST/JSON architecture it could be a viable alternative.

Chapter 6

Implementation

In this chapter we present an implemented test environment in which the performance of GraphQL as well as REST could be measured. The details of the implementation as well as a system overview are included so that the experiments and performance reviews can be duplicated if any one wants to verify the results. The goal with the implementation is to create a system where it is possible to answer the following question from the problem description:

- Can the communication between a web application and a system of microservices become more efficient with the query language GraphQL, compared to HTTP requests and a REST API?

6.1 System overview

The test environment consists of a microservice system with four different microservices hosted in Azure Service Fabric and a test client built as a web application. The services are implemented in .NET and the web application in HTML and JavaScript. The microservices are accessed through a load balancer which redirects traffic from an open access points to either the REST or GraphQL gateway. The gateways exposes a REST and GraphQL API respectively and both have communication options to all microservices. See Figure 6.1 for a system design overview.

6.2 Microservices

The microservices in this system are only designed to access data from a database and not to do any actual computations. Each service has access to data that is only available to that specific service. The services communicates internally with the gateways through the built in functionality of remote

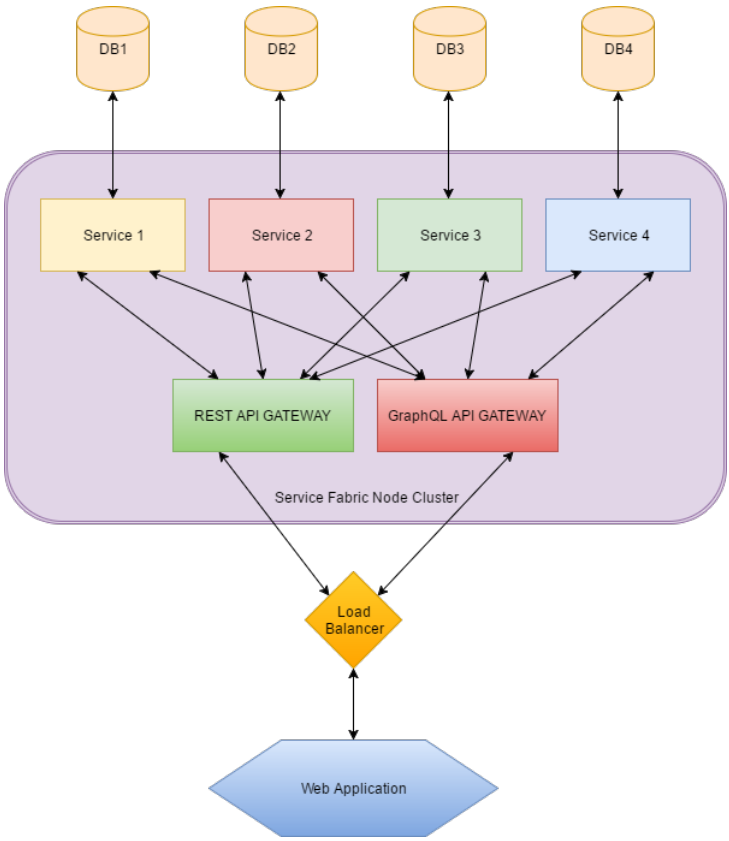

Figure 6.1: System overview

procedure calls. The databases are SQL databases provided by Microsoft and are hosted on a separate virtual machine outside of the Service Fabric cluster.

The services are hosted in a Service Fabric cluster with three nodes. All services are hosted on the same node as an application container but can be scaled separately if needed. Each node is a virtual machine of the type `Standard A0` with specifications shown in Table 6.1.

Hardware	**Specification**
Server	Located in West Europe
CPU	Intel Xenon E5-2660 @2.2 GHz, single core
RAM	768 MB
Operation System	Windows server 2012 R2 64-bit

Table 6.1: The hardware specifications of the servers that the system is hosted on

6.2.1 Test data

The data that can be accessed by the system is showed in Figure 6.2. Each service has access to one of the tables: animals, cars, persons, or buildings. Each field in the table holds exactly 6 characters in order to get predictable data objects that can be reproduced.

