NTNU
Norwegian University of
Science and Technology

Optimizing Inter-Service Communication Between Microservices

Martin Storø Nyfløtt

Applied Computer Science
Submission date: December 2017
Supervisor: Rune Hjelsvold, IDI

Norwegian University of Science and Technology
Department of Computer Science

Preface

This thesis explores approaches for optimizing the communication between services in a microservice architecture. Limitations of the REST architectural pattern within the microservice pattern is investigated through a literature review, and alternative protocols to HTTP/1.1 are explored for communication between services.

This work was conducted during fall semester in 2017 under Department of Computer Science (IDI) at the Norwegian University for Science and Technology (NTNU).

The readers of this thesis are expected to have a background in information technology and should be familiar with the basics of computer networking and architecture.

15-12-2017

Acknowledgment

First and foremost, I would like to thank my supervisor prof. Rune Hjelsvold for giving invaluable advice, guidance and continuous feedback throughout this period. I would also like to thank my friends and family for all help and support which has made this work possible. I would also especially thank prof. Christopher Frantz for giving his advice and opinion which has had great impact on this work. Another person I would like to thank is my friend Pooya Fatehi for making the illustration of an ant that represents a microservice on the front-page of this thesis.

Abstract

The microservice pattern is a new alternative to architecting back-end systems. Instead of building large, monolithic systems that lead to issues related to scalability, maintainability and extensibility, systems are built as a set of small, independent services – microservices. Even though these services operate independently to a certain degree, there is often communication between them. Going from a monolithic system to a distributed system, the networking communication becomes a challenge.

Representational State Transfer (REST) is a commonly used architectural pattern for designing service interfaces. While REST promotes generalization of endpoints through uniform interface, this leads to more overhead when transferring documents. This thesis discuss issues of REST and discusses them in context of microservices through a literature review. Some of the discussed issues relates to how concepts under REST should be approached where much discussion relates to how the linking between representations should be approached. There are also other issues relating to service discovery, transactions, security, and reliability.

HTTP is often being used as a fundamental transfer protocol to implement a system following the REST architectural pattern. Other protocols have been explored in this thesis to determine their behavior in a microservice architecture. HTTP/1.1, HTTP/2 without encryption, and Constrained Application Protocol (CoAP) were tested under different latencies (0ms, 5ms, 10ms) as the messaging protocol between microservices in the AcmeAir benchmarking system. Results show that HTTP-/1.1 was able to provide higher throughput compared to HTTP/2, whereas CoAP had a lower request throughput under all latencies. The differences in the latencies did, however, even out between the protocols under higher latencies. Considering bandwidth, HTTP/2 used least bandwidth followed by CoAP and HTTP/1.1. Some of the behavior of the CoAP protocol can be explained by framework limitations. This has resulted in contributions to the Californium Java CoAP framework by avoiding thread creation during request processing and introducing asynchronous APIs in the framework for HTTP messaging. Furthermore, CoAP required some request headers to be implemented as they were not present in the framework or the protocol. It is argued this leads to higher coupling compared to the other protocols.

Acronyms

ABAC - Attribute Based Access Control

ACL - Access Control Language

API - Application Programming Interface

AST - Accountable State Transfer

BSON - Binary JSON

CD - Continuous Development

CI - Continuous Integration

CoRE-RD - Constrained RESTful Resource Discovery

CPI - Cycles Per Instruction

CRUD - Create Read Update Delete

DNS - Domain Name System

DNS-SD - DNS Service Discovery

DTLS - Datagram TLS

EC2 - Amazon Elastic Cloud Computing

ECS - EC2 Container Service

FEC - Forward Error Correction

GC - Garbage Collection

h2c - HTTP/2 over TCP (no encryption)

h2 - HTTP/2 over TLS (encrypted)

HATEOAS - Hypertext As The Engine Of Application State

HTTP - Hypertext Transfer Protocol

HTTPS - HTTP Secure (HTTP with encryption)

IANA - Internet Assigned Numbers Authority

I/O - Input/Output

IoT - Internet Of Things

ISC - Inter-Service Communication

JIT - Just In Time

JSON - JavaScript Object Notation

LPAR - Logical-Partition Mechanism

MIME - Multipurpose Internet Mail Extensions

QoS - Quality of Service

QUIC - Quick UDP Internet Connections

ReLL - Resource Linking Language

REST - Representational State Transfer

RMM - Richardson Maturity Model
ROA - Resource Oriented Architecture
RPC - Remote Procedure Call
RTT - Roundtrip Time
SLA - Service Level Agreement
SOAP - Simple Object Access Protocol
SOA - Service Oriented Architecture
TCC - Try-Confirm/Cancel
TCP - Transmission Control Protocol
TDD - Test Driven Development
TLS - Transport Layer Security
UDP - User Datagram Protocol
URI - Unique Resource Identifier
WS-* - Web Service Stack
XML - eXtensible Markup Language
XMPP - Extensible Messaging and Presence Protocol
XP - Extreme Programming
XWADL - XMPP Web Application Description Language

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic

The microservice architecture pattern is an emerging architectural pattern that has gained much popularity in the industry for architecting backend systems. Typically, server-side applications are often built in one or several large code-bases, often developed as monoliths. Multiple tech companies such as Amazon, LinkedIn, Netflix, and SoundCloud had large, monolithic architectures which limited their abilities in terms of scalability and extensibility [1, 2]. These monoliths were then refactored into a set of smaller, independent services that communicate together. These services are often referred to as "loosely coupled services with bounded context" [3]. Being forefronted, and popularized by the previously mentioned tech companies, there is no consensus on a concrete definition of the microservice pattern itself. However, the pattern can be described as a way to develop a system consisting of small, independent services that are centered around business capabilities and run in isolation from each other [1, 4, 5]. Despite the services being independent from one another to a certain degree, the services can still communicate in a synchronous or asynchronous fashion, typically using messaging queues or through Application Programming Interfaces (APIs) following concepts from the Representational State Transfer (REST) pattern, typically using Hyper Text Transfer Protocol (HTTP) [1, 4].

Some of the benefits of the microservice architecture comes from the independent nature of the services: each service can be developed, tested, and deployed independently. The services are then typically developed by independent teams which allows services to be more specialized towards their use-case and being implemented in different languages (polyglot programming). Considering that the system is made up of loosely coupled, ideally independent services, if one service would become unavailable, this would have a limited effect on the whole system availability by utilizing patterns such as circuit breakers and designing services for failure [6].

Microservices on the other hand, introduce several challenges. Additional costs are required in terms of operational costs and administrating a larger set of services compared to fewer monolithic services. Complexity is rather shifted from the application towards the infrastructure [7]. Small-scale systems may not benefit from the pattern due to additional overhead and administration of the services. There is also extra cost in terms of network communication which can negatively affect performance. Other challenges relate to service discovery, security, service granularity (how large should a service be), transactions, logging, API versioning among the various services, and general issues from implementing a distributed system.

A much discussed topic regarding microservices is its relationship with Service-Oriented Architectures (SOA). Some consider SOA and the microservice architecture two distinct patterns due to how the two patterns assert problems related to communication heterogeneity, service granularity, security, transactions, and service ownership [8]. However, others argue that the microservice pattern is a way to do SOA, much like scrum is an agile development methodology [9]. Additionally, Zimmermann argues that microservices do not violate any SOA principle, but rather embrace existing patterns [10].

1.2 Keywords

Microservices, web services, REST, RPC, web, HTTP, CoAP, inter-service communication, service-oriented computing, distributed computing, architecture, service-oriented architecture

1.3 Problem Description

Inter-service communication (ISC) refers to the communication between microservices. The additional network overhead is largely recognized as one of the major bottlenecks that negatively affects performance when communicating between multiple services [8, 5, 11, 12, 13]. Zimmermann does also raise the question whether there are alternatives to using REST in the context of microservices and whether web protocols are sufficient for communication [10].

As mentioned earlier, communication can be either synchronous or asynchronous [14]. *Synchronous communication* means a call will block from the service consumer until the requested operation has completed. With *asynchronous communication*, on the other hand, the requests may be queued e.g. in a messaging queue and processed at a later point in time. This makes it possible for the client to start working on other tasks independently of when the request is further processed.

This work aims at examining the overhead in inter-service communication, proposing several methods for mitigating the extra overhead in addition to benchmarking several protocols for inter-service communication. The adoption of REST within the microservice pattern to identify limitations and challenges of REST is also explored.

1.4 Justification, Motivation and Benefits

Microservices is an emerging architectural pattern and thus has a limited amount of research and literature [2, 15]. This project would contribute to the overall understanding of the microservice pattern. Additionally, it would contribute to the industry by giving advice to new microservice implementations, in addition to proposals of new network protocols. Potentially, existing systems may benefit from the findings of this project and can use the results in order to achieve better performance. The reduced response time may also lead to less computing resources being used. As a result of this, systems can save energy and contribute to greener systems. Microservices is a pattern that is often used in context of cloud computing [1], and many cloud vendors use a pay-as-you-go model where users of the cloud platforms are billed, e.g. for bandwidth usage. Reducing the amount of bandwidth one service uses would also lead to less costs for hosting the microservices.

Stakeholders that can potentially benefit from the results in this work would include: system architects seeking guidance and advice on microservices, protocol developers designing the protocols being used for ISC, service-providers that provide platforms and support for microservices, and developers and researchers looking for related work, and knowledge regarding microservices.

1.5 Research Questions

This project will aim at answering the following research questions:

- R1: REST limitations: What are the architectural limitations of using REST in microservices?
- R2: Benefits and limitations of HTTP/2: For ISC, what are the benefits and limitations of HTTP/2 compared to HTTP/1.1 in terms of performance and latency?
- R3: How does CoAP compare to HTTP: How can CoAP be applied in a microservice system, and what are the challenges of using CoAP instead of HTTP?
- R4: Latency in ISC: Which factors impact latency in ISC and how can these be reduced?

1.6 Planned Contributions

This project will contribute to the area of microservices, a rather new research area with limited literature [2, 15]. It will provide more knowledge and insight into the microservice pattern and can act as a resource for how to approach certain problems within microservices and service orientation. Ultimately, the project may find better approaches for performing ISC that can result in lower response times.

These results may also be generalized and used outside the microservice area, and can potentially inform the creation of newer protocols. The results of this thesis has already lead to contributions to the Californium Java CoAP framework.

1.7 Outline

A general introduction is given to the reader in the topic of REST, microservices, and inter-service communication in Chapter 2. The research question regarding architectural limitations of REST in microservices is first discussed by a literature review in Chapter 3. After this, on a concrete level, HTTP/1.1, HTTP/2, and CoAP are compared in AcmeAir under different latencies in Chapter 4 to answer the research questions regarding latency and protocols. A conclusion is given in Chapter 5 where each research question is summarized based on the findings in the previous chapters.

2 Background

2.1 Microservices

Microservices is an emerging research topic in academia. Alshuqayran et al. [15] present a systematic literature review of the topic, and found that the term microservices was first used in 2010. The term was, however, used for a framework for automatic service description instead of an architectural pattern [16]. The survey by Alshuqayran et al. argue that the term was first defined by Fowler and Lewis in 2014 [1], although Fowler states that the term was first discussed by a group of architects in a workshop near Venice, May 2011, in order to describe a new architectural style the participants had observed.

A systematic mapping survey by Vural et al. [2] discusses challenges and definitions of microservices. They found that much of the published formal literature are solution proposals or evaluation research, and that there is a lack of opinion papers and experience reports. Furthermore, they discovered challenges related to integration and deployment, are most frequently discussed in the literature.

There is much discussion related to the actual definition of microservices. While Fowler describes microservices as an architectural pattern [1], Zimmermann [10] discusses whether microservices is a concrete approach to SOA or whether it can be considered as a new, novel architectural pattern. He argues based on a literature review comparing positions in the industry and academia, that microservices does not bring any conceptually new properties compared to SOA, but as an approach towards SOA using "state-of-the-art software engineering practices" [10]. He also mentions that definitions of microservices include terms such as development process and organization related terms, whereas other architectural patterns such as REST are defined as a set of constraints on a more abstract level [17]. For example, Aderaldo et al. [18] propose a set of requirements for a microservice benchmarking system, where version control of source code is one of the proposed requirements.

The debate of the different definitions of microservices can be seen in the discussion with Pautasso et al. [19], that emphasized that each of the discussing parties have a different concrete definition of microservices.

Considering REST and SOA being defined as a set of principles or constraints, many microservice definitions include for examples that light-weight containers are used to deploy the applications, in addition to embracing DevOps[1] and au-

[1]A paradigm or culture where there is close relationship between development and operations of a

tomatization paradigms to simplify deployment. It would be reasonable to assume that these practices comes as a result of having to deploy a system consisting of multiple autonomous services and that they are not directly tied to any particular development process or methodology.

2.1.1 Microservice definition

In order to simplify discussion in this thesis, the term microservice being used in this thesis is defined as:

> Microservices is an architectural pattern where an application is composed of small, loosely coupled, ideally independent (micro)services. The unity of these services form through the concept of emergence, the whole application itself. The independent nature of each service isolates application failure to a smaller segment of the application.

A particular microservice is defined as:

> A service within a microservice system which is typically designed around, but not limited to a particular business use case. This service communicate using a standardized, light-weight messaging mechanism that promotes loose coupling with other services.

As a result of the independence of microservices, these services can be written in different programming languages, thus enabling polyglot programming. The isolation of the different services comes at the cost of increased network traffic and increased application complexity. Caching or data duplication between services can be used to reduce the network communication and dependency of other services. Paradigms such as continuous delivery (CD), containerization, and DevOps can be introduced to simplify the development and deployment efforts of the application.

The earlier definition specifies that a microservice should be *small*. However, having a too small of a microservice can result in an increase of network communication and architectural complexity, also referred to as a nanoservice [20]. On the other hand, a large enough service would result in a monolith that becomes less agile and is difficult to maintain in the long run. The difference between a nanoservice and a microservice would be that the nanoservice outweighs its utility due to overhead from the additional efforts required to develop, maintain, and operate the service. Some practitioners argue that a microservice size should be determined in terms of amount of lines written, the amount of weeks it should take to write a new service, or centered around the two-pizza rule proposed by Amazon [21, 1]. The proposed definition of *microservice size* being used in this thesis is defined as follows:

computer system.

> The size of a microservice should be centered around one single business capability where the utility of the service should outweigh the overhead associated with architectural complexity, operational costs, and network overhead, but not result in a generalized, monolithic service which leads to a service with high complexity.

The previously mentioned metrics (code size, team size, etc.) can be considered as appropriate tools as an attempt to follow this definition.

Furthermore, this thesis uses the terms *service consumer* and *service provider*. Both of these can be microservices but are used in a more abstract context where the service consumer is a client making a request to the service provider, which is a server providing some service to the service consumer.

2.1.2 Open issues in microservices

Microservices being a distributed system, inherit the problems of distributed computing. Complexity of the application is (compared to a monolithic architecture) shifted from the application itself towards the supporting infrastructure of the application. There are however various strategies to mitigate these issues. To summarize, these are the following typical issues associated with microservices:

- General issues with a distributed system (Latency, data management, interface design, logging, infrastructure supporting the system) [10, 22].
- Formalization-specific issues (Service size, design, relationship with SOA) [10].
- Moving from a monolith (one service or application) to multiple microservices leads to an increase of administrating and orchestrating these services [4].
- A general lack of "best-practices" (e.g. how a monolith should be decomposed to microservices) [10, 23].
- Increased architectural complexity (higher cognitive load) and development costs (sometimes referred to as the "microservice premium") [19].
- Security (larger attack area, trust, heterogeneity) [5].

Fazio et al. [24] describe some of the major challenges when deploying a microservice application to the cloud. One challenge being discussed is the heterogeneity of different deployment frameworks in different clouds. They also mention that one research challenge in microservices is knowing how microservices should be most efficiently deployed, either in the same container, on the same virtual machine, the same physical host, or in the same network. This is also supported by Salah et al. [25] who investigated the performance differences of deploying an application on a Amazon Elastic Cloud Computing (EC2) instance compared to the EC2 Container Service (ECS).

Klock et al. [21] propose a model for clustering together microservices based on workload on the services. Their solution enables gathering operational data from existing running microservices and based on this data they suggest a deployment model for optimizing the location of the different microservices. Using their model, they managed to optimize an existing microservice system which resulted in 23% higher request throughput. Their proposal also makes it possible to group together microservices based on different types. Some microservices may be CPU-intensive by optimizing an artificial neural network, or GPU-intensive by rendering video files.

In Microservice Tenets, Zimmermann [10] argues based on a review of industry practices and formal literature some of the major challenges when applying the microservice architecture. The most recurring ones are the issues related to distributed computing: "data integrity and consistency management, service interface design and evolution, and application/service management (including application and infrastructure security management)" [10]. As mentioned earlier, he touches the debate regarding the particular definition of microservices. Many practitioners define the architectural pattern together with implementation-specific details such as that the microservices needs to be deployed using a continuous integration (CI) pipeline, etc.

Another debated topic in terms of the microservice pattern is its relationship with SOA. Zimmerman argue that the microservice pattern is not a new novel architectural pattern, but can be considered as an approach towards SOA while also embracing new development strategies such as test-driven development (TDD), and extreme programming (XP) [10].

2.2 Inter-Service Communication

As mentioned earlier, there are various strategies for inter-service communication. Communication can be synchronous or asynchronous. This communication can be between one (one-to-one) or several (one-to-many) services.

Richardson [14] describes various patterns in inter-service communication (Summarized in Table 1). Synchronous communication is used when the call from the service consumer is blocked until the operation of the requested task has completed and the result is provided back as a result. This is a typical behaviour for RESTful HTTP APIs where the entire operation is completed once the service consumer receives the response (Request/Response). Compared to synchronous communication, with asynchronous, the request is processed at a later point in time, outside the request-response cycle by the service consumer. It may for example be put in a message queue and later processed by a worker service. The service consumer may or may not receive any notification of the completion of the task. It is possible to incorporate asynchronous communication using REST APIs (request/asynchronous response), where the worker node could perform a request back to the service consumer through a push mechanism, or by having the service consumer pull for updates. This may however introduce the notion of state on the service which would break the principle of statelessness within REST [17]. The service consumer may also not care about the result (notification), which is more of a fire-and-forget approach. Services can also subscribe to events happening in other services. One service can push out a message to several other subscribed services (publish/asynchronous responses).

	One-to-one	One-to-many
Synchronous	Request/response	-
Asynchronous	Notification Request/async. response	Publish/subscribe Publish/async. responses

Table 1: Inter-service communication patterns [14].

However, Bonèr [26] argues that synchronous communication should be avoided, and is by some considered as an anti-pattern within microservices. This is due to the fact it creates a higher coupling between the different microservices. Westhide [27] claims it violates the concept of having services working in isolation. Although there are cases where services may need information from other services, and these dependencies are resolved through synchronous calls, it should be avoided. Instead, data should be replicated across different services through notification mechanisms or events, often referred to as *event-driven messaging* [28, 4].

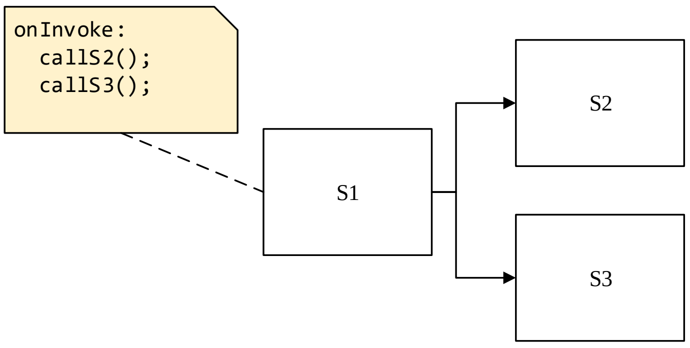


Figure 1: Orchestration between services. When the service S1 is invoked, it will directly call the services S2 and S3.

The collaboration between services is typically described as either orchestration (Figure 1) or choreography (Figure 2) [4, 5]. In orchestration, a central mediator is responsible for telling each service what should happen. This could for example be a service directly invoking other services through synchronous calls. With choreography, there is no common mediator. Instead, the publish/subscribe pattern is used for invoking other services. Contrary to the previous example regarding orchestration, the service does not directly know which services to invoke. Instead, services that need to be invoked register themselves – or subscribe, to events from the service. Within microservices, choreography is preferred over orchestration due to how choreography leads to a lower coupling between the services [10, 4]. Dragoni et al. [5] argue orchestration leads to tighter service coupling and uneven distribution of responsibilities. Richards [8] describes this as "high efferent" coupling, which means a service is highly dependent on other services to perform its task. He also mentions that choreography can lead to worse application performance due to the extra cost of network communication between different services.

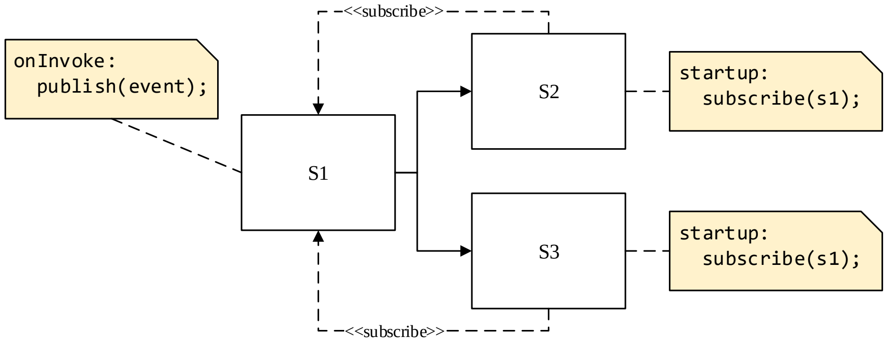


Figure 2: Choreography between services using the publish/subscribe pattern.

Villamizar et al. have investigated the performance impact and costs on a monolithic system compared to a microservice system through a use-case in the Amazon cloud [12]. Their system consists of two parts: One performing a heavily computational operation in order to determine a payment plan according to a provided set of parameters, while the other part would interact with a relational database in order to obtain stored records of user data. Performance analysis was done using Apache JMeter[2] in order to generate and analyze responses from the two setups. They find that the monolithic architecture was performing about 1000ms faster

[2]https://jmeter.apache.org/

than using microservices on the computationally heavy service, due to the additional networking cost. The differences were however far lower on the service interacting with the database (about 80ms difference), where the microservices performed faster. They conclude that due to the granularity introduced by microservices, it makes it possible to specialize configuration of different services, and therefore potentially reduce cost but can also in some cases lead to faster response times. They acknowledge the additional complexity introduced by microservices, but also how issues that are typically dealt with on application level must be handled on a higher level, for example by the infrastructure. Their use-case, however, is rather limited in terms of complexity regarding amount of services and the collaboration between those.

The literature suggests various approaches for reducing latency between microservices. Richards [8] suggests to combine services to more coarse-grained services whenever a set of services has to communicate and this negatively impacts performance. Bass [11] suggests deploying the microservices in close approximation, either in same data-center or rack, or within the same virtual host/container. This ensures that the network round-trip-time (RTT) is low between the services that need to communicate.

2.3 REST limitations

Zimmermann poses the question regarding whether there are alternatives to using REST and web protocols within microservices [10]. One of the reasons REST with HTTP being frequently used between microservices is due to the ubiquity of the protocol; developers are often familiar with the protocol, and how the architectural style creates weaker coupling between services [10]. However, Ranney [29] argues that REST does not scale well for a system consisting of, for example, thousands of microservices, where issues arise regarding service contracts, documentation, and the lack of type-checking in JavaScript Object Notation (JSON). He does also argue that the usage of caching is not as relevant between services as it is in for example a browser. Although REST is not limited to using JSON as transport format, it is considered more common in context of REST due to its platform independence, lightweight nature, and readability [30, 31]. eXtensible Markup Language (XML) is another much used format, but is considered as more heavy-weight in terms of size and parsing time [32]. Other alternatives to JSON, such as binary JSON (BSON) [33] have emerged in order to address these issues.

Huang et al. [34] discuss some of the limitations of REST-style APIs in context of mobile devices. They make a set of observations regarding the limitations of REST. One of their observations is regarding dependency between API calls. If a call that would obtain the latest orders by a customer only accepts an ID of the customer, but the service consumer does only have the registered email, the service consumer would have to make an additional request in order to obtain the customer ID. Another identified limitation is the extra overhead caused by unused fields or attributes in the exchanged documents. When a set of different services rely on one service and the different services require a slightly different set of information, one approach is to generalize this into the same document being exchanged. They also observe that REST makes APIs less flexible for multi-image retrieval both due to the additional required requests and the limitation of image size/quality. This observation can be generalized in how REST limits clients from requesting multiple resources. They propose an API query language (AQL) in order to avoid the limitations by REST. However, their architecture introduces a central component between the service consumer and the service for request aggregation which makes it less suitable for microservices. The AQL does also introduce a higher coupling between client and server as the client is required to know more about the domain model considering attributes of the retrieved documents needs to be specified in the request.

2.4 Benefits and limitations of HTTP/2 and QUIC

HTTP is by many considered as the primary choice of protocol in combination with REST [1, 8]. HTTP/1.1 was developed in 1999 [35], and the last revision of the protocol was introduced with HTTP/2 in 2015 [36]. Compared with HTTP/1.1, HTTP/2 is a binary protocol that enables faster parsing of messages and more efficient bandwidth usage by having smaller requests in addition to header compression. It also introduces multiplexing which allows clients to send multiple requests through the same Transmission Control Protocol (TCP) connection before any response is received. HTTP/2 was based on the SPDY protocol, an experimental protocol developed by Google in order to obtain faster page load times [37]. Quick UDP Internet Connection QUIC [38] is also developed by Google, however, unlike HTTP, uses User Datagram Protocol (UDP) instead of TCP and acts as a transport protocol for HTTP/2.

Megyesi et al. [39] compares how SPDY, HTTP/1.1, and QUIC perform under simulated networking conditions with various RTTs, bandwidths, and packet loss. Their analysis involved running requests towards websites hosted officially by Google. In order to simulate networking conditions, a server intercepting the traffic was set up in order to simulate various types of networking conditions using NetEm [40], a tool for emulating networking latency. They found that QUIC performed poorly compared to the other protocols when downloading large object sizes. The protocol did however provide better performance compared to HTTP/1.1 and SPDY with high RTT and during high packet loss. Since SPDY and QUIC enables multiplexing, the protocols did also perform better regarding small object sizes. In the context of microservices, the exchanged objects are typically considered small. However, the main finding is that the latency has the biggest impact on which protocol should be used. Their findings shows that HTTP/1.1 performed best with large object sizes, high RTT and packet loss. This supports that SPDY and QUIC could be considered a reliable alternative to HTTP/1.1 in the context of microservices, considering the services communicate through a more or less reliable network. Although SPDY was used as a baseline for HTTP/2, there are various number of changes in the two protocols which may limit the amount of generalizations that can be made from this study towards HTTP/2. However, their study does nevertheless highlight how multiplexing impacts performance for small object-sizes, which is essential for microservice communication.

Carlucci et al. [41] found similar results. However, their setup used on-site servers and NetShaper[3] to simulate networking conditions. They also demonstrate how TCP and QUIC handles congestion and packet loss using the Forward Error Correction (FEC) in QUIC. Their results show that FEC has a negative impact on performance and TCP was able to provide better response times during heavy packet loss. QUIC did also perform worse on a high-speed link compared to the two other protocols due to how it deals with congestion window.

There are a few publications focusing on HTTP/2 performance in particular, but these focus on browser interaction with web-pages and not web APIs. Their results can nevertheless be generalized towards web APIs as the protocols are used together with REST. Sill [42] does for example argues that future API architectures would shift away from human-readable protocols towards machine-optimized protocols. This can be seen in how HTTP/2 is a binary protocol and is more optimized for machine interaction and not primarily browser-interaction. He also gives the example of gRPC[4], which is an RPC framework by Google that enables developers to declare interfaces in a language-agnostic language that can then be compiled to other languages. Although HTTP/2 offers newer features such as multiplexing that can mitigate some of the issues of the HTTP protocol identified by Huang et al. [34], it is an interesting area showing how multiplexing may affect congestion and response times. The additional header compression would lead to smaller requests/responses, thus reducing traffic between services. QUIC does additionally reduce the network interaction by avoiding the TCP handshake and the additional traffic caused by TCP ACK-ing. Performance impact of the newer HTTP protocols in context of RESTful APIs does, however, remain an unexplored area within the literature.

[3]http://netshaper.sourceforge.net/
[4]http://www.grpc.io/

2.5 How does CoAP compare to HTTP

The Constrained Application Protocol (CoAP) [43] emerged as an alternative to HTTP in the Internet of Things (IoT) field where resource-constrained devices are limited in terms of processing and energy capabilities [44]. HTTP is considered as too heavy for some of these devices, especially HTTP/2 which has also received criticism regarding its complexity and violation of the protocol layering principle [45]. Unlike HTTP/1.1, CoAP is a binary protocol running on UDP. The simplicity of the protocol does also restrict verbs to CRUD operations (GET, PUT, POST, DELETE).

Shi et al. [46] investigate using CoAP for communication between mobile and IoT devices where their backend is implemented as microservices. In their work, they perform performance tests on the CoAP protocol between a Raspberry Pi and a laptop through a wireless network and BLE 4.1 (Bluetooth Low Energy). Their tests are executed on an isolated private network in addition to a public network, that being a Starbucks network. Their results are however questionable due to external validity concerns in respect to of their network configuration. There is also no comparison with other protocols such as HTTP or raw UDP, which would provide a baseline for validation of the protocol performance. Their results show that the average round trip time caps at approximately 16ms for >73 concurrent clients, which may not be reasonable in a real world scenario where the round trip time should continue to increase beyond 16ms as more requests are created.

Kovatsch et al. [47] present a CoAP Java framework with additional improvements towards a multi-threaded architecture and perform performance comparisons against "state-of-the-art" HTTP servers. Their framework named Californium[5] is available under the Eclipse open source umbrella and targets devices that are not directly considered as resource constrained, thus prioritizing performance over resource usage. In their performance analysis, they compare sending a simple "Hello world" message over CoAP using Californium, compared with several other CoAP frameworks. They argue that there is a lack of a benchmarking tool for load-testing CoAP solutions, thus developing their own benchmarking tool named CoAPBench. ApacheBench[6] is used for benchmarking the HTTP servers. Both of these tools are distributed over several computers in order to achieve high concurrency load towards the tested server. Their results show that Californium provides about the same performance as HTTP/1.1 using the keep-alive flag, however, demonstrates better scalability for a high number of concurrent clients (>80 clients). Without the keep-alive flag, a new TCP connection has to be established for each HTTP request. Their results demonstrate a 33-60 times increase factor for this scenario.

[5]https://eclipse.org/californium
[6]https://httpd.apache.org/docs/2.4/programs/ab.html

It is argued that this is due to the extra overhead caused by the TCP protocol in addition to overhead from the additional headers added to each request, which CoAP avoids.

The work by Kovatsch et al. [47] demonstrates that CoAP may perform better than the HTTP protocol, but their experiments are however limited to a narrow use-case scenario with a small requests payload size. Additionally, although their design for testing the HTTP servers is distributed among several computers, ApacheBench uses a blocking, single-threaded design to send HTTP requests which may limit HTTP/1.1 from demonstrating its actual performance. Even though CoAPBench adheres to the same congestion control by waiting for the response for each request before sending next request, it relies on an entirely different threading model which may have an impact on the results. Their publication is nevertheless novel in comparing CoAP and HTTP on servers instead of resource-constrained devices.

Daniel et al. have explored the performance differences between SPDY, HTTP/1.1, and CoAP [48] which demonstrates that CoAP performs significantly better performance–wise compared to both SPDY and HTTP/1.1 in terms of download time and transferred bytes. They do also suggest improvements to SPDY and HTTP by reducing the amount of headers sent and using TCP Fast Open, which enables sending data during the TCP handshake. Their performance evaluation is done in various simulated networking environments using Netem [40]. Compared to the work by Kovatsch [47] which has a focus on performance in the context of concurrency, this study focuses on response times and identifying protocol overhead. However, this study is done in a more constrained environment with a bandwidth limit of 20Kbps and latency of 20ms. Due to the added network latency, CoAP performs significantly better due to avoiding the overhead from TCP. Additionally, this study demonstrates how the various protocols perform with different object sizes.

3 Limitations of REST in microservices

This chapter aims to address the research question regarding limitations of REST within microservices (R1) by identifying limitations of REST in the existing literature, then discussing their relevance and applicability in the microservice pattern. A general introduction to the history of distributed computing and REST is given before the results of the literature search is presented, followed by a discussion towards microservices and the findings.

3.1 Background

In the work "A note on Distributed Computing" [49] by Waldo et al., the authors discuss some of the central issues regarding the view of unified objects across a distributed system. These issues were specific to network latency, coordination of shared address space, handling of failures, quality of service, and concurrency. It was argued that the interfaces do not give a good enough indication that a certain method call would invoke a function on another machine. This could lead to large performance problems: a system with a large amount of remote procedure calls would suffer from network latency when the system started communicating across larger distances. Furthermore, a brief summary was given by the authors on the history of distributed computing with a new paradigm emerging approximately each decade: message-orientation in the 70s, procedure-orientation in the 80s, and object-orientation in the 90s.

REST was introduced by Fielding in 2000 [17]. Within REST, the idea is that parties exchange documents through resources through a unified interface. The architectural pattern is used to describe many of the components that allow the web to scale, where in the mid-90s the Internet faced scaling problems due to exponential growth. One of the problems at this time was the lack of protocols that would allow caching of information [50]. REST was developed together with HTTP 1.1 and is often used as an example how the web became successful through the use of caches and Uniform Resource Identifiers (URIs). There are six constraints in REST: Client-server, stateless, cache, uniform interface, layered system, and code on demand.

3.1.1 Client-server

The client (service consumer) is accessing services, provided by a server (service provider). By having a clear separation of concerns regarding the client and the server, it leads to simplification of the system and makes it easier scalable. For example, all the user-interface functionality can remain on the client, while all business-logic can be implemented on the server. These two can then be scaled and developed independently to a certain extent.

3.1.2 Stateless

All information required to process one request between the server and the client should be in one request only. Application state is maintained by the client and there should not be any context on the server that needs to be taken into account when processing the request.

3.1.3 Cache

It should be possible to express what information can be cached between the server and clients. This mitigates the problem of network latency as information can be cached on the client or on an intermediary (proxy) before reaching the server.

3.1.4 Uniform interface

An important feature of REST is that all clients are using the same interface on the server. Although the clients may use different libraries supporting the communication towards the server, the actual interface in order to exchange information remains the same. Fielding states that this constraint is "optimizing for the common case of the Web" [17].

In order for a interface to be uniform, Fielding describes four constraints:

Identification of resources

Each interaction towards the system happens towards a resource. A resource could be a web-site, an image, or a document listing customer orders. These are then accessed through a URI scheme where the scheme is based on entities and the naming of the scheme use nouns for the resource names.

Manipulation of resources through representations

When being accessed, each resources provides a representation of themselves (i.e. an HTML or JSON document). When a client wants to update a particular property of a resource, a representation of the updated resource is provided to the server. For example, to update the name of a particular user, a representation of that resource is sent to the serer with the updated name. Clients may however specify what format they want representations in, for example JSON or XML.

Self-descriptive messages

One single request should contain enough information to process the certain request. The semantics of the resource is apparent through the URI, and how it should be processed is expressed through the content-type of the representation.

Clients can also manipulate resources using a common set of actions on the resource, for example the verbs in HTTP; POST, GET, PUT, and DELETE for CRUD (create, read, update, delete) operations.

Hypermedia as the engine of application state (HATEOAS)

Much like how hypermedia plays a central role of how information is linked together, application state should be driven by the same principle. Each representation would then provide a set of URIs for valid state transitions that the client can make. The client should then rely on the URIs provided in the representations instead of relying on hard-coded URIs on the client itself. An example of this is how users are interacting with the web. The user can click on hyperlinks to perform a state transition in the browser. These links are specified in the obtained HTML documents or dynamic scripts from the web server and are not crafted by the browser itself.

3.1.5 Layered system

The layered system constraint can be compared to the client-server constraint, but takes the idea further by indicating each component (or layer) should only be concerned about its own responsibility and interact with other components through higher levels of abstraction that they provide. As an example, a client would interact towards the interface exposed by the server, and is not concerned about the implementation of what the client is trying to access, nor any intermediaries the request reaches before reaching the server.

3.1.6 Code on demand

Code on demand is an optional constraint in REST. The purpose of this constraint is that the server is able to provide *code* that will extend the functionality of the client. One example is how a browser may access a web-page that uses JavaScript for dynamically rendering the contents of a web-page. The HTML document of the website would reference a JavaScript file that would then instruct the client how to render the web page.

In the area of services, one service consumer may use code provided by another service to interpret or execute logic required to process a certain request. However, this may be in contradiction to the client-server and layered system constrain as responsibility leaks outside the service consumer and introduces higher coupling since the code provided by the server is tied to one particular language or platform.

3.1.7 Richardson Maturity Model

There has been much discussion regarding how *RESTful* a certain service is, and how to quantify the actual *RESTfulness* of a service. The Richardson Maturity Model (RMM) [51] is a model that identifies four levels of RESTfulness for services using

HTTP, each level is required before being applicable to the next one:

0. Applications use HTTP as a tunnelling protocol for remote interactions.
1. The application model is exposed as a set of resources.
2. Appropriate use of verbs, response types, codes.
3. Exchanged documents follow the HATEOAS constraint.

Level 1 is a typical level that can distinguish e.g. Simple Object Access Protocol (SOAP) services from REST services. A REST service may expose a user document on the URI `/users/1`, while an RPC-based service would accept requests on for example `/userService`. Level 2 indicates the service use the mechanisms provided by HTTP such as response codes to indicate what happened during the processing of the request, and response types to express how requests and responses should be interpreted. The last level indicates the service should be compliant with the HATEOAS constraint in REST.

3.1.8 RPC-Style and REST

After REST was introduced in 2000 [17], there was much discussion regarding how REST would be applicable to *enterprise systems*. In that area, services often used RPC-style APIs such as SOAP and other technology in the web service stack (WS-*). Many argued that REST was too immature to handle enterprise requirements such as reliable message transferring and ACID[1] transaction support [52]. In the work by Pautasso et al. [52], the strengths and weaknesses between RESTful HTTP APIs and WS-* APIs are discussed. The focus is on how WS-* and REST compares in terms of technology and on a conceptual level. The paper concludes that WS-* is more appropriate for enterprise applications that may require integration with heterogeneous systems and strict Quality of Service (QoS) requirements (i.e. transactions).

While REST is an architectural pattern, WS-* is a standardized technology stack. There is a lot of discussion comparing WS-* and REST where some describe the discussion as "biased and religious" [52], which lead to Fielding make a post to clarify some of the confusion around REST as an architectural pattern and concrete technologies [53].

3.1.9 Resource Oriented Architecture (ROA)

As REST is an architectural pattern, ROA is an architecture that follows RESTful principles, being described by Richardson et al. in 2007 [54]. ROA can be summarized that a system is modelled as resources (specified by a URI), where the resources are accessed through CRUD operations using the HTTP verbs POST, GET,

[1]Atomicity, Consistency, Isolation, Durability, a set of requirements often used in context of transactions.

PUT, DELETE as part of the uniform interface. These resources provides representations that are linked, following the HATEOAS constraint. Additionally, the same concept of statelessness is mentioned by Richardson et al.

3.1.10 The place of REST within microservices

REST is often considered as the go-to strategy when architecting distributed systems as developers are often familiar with the web technologies, even though there are RPC-frameworks being actively developed such as Apache Thrift[2] and gRPC[3] [4]. As stated earlier, microservices should communicate using light-weight protocols and encourage loose coupling. How REST is approached would greatly impact the coupling towards other services. For example, if a service consumer is integrating towards a service that is level 3 on the RMM, it would be higher coupled if all the URIs are hard-coded rather than embracing the capabilities of HATEOAS.

3.2 Methodology

This chapter contains a literature survey in order to answer the research question regarding limitations of REST within microservices. The keywords REST AND (limitations OR challenges) AND (web OR services OR architecture) were used across the databases ACM, IEEExplore, ScienceDirect, Springer Link, together with a high-level search in Google Scholar. The most relevant papers were selected and those papers who did not mention limitations or challenges of applying REST were not investigated. Relevant references were also taken into account while performing the literature survey. The papers were grouped according to the problems or limitations identified to REST in the papers.