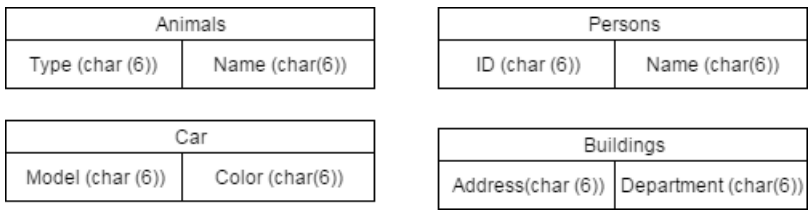

Figure 6.2: The data tables where the test data is stored

6.2.2 REST API

The REST API is implemented in C# with the framework ASP.NET [4]. The API offer endpoints for fetching all the data that a single service has to offers and an endpoint which fetches a bundle of data with all the information available in all services. For serialisation the API only accepts messages in a JSON format. The HTTP endpoints are designed in the following way:

- /api/person/
- /api/animal/
- /api/car/
- /api/building/
- /api/Databundle/

6.2.3 GraphQL API

The GraphQL endpoint is also hosted with a ASP.NET HTTP endpoint that can be accessed through `/api/graphql`. The GraphQL engine that parses requests and builds responses is built with the `graphql-dotnet` [10] library for .NET. The GraphQL service is designed with a schema that supports one type of query. The query has four fields of data where each field is a list of data from one service.

6.3 Web application

The web application is built to fetch data from the microservices so that the performance of GraphQL and REST can be meassured. The web application is built using HTML and Javascipt with the framework React. Accessing the REST API is done with the HTTP client Axios [7].

In order to format and send GraphQL requests the GraphQL client Apollo [3] for React was used. Apollo is built to make communication with GraphQL servers through HTTP easier.

The web application measures the total time it takes to receive a response to a request from both the GraphQL and REST service.

Chapter 7

Results

In this chapter the results from performance tests done in the implemented test environment will be presented. A set of test cases where a varying amount of data is gathered from the microservices will be defined so that we can observe how GraphQL and REST performs in different scenarios. Separate tests where we observe how taxing it is for the CPU to handle each communication type will also be presented.

7.1 Test 1

The first test consists of fetching a fairly small amount of data from each service using both the REST and GraphQL interfaces. The latency when fetching data from a cloud hosted system can vary a lot. In order to get reliable data 100 requests are sent with a second long pause between each request to ensure that the load on the system has minimal effect on the result.

Result

The response time is measured as a complete round trip in the system including the time it takes for the microservices to process and fetch data from the databases. The corresponding response time for each request is shown in Figure 7.1.

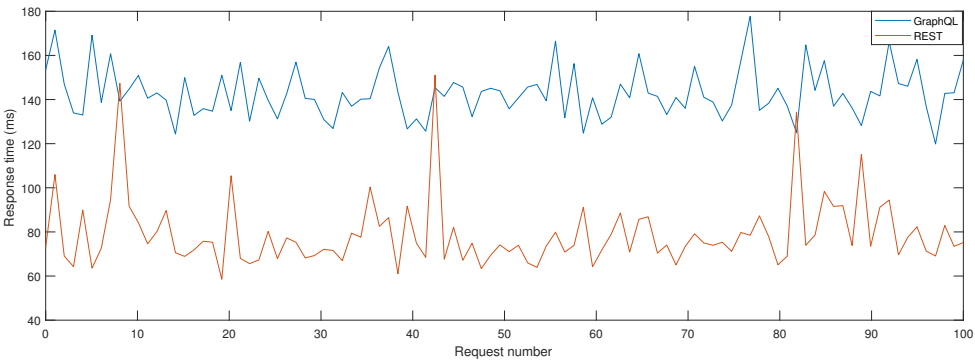

Figure 7.1: The response times when fetching data from the GraphQL and REST interfaces. (Test 1)

Communicating with the system through REST resulted in a lower response time compared to GraphQL. The REST communication got the fastest response in `58.53` milliseconds and the slowest response in `151.08` milliseconds. GraphQL performed considerably worse with a fastest response time of `119.86` milliseconds and a slowest response time of `177.72`. The comparison of the average response time for REST and GraphQL is shown in Figure 7.2. REST had an average response time of `78.94` milliseconds and GraphQL had an average response time of `142.92`, making GraphQL `81.05%` slower on average.