3.3 Results

3.3.1 Limitations in REST constraints

In his dissertation describing REST [17], Fielding describes some drawbacks regarding the different architectural constraints. With the stateless constraint, repetitive data may be introduced in messages can result in decreased application performance since no shared context is stored on the server across requests. Additionally, he mentions this constraint relies on correct implementation across various clients of the semantics used by the system in order to achieve consistent application behavior across heterogeneous clients. One downside with the caching constraint is if data is updated frequently, it could introduce a delay between when clients are able to observe the updated data. The downsides he mentions regarding the uniform interface refer to how each client would receive the same representation regardless

[2]`https://thrift.apache.org/`
[3]`https://grpc.io/`

of its needs. In other words, it limits specialization of representations for different clients. This would also introduce extra overhead in the transferred messages. In the optional constraint code on demand, there may be trust issues regarding the provided code that clients would execute. Fielding also mentions that this can increase the complexity of the system.

In another work by Fielding et al. [55], a description of REST is given, similar to [17]. However, it is more focused on its relationship with the HTTP protocol. It is also mentioned that REST was developed for "large-grain data transfers rather than computation-intensive tasks" [55]. Under future work, the authors mention that REST could be extended to include QoS and continuous data streams (e.g. video streaming). However, the main focus of future development of REST would be in support of newer versions of the HTTP protocol.

3.3.2 The one "true" REST API

There has been much discussion regarding when an API should be considered RESTful or not, or *how* RESTful it is. Fielding argues that in order for an API to be considered RESTful, they must be hypertext-driven [56]. Failing to do so would result in higher coupling between client and server, and would be similar to how the coupling is in RPC-style APIs where the client are more tied due to coupling either through service descriptions or hard-coded URIs. An example can be how users typically interact with websites. When accessing a website, a user would typically enter the domain name of the website, then follow links to discover new information on other pages and possibly other websites. The user could bookmark webpages within that particular website to make it easier to find information. However, doing so would result in higher coupling as if the URI scheme changes and the old URL does not redirect to the new URL, the bookmark would result in a "not found" page.

Davis [57] argues many practitioners of RESTful implementations fails to follow several of the REST constraints in addition to a general lack of understanding of them. He uses the analogy of how the web works in order to describe how REST could be implemented and compare it against implementations that violates REST principles. In terms of the adressabillity of resources, he argues that many implementation of search functionalities rely on the use of POST, e.g. to search for available products on a website. These search implementations keep the search criteria inside the request body instead of the resource URI, thus breaking bookmarkability and caching of the search results. Another discussed challenge is related to uniform interface is where clients may submit partial representations of a resource through the HTTP PUT verb, making it possible for two clients to submit partial representation of a resource where the result could be an inconsistent state of the resource.

Davis does also mention the issue related to how HTTP verbs are supported in various frameworks, and how firewalls and proxies may sometimes prevent those requests from being processed as they do not recognize the HTTP verb being used. A common workaround is to instead rely on an HTTP POST request which has the HTTP verb specified in a header, e.g. `x-HTTP-method-override`.

Another issue being discussed by Davis is the question related to whether wrapper frameworks should be developed in parallel with the REST API itself. It is argued that the creation of wrapper APIs often lead to deficiencies in the interface itself due to higher coupling between the wrapper framework and the REST interface. Another argument is that the wrapper frameworks only acts as object serializers and promotes RPC-style interaction where the REST interface is hidden. Davis does also highlight the problem of processing semantics in relation to the media type. He use the example of Atom, where the media type both expresses the actual type of the transmitted representation together with how it should be processed. This leads to service consumers having to explicitly know how links should be processed, thus leading to higher coupling between the service and the service consumer. Furthermore, he brings up the issue related to HATEOAS and argue few implementations follows this constraint. He brings up the argument that hyperlinks are an essential part of how the web works. Users do for the most part not craft links when surfing the web, but follow hyperlinks to make transitions between web pages. The same way should APIs interact under HATEOAS; URIs should not be hard-coded in the service consumer but be crafted based on possible state transitions from the representations provided by the service. Failing to do so leads to a higher coupling between the service and service consumer as the URIs needs to be hard-coded.

Davis concludes that many REST services fails to follow many of the constraints defined by Fielding [17] and that many of the developers lack a general understanding of these principles.

Fernandez et al. [58] build a model based on the work by Fielding [17] in order to gain a better understanding and make it easier to extend the architectural pattern. They found that scalability, simplicity, and extensibility are central properties within the pattern. They also argue that security is an aspect that is not described by REST, but will be handled by implementation details of the pattern, for example using HTTPS as the transport layer. Another property being discussed is regarding HATEOAS. They argue REST was designed for human interactions, and not machines. Creating a service that would conform with the HATEOAS constrain would lead to providing semantic hypermedia, which they argue would increase the complexity of the service implementation.

Vinoski [59] discuss how REST encourages low coupling through uniform interface. He does also mention some of the challenges of using REST opposed to an RPC-style API. The payloads may be larger, as the interfaces should be generalized due to the uniform interface constraint. RPC does on the other hand allow developers to specialize the interfaces, effectively reducing the payload size at the cost of higher coupling. Vinoski does also bring up an issue related to limitations in regards to MIME-types (Multipurpose Internet Mail Extensions) when using HTTP as the transport protocol. MIME-types are used in the `Accept` and `Content-Type` headers to express what formats the client can understand and to specify what format is being transmitted. These are limited to a certain set of MIME-types specified by Internet Assigned Numbers Authority (IANA) [60]. It is possible to propose new MIME-types to IANA, or even implement these explicitly. However, doing so would require all service consumers to support the proposed MIME-type, which in some cases may not be possible either due to political or technical reasons. Another limitation of MIME-types are composite types where a representation may relate to multiple content types, for example an XML-wrapped JSON document. He concludes that RPC-style APIs attempts to extend programming paradigms for the sake of simplicity into distributed computing at the cost of higher coupling, maintainability, extensibility, and scalability.

In another article, Vinoski [61] continues the discussion regarding RPC and distributed computing. He also discusses the work by Waldo et al. [62] which refers to how distributed computing impose an entirely different requirement on function calls compared to having those calls in the same process or module. The goal of many RPC-oriented frameworks is to make the invocation of a function on a remote location as simple as calling a function. The function will serialize the required arguments, and invoke the remote function. When the remote function is invoked, the call stack may also include the frames from the original caller, making it possible for exceptions to be caught on the client. The main problem being discussed by Waldo and Vinoski is that this can make the developer writing code less aware that a specific function call leads to a remote invocation. Costs related to network latency, handling of shared memory, concurrency, and partial failures are large problems associated with RPC-style approaches. Vinoski does state that while developers are leaning more towards REST-style approaches, many implementations are closely related to RPC and distributed objects since many ignore the hypermedia constraint in REST.

3.3.3 Transactions

In the work by Mihindukulasooriya et al. [63], some of the major challenges related to REST and transactions are discussed. Their paper lists patterns that support transactional support within REST, however they conclude many of these patterns contradict properties within REST. One of the major challenges is how transactions impose the notion of state throughout a system, which is a violation of the stateless constraint in REST. In a decentralized system, the particular services involved in a transaction need a mechanism for coordination, agreement, and failure-recovery. Some implementations attempt to solve the notion of state through the creation of temporary resources. It is argued these resources are expressing application state and not resource state, and is therefore not entirely RESTful. Furthermore, the same data in a service may be exposed through different resources. This does further complicate how synchronization should be approached. In some models, the aspect of resource locking is used. They mention issues regarding resource locking would be related to deadlocks, clients not adhering to appropriate usage of locks, and availability of resources. The paper concludes that the major challenge would be to define a simple and efficient protocol that allows transactions while at the same time adhering to the constraints under REST.

Pautasso et al. propose a Try-Confirm/Cancel (TCC) pattern where transactions are implemented by the use of confirmation links to achieve atomicity across a transaction [64]. In their prototype, they take the example of booking of flights between two different airlines. After creating the booking, the resource would return a document containing a link for payment. Each booking would have a deadline of 24 hours for payment, and in this period the client can make a DELETE on the booking resource to delete it, or a PUT to confirm it with the appropriate confirmation details after having processed the payment. An example of a successful booking using the TCC pattern can be seen in Listing 3.1. The requests prefixed with 1 would involve seat reservations. Requests prefixed with 2 is the creation of the actual booking, while request 3 is confirming the booking. Their example is, however, limited in that it does not show how rollbacks should be handled e.g. if a user does no longer have any money left in their account between 3a and 3b and is therefore unable to confirm the payment of the second flight. One approach would be to confirm to the booking resource after the payment for both flights have been performed. It could be considered unreasonable for a flight company to always roll back payments on DELETE of a confirmed booking. A more appropriate solution could be that a payment would return a reference number or code that can be used as a confirmation token to the booking resource.

```
1  GET swiss.com/flight/LX101/seat -> HTTP 200 - OK
1  GET easyjet.com/flight/EZ999/seat -> HTTP 200 - OK
2  POST swiss.com/booking -> HTTP 302 - FOUND,
                             Location: /booking/A
2  POST easyjet.com/booking -> HTTP 302 - FOUND,
                               Location: /booking/B
2  GET swiss.com/booking/A -> HTTP 200 - OK,
                              Confirm URI: /payment/A
2   GET easyjet.com/booking/B -> HTTP 200 - OK,
                                Confirm URI: /payment/B
3a  PUT swiss.com/payment/A -> HTTP 200 - OK
3b  PUT easyjet.com/payment/B -> HTTP 200 - OK
```

Listing 3.1: A list of the involved requests and responses involved while performing a booking between two airlines using the TCC pattern proposed by Paytasso et al. [64]

da Silva Maciel et al. [65] propose a transaction model for REST named *Optimistic Concurrency Control* in order to solve the lost update problem in which two clients overwrite each others changes. It uses a version number on representations in order to validate whether the resource state has changed between calls. These version numbers are attached to the representations. A service can then reject a request to update a resource if the version number is not equal to the one that the service is holding. For each modification to the resource, the version number is incremented. The case where a request between two parties is denied due to conflicting state is shown in Figure 3. In this example, Alice and Bob want to make sure that some resource has the value "foo", then update it accordingly. Alice wants to update the resource to "bar", while Bob wants to update it to "don". Alice does first make a request, and sees that the resource returns a representation with the content "foo" which is version 1. Bob is very eager to make the update and makes another request and receives the same response as Alice. Before Bob manages to make the update, Alice makes a request to update the resource to "bar" together with the current version number of the resource. The resource accepts the update, and returns an OK to Alice. Bob, unaware of Alice having updated the resource thinks the resource is still at version 1 and makes a request to update the version 1 resource to "don". The resource sees that Bob is not aware of the update being done by Alice due to the old version number, and rejects the update request with a CONFLICT. This pattern is considered optimistic since it assumes that conflicts rarely occur.

3.3.4 Service discovery

One of the discussed differences regarding WS-* compared to REST by Pautasso et al. [52] is that REST does not describe how service discovery should be implemented. In the work by Shang et al. [66], the various challenges of IoT devices using TCP/IP are discussed. In particular REST together with HTTP is discussed. IoT devices often communicate through a RESTful API using either HTTP or CoAP. These are often typically secured using Transport Layer Security (TLS) and Datagram TLS (DTLS). They argue the encryption together with overhead from using REST introduces extra overhead. In a dynamic environment, the proxies may often

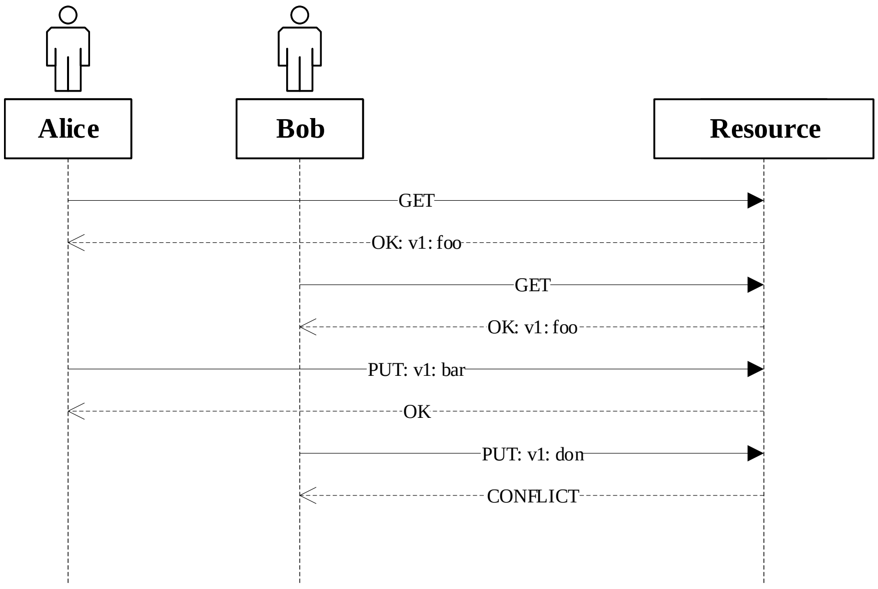


Figure 3: How a conflicting state is handled in the Optimistic Concurrency Control pattern.

not be able to cache a significant enough amount of requests in order for the clients to benefit from it. Furthermore, the stack for supporting HTTP can be considered too much from too resource-constrained devices. One approach for service discovery being discussed is Domain Name System Service Discovery (DNS-SD) which enables discovering other devices on a local network, such as printers and other computers. This approach is criticized for not taking into account that IoT devices are often considered resource-constrained and that they lack the necessary infrastructure required for supporting DNS-SD. An alternative to DNS-SD being discussed is Constrained RESTful Resource Discovery (CoRE-RD) [67]. CoRE-RD has a different approach to service discovery compared to DNS-SD by discovering resources and their capabilities instead of actual services. Furthermore, it is implemented on top of the CoAP protocol.

Stanik et al. propose a messaging protocol using Extensible Messaging and Presence Protocol (XMPP) [68] for inter-cloud communication [69]. The purpose of using XMPP instead of HTTP is to avoid some of the limitations by using HTML with REST, being the lack of service discovery and a limit of verbs that can be used on resources. Their protocol defines two XML documents: One for interacting with resources, and one for describing the capabilities of resources using XMPP Web Application Description Language (XWADL). XWADL makes it possible to generate code stubs for remote interaction as it provides a description of the resources. Interactions with resources happen through XML documents (XML-REST schema) that wraps the actual representations. An example of this can be seen in Listing 3.2.

```
<iq type="result"
    from="company-a.com/openstack"
    to="requester@company-b.com/rest-client"
    id="rest2">
  <resource xmlns="urn:xmpp:xml-rest" path="/compute">
    <method type="POST">
      <response mediaType="text/uri">
        <representation>
            xmpp://company-a.com/openstack#/vm1
        </representation>
      </response>
    </method>
  </resource>
</iq>
```

Listing 3.2: An example of a resource representation in XML-REST by Stanik et al. as a result of creating a virtual machine in OpenStack through a POST request [69].

One of the important differences between their proposal [69] and REST with HTTP is how resources accept an arbitrary number of *actions* to be performed on them together with a method. While the methods in their implementation is limited to CRUD operations, the particular action allows fine-grained operation invocation on resources. The authors state that this enables RPC-style invocation of methods similar to SOAP and XML-RPC.

3.3.5 Extensions of REST

In the work by Khare et al. [70, 71], several patterns are proposed to deal with various issues regarding REST in a distributed system. The biggest problem being discussed is how latency becomes a problem in distributed problems. Considering a system running on one local machine, although there may be forms of thread synchronization mechanisms to deal with concurrency issues, business logic may rely on direct memory to access values in the system. This can become challenging when a system relying on low response times becomes distributed, as network latency can make processes take much longer time. One example being discussed is how the observed value of resources change over time, e.g. stock prices. The issue of latency complicates how simultaneous agreement where all parties need to agree on a certain criteria is performed between different services. The authors do also mention one limitation of REST is messaging is limited to a synchronous request-response protocol. One particular problem being discussed is the "lost update" problem, where if two clients would write their changes to a resource simultaneously, the changes of one client would overwrite the other client resulting in lost data.

In order to mitigate these limitations within a distributed system, additional patterns have been proposed to extend REST by Khare:

A+REST (Events)

The A+REST (Asynchronous REST) pattern can be described with the publish-subscribe pattern under Section 2.2. A client would "*watch*" for updates on a specific resource, then the resource would notify the client when the update occurs. This solves the problem where a client has to poll at a certain interval to synchronize its understanding of a resource.

R+REST (Message routing)

The R+REST pattern attempts to solve the issue where one request needs to reach multiple services before reaching a final service. An example given by Khare in [71] is how a printer service may require a printing request to reach multiple other services before reaching the final printer service. When printing, a user may want to attach a cryptographic watermark on the printed papers. After being printed, the amount of pages needs to be reported to a payment accounting service for billing the user about how many pages have been printed. Typically, each of these steps are presented as intermediary servers that acts as proxy servers before reaching the printing service. The message routing pattern eliminates some of the response links, reducing the amount of roundtrips each message must make in order to perform a print operation. Instead of waiting for a response from the next service to perform its operation, the service would route the request to the next service. The next service then sends the response back directly to the client making the request. When the printer service is done printing, it can notify the payment accounting service about the amount of printed pages (which then in turn notifies the user) instead of sending responses through a nested chain of intermediary services that would wait for a response. This can be compared to how asynchronous messaging is performed with microservices where a service-provider may notify another service other than the service consumer when a certain request is processed.

REST+D (Delegation)

The delegation pattern attempts to deal with the "lost update problem" mentioned earlier by introducing an intermediary synchronization service (mutex lock object) that encapsulates an origin service. The synchronization service guarantees mutually exclusive access to the encapsulating service by maintaining a registry of which clients are accessing which resource. The resource can be considered as locked once a request goes through it until it is instructed to be unlocked, or until it times out.

REST+E (Estimation)

REST+E describes how REST *estimates* current values using caches or lower level protocol mechanisms. A value is considered to be the current value when it is present in a cache until the cache is invalidated.

3.3.6 Linked documents

Liskin et al. [72] propose a system for adding hyperlinks to resources using a proxy, enabling existing level 2 systems to reach level 3 on the RMM scale. They argue many practitioners often do not implement HATEOAS in their system as it involves duplicating business logic and URI schemes. Their proxy server would rely on a modeling language that describes which state transitions the representations would be in. Their proposal is however limited to XML documents and relies on a heavy use of XPath2[4] for parsing the transmitted documents. In their conclusion, they reiterate how HATEOAS results in weaker coupling as clients do not need to know the entire domain model of the system. In addition, clients become more robust for changes in URI schemes.

With RESTler, Alarcon et al. [73] presents a crawler for REST services. The crawler would rely on a metamodel named Resource Linking Language (ReLL) that describes how the different representations provided by a resource are linked together. RESTler would then produce a typed graph of the various representations it obtained and their relations. As an example, they traverse the webpage of UC Bakerly and parts of the Twitter API. They conclude by stating their solution is limited in terms of acting as a service descriptor by means of not being able to express context of use of the resources and authentication.

3.4 Authorization and authentication

The work by Zou et al. [74] discusses the relationship between REST and Service Level Agreement (SLA) in terms of accountability. They argue REST was not designed to fit enterprise requirements such as who should be accountable when a SLA is breached, which is crucial in cloud environments. In their work, they propose Accountable State Transfer (AST), which is an architecture to solve the issues REST has in relation to accountability. In AST, service contracts are identified by an URI which is referenced in metadata fields in requests/responses. They define a service contract as follows: "an electronic representation of a traditional contract

[4]https://www.w3.org/TR/xpath20/

that captures the essential contractual information including involved parties, domain specific terms, obligations for each party, contract execution states and rules that determine those states" [74]. Furthermore, they argue that the major challenge of REST is knowing which contract relates to a service in addition to tracking progress of contract execution and breaches. The AST introduces two new architectural components: The contract manager and a contract monitor. The contract monitor is responsible for monitoring the interaction between service and service-consumers, and communicates with the contract manager which determines the state of a contract execution. While AST is proposed as an extension of REST, it provides a framework for managing contracts between services and service-consumers.

RestACL by Huffmeyer et al. propose an Access Control Language (ACL) for REST services [75]. RestACL provides a solution for dealing with access control in REST, as according to the authors, REST does not specify how this should be handled. Their solution use Attribute Based Access Control (ABAC), which allows rules to be based upon attributes of objects. Their implementation is using JSON for expressing the ACL rules and consists of domains, templates, parameters, and policies. The domain is used for expressing the resource model being used in the REST services. This enables associating policies with each resource. Parameters enables providing policies for resource attributes in a similar manner, and templates provides a generalization of access rules where multiple resources may use the same access rules.