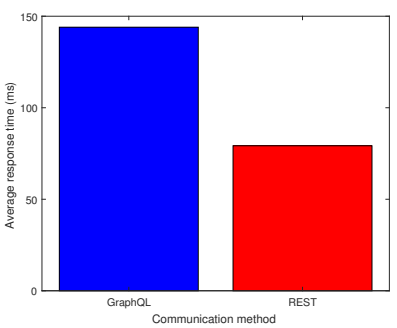

Figure 7.2: The average response times of GraphQL and REST compared. (Test 1)

7.2 Test 2

The second test case uses the same set up as the first, but in this case we measure the performance when handling requests for a larger amount of data. The databases were extended to each contain 25 rows.

Result

The measured response time in this test showed that GraphQL lags behind response time when the data amount is increased. An overview of the response times for all 100 requests using both REST and GraphQL is shown in Figure 7.3. Communication through REST had the fastest response time with `63.04` milliseconds and the longest response time with `260.39` milliseconds. GraphQL could not match the number produced by REST with a fastest response time of `203.00` milliseconds and the longest response time of `840.56` milliseconds. A comparison of the average response time in test case 2 is shown in Figure 7.4. Fetching the data through a REST interface took on average `100.35` milliseconds while fetching data using GraphQL on average took `256.80` milliseconds. Data fetching using GraphQL in test case 2 were on average `256%` slower than fetching the same data with REST.

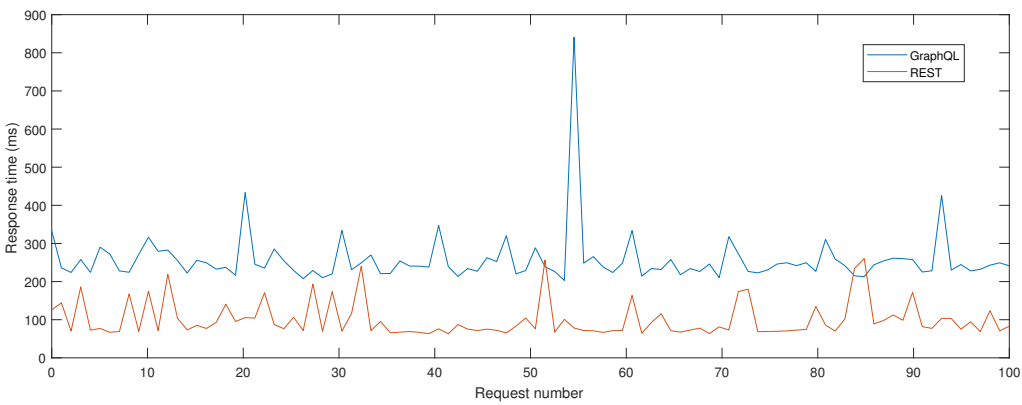

Figure 7.3: The response times when fetching data from the GraphQL and REST interfaces. (Test 2)

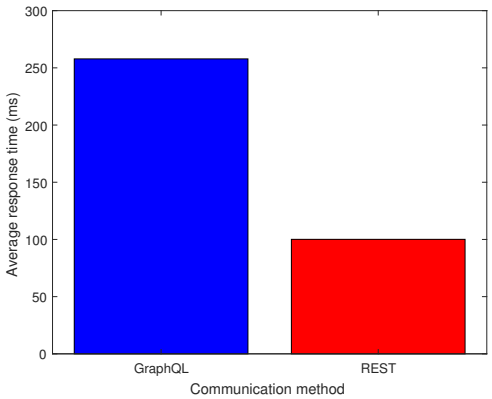

Figure 7.4: The average response time for communicating with microservices using REST and GraphQL. (Test 2)

7.3 Test 3

In the third test case we increased the data amount further, requesting 50 rows of data from each service. The third test case followed the same procedure as both previous test cases.

Result

The third test case further proves that the GraphQL frameworks fails to keep up with the REST architecture performance wise. The response times for GraphQL and REST are shown in Figure 7.5. REST had a minimum response time of `65.25 ms` and a maximum response time of `185.91 ms`. GraphQL performed much worse with a fastest response time of `301.14 ms` and a slowest response time of `759.84`.

A comparison of the average response time for GraphQL and REST is shown in figure 7.6. We see no performance decrease from the REST service in this this test compared to test case 2. The average REST response time were actually faster than in test case 2 with `84 ms`. The average response time for GraphQL in this test were `357.89 ms`. The average GraphQL response were 426 % slower than the REST response, a relative increase of 179 percentage points compared to test case 2.