Field et al. [76] proposes a framework for obligation fulfillment. An obligation is defined as non-functional or a cross-cutting system requirement. One example being used is handling of electronic health records. When a doctor is accessing the records of a certain patient, said patient should receive a notification about who and when accessed their health records through a secure channel. Those obligations can be imposed by policies or requirements that are introduced after the system has started operating. Their work describes how this can be implemented within a REST architecture without coupling the necessary logic of obligation fulfillment together with business logic in addition to a proposal to extend the security module in the Java Spring framework to support their approach. In their implementation, the various obligations are modelled as resources available through a REST API. These resources can then be referenced in metadata fields and configuration files. In their implementation for the Spring framework, the configuration files can reference the obligations during configuration time of the application. Filters in the application will intercept requests before they get processed can then act appropriately depending on the referenced obligation.

WS-Agreement is a specification that enables two parties to agree on a SLA in context of service consumption. The work by Kubert et al. [77] explores how the

WS-Agreement standard can be implemented together with REST. WS-Agreement is also relying on other WS-* standards that relates with SOAP. Their work aims to re-implement the WS-Agreement and make adjustments to the standard such that it is more appropriate in context of REST. They found that it is not possible to directly port the standard directly to REST considering the standard as-is leads to an RPC-style implementations that use technologies such as SOAP for communication. In their implementation, they implement the various concepts of the specification as resources (Templates, agreements, and their associated states). The authors have a discussion regarding the implementation of application state. They argue having application state on the client is "kind of misleading" [77] considering how resources can also represent state. The discussion seems rather confusing considering they bring up the argument regarding whether clients should maintain the resource state and specify these in the requests. Given the example where a browser is requesting a web-page from a web server in a RESTful fashion, the web server does not care about the processing of the request in context of the previous requests. The client may, however, keep a list of the previous URIs visited and their representations for caching in order to prevent requesting them again. The actual web-page itself being requested may however, be changed over time. If it is a news-page, it may change multiple time in an hour. The client will then capture the state of the resource through a representation at the moment in time when a request is made.

3.5 Discussion and future work

3.5.1 Discussion

An architectural pattern can be described by a set of constraints. These constraints are then applied to a system in order to have a certain (positively) desired outcome. In a presentation by Fielding on the REST architecture [50] he reiterates that the constraints in an architectural pattern is not mandatory but only used to achieve the desired outcome that the particular constraint would impose on the system. It is therefore important to question whether the certain constraint would be in violation of other parts or requirements of the system and how these problems could be solved.

Compared with RPC-style APIs, REST can provide a weaker coupling between service-consumer and the service provider. However, one challenge is how much of an impact HATEOAS would have on coupling between clients. It is argued that HATEOAS may become challenging to implement due to duplication of business and URI schemes [72]. However, there are standards that specify how this can be achieved through for example HAL (Hypertext Application Language) [78].

Some of the surveyed papers argue the lack of service discovery is a limitation of REST. It might be arguable that HATEOAS and the uniform interface constraint does however provide mechanisms that can partially be used for service discovery. In HTTP, it is possible to access resource metadata through the HEAD verb, and which HTTP verbs the resource can accept using the OPTIONS verb [54] (Uniform interface). Which state transitions can be made from one resource can be expressed in the representation (HATEOAS) [56]. Clients who have a hard-coded URI scheme would have a higher coupling in addition to a certain understanding of the domain model that the API exposes. Those clients are required to rely on some documentation (either before run-time, or as a list of URIs loaded at run-time) that would describe how the service-consumer would interact with the service-provider. Another challenge regarding HATEOAS would be the coupling on the exchanged documents themselves and how much domain-specific knowledge should remain in the service-consumer as discussed by Vinoski [59].

Perhaps the most significant limitation of REST for microservices would be how REST limits specialization of interfaces in relation to performance. The uniform resource constraints states: "The REST interface is designed to be efficient for large grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction." [17]. This would prevent specialization of the interface where for example specific type of clients would receive trimmed-down documents to prevent additional data transfer overhead. Another question would be whether microservices should be optimized for common usage, or specialize themselves towards other services. Having specialized services would lead to higher coupling and reduce the flexibility and ability to replace a certain service.

Considering microservices as a way of approaching SOA, many of the problems of applying REST in a microservice system would be inherent from previous discussions where REST was compared with RPC-style APIs such as the WS-* stack. Even though many of the discussions were considered as heated and that they were comparing two conceptually different entities (a standardized implementation to an architectural style) [53, 12], many of the challenges remain important from that discussion. Issues regarding QoS and transactions are still relevant when considering a REST architecture.

In the microservice architecture, REST would be limited to the synchronous one-to-one ISC pattern. The additional patterns proposed by Khare [70, 71] describe how RESTful interaction can be used in synchronous in addition to one-to-many communication. These are patterns similar to the ones described under Section 2.2. Furthermore, the message routing pattern can be compared to how microservices would use message queuing for asynchronous message passing. Implementing asynchronous calls does on the other hand introduce an implicit state between requests in terms of the request-response lifecycle.

Transactions remain a challenge in RESTful systems, and is perhaps even more of an issue in a microservice architecture due to how services should have low coupling between them. Many of the patterns attempting to implement transactions in a RESTful fashion often lean towards the creation of temporarily resources, which is argued should not be considered RESTful as the resources themselves represent application state in order to not be in violation of the stateful constraint [63].

In context of microservices, transactions could be considered an anti-pattern due to how they lead to high coupling between services. Instead, it might be worth considering to combine services that are required in a transactions. However, if this is not possible, other patterns such as eventual consistency is a much considered alternative to ACID transactions [4].

There were multiple issues being discussed in relation to using HTTP as the transport protocol. Although none of the mentioned papers specified any particular HTTP version, most of them mentioned limitations from max URI lengths, extensibility of MIME and content types, and lack of support of various HTTP verbs in proxies and firewalls. It would be questionable whether some of these limitations could be considered anti-patterns. Having a HTTP header that is large enough it is rejected by the server could be a symptom that there is too much meta-data (headers) or the URI is too long, and some of this information should be moved to the HTTP body instead. This is on the other hand not possible with certain verbs such as GET, which does not allow a HTTP body. The issue regarding extensibility of MIME and content types may only be related to specific use-cases. Microservices should rely on a commonly used (and understood) exchange formats, such as JSON or XML. Introducing entirely new MIME or content types would lead to significantly higher coupling between client and server. It may also require specialized implementations of parsers for those types in addition to implementing the MIME and content-types themselves.

Others have proposed to use alternative protocols such as the XMPP protocol [69] by Stanik et al. The goal of their proposal was to solve limitations introduced by the HTTP protocol in regards to limitations of HTTP verbs and lack of service discovery. Their proposal does however enable developers to implement RPC-style APIs in the form of action elements in the requests that can be performed on the target resource. Although their proposal involves using the GET, PUT, POST, and DELETE verbs for CRUD operations in order to comply with the uniform interface constraints, the fact they are introducing an arbitrary set of operations through actions is a violation of the uniform interface constraint. This approach can be compared with how functions are executed on distributed objects in for example SOAP RPC-style APIs, and is heavily criticized by Fielding in [53].

While there are multiple proposals for how to implement HATEOAS, the main challenge when applying it is providing a simple and efficient framework for developers to implement the services. As stated by Davis [57] and Fielding [56], the lack of HATEOAS in REST services limits clients from acting as thin clients, leading to knowledge of the domain model being implemented on clients through the expression of hard-coded URIs. Davis further argues that one of the reasons for this may be that REST frameworks does little to promote the HATEOAS constraint in particular.

As stated earlier, microservices that embrace HATEOAS lead to a lower coupled system. It makes it simpler to change the URI scheme in services and reduces the amount of domain logic being implemented in the service consumer. It does also lead to a higher chain of responsibility due to how the service consumers sees which state transitions it can make through hypertext. This embraces the principle that each microservice should be a self-containing service with low coupling towards other services. Davis makes the example of a web without hypertext [57], making it much more difficult for users to browse the web as they are required to manually craft the URI for each website. This can also be said for other media that a web page may require, such as style sheets, scripts, images, and videos. Instead of using a browser, alternatively, the user would have to use applications that contain domain-specific knowledge for each web area that they would want to explore. In other words, hypertext is fundamental to how users browse the web: The browser does not contain any domain-specific logic. Instead, it makes requests on behalf of the user and displays the available state transitions that can be made by the use of hypertext.

Services using REST using inter-service communication are on the other hand required to be aware of a certain level of the domain model being exposed by a service in order to act appropriately. Even though the URIs are crafted based on hypertext from responses, the service consumer does still require an entry-point to make the first request in order to see which state transitions are available from a service. The service consumer would still be bound to the service contract in terms of the document structure and contextual use of the calls to the service.

3.5.2 Future work

Following the HATEOAS discussion, one field of research that was not investigated was how frameworks could further embrace HATEOAS in application design. Davis [57] argues not only the lack of understandability but also the support for HATEOAS in REST frameworks is a major factor keeping back developers from embracing HATEOAS in their applications. This is further supported by Liskin [72] who argue developers would have to duplicate URI schemes and business logic in order to expose state transitions. Websites is a much used example of an implementation of HATEOAS. One research topic would be investigating which and how programming paradigms and patterns can be taken from the creation of dynamic webpages and applied to the expression of hypermedia in REST APIs.

Another question in terms of HATEOAS is how many state transitions should be expressed in each document. It may also not be known at implementation-time which URIs a service consumer may request. One of the goals of SOA is to provide new business capabilities through the sum of other services [4]. Relying on HATEOAS to forge URIs could in some cases lead to additional requests in order to obtain a link for the intended state transition. Having too many links in the representation would lead to larger representations, while too few would lead to multiple requests being involved to discover the appropriate state transition.

4 Protocol Comparison in Acme Air

This chapter addresses the research question regarding benefits and limitations of HTTP/2 as an alternative to HTTP/1.1 for ISC (R2), whether CoAP can be considered as an alternative protocol to HTTP for ISC (R3), and investigates approaches for optimizing existing implementations (R4). This is done by using the CoAP and HTTP/2 protocol in the Acme Air benchmarking system [79] and investigating their behavior under different latencies. The latencies would simulate the distance between two microservices.

4.1 Background

The work by Ueda et al. [79] investigates an open-source bench-marking tool for analyzing the behavior of web services. They argue this is the first study being done on microservice performance in particular. This is done together with a monolithic reference implementation to see the impact of moving a system to a microservice architecture. In their findings, they found that their microservice implementation performed significantly slower in terms of response time (79.2%) than the monolithic implementation. Furthermore, they compare the microservices implemented in Java and NodeJS. Additionally, the authors investigate the performance on both hardware (cycles per instruction and code path lengths) and software level (time being spent in modules/layers). The authors also seek to investigate the impact of container virtualizaton by testing different Docker networking configurations and comparing running the services on the host against running the services inside Docker containers.

Their work is based on AcmeAir, which is a fictitious airline website for benchmarking web services. This has then been extended to microservices by splitting up the service into separate services. Their Java implementation is using IBM Websphere Liberty as an application container where the REST API is implemented using JAX-RS, a Java Enterprise framework for building REST APIs. The NodeJS implementation uses the Express framework for implementing the REST API. The AcmeAir website is a static website that interacts with a REST API using JavaScript for dynamic loading of information. There are six services involved in AcmeAir: main-service, auth-service, booking-service, customer-service, flight-service, and support-service. These services are encapsulated by one common nginx server which acts as the API-gateway in AcmeAir.

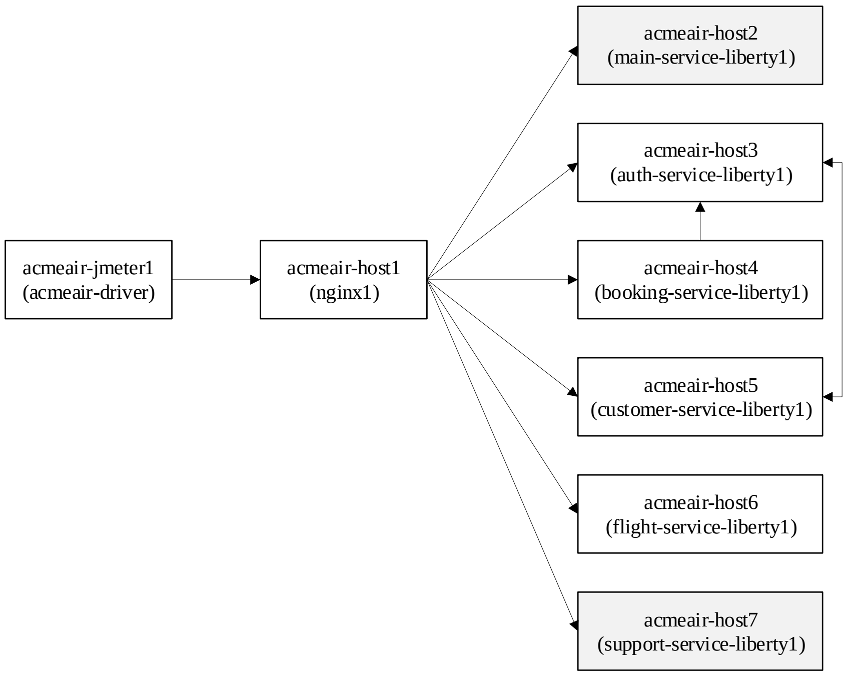


Figure 4: Overview of the involved hosts/services in the experimental setup. The host with gray background are not covered by any of the test cases in the acmeair-driver.

Requests towards the AcmeAir system are made using the *acmeair-driver* which is a set of test cases configured in Apache JMeter. These test cases simulate a work-flow throughout the system, but target only certain services through the exposed API and not the website of the system. The different services involved in the test cases are represented as the white services highlighted in Figure 4. Typically, the test cases requires a signed-in user. The different test cases can be listed as follows:

- **Login**
  `POST /auth/acmeair-as/rest/api/login`
  Logs in a user.
- **Query Flight**
  `POST /flight/acmeair-fs/rest/api/flights/queryflights`
  List available flights from airport X to airport Y at a certain date.
- **List Bookings**
  `GET /booking/acmeair-bs/rest/api/bookings/byuser/<email>:`

List the different flight bookings a user identified by the email has registered.

- **Logout**
  `GET /auth/acmeair-as/rest/api/login/logout:`
  Logs out the user.
- **Book Flight**
  `POST /booking/acmeair-bs/rest/api/bookings/bookflights:`
  Books a flight for a user.
- **View Profile Information**
  `GET /customer/acmeair-cs/rest/api/customer/byid/<email>:`
  Views profile information for a user.
- **Update Customer**
  `POST /customer/acmeair-cs/rest/api/customer/byid/<email>:`
  Updates information about a certain user.
- **Cancel Booking**
  `POST /booking/acmeair-bs/rest/api/bookings/cancelbooking:`
  Cancels an existing booking.

The different endpoints that each test targets differ in both content type and the length of the response. The login endpoint accepts login credentials in the content type `application/x-www-form-urlencoded`, while logout does not accept any input data. They both produce a response in `text/plain` with the content "logged in" or "logged out". The `queryflights` resource can produce longer JSON documents (content type `application/json`) which can be multiple kilobytes long.

These tests are declared in a JMX script together with internal components in Java for JMeter to build and parse requests. The JMX script contains parameters for the various requests, expressing how the different tests should be executed, and how the result of the tests should be processed after their execution. When JMeter starts, it will use the JMX script together with the compiled Java classes containing handling of the retrieved responses and run for 10 minutes. During this period of time, there is a small ramp-up window in which 10 threads are spawned for sending/processing requests. The result of each request is written to a CSV file.

Cycles per instruction (CPI) is the average number of clock cycles per CPU instruction. It is determined using an in-house tool. A high CPI would indicate that more time is being spent per instruction. This may the case because of cache misses and other hardware-level bottlenecks. The code path length is also determined using an in-house tool and is the amount of CPU instructions required to process one HTTP request to the AcmeAir system. This metric was used to determine if the code path would change across different configurations in their benchmarking.

The AcmeAir system is deployed in Docker containers on an IBM z13 mainframe. The mainframe allows partitioning of the hardware using logical-partition mechanism (LPAR) that enables isolating running operating systems. The partition for running the acmeair-driver was allocated to 64 cores with 1TB of RAM, while the partition for hosting the AcmeAir system had 16 cores with 2TB RAM. Each partition ran SUSE Linux Enterprise Server 12 SP1.

In their results, they found that the overhead from the Docker bridge networking infrastructure had a significant impact on performance with a 33.8% hit on performance compared to processes running on the host. The code paths were on average about 3 times as long in the microservice system compared to the monolithic architecture. One major difference was observed between the NodeJS and Java process. The NodeJS microservices used a higher amount of time in the native parts of the runtime compared to Java, where the reason being that Java has more of its networking stack implemented in Java, whereas NodeJS rely more on native implementations in C++. Regarding CPI, the authors saw a trend of NodeJS having a smaller CPI compared to Java. Furthermore, the NodeJS implementation had a higher cache-miss rate in the microservice implementation compared to Java which was different from the monolithic implementation where Java had a higher miss rate.

In context of the microservice architecture, the work by Ueda et al. is rather limited. The microservices in AcmeAir communicate using synchronous API calls using HTTP/1.1 over a RESTful API. There is no asynchronous messaging through message queues or publish/subscribe patterns being used. Some of the microservices also interact with each other, for example service that require authentication rely on sending an HTTP call to the auth-service for each request they process. As argued earlier (Section 2.1.1), an alternative would be to duplicate information across services to avoid the communication overhead. Furthermore, in terms of the Richardson maturity model, the REST API provided in AcmeAir does not rely on HATEOAS, but use hard-coded URIs for the resources that are being used. Some of the REST endpoints are also leaning more towards RPC style that uses verbs for resource names (e.g. `cancelbooking`). The API does also use cookies for authentication, which has been criticized by Fielding for violating several constraints of REST [17]. The test cases in the acmeair-driver is also only limited to GET and POST requests.

While Acme Air does not focus on comparing protocols, it provides a reference and a base for analyzing microservices during runtime. Furthermore, how the different requests are structured in both size and content type makes it possible to see more in detail how different protocols scale across different payload sizes.

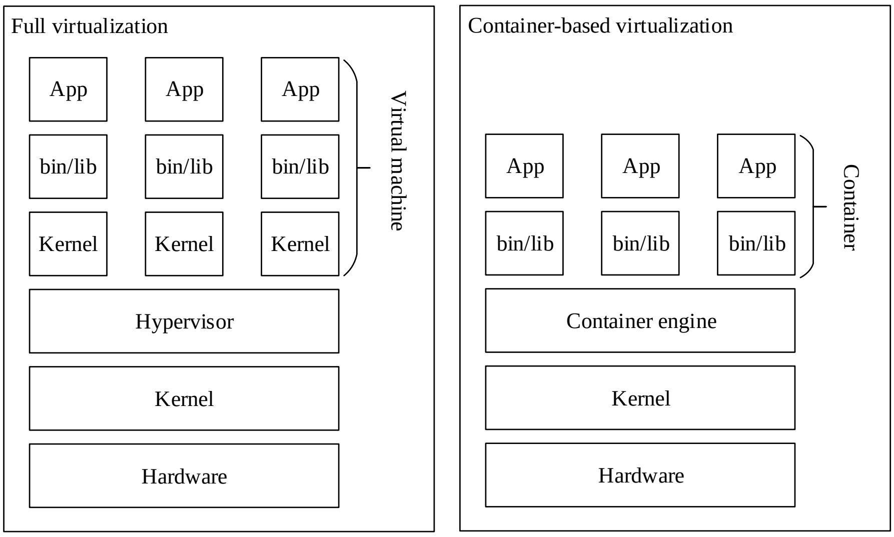


Figure 5: In container-based virtualization, instances shares the same kernel as the host, enabling more efficient resource usage at the expense of isolation compared to traditional virtualization.

4.1.1 Container virtualization

Docker is a container virtualization technology that enables encapsulations of applications from each other and lets these run in what they see as an isolated environment. It can be compared to virtual machines where Docker does not virtualize the kernel per Docker instance, which enables faster start-up times of containers and lower resource footprint compared to full virtualization (Figure 5). This does however come at the cost of isolation and security. A compromised virtual machine requires escaping the virtual machine and the hypervisor (the software managing virtual machines) in order to get access to the host kernel. Some methods of better securing Docker containers can be using control groups, SELinux and avoid running privileged/elevated containers [80]. Processes running in Docker containers will see their own isolated file system and will be able to access and execute processes available in that particular container. The containers are often stripped-down versions of a certain operating system as the containers should only contain what is required to execute the particular process. Docker has received much attention as it simplifies development and deployment of applications. Container virtualization is also limited to certain platforms. For example, Docker only supports Windows and Linux containers[1]. Running a Linux container on Windows requires a hypervisor (e.g. Hyper-V) that then would virtualize the shared Linux kernel since a Linux container cannot directly interact with the Windows kernel. [81]

In Docker, there are three central concepts: Volumes, images, and containers. Docker images are used as templates to spawn containers. Compared to how regular applications run, the image can be thought of as the binary files for a process, where the process itself is the container in Docker. The images are built from a Dockerfile, which is a text file declaring which image it should be based on, where applications should be stored within the image, and what happens when the container starts. The images are built by using layers. This means that one image is the base image together with the changes introduced by a Dockerfile on top. This makes it possible to re-use layers across different images. If multiple images use the same base-image, they would just reference the layer instead of copying it for each image. Docker volumes are storage areas for Docker containers to persist data.