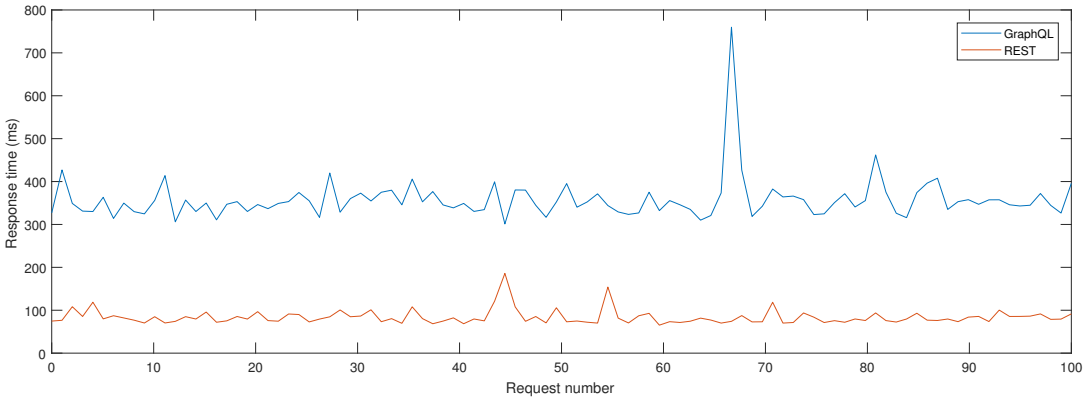

Figure 7.5: The response times when fetching data from the GraphQL and REST interfaces. (Test 3)

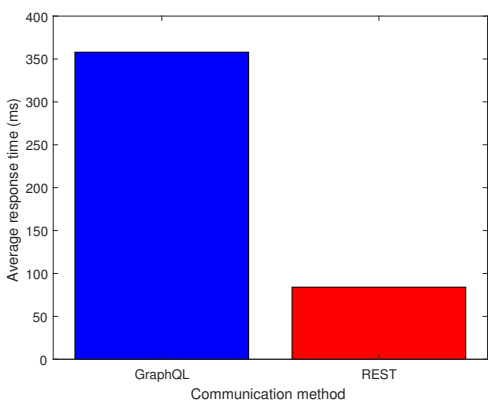

Figure 7.6: The average response time for communicating with microservices using REST and GraphQL. (Test 3)

Chapter 8

Conclusions

The results from the tests show that GraphQL could not match the REST/JSON interface in terms of response time. Even when fetching few data resources the GraphQL parser causes the response time to suffer. Since GraphQL is tied to using JSON as a serialisation method it is limited to a sub par serialisation technique.

What GraphQL offers instead of great performance is a much easier way of fetching specific data including only the fields that a client wants. The problem with a REST interface is that every resource is mapped to a URI, and if different applications want custom made data structure for their need a new REST endpoint have to be made for each case. With GraphQL only the most detailed structure handling the same data resources have to be created, from that resource each client can specify exactly which field they want to fetch. This eliminates large interfaces which are hard to maintain and have no clear rules for how to handle several versions of the same API. GraphQL can in this specific scenario, where few data resources are fetched at a time be a good idea to use. But if performance is the most important metric it is not a strong contender, at least not the .NET implementation presented in this thesis.

The REST architecture also have the possibility of an even better performance, both in terms of lower response time and a smaller message size if a binary serialisation protocol were used instead of the JSON format.

It is however not possible to say if using a RPC framework such as gRPC or Thrift and binary serialisation is better in a microservice architecture. Opting to use the frameworks and binary serialisation forces you to give up the freedom and corner stone of microservices that is a true cross platform tolerant architecture. It ultimately becomes a trade off between limiting the system to use specific languages and tools and a higher performing communication method that is more troublesome to implement.

The choice of infrastructure on which to host the microservices on is also a

question that has no definitive answer. For raw performance using containers instead of virtual machines were shown to be more effective with better CPU performance and less memory required. For small systems where monitoring of the system is not a priority and no massive data processing is required the AWS platform Lambda is a good alternative which outperforms regular virtual machines.