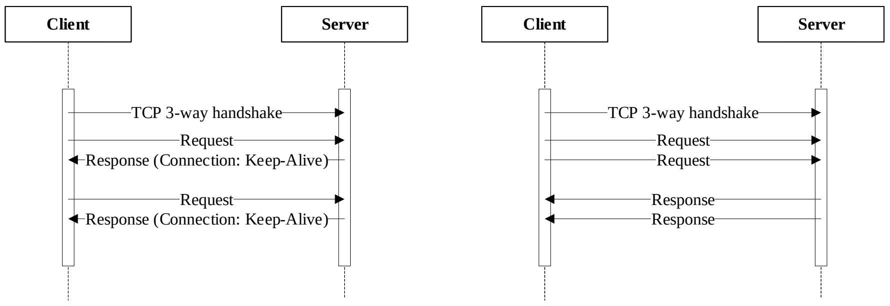
The Docker networking stack allows containers to communicate using two modes: bridge or host networking. With the bridge mode, the containers will communicate with a virtualized networking adapter compared to host networking where the container will use the same networking stack as the host. Bridge networking allows containers to listen on ports that would otherwise collide with other ports in other processes on the host or Docker networks.

In the work by Ueda et al. [79], the Docker infrastructure did not lead to any significant impact on the input/output (I/O) performance with the exception of bridge mode which degraded performance by 33.8% in throughput together with a higher number of cache misses. This is due to the extra overhead introduced by the virtualized networking interface. They argue that developers should carefully consider which networking mode is being used with Docker, and argue that the bridge mode should only be used if it is not possible to use host networking.

4.1.2 Protocols

In AcmeAir, the HTTP/1.1 protocol is used for inter-service communication [79]. Additionally, the keep-alive flag is used to reduce the amount of TCP handshakes across requests made within the system (see Figure 6a). In comparison, HTTP/2 has the keep-alive functionality built into the protocol and performs connection reuse by default [36]. HTTP/2 does also allow multiplexing which avoids the problem of ahead-of-time blocking in HTTP/1.1 (see Figure 6b). CoAP, on the other hand, does not have any initial hand shake as it is an UDP-based protocol [43].

[1] https://blog.docker.com/2016/09/dockerforws2016/



(a) How keep-alive is used in HTTP/1.1 to re-use TCP connections.

(b) How multiplexing also enables parallel requests in HTTP/2.

Figure 6: Comparison of how two requests may interact with a server in HTTP/1.1 and HTTP/2.

4.1.3 nginx

nginx (pronounced "engine X") is a web server that is also used as reverse-proxy in front of web applications. It has gained much attraction during the past decade for providing simple configuration and high performance [82]. nginx supports both HTTP/1.1 and HTTP/2 for front-end connections, however is limited to HTTP/1.1 on forwarded connections. In nginx, each request is processed by a single-threaded worker process. This worker process uses asynchronous non-blocking I/O to communicate. This solves the problem of slow connections affecting the processing of other connections and enables more efficient processing of requests under high load. [83]

4.1.4 nghttp2

nghttp2 is a C-library for the HTTP/2 protocol. The authors of the library has also implemented a web proxy utilizing the library named nghttpx[2]. One of the special properties of nghttpx is how it can operate as a translator between two protocols. It can, for example, accept incoming HTTP/1.1 requests and forward them to a HTTP/2 server using HTTP/2. nghttpx uses an event-driven, non-blocking I/O model similar to nginx. However, nghttpx is only a proxy and does not enable hosting of web services.

4.1.5 Californium

As mentioned in Chapter 2, Californium is a Java framework under the Eclipse open source umbrella for building applications using the CoAP protocol [47]. It

[2] https://nghttp2.org/documentation/nghttpx.1.html

uses a multi-threaded model with a blocking socket for I/O communication. One of the modules in Californium is Californium-Proxy which acts as a proxy that can translate messages between CoAP and HTTP. Californium is licensed under the EPL+EDL dual license. It was first introduced by Kovatsch et al. in 2012 [84] and has then later received attention regarding optimization such as the threading model described in [47].

4.2 Methodology

4.2.1 Infrastructure

The AcmeAir system is deployed on an OpenStack[3] private cloud. This involved 7 separate instances for hosting AcmeAir services, and one instance for the acmeair-driver. All of these hosts are instances running CentOS 7[4] which have 4GB RAM, 2 vCPUs, and 40GB of disk storage. Each of the components in AcmeAir are executed in Docker containers, each on their respective host. The different services in AcmeAir are for the most part Java applications with the exception of nginx1, which acts as the API gateway. The Java applications are hosted in a WebSphere Liberty application server.

In order to achieve a higher level of isolation between the services to simplify latency simulation, each service is deployed on a separate host with the exception of the database belonging to each service. Considering a more real-world scenario might have a database on a separate host optimized for database services, this experiment is investigating the impact of different protocols for inter-service communication under different latencies. Having the database on the same host as their belonging service makes it easier to see the impact of different protocols under different latencies as database interaction does not appear on the outgoing traffic from the host. Orchestration of these services was done using automated scripts that connect to each host and start the services. An overview of the different hosts together with their interaction can be seen in Figure 4.

4.2.2 Simulating networking latency

Latency between each service is simulated using the tc (Traffic Control) utility program in Linux[5]. The following command is used in order to add a latency of 5ms to the interface `eth0` for outgoing traffic:

```
# tc qdisc add dev eth0 root netem delay 5ms
```

This adds a traffic rule to the queuing discipline (qdisc) for the `eth0` interface. When applications are sending packets on the network, those packets are queued to the qdisc for that particular interface. The kernel will then attempt to forward

[3]https://www.openstack.org/
[4]https://www.centos.org/
[5]https://linux.die.net/man/8/tc

these packets from the qdisc to the network interface driver. In the command above, `root` will refer to the root of the networking interface since traffic class rules are not used for configuring the traffic.

The `netem` keyword in the command above is a networking emulation tool for the purpose of testing applications and protocols under different conditions such as latency and packet loss [40].

As the command only introduce delay on outgoing traffic, it is executed across all service hosts, including the acmeair-jmeter1 host to achieve simulated latency on both inbound and outbound traffic. As a result of this, introducing 5ms simulated latency then results in 10ms additional round trip time between two service hosts.

4.2.3 Experimental setup

The AcmeAir system is executed under 0ms, 2ms, and 5ms simulated latency with the protocols CoAP, HTTP/1.1 with keep-alive, and h2c (HTTP/2 over TCP without encryption). Each test run would involve a warm-up period of 2 minutes before sending requests for 10 minutes using one of the previously mentioned protocols being used. The reason for this warm-up period is to reduce the chance of classes and instances, since this would require lazy-loading or JITing (Just in Time Compilation) introduce noise or high response times that are not representative for the protocols. The version numbers of each server and framework being used is listed in Appendix A.

The acme-air driver is configured to send requests for 10 minutes across 10 threads with a ramp-up period of 30 seconds. The 30 seconds ramp-up period leads to the threads starting up gradually and leads to a gradual increase of work load on the system. Each request sent from acmeair-driver towards the acmeair system use HTTP/1.1 independently of which protocol is being tested. Furthermore, these requests use the keep-alive flag to avoid creating new TCP connections between requests.

During each test run, JMeter will write response meta-data such as timestamp, request URL, response time to a CSV file. Additionally, a background process will query each of the service hosts each second about their status to gather information about network and CPU usage. This monitoring service is implemented as a Java JAX-RS REST API that exposes information from the Linux command `ifconfig` for network statistics, and uses the Java function `ManagementFactory.getOperatingSystemMXBean()`[6] to get information about CPU usage. These are made available from two separate URIs `/systemStatus` and `/systemStatus/network` on each service host. Each time

[6] https://docs.oracle.com/javase/7/docs/api/java/lang/management/ManagementFactory.html

these are queried, their JSON representation is saved to a file together with the timestamp of when they were queried. At the end of each test run, these JSON files together with the request log from JMeter are parsed and imported into a MariaDB database for archival and data analysis.

The network utility command `ifconfig` in Linux lists the available networking interfaces in Linux. This command lists statistics for each interface, such as packets and bytes received, sent, dropped, together with the current status of the interface such as the associated IP address and hardware address. Docker allows the creation of networks on the local host. This enables for example the ability of containers to communicate using the name of the containers as the hostname in HTTP requests. In Acmeair, this is a commonly used strategy for communication across containers where each host has its own Docker network. On each host, a Docker network is created using the `--network` flag in `docker run`. This network is then represented as a separate interface in `ifconfig`. In the experimental setup for AcmeAir, the `eth0` interface is the interface for outgoing traffic. `lo` is for local loopback which typically handles traffic to localhost.

The monitoring service will expose network statistics for all of the interfaces listed by `ifconfig`. However, only the `eth0` interface is paid attention to when analyzing the results. This is due to the external communication out from the node will happen on the `eth0` interface where the simulated latency is also added.

4.2.4 Protocol adapters

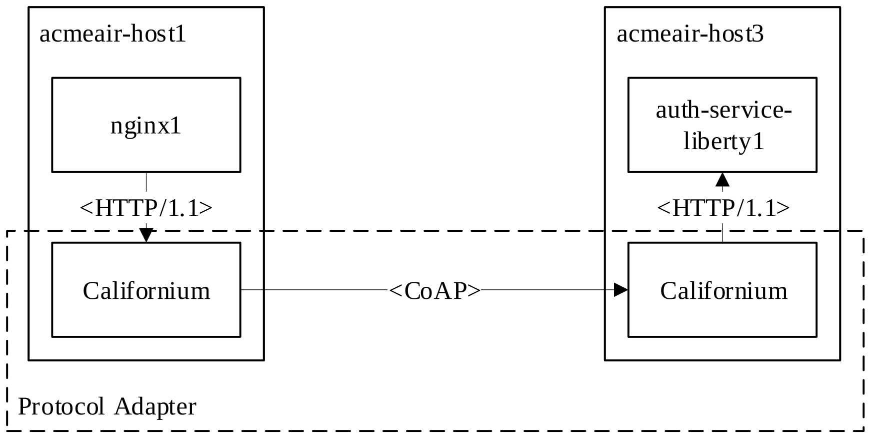


Figure 7: How requests are translated by intermediary proxies for the CoAP protocol. For h2c and HTTP/1.1, nghttp2 and nginx are used instead of Californium.

The Acmeair system use HTTP/1.1 for inter-service communication. In order to test different protocols without having to modify the implementation of AcmeAir services, the requests are forwarded to a proxy server that will then translate the requests to the protocol being tested (CoAP, h2c, or HTTP/1.1) on the same host. The request is then sent to the host for the target service where another proxy will translate the request back to HTTP/1.1 and forwarded to the target service. This flow is illustrated in Figure 7 for CoAP, where Californium is used to translate between HTTP/1.1 and CoAP for requests to the auth service. nghttp2 is used for h2c, and nginx for HTTP/1.1.

4.2.5 Changes to Californium

The HTTP proxy feature is used in Californium for translating requests between CoAP and HTTP. Initially, this feature demonstrated much worse performance compared to HTTP. By investigating the code of Californium, it became apparent that for each incoming HTTP request, Californium would create two new threads: One thread for the creation of the CoAP request, and one for sending a HTTP response back to the client, awaiting a CoAP response from the first thread. Conceptually, these two threads perform their work sequentially as the second thread is waiting for the response from the first one. Additionally, the Californium threading model uses the concept of worker threads for both sending and receiving messages on a blocking UDP socket API [47]. These two threads were eliminated by refactoring the code such that the code is executed sequentially without having to create two new threads.

Another problem regarding performance was the blocking API being used for sending HTTP requests. This had a negative impact on performance as it would block one thread until the HTTP response is received. It may also limit the ability to do data processing before TCP ACKs are being sent (depending on the framework being used) [85]. The API originally being used for creating outbound HTTP calls from the Californium framework was `org.apache.http.impl.client-.DefaultHttpClient`. Instead, the Apache HttpAsyncClient[7] was introduced to make outbound HTTP requests.

After introducing the asynchronous HTTP API and removing unnecessary threads, a few race conditions appeared. One issue was related to how two requests could receive parts of the response of another request if sent through the same connection. In Californium, the interaction between a client and a server is represented as an `Exchange` object [47]. This object has an UDP endpoint associated with it that handles incoming and outgoing traffic. The problem was that two exchanges would use the same endpoint for outgoing CoAP requests, leading to concurrent

[7] `https://hc.apache.org/httpcomponents-asyncclient-4.1.x/index.html`

responses getting intermixed with the original HTTP request. To solve this issue, a pool of `EndPointManager` instances were used to take endpoints from upon a CoAP request creation. These were returned to the pool once the CoAP response was received.

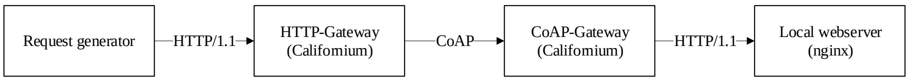


Figure 8: Experimental setup for profiling the Californium framework.

In order to avoid setting up the entire AcmeAir system to analyze the performance issues in Californium discussed earlier, a smaller setup was used on a local machine. It involves a C# .NET Core application sending HTTP requests to a Californium HTTP gateway which sends a CoAP request to a Californium CoAP-gateway, which in turn requests a HTTP resource on a local nginx HTTP server (Figure 8). This enables an simplified analysis of the behavior of the different components. Alternatively, one could have attached to the containers and remotely analyze those in the AcmeAir system during benchmarking. The C# .NET Core application would time the request and produce an average of the total request time. Additionally, the lightweight-java-profiler[8] by Jeremy Manson was used to capture stack traces at certain intervals. A flame graph is then produced using the FlameGraph[9] library by Brendan Gregg to visualize which stack traces occurred most frequently. Although this method simplifies setup for faster analyzing the performance behavior, there is only one type of request being sent through the system. The system is also running on a different environment than the one AcmeAir system was deployed on, which could introduce bias in terms of framework differences and interrupts from other processes.

The lightweight-java-profiler is implemented as an agent that is loaded together with the Java application being profiled through the `-agentpath` argument to the Java process. This agent will capture stack traces of all running threads in the Java VM and write those to a file at a certain interval which defaults to 100ms. The stack traces are then written to a file. Jeremy Manson argues that this profiler does not introduce as much overhead as other profilers, since it does not require to stop the entire JVM to capture stack traces and operates asynchronously. [86]

A flame graph is a visualization of how frequent code paths occurred within a set of stack traces. Flame graphs are produced by providing the set of stack traces from light-weight-java-profiler to FlameGraph. Figure 9 shows a small segment of

[8]`https://code.google.com/archive/p/lightweight-java-profiler/`
[9]`https://github.com/brendangregg/FlameGraph`

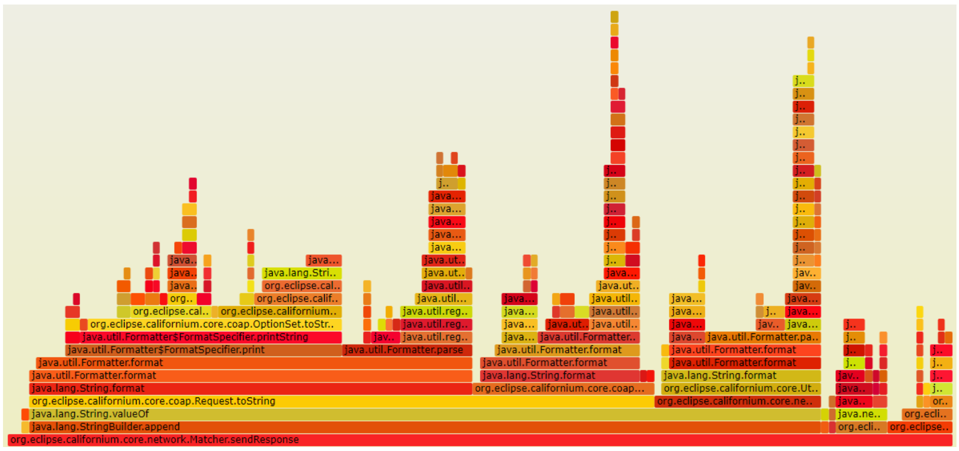


Figure 9: An excerpt of a flame graph of the CoAP gateway.

the result of profiling a Californium Proxy that is focused on the `sendResponse` frame. The y-axis indicates the size of the stack, while the x-axis indicates the amount of samples that particular frame has within the runtime of the application. The colors of the frames are only to distinguish them and do not carry any further meaning. A wider stack is an indication that a larger amount of time was spent within that particular graph. In Figure 9, a large amount of the stacks took place in `java.lang.StringBuilder.append` and `java.lang.String.valueOf`, which was called from the `Matcher.sendResponse` function inside the Californium framework.

The Californium framework uses the `java.util.Logging` API for logging status messages. Logging happens by providing a message together with a level indicating how important the log message is. Some examples of log levels are "fine", "info", "warning", "severe" (from lowest level to highest). An application is configured to log messages above a certain log level. That means all messages below the level is discarded. For example, if the application has set its log level to "warning", the `send()` function in Listing 4.1 does not write any messages.

```
class Response {
    private static final LOGGER
            = Logger.getLogger(Response.class.getName());
    ...

    public void send() {
        LOGGER.log(Level.INFO, "Sending response");
        ...
    }

    public void write(Parameter param) {
        LOGGER.log(Level.INFO,
                "Adding param: " + param.toString());
        ...
    }
}
```

Listing 4.1: Example of how messages can be logged with java.util.Logging.

The implementation of the log function will first attempt to check if the log level permits writing the log message. If it does, it will do additional operations on the log message (e.g. string formatting or object serialization) before writing the log message to a configured destination (e.g. file, network, or standard output).

One problem was the use of string concatenation instead of log formatting for constructing log messages in the Californium framework. An example of this can be seen in the `write(Parameter)` function in Listing 4.1. In this case, `toString()` is called on the `Parameter` object param, the string is concatenated with the "Adding param: " string, and is then passed to the log function. If the log level is set to a higher level than info, the log message is discarded. It would then not have been necessary to call the `toString()` function on the `param` object. Furthermore, the `toString()` function may be an expensive call in terms of CPU cycles or memory usage in order to construct the string depending on the implementation of the `toString()` method.

The `java.util.Logging` API allows log messages to be provided with parameters. This makes it possible to pass a parameter that is then formatted into the log message before stored, allowing to check if the log level allows writing to log before the log message is constructed and `toString()` is called on the provided parameters. An example of the write function from Listing 4.1 can be seen in Listing 4.2 where `param` is passed as an argument to the log function.

```
public void write(Parameter param) {
    LOGGER.log(Level.INFO, "Adding param: {0}", param);
    ...
}
```

Listing 4.2: A refactored version of write in Listing 4.1 where the param object is passed as a parameter to the log function.

A large number of functions in the Californium framework used the logging API without using log formatting which resulted in many frames related to the `toString()` method appear on the flame graph discussed earlier. This issue was largely related to incoming and outgoing requests where the request/response object was attached to the log message. By using passing log parameters as parameters instead of relying on string concatenation, the `toString()` method did no longer appear on the flame-graphs as they were no longer called.

Another encountered issue after changing to asynchronous HTTP connection handling was that session states became intermixed between requests. If one user signed in, all other users would have that particular session, and if the user signed out, all users would be signed out. In AcmeAir, the authentication mechanism is implemented by the use of session cookies. When the CoAP-gateway would make a request and a state was introduced through cookies, all other request would receive this state. By disabling cookie management in the HttpClient API, this issue became resolved.