8.1 Future work

To better understand the microservice pattern and to give a more complete picture of the architecture a review of design choices for crucial parts of microservices such as service registry, service discovery and inter process communication could be done. It was my intention to include some of these parts in this report but as it turned out that would have required a lot more time and would result in a far longer report.

We would like to extend the performance review of GraphQL to include implementations in different languages such as JavaScript to see if the performance is equal across different platforms.

Performance tests of REST using a different serialisation format and RPC technologies could also be implemented into the test environment to create a complete bench marking tool where actual numbers for each communication method could be compared with numbers instead of relying on a performance review from scientific papers.

Chapter 9

Acknowledgements

I would like to thank Dohi for giving me the opportunity to do this project and for presenting me with an interesting subject. I would especially like to thank my handler at Dohi: Jonas Karppinen, and my supervisor at Umeå University: Mikael Rännar, for their help and support.

References

[1] Amazon web services. https://aws.amazon.com (Visited 2017-05-13).

[2] Apache thrift documentation. https://thrift.apache.org/ (Visited 2017-05-13).

[3] Apollo official website. http://www.apollodata.com/ (Visited 2017-05-13).

[4] Asp.net documentation. https://www.asp.net/ (Visited 2017-05-13).

[5] Aws lambda faq. https://aws.amazon.com/lambda/faqs/ (Visited 2017-05-13).

[6] Aws lambda platform. https://aws.amazon.com/lambda/ (Visited 2017-05-13).

[7] Axios on github. https://github.com/mzabriskie/axios (Visited 2017-05-21).

[8] Cloud computing (wikipedia). https://en.wikipedia.org/wiki/Cloud_computing (Visited 2017-05-13).

[9] Extensible markup language (xml) 1.0 (fifth edition) w3c recommendation 26 november 2008. https://www.w3.org/TR/REC-xml/ (Visited 2017-05-15).

[10] graphql-dotnet on github. https://github.com/graphql-dotnet (Visited 2017-05-13).

[11] Graphql official webpage with documentation. http://graphql.org/ (Visited 2017-05-13).

[12] grpc documentation. http://www.grpc.io/ (Visitied 2017-05-14).

[13] The javascript object notation(json) data interchange format rfc 7159 (ietf). https://tools.ietf.org/html/rfc7159 (Visited 2017-05-13).

[14] Microsoft azure. https://azure.microsoft.com (Visited 2017-05-13).

[15] Protocol buffers documentation. https://developers.google.com/protocol-buffers/ (Visited 2017-05-13).

[16] Remote procedure call (wikipedia). https://en.wikipedia.org/wiki/Remote_procedure_call (Visited 2017-05-14).

[17] Xml soap w3schools.com. https://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (Visited 2017-05-13).

[18] S. Kami Makki A. Sumaray. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *ICUIMC '12 Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, 2012.

[19] L. Byron. Graphql: A data query language. http://graphql.org/blog/graphql-a-query-language/ (Visited 2017-05-15), September 2015.

[20] A. Messina et al. The database-is-the-service pattern for microservice architectures. In *International Conference on Information Technology in Bio- and Medical Informatics*, volume 9823, pages 223–233. Springer, Cham, 2016.

[21] C. G. Kominos et al. Bare-metal, virtual machines and containers in openstack. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017.

[22] F. Belqasmi et al. Soap-based vs. restful web services a case study for multimedia conferencing. *IEEE Internet Computing*, 16 Issue: 4:54–63, July-August 2012.

[23] M. Villamizar et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, 2015.

[24] M. Villamizar et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[25] Popic et al. Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*, 2016.

[26] R. K. Barik et al. Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016.

[27] J. Weber F. Montesi. Circuit breakers, discovery, and api gateways in microservices. Technical report, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark, 2016.

[28] M. Fowler. Microservices by martin fowler, 2014. `https://martinfowler.com/articles/microservices.html` (Visited 2017-05-13).

[29] J. Grabis J. Tihomirovs. Comparison of soap and rest based web services using software evaluation metrics. *Information Technology and Management Science*, 19:92–97, December 2016.

[30] A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, 2015.

[31] K. Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, 2012.

[32] S. Newman. *Building Microservices*. O'Reilly Media Inc., USA, 2015.

[33] C. Richardson. Building microservices: Inter-process communication in a microservices architecture. `https://www.nginx.com/blog/building-microservices-inter-process-communication/` (Visited 2017-05-13), July 2015.