These improvements to the Californium has been submitted upstream and merged in a pull request [87]. The performance impact was measured to double the throughput capacity on the master branch of Californium by one of the maintainers. To summarize, these issues were encountered while optimizing the Californium Proxy:

- Unnecessary thread creation when forwarding HTTP requests to CoAP requests.
- While the Californium framework has an asynchronous API based on worker threads, the HTTP requests made from the Californium used a blocking synchronous API.
- As an effect of higher request throughput, some requests could attempt to use the same message exchange for messaging, causing some response data to appear in the wrong responses.
- Sessions set using cookies became set across all requests due to cookie management not being turned off by default in the asynchronous HTTP API.
- Lack of log formatting caused a high number of objects to be serialized to string when they were not needed, leading to resources being spent on unnecessary object and string serialization.

In CoAP, header names are represented as options associated by an option number instead of strings. This makes the amount of headers that can be sent through the CoAP proxy limited to only the ones defined by the CoAP specification in Section 12.2 [43]. It is on the other hand possible to introduce new options. However, these options would lead to a higher coupling between client and server as these are communicating using option numbers that is not set by the CoAP specification. The same issue as with header goes with content types. These are also implemented as numbers. Only the content types for `application/x-javascript`, `text/css` and `application/x-www-form-urlencoded` had to be introduced. Additional introduced headers were related to cookie management through the `set-cookie` and `cookie` header.

The Californium framework use a threading model with several worker threads that are involved in processing a request. Incoming requests are handled with the class `UDPConnector` which use a blocking call on a `DatagramSocket` for receiving UDP datagrams. A pool of worker threads (network stage threads) are awaiting for incoming messages on this call, which is default to 1 worker thread. The incoming messages are copied off the message buffer and given to the `InboxImpl`, which will execute the incoming data on thread from a pool of workers threads. This stage is called the "protocol stage" which will handle protocol-related work such as parsing of CoAP messages. The same thread will be used for any logic that is running on top of Californium. When creating a response, the response is queued in a `BlockingQueue` of outgoing messages where another set of networking stage threads are waiting to send messages out on the `DatagramSocket`.

It might be questionable whether this threading model has a positive impact on performance in context of this work, as the two stages are doing for the most part CPU-intensive work that is related to protocol-parsing and business logic. However, this might be beneficial if expensive CPU-intensive operations such as encryption that would require more CPU work for processing compared to simple parsing operations, or if the business logic would use blocking API calls. It is also beneficial at higher concurrency load, as discussed by Kovatsch et al.. [47]. While the setup in this experiment used 10 concurrent clients from the same host, the work by Kovatsch et al. used multiple hosts to gain a higher workload on the target server.

4.3 Results

Protocol	**Lat.**	**Requests**	**Mean**	**Max**	**Min**	**Std.dev.**
coap	0 ms	895542	6.4042 ms	245 ms	2 ms	4.0090 ms
h2c	0 ms	1030060	5.5625 ms	108 ms	1 ms	2.8164 ms
http1.1	0 ms	1152871	4.9634 ms	89 ms	1 ms	2.8448 ms
coap	2 ms	381152	15.2300 ms	108 ms	10 ms	4.2328 ms
h2c	2 ms	386191	15.0327 ms	103 ms	10 ms	4.1621 ms
http1.1	2 ms	397553	14.5938 ms	85 ms	9 ms	4.1093 ms
coap	5 ms	195262	29.8653 ms	118 ms	22 ms	6.8693 ms
h2c	5 ms	197126	29.5732 ms	133 ms	22 ms	6.9712 ms
http1.1	5 ms	199428	29.2244 ms	117 ms	22 ms	7.0784 ms

Table 2: Summary of the requests.

4.3.1 Response times

The response times per service is summarized in Figure 10 and Table 2. CoAP is standing out from h2c and HTTP/1.1 in terms of standard deviation and average response time under no latency. HTTP/1.1 and h2c performed more equally where HTTP/1.1 had a lower average response time for all cases.

When the system is delaying each packet with 2ms, the differences start to even out between the different protocols (Figure 10). From having the highest standard deviation on no simulated latency, CoAP has a lower standard deviation under

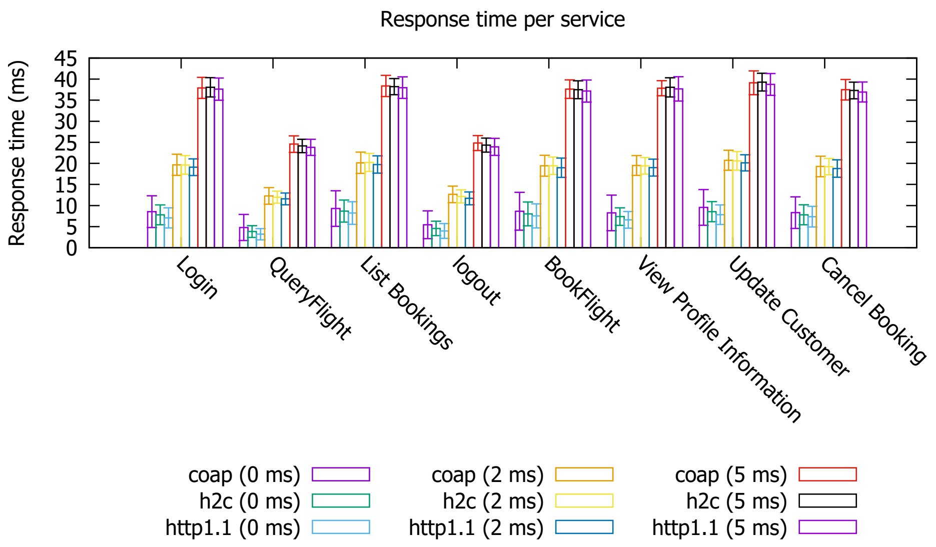


Figure 10: Response times across the different simulated latencies.

5ms simulated latency. As seen between no simulated latency and 2ms simulated latency, the differences across all protocols become more even with 5ms simulated latency (in terms of mean response time, std. dev., and amount of requests).

4.3.2 Network usage

The network usage is summarized in Figures 11, 12, 13, and 14. The nginx1 service was excluded from these graphs and added as Appendix B. The services main-service and support-service were excluded as they are not involved in any of the benchmarking test cases.

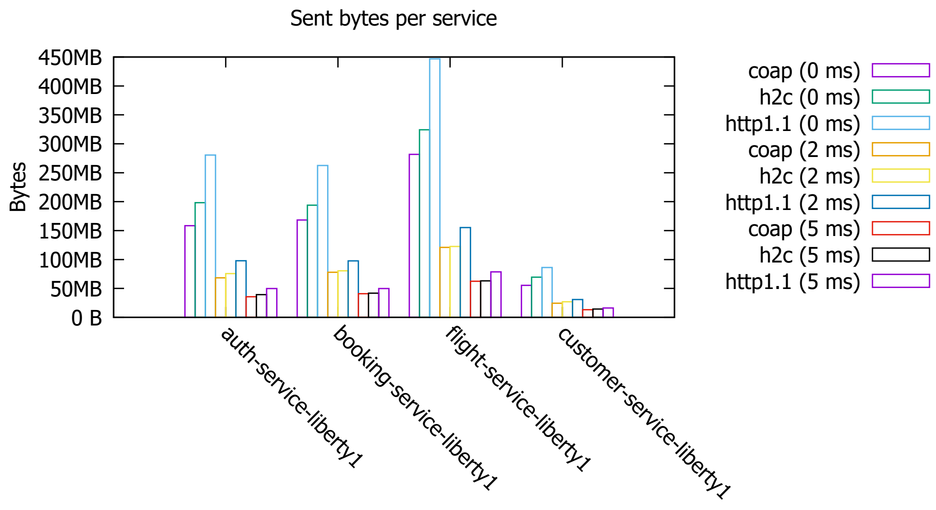


Figure 11: Amount of bytes sent per service.

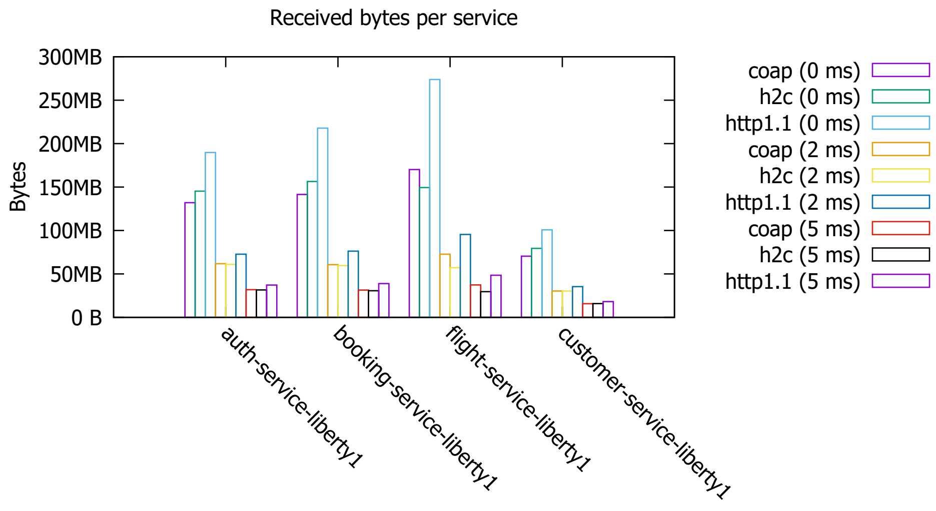


Figure 12: Amount of received bytes per service.

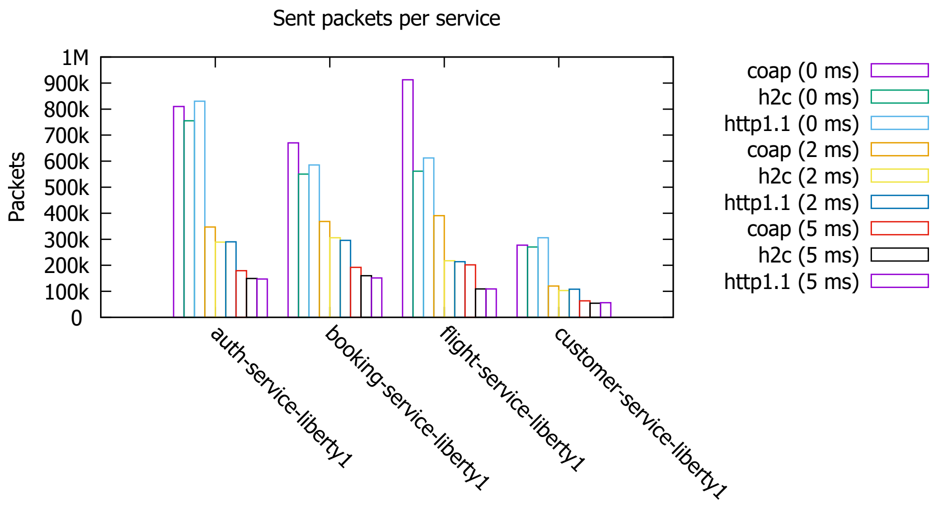


Figure 13: Amount of sent packets per service.

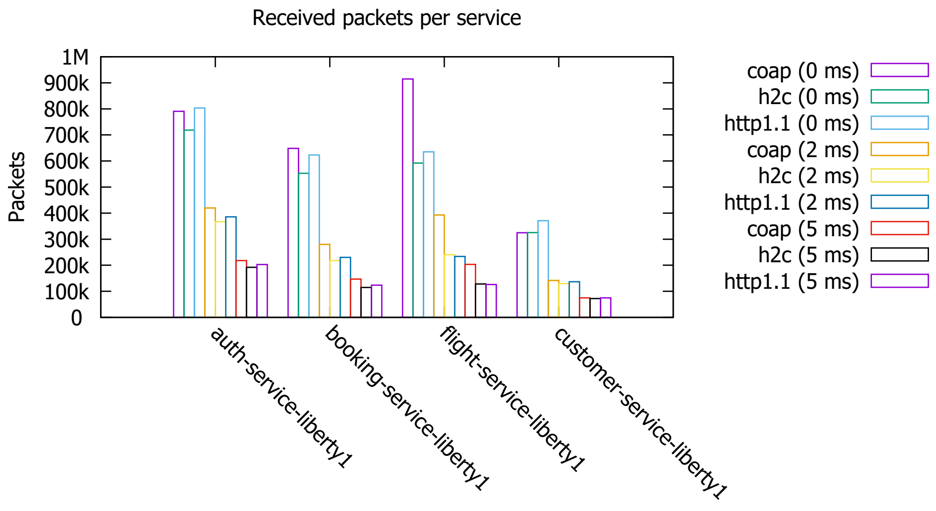


Figure 14: Amount of received packets per service.

To summarize, the differences in response times between the protocols even out towards 5ms simulated latency. This shows that the amount roundtrips involved in each protocol to process a request does not differ much between the three protocols. By dividing the total amount of bytes and packets on each protocol under each simulated latency by the total amount of requests, it is possible to see a more detailed picture of how the protocols compare:

Protocol	**Latency**	**Bytes/request**	**Packets/request**	**Bytes/packet**
coap	0 ms	3224.6675 B	12.1271	265.9063 B
h2c	0 ms	3143.6767 B	8.6052	365.3242 B
http1.1	0 ms	3782.2935 B	8.4253	448.9194 B
coap	2 ms	3273.4470 B	12.6509	258.7531 B
h2c	2 ms	3213.5299 B	9.4182	341.2046 B
http1.1	2 ms	3849.4307 B	9.2052	418.1797 B
coap	5 ms	3303.8630 B	12.8083	257.9475 B
h2c	5 ms	3248.9165 B	9.6372	337.1219 B
http1.1	5 ms	3896.4796 B	9.5806	406.7032 B

Table 3: Overview of packet count and transferred bytes per request.

Table 3 shows how the protocols compare per request. While CoAP sends more packets per request, the packets are considerably smaller compared to h2c and HTTP/1.1. This does not affect the response time as these packets are sent in sequence and do therefore not result in waiting for a response before sending the next packet. h2c does, on the other hand, have a similar behavior compared to HTTP/1.1 when it comes to packets per request, but is resulting in fewer transferred bytes per request under all latencies.

4.3.3 CPU usage

The CPU usage on each service under each latency and protocol is summarized in Figure 15 with the average CPU usage together with the standard deviation as error bars. It becomes apparent that across all latencies, CoAP is producing most CPU load under all latencies on the API-gateway (nginx1). One possible cause for this is that the Californium framework may be less performant on translating HTTP requests to CoAP requests compared handling incoming CoAP requests and translating them to HTTP requests. This trend changes on the other services where the differences are marginal, for example on the customer-service where the protocols had a much lower difference across the different latencies. A common trend, however, is that the differences in the protocols in terms of CPU and response times decrease as the simulated latency increases.

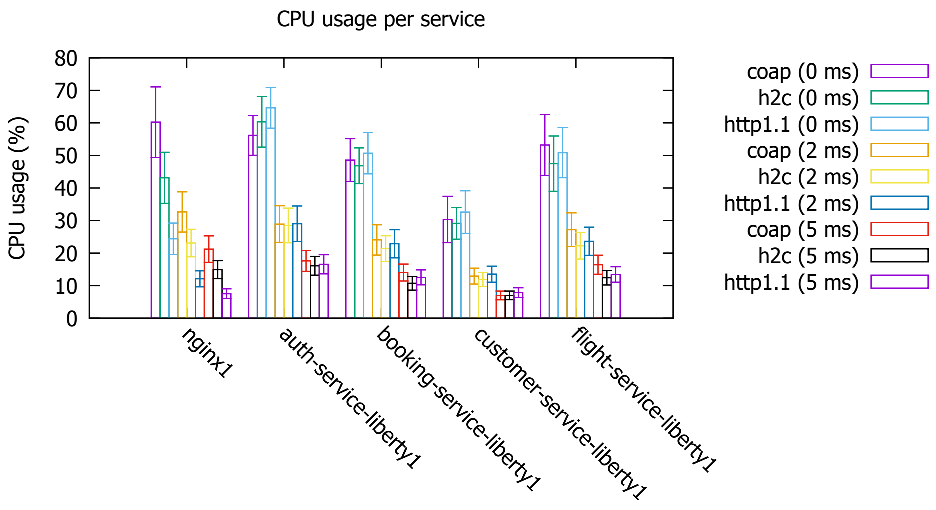


Figure 15: CPU utilization across services per latency and protocol.

4.4 Inspecting Californium performance

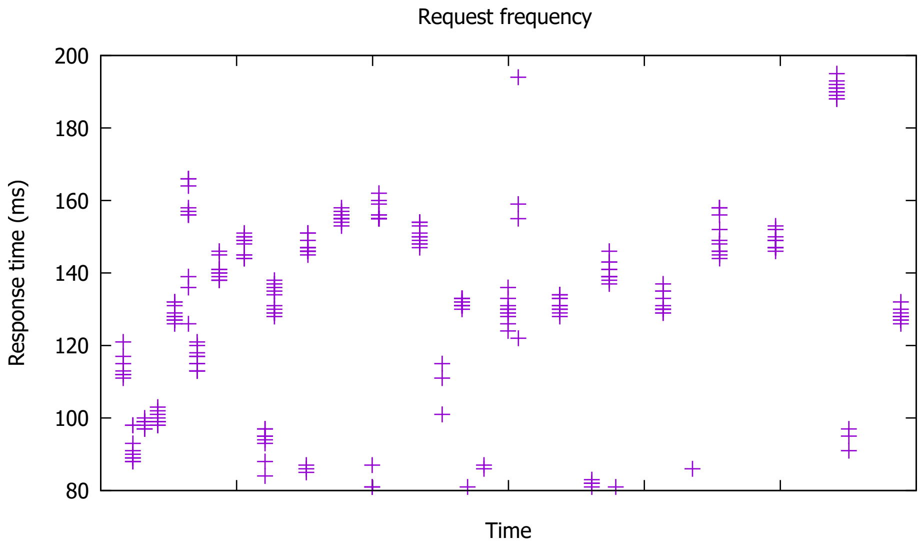


Figure 16: Frequency of CoAP requests with response times in the range <80ms, 200ms>.

CoAP with no simulated latency is standing out from the other configurations in terms of CPU usage on nginx1 and standard deviation for response times. Compared to other protocols and latencies, the protocol has a high number of requests that are clustered together with high response time. This can be seen in the frequency diagram on Figure 16.

4.4.1 Profiling

The clustering of the request occurrence indicates that a recurring process might be delaying the response processing, thus resulting in the clustered requests in Figure 16. Java is a managed programming language in which memory is managed by the runtime in terms of garbage collection (GC). GC is a process that starts at given intervals or when memory usage reaches a given threshold, such as heap size. In order to investigate whether GC has a significant impact on the system performance, a smaller setup is used on a local machine where a HTTP gateway accepts HTTP requests, translates them to CoAP requests and forwards them to another CoAP proxy which will then forward the request to an nginx server (Figure 8). A program will send HTTP requests through the system as long as it receives a response. Java Visual VM[10] is then used for inspecting the runtime behavior of the system in terms of memory usage.

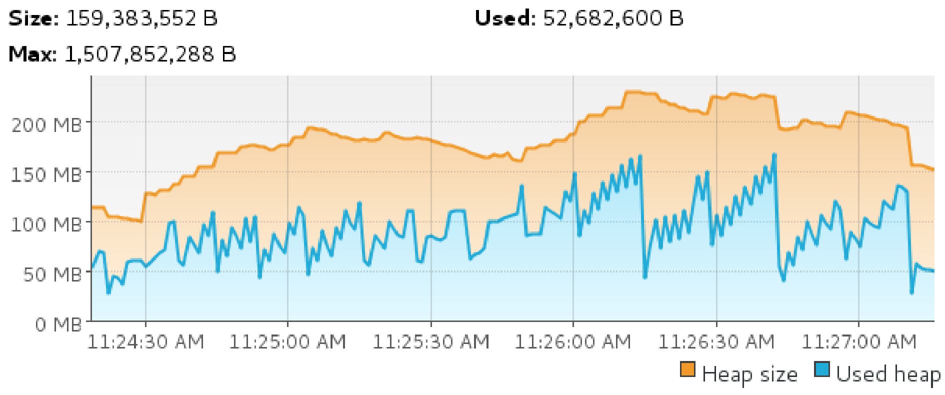


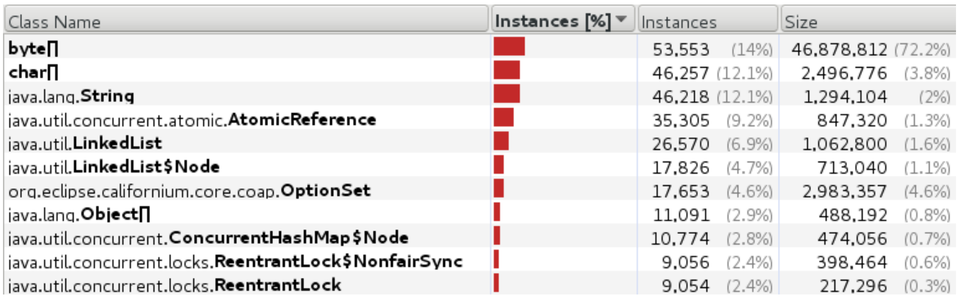
Figure 17: Memory usage of the HTTP gateway using Californium in Java Visual VM.

Figure 17 shows the memory usage of the HTTP gateway over time as requests are passing through. Heap size is the maximum allowed memory the application can use of the allocated memory to the Java process (similar to how much memory the Java process is using), while used heap is the amount of memory that is being used at each time. The used heap does also include garbage, which is memory that

[10]https://visualvm.github.io/

is no longer used or pointed to.

Objects in the Java VM can be either short or long-lived. Once there are enough short-lived object, a minor GC will begin and go through the young generation of short-lived objects. The minor GC will cause a so-called *stop the world event*, which will pause the Java VM until the GC is done, causing a full stop of the executing program. Typically, the minor GC is short and does typically not have a big performance impact. However, if the short-lived objects exists long enough, they will typically be marked long-lived and put in the old generation. A major GC is required to remove allocated memory from the old generation, which will also cause a stop the world event. Considering the old generation is in most cases much larger than the young generation, a major GC will take much more time than a minor GC. [88]



Class Name	Instances [%]	Instances	Size
byte[]	▬	53,553 (14%)	46,878,812 (72.2%)
char[]	▬	46,257 (12.1%)	2,496,776 (3.8%)
java.lang.**String**	▬	46,218 (12.1%)	1,294,104 (2%)
java.util.concurrent.atomic.**AtomicReference**	▬	35,305 (9.2%)	847,320 (1.3%)
java.util.**LinkedList**	▬	26,570 (6.9%)	1,062,800 (1.6%)
java.util.**LinkedList$Node**	▬	17,826 (4.7%)	713,040 (1.1%)
org.eclipse.californium.core.coap.**OptionSet**	▬	17,653 (4.6%)	2,983,357 (4.6%)
java.lang.**Object[]**	▬	11,091 (2.9%)	488,192 (0.8%)
java.util.concurrent.**ConcurrentHashMap$Node**	▬	10,774 (2.8%)	474,056 (0.7%)
java.util.concurrent.locks.**ReentrantLock$NonfairSync**	▬	9,056 (2.4%)	398,464 (0.6%)
java.util.concurrent.locks.**ReentrantLock**	▬	9,054 (2.4%)	217,296 (0.3%)

Figure 18: List of objects during runtime and their resource usage in a heap dump.

Figure 18 shows a heap dump of the HTTP gateway. The type which occupies the most memory in the heap is `byte[]` which use 72.2% of the heap, followed by `char[]` (3.8%) and `java.lang.String` (2%). Most of the `char[]` instances are internal representations of `java.lang.String`. A large amount of the allocated strings are segments of the HTTP query string, which is being split up to determine where the request should be forwarded. Most of the instances of `byte[]` are used in instances such as `UDPFragment`, `Request`, `Response`, `LinkedList$Node` (internal structure of `java.util.LinkedList`, where most references are from the `OptionSet` class), and in `Exchange$KeyMID` to represent addresses. `UDPFragment` is being used to hold data about a received UDP fragment. `Request` and `Response` are classes that deal with request/response handling inside the Californium framework. The `OptionSet` class represents metadata about a request. All of these classes inside the Californium framework actively allocates new byte arrays once required for sending, receiving, and processing requests. This results in excessive garbage which the GC will clean up in certain intervals, resulting in the frequent allocation/deallocation seen in the graph on

Figure 17.

4.4.2 Impact of GC events on response times

In order to accurately determine whether the impact of major GC events has a significant impact on the response time, the same proxy-setup as described earlier in Section 4.2.5 is used. Additionally, the request time together with their response time is saved to a CSV file from a request generator. In the two Java processes running the Californium framework for translating messages between HTTP and CoAP, a timestamp is written to a CSV file for each major GC event using the `com.sun.management.GarbageCollectionNotificationInfo` class for GC notifications. The request generator is a C# .NET Core application that will send requests as long as it receives a response for 5 minutes. A warmup period of 30 seconds is used for making sure all components are fully loaded, and to make sure a larger amount of the heap memory size is being used leading to more frequent GC events than once the system is started.

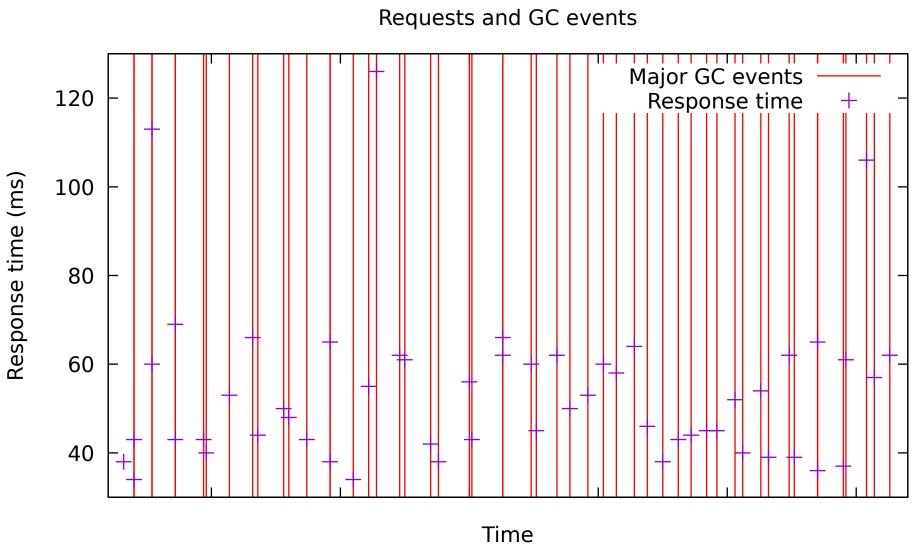


Figure 19: Major GC events and response time extremes (greater than or equal to 30ms).

Figure 19 shows the all request response time extremes above a certain threshold (more than or equal to 30ms) based on the distribution of response times. In total, the response time average was 1.18ms with a standard deviation of 0.96ms with a total of 252636 requests. The HTTP gateway had 29 major GC events, while the CoAP-gateway had 25 events. It can be seen on the graph that each GC event has a corresponding request extreme. Of the 53 identified extremes, one did not have a corresponding GC event (response time of 38ms). This extreme may have been caused by other factors such as system interrupts.

Considering the large amount of memory being freed by GC (see Figure 17) and its occurrence with response extremes, the excessive garbage caused by frequent memory allocation of short-lived objects would be a major factor leading to spikes in response times.

4.5 Discussion and future work

4.5.1 Discussion

The results shows that both HTTP/2 and CoAP performed worse than HTTP/1.1 in terms of response time. However, HTTP/2 used least bandwidth, followed by CoAP. One reason HTTP/1.1 resulting in such low latencies could be due to how optimized nginx is compared with nghttp2. Despite the attempts to optimize the Californium framework, there is a significant difference in response times between Californium compared to nghttp2 and nginx. One possible cause would be the lack of memory buffer re-usage for sending, receiving, and processing data which then in turns leads to excessive GC.

One major differences between CoAP and HTTP is if headers are not supported by the CoAP standard or the CoAP framework, some development effort is required to properly map the unsupported headers. Considering that application-specific changes are required on protocol level, it would also lead to higher coupling between the business logic and the protocol.

To answer the research question R3 regarding CoAP and HTTP, based on the experience from using CoAP in AcmeAir, CoAP can easily be implemented between microservices without changing URI schemes by the use of proxies. However, due to how CoAP implements headers as IDs instead of using string like in HTTP, some effort may be required to set up an appropriate mapping between the two proxies for correct header translation. Unknown headers would not be automatically mapped, and therefore rejected.

Regarding research question R2 that questions the differences between h2 and HTTP/1.1, the major observed difference in this experiment was the bandwidth usage between the two protocols. The differences (latency) between the two protocols decreases with higher latencies, which may be an indication that there is an optimization opportunity in nghttp2. Considering performance limitations with no latency tend to be framework-related, it might be said that framework bottlenecks become dominant with no simulated latency. With a higher amount of latency between each request, the resource provider is able to perform other tasks. This can be seen in how also the CPU usage is reduced on higher latencies. That the protocols gets more even in terms of latency is an indication that there is no difference in amount of round-trips required to process each message.

However, this study is limited to the architecture of AcmeAir. The service in AcmeAir use only synchronous messaging and no asynchronus messaging is involved. Furthermore, these messages use a limited set of HTTP verbs and some of the interfaces may not be entirely RESTful due to the use of cookies, lack of HATEOAS, and use of verbs instead of nouns for the resource names. The protocol being used for direct service-to-service (e.g. between booking-service and auth-service) did however use HTTP/1.1 regardless of what protocol was tested. This lead to more traffic unrelated to the tested protocol become reflected in the measured traffic. This does however not get reflected in the results as HTTP/1.1 is used regardless of simulated latency and tested protocol between these services.

4.5.2 Future work

Future work would involve *optimizing the Californium framework to reuse buffers*. It would be expected that this leads to less time being spent on garbage collection and memory allocation, which would lead to a less skewed normal distribution of the response times and less peaks/extremes in response times.

Considering the performance impact of blocking vs. non-blocking, it would be worth further *investigate the impact of using an asynchronous model for message processing in Californium* instead of relying on thread pools. In a post by Stephen Cleary [89], he explains how the `async` keyword in .NET avoids using any thread for waiting on application level in a pure asynchronous threading model. Instead of a thread waiting for a reply, the operating system relies on Direct Memory Access (DMA) for the device to read directly from the application memory, then signal using a system interrupt back to the operating system and then back to the application indicating the I/O operation is complete. It would be interesting to see how much of a performance impact this model has compared with the threading model being used in Californium.

SPDY was not benchmarked in this setup due to how it is still considered an experimental protocol by Google and the only reference implementations during the time of writing is a Go library and the Chromium browser. *When a major server vendor such as nginx would provide support for SPDY for backend communication, it would be interesting to see how the protocol would compare* with HTTP/2 in terms of both latency and bandwidth usage. It would however be expected that the protocol behaves similar to HTTP/2 in terms of response times and bandwidth usage due to their relation.

The reason for using nghttp2 instead of nginx for h2c was due to nginx not supporting HTTP/2 for backend communication. While it is still supported for front-end communication, it is yet to be implemented since it would involve significant changes to the internals of nginx[11].

One aspect that was not much investigated was *profiling and analyzing the runtime of nghttp2*. Even though nghttp2 supports HTTP/1.1 like nginx, the implementation of h2c was not investigated together with the threading model being used. nginx was also not investigated.

Another aspect that would be worth investigating would be *the effect of encryption on the different protocols and how this would impact their behavioral characteristics*. The different protocols were not tested with any encryption in this experiment due to time constraints, but would be worth investigating considering how microservices would often be deployed in a public cloud environment and the wide-spread usage of encrypted HTTP channels in general.

CoAP provides a publish/subscribe implementation on protocol level to observe when resources change their state [43] in addition to a mechanism for service discovery [67]. Other future work would *investigate whether the publish/subscribe mechanism in CoAP can be applied in a microservice system*.

The HTTP/2 protocol allows multiplexing where multiple response/requests can occur over the same TCP-connection. Another aspect for future work would be *how can multiplexing be utilized for efficient service consumption of APIs*.

The AcmeAir system does only use synchronous communication. Future work would involve *extending the AcmeAir system to also use asynchronous messaging* (e.g. message queues and publish/subscribe pattern) for ISC.

[11]https://trac.nginx.org/nginx/ticket/923

5 Conclusion and future work

5.1 Conclusion

The microservice architecture pattern emerges as a new alternative to architecting applications for large-scale deployments. The emergence of newer tools for developing and orchestrating these services made it possible to develop services of more granular size. While many applications were implemented as monoliths, the need for faster scaling in terms of throughput and introduction of new features together with the introduction of ubiquitous cloud computing formed the fundamental needs for microservices. Monoliths became difficult to scale and deploy in cloud environments, where computing resources became more accessible compared to hosting a system in-house.

Implementing a microservice system does however come at the cost of increased architectural complexity, increased network traffic, and efforts of orchestrating and maintaining this distributed system.

R1: REST limitations: What are the architectural limitations of using REST in microservices?

While REST appears as an alternative to RPC-style APIs that brings lower coupling between service consumers and service providers, the limitations of the architecture are still the same from previous discussions of the architectural style regarding REST and RPC. These limitations relate to transaction support, security, reliable message transfer, together with how HATEOAS should be practiced. Although there are patterns that extend REST in an attempt to introduce transactions, these patterns are often criticized for introducing application state through the semantics of the interface. It may on the other hand be questionable whether microservices should introduce transactions at all as many argue a microservice system should instead rely on eventual consistency [4].

As the literature survey in Chapter 3 show, there are some side-effects of using REST. One of the constraints in REST (Uniform interface) states that requests should be "optimized for the common use of the web" [17]. In a microservice system, this would mean that a service should not provide specialized interfaces towards other services but provide one, common, generalized interface towards each service. Prohibiting interface specialization increased the overhead in the transferred messages, negatively affecting performance. On the other hand, this leads to a lower coupling between services.

Many of the discussed problems together with REST was related to the transport protocol and not the architectural pattern itself. HTTP being one of the most commonly associated protocols with REST, there are several issues being related to the use of HTTP together with REST such as service discovery and that it is a synchronous request-response protocol that does not enable asynchronous communication without the use of web hooks or introducing application state through temporarily resources.

R2: Benefits and limitations of HTTP/2: For ISC, what are the benefits and limitations of HTTP/2 compared to HTTP/1.1 in terms of performance and latency?

The results from using h2c in AcmeAir show that the protocol itself does not improve the response time, but lead to a lower bandwidth usage. HTTP/1.1 had on the other hand a lower response time than h2c with a higher bandwidth usage. The difference between these two protocols in terms of latency became smaller as the latency increased. One possible reason HTTP/1.1 was able to outperform in terms of response time across all latencies may be due to HTTP/1.1 have been existing for a longer time than HTTP/2, which gives more time for web-server developers to optimize the protocol implementations for both throughput and resource efficiencieness.

HTTP/2 introduces a number of mechanisms for message transmission over HTTP/1.1. While HTTP/1.1 requires the use of the keep-alive flag in order to reuse connection, HTTP/2 maintains this natively. The protocol also introduce multiplexing which enables multiple requests/responses to be in-flight over the same TCP-connection. Fundamentally on protocol level, HTTP/2 is a binary protocol opposed to previous standards which were ASCII based. Even though this makes the protocol less readable for humans, the binary protocol enables more efficient parsing of messages.

R3: How does CoAP compare to HTTP: How can CoAP be applied in a microservice system, and what are the challenges of using CoAP instead of HTTP?

One of the major limitations when using CoAP over HTTP is the amount of frameworks being available at hand. CoAP being developed for constrained IoT devices does pose some fundamental differences compared to HTTP. Although the protocol provide GET, PUT, POST, DELETE verbs similar to HTTP, the protocol use a high degree of flags instead of using strings for presenting verbs, headers, and content-types. This limits the supported headers and content-types to the ones being provided in the protocol specification. In a microservice system, this introduces higher coupling between two communicating end-points, where application-specific headers and content-types must be implemented in the proxies on both the sender and receiver end.

The approach towards introducing CoAP in AcmeAir use intermediary proxies to translate requests between HTTP and CoAP. By using these proxies, the frameworks for the protocols are not restricted to one language or platform. On the other hand, CoAP does still have fewer implementations compared to HTTP due to the widespread adoption and maturity of the HTTP protocol. This approach does also lead to more architectural complexity by introducing new intermediaries that process requests and may also reduce response times compared to direct communication.

Compared with h2c and HTTP/1.1, CoAP provided a much lower request throughput without any simulated latency, possibly due to framework limitations. It did however lean towards similar response times as HTTP/1.1 and h2c. Despite efforts to optimize the framework in terms of asynchronous communication, threading, and synchronization, the protocol remains behind both h2c and HTTP/1.1. One of the reasons may be due to memory fragmentation as a result of frequent allocation of temporarily message buffers which remains a possible aspect for optimization.

Although CoAP appears as a light-weight alternative to HTTP, the use of header flags can possibly lead to higher coupling between endpoints if the application use headers not present in the CoAP standard specification. It does on the other hand provide mechanisms beneficial in a microservice system that HTTP does not provide such as service discovery and a publish/subscribe mechanism.

R4: Latency in ISC: Which factors impact latency in ISC and how can these be reduced?

While it is possible to place two services in close approximation (e.g. on the same host or same data center opposed to two server regions), the actual way the services communicates also has an affect on the latency. Which protocol, interface design (in terms of granularity), and implementation are all factors in the latency of messages processing.

A protocol may for example promote loose coupling by providing a set of widely used semantics and standards. It could also provide efficient message transferring by the use of compression algorithms or a binary protocol. However, even though the messages can be relatively small in one protocol, the impact on latency based solely on message size may not have an effect on the latency depending on the Maximum Transmission Unit (MTU) size allowed on the network. HTTP/1.1 being a much used protocol for service-consumption, newer protocols such as HTTP/2 is designed to be more machine-readable and efficient. Surprisingly, HTTP/1.1 did however outperform HTTP/2 in terms of response times, where the differences evened out as latency increased.

There are also implementation-specific details that affect the latency. Understanding the run-time in order to allow high throughput of requests is a key element. Analysis through profiling and application resource usage data (CPU, memory usage, etc.) can help developers get a better understanding of how their application is behaving like it is. The analysis of the performance in the Californium framework highlighted the importance of efficient resource usage in terms of threads and asynchronicity. This analysis lead to implementation improvements that has been submitted and merged into the Californium project. However, the architecture of the run-time may be a limit in how rapid an application is able to respond to a request. Too much switching between threads can lead to a higher cache-miss. Other models such as asynchronous/non-blocking programming models has become more ubiquitous and a necessity for high-performance networking applications like nginx.

Understanding the implications of the architectural decisions being made in a microservice system is important in order to create an efficient distributed application. Considering the microservice pattern introduce more complexity compared to monoliths, a small change to one component may have a larger effect on the overall system picture. Being able to monitor the behavior of the microservices in addition to carefully design and test them becomes an increased challenge due to the nature of distributed systems.

5.1.1 Future work

Some limitations regarding latency is much blamed on the Californium framework. Future work would then be to further explore the effects of performance optimizations on the framework itself, and the side effects it would have on the experimental setup in AcmeAir. It is argued some of the limitations is due to inefficient use of memory buffers, this may also affect both CPU utilization and response times. Further optimizations of the Californium framework can make the protocol more attractive for backend Java systems as the protocol provides other mechanisms suitable in a microservice system such as service discovery and support for the publish/subscribe pattern.

Another research area would be the inter-linked set of services as a result of applying the HATEOAS constraint. Much like how the web emerged, HATEOAS would enable a similar set of inter-linked services to emerge, but designed for machines and not directly humans. There are multiple challenges that needs to be faced in order to allow such a network of services to merge: How are service boundaries defined between the services, service life-cycle, authorization, context of use, service contracts, service discovery, and how to define the correct level of available state transitions. HATEOAS does also not express for example how a document should be formed that a resource would accept e.g. through POST. This can be compared to one of the main objectives in SOA; although services are interlinked today, embracing HATEOAS would enable a more loosely coupled network of exchanging services. This would enable a client to explore and browse services without having to refer to a set of documentation in order to understand how to interact with them.

Bibliography

[1] Fowler, M. & Lewis, J. Microservices. [updated 25.04.2014, cited 29.09.2016]. URL: http://martinfowler.com/articles/microservices.html.

[2] Vural, H., Koyuncu, M., & Guney, S. 2017. A systematic literature review on microservices. In *International Conference on Computational Science and Its Applications*, 203–217. Springer.

[3] Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., & Nieh, J. 2015. Synapse: A microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 21:1–21:16, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/2741948.2741975, doi:10.1145/2741948.2741975.

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

[5] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. 2016. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

[6] Montesi, F. & Weber, J. 2016. Circuit breakers, discovery, and api gateways in microservices. *arXiv preprint arXiv:1609.05830*.

[7] Rhubart, B. Architect: Microservices and soa. Available from: http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html. [updated 03.2015, cited 29.09.2016].

[8] Richards, M. 2016. *Microservices vs. Service-Oriented Architecture*. O'Reilly.

[9] Savchenko, D., Radchenko, G., & Taipale, O. 2015. Microservices validation: Mjolnirr platform case study. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, 235–240. IEEE.

[10] Zimmermann, O. 2016. Microservices tenets. *Computer Science - Research and Development*, 1–10. URL: http://dx.doi.org/10.1007/s00450-016-0337-0, doi:10.1007/s00450-016-0337-0.

[11] Bass, L. 2015. Microservice position paper. *The 11th SEI Architecture Technology User Network (SATURN) Conference*. URL: https://github.com/michaelkeeling/SATURN2015-Microservices-Workshop/blob/master/saturn2015-position-papers/bass-microservices-workshop-position-saturn2015.pdf.

[12] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. Sept 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, 583–590. doi:10.1109/ColumbianCC.2015.7333476.

[13] Kookarinrat, P. & Temtanapat, Y. 2016. Design and implementation of a decentralized message bus for microservices. In *Computer Science and Software Engineering (JCSSE), 2016 13th International Joint Conference on*, 1–6. IEEE.

[14] Richardson, C. Building microservices: Inter-process communication. Available from: https://www.nginx.com/blog/building-microservices-inter-process-communication/. [cited 01.10.2016].

[15] Alshuqayran, N., Ali, N., & Evans, R. 2016. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, 44–51. IEEE.

[16] Villamor, J. I. F., Fernandez, C. A. I., & Ayestaran, M. G. 2010. Microservices: Lightweight service descriptions for rest architectural style. In *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence, ICAART 2010*, Setubal, Portugal. INSTICC, Institute for Systems and Technologies of Information, Control and Communication. URL: http://oa.upm.es/8128/.

[17] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[18] Aderaldo, C. M., Mendonça, N. C., Pahl, C., & Jamshidi, P. 2017. Benchmark requirements for microservices architecture research. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, 8–13. IEEE Press.

[19] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. 2017. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1), 91–98.

[20] Rotem-Gal-Oz, A. Practical soa. Available from: http://arnon.me/wp-content/uploads/2010/10/Nanoservices.pdf. [updated 10.2010, cited 30.09.2016].

[21] Klock, S., Van Der Werf, J. M. E., Guelen, J. P., & Jansen, S. 2017. Workload-based clustering of coherent feature sets in microservice architectures. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, 11–20. IEEE.

[22] Kratzke, N. 2017. About microservices, containers and their underestimated impact on network performance. *CoRR*, abs/1710.04049. URL: http://arxiv.org/abs/1710.04049, arXiv:1710.04049.

[23] Di Francesco, P. 2017. Architecting microservices. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, 224–229. IEEE.

[24] Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., & Villari, M. 2016. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5), 81–88.

[25] Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. 2017. Performance comparison between container-based and vm-based services. In *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*, 185–190. IEEE.

[26] Bonèr, J. Life beyond the illusion. Voxxed Days, 2016. URL: https://www.youtube.com/watch?v=Nhz5jMXS8gE.

[27] Westhide, D. Why restful communication between microservices can be perfectly fine. Available from: https://www.innoq.com/en/blog/why-restful-communication-between-microservices-can-be-perfectly-fine/. [updated 06.03.2016, cited 02.12.2016].

[28] Thönes, J. Jan 2015. Microservices. *IEEE Software*, 32(1), 116–116. doi:10.1109/MS.2015.11.

[29] Ranney, M. What i wish i had known before scaling uber to 1000 services. GOTO 2016, 2016. URL: https://www.youtube.com/watch?v=kb-m2fasdDY.

[30] Riva, C. & Laitkorpi, M. 2007. Designing web-based mobile services with rest. In *International Conference on Service-Oriented Computing*, 439–450. Springer.

[31] Sill, A. Sept 2016. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5), 76–80. doi:10.1109/MCC.2016.111.

[32] Maeda, K. 2012. 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*, 177–182. IEEE.

[33] Bson (binary json): Specification. http://bsonspec.org/spec.html. (Accessed on 2017.12.05).

[34] Huang, C.-C., Huang, J.-L., Tsai, C.-L., Wu, G.-Z., Chen, C.-M., & Lee, W.-C. 2014. Energy-efficient and cost-effective web api invocations with transfer size reduction for mobile mashup applications. *Wireless networks*, 20(3), 361–378.

[35] Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L., Leach, P. J., & Berners-Lee, T. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[36] Belshe, M., Peon, R., & Thomson, M. Hypertext transfer protocol version 2 (http/2). RFC 7540, RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7540.txt.

[37] Belshe, M. & Peon, R. Spdy protocol. Internet-Draft draft-mbelshe-httpbis-spdy-00, IETF Secretariat, February 2012. URL: http://www.ietf.org/internet-drafts/draft-mbelshe-httpbis-spdy-00.txt.

[38] Hamilton, R., Iyengar, J., Swett, I., & Wilk, A. Quic: A udp-based secure and reliable transport for http/2. Internet-Draft draft-tsvwg-quic-protocol-02, IETF Secretariat, January 2016. URL: http://www.ietf.org/internet-drafts/draft-tsvwg-quic-protocol-02.txt.

[39] Megyesi, P., Krämer, Z., & Molnár, S. Comparison of web transfer protocols. High Speed Networks Laboratory, Department of Telecommunications and Media Informatics, Budapest, January 2016. URL: http://proprogressio.hu/wp-content/uploads/2016/01/MolnarSandor_2015.pdf.

[40] Hemminger, S. et al. 2005. Network emulation with netem. In *Linux conf au*, 18–23.

[41] Carlucci, G., De Cicco, L., & Mascolo, S. 2015. Http over udp: an experimental investigation of quic. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 609–614. ACM.

[42] Sill, A. 2015. Invention, innovation, and new apis. *IEEE Cloud Computing*, 2(2), 6–9.

[43] Shelby, Z., Hartke, K., & Bormann, C. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. URL: http://www.rfc-editor.org/rfc/rfc7252.txt.

[44] Bormann, C., Castellani, A. P., & Shelby, Z. 2012. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2), 62.

[45] Kamp, P.-H. 2015. Http/2.0: the ietf is phoning it in. *Commun. ACM*, 58(3), 40–42.

[46] Shi, H., Chen, N., & Deters, R. 2015. Combining mobile and fog computing: Using coap to link mobile device clouds with fog computing. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, 564–571. IEEE.

[47] Kovatsch, M., Lanter, M., & Shelby, Z. 2014. Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the*, 1–6. IEEE.

[48] Daniel, L., Kojo, M., & Latvala, M. 2014. Experimental evaluation of the coap, http and spdy transport services for internet of things. In *International Conference on Internet and Distributed Computing Systems*, 111–123. Springer.

[49] Kendall, S. C., Waldo, J., Wollrath, A., & Wyant, G. A note on distributed computing. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1994.

[50] Roy, F. A little rest and relaxation. ApacheCon, 05 2008. URL: http://roy.gbiv.com/talks/200804_REST_ApacheCon.pdf.

[51] Richardson, L. Justice will take us millions of intricate moves. QCon, 2008. URL: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.

[52] Pautasso, C., Zimmermann, O., & Leymann, F. 2008. Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings*

of the 17th International Conference on World Wide Web, WWW '08, 805–814, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/1367497.1367606, doi:10.1145/1367497.1367606.

[53] Fielding, R. T. On software architecture. [updated 22.04.2018, cited 30.10.2017], 2008. URL: http://roy.gbiv.com/untangled/2008/on-software-architecture.

[54] Richardson, L. & Ruby, S. 2008. *RESTful web services*. O'Reilly Media, Inc.

[55] Fielding, R. T. & Taylor, R. N. 2002. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2), 115–150.

[56] Fielding, R. 09 2008. Rest apis must be hypertext-driven » untangled. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. (Accessed on 02.11.2017).

[57] Davis, C. 2012. What if the web were not restful? In *Proceedings of the Third International Workshop on RESTful Design*, 3–10. ACM.

[58] Fernandez, F. & Navón, J. 2010. Towards a practical model to facilitate reasoning about rest extensions and reuse. In *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, 31–38, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/1798354.1798383, doi:10.1145/1798354.1798383.

[59] Vinoski, S. March 2008. Demystifying restful data coupling. *IEEE Internet Computing*, 12(2), 87–90. doi:10.1109/MIC.2008.33.

[60] Freed, N., Klensin, J., & Hansen, T. Media type specifications and registration procedures. BCP 13, RFC Editor, January 2013. URL: https://tools.ietf.org/html/rfc6838.

[61] Vinoski, S. July 2008. Convenience over correctness. *IEEE Internet Computing*, 12(4), 89–92. doi:10.1109/MIC.2008.75.

[62] Waldo, J., Wyant, G., Wollrath, A., & Kendall, S. 1997. A note on distributed computing. In *Mobile Object Systems Towards the Programmable Internet*, 49–64. Springer.

[63] Mihindukulasooriya, N., Esteban-Gutiérrez, M., & García-Castro, R. 2014. Seven challenges for restful transaction models. In *Proceedings of the 23rd International Conference on World Wide Web*, 949–952. ACM.

[64] Pautasso, C. & Babazadeh, M. 2013. The atomic web browser. In *Proceedings of the 22nd International Conference on World Wide Web*, 217–218. ACM.

[65] da Silva Maciel, L. A. H. & Hirata, C. M. 2009. An optimistic technique for transactions control using rest architectural style. In *Proceedings of the 2009 ACM symposium on Applied Computing*, 664–669. ACM.

[66] Shang, W., Yu, Y., Droms, R., & Zhang, L. Challenges in iot networking via tcp/ip architecture. Technical report, NDN Project, Tech. Rep. NDN-0038, 2016.

[67] Shelby, Z., Koster, M., Bormann, C., & der Stok, P. V. Core resource directory. Internet-Draft draft-ietf-core-resource-directory-05, IETF Secretariat, October 2015. http://www.ietf.org/internet-drafts/draft-ietf-core-resource-directory-05.txt. URL: http://www.ietf.org/internet-drafts/draft-ietf-core-resource-directory-05.txt.

[68] Saint-Andre, P. Extensible messaging and presence protocol (xmpp): Core. RFC 6120, RFC Editor, March 2011. URL: http://www.rfc-editor.org/rfc/rfc6120.txt.

[69] Stanik, A. & Kao, O. 2016. A proposal for rest with xmpp as base protocol for intercloud communication. In *Information, Intelligence, Systems & Applications (IISA), 2016 7th International Conference on*, 1–6. IEEE.

[70] Khare, R. & Taylor, R. N. 2004. Extending the representational state transfer (rest) architectural style for decentralized systems. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 428–437. IEEE.

[71] Khare, R. *Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems*. PhD thesis, University of California, Irvine, 2003.

[72] Liskin, O., Singer, L., & Schneider, K. 2011. Teaching old services new tricks: adding hateoas support as an afterthought. In *Proceedings of the Second International Workshop on RESTful Design*, 3–10. ACM.

[73] Alarcón, R. & Wilde, E. 2010. Restler: crawling restful services. In *Proceedings of the 19th international conference on World wide web*, 1051–1052. ACM.

[74] Zou, J., Mei, J., & Wang, Y. 2010. From representational state transfer to accountable state transfer architecture. In *Web Services (ICWS), 2010 IEEE International Conference on*, 299–306. IEEE.

[75] Hüffmeyer, M. & Schreier, U. 2016. Restacl: An access control language for restful services. In *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, 58–67. ACM.

[76] Field, J. P., Graham, S. G., & Maguire, T. 2011. A framework for obligation fulfillment in rest services. In *Proceedings of the Second International Workshop on RESTful Design*, 59–66. ACM.

[77] Kübert, R., Katsaros, G., & Wang, T. 2011. A restful implementation of the ws-agreement specification. In *Proceedings of the Second International Workshop on RESTful Design*, 67–72. ACM.

[78] Kelly, M. Json hypertext application language. Internet-Draft draft-kelly-json-hal-08, IETF Secretariat, May 2016. URL: http://www.ietf.org/internet-drafts/draft-kelly-json-hal-08.txt.

[79] Ueda, T., Nakaike, T., & Ohara, M. 2016. Workload characterization for microservices. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, 1–10. IEEE.

[80] Morabito, R., Kjällman, J., & Komu, M. 2015. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, 386–393. IEEE.

[81] Merkel, D. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2.

[82] Reese, W. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173), 2.

[83] Soni, R. *Nginx Core Architecture*, 97–106. Apress, Berkeley, CA, 2016. doi:10.1007/978-1-4842-1656-9_5.

[84] Kovatsch, M., Mayer, S., & Ostermaier, B. 2012. Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, 751–756. IEEE.

[85] Storø Nyfløtt, M. An exploratory approach towards using coap for inter-service communication among microservices. IMT5251: Advanced Project Work, 12 2016.

[86] Jeremy, M. Java concurrency (&c): Why many profilers have serious problems (more on profiling with signals). https://jeremymanson.blogspot.

no/2010/07/why-many-profilers-have-serious.html. [Accessed on 10.11.2017].

[87] Storø Nyfløtt, M. 11 2017. Performance improvements to californium-proxy by martinmine · pull request #479 · eclipse/californium. https://github.com/eclipse/californium/pull/479. [Accessed on 07.12.2017].

[88] Williams J, M. Java garbage collection basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. [Accessed on 07.11.2017].

[89] Stephen, C. There is no thread. https://blog.stephencleary.com/2013/11/there-is-no-thread.html. [Accessed on 17.11.2017].

A List of frameworks and servers

nginx	1.13.7
nghttp2	1.12.0
Californium	1.0.6
httpasyncclient	4.1.3

Table 4: Servers, frameworks and versions being used.

B nginx1 usage data

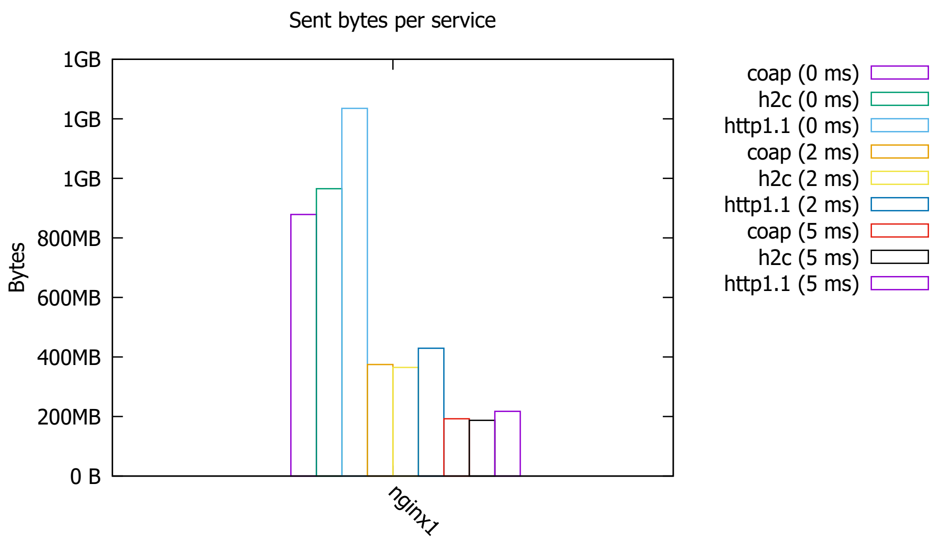


Figure 20: Amount of bytes sent in nginx1.

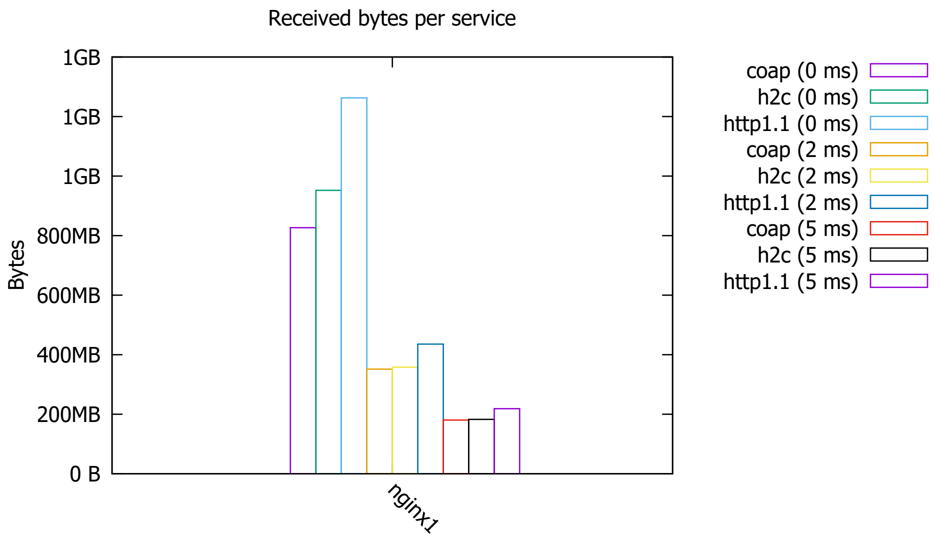


Figure 21: Amount of received bytes in nginx1.

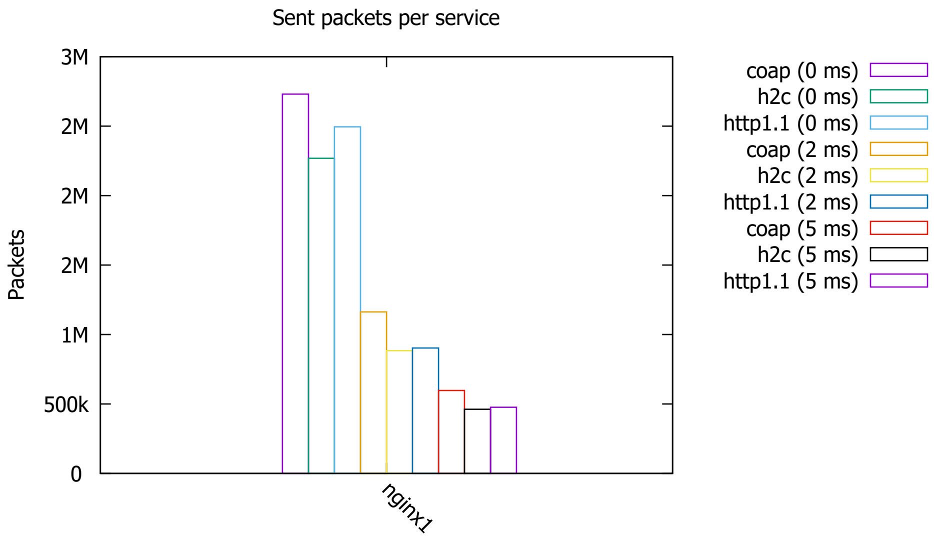


Figure 22: Amount of sent packets in nginx1.

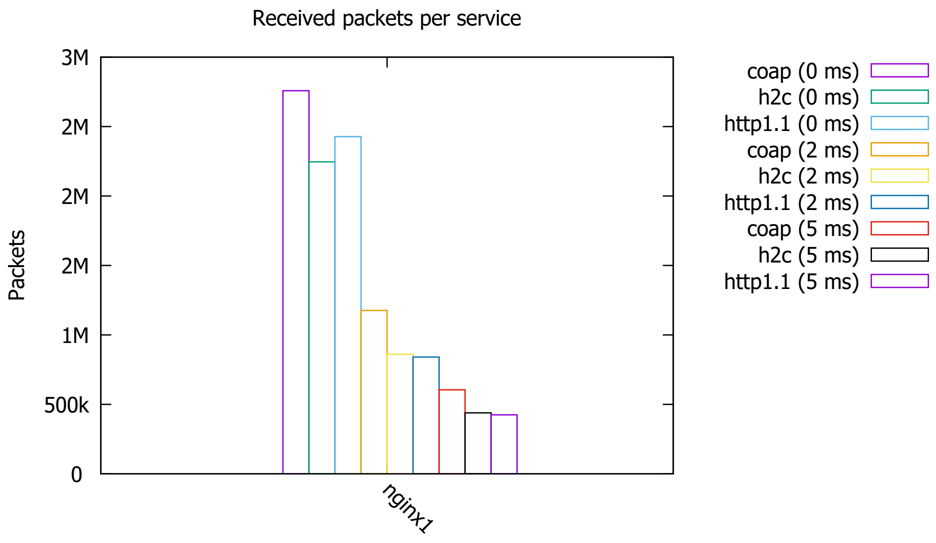


Figure 23: Amount of received packets in nginx1.