CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluating RPC for Cloud-Native 5G Mobile Network Applications

Master's thesis in Computer science and engineering

Rasmus Johansson, Hanna Kraft

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Evaluating RPC for Cloud-Native 5G Mobile Network Applications

Rasmus Johansson, Hanna Kraft

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Evaluating RPC for Cloud-Native 5G Mobile Network Applications

Rasmus Johansson and Hanna Kraft

Supervisor: Romaric Duvignau, Department of Computer Science and Engineering
Advisor: Maysam Mehraban, Ericsson
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Evaluating RPC for Cloud-Native 5G Mobile Network Applications

Rasmus Johansson
Hanna Kraft
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis investigates the communication between services in 5G network functions. The development of the *5G Core* (5GC) is by design increasing the amount of communication needed in the control plane. The reason for this is the migration to the cloud and the adoption of a microservices architecture. The telecommunications domain sets strict requirements on performance, which implies the need for the implementation of inter-service communication to be carefully constructed. This thesis evaluates the use of *Remote Procedure Call* (RPC) as inter-service communication in a 5GC network function. The purpose is to evaluate whether RPC frameworks will fulfill the requirements of inter-service communication and the strict requirements on telecom applications. The frameworks evaluated are gRPC and Apache Thrift. We also compare the frameworks to a TCP solution since this is the approach currently considered for this use case and a solution with minimal overhead to the communication. The evaluation is both quantitative, with benchmarks on latency, throughput and CPU usage, and qualitative where qualities such as availability and ease of development are evaluated. From the evaluation, we can conclude that using RPC frameworks would suit most needs. Even if the evaluated RPC frameworks perform slightly worse than a reference TCP solution in the quantitative evaluation, they can provide many other benefits such as bidirectional streaming RPC and high-availability features. Among the evaluated RPC frameworks, Apache Thrift stands out slightly in terms of performance, while gRPC stands out in the qualitative evaluation.

Keywords: RPC, inter-service communication, 5G, 5G Core, Network Function, Microservices, Cloud-Native.

Acknowledgements

We would like to thank our supervisor **Romaric Duvignau** and our examiner **Vincenzo Massimiliano Gulisano**. We would also like to thank our advisor at Ericsson, **Maysam Mehraban** and our manager at Ericsson **Marcus Oscarsson**.

Rasmus Johansson and Hanna Kraft, Gothenburg, November 2020

Contents

List of Figures

Acronyms

3GPP *Third Generation Partnership Project.* 5
5GC *5G Core.* v

AMF *Access and Mobility management Function.* 5
API *Application Programming Interfaces.* 1

CNCF *Cloud Native Computing Foundation.* 6
COTS *Commercial-Off-The-Shelf.* 5

GUAMI *Globally Unique AMF Identifier.* 6
GUTI *Globally Unique Temporary Identifier.* 3

IDL *Interface Definition Language.* 10

NF *Network Functions.* 1
NFV *Network Function Virtualization.* 5

QPS *Queries Per Second.* 18

RAN *Radio Access Network.* 5
REST *Representational State Transfer.* 7
RPC *Remote Procedure Call.* v

SBA *Service-Based Architecture.* 1
SBI *Service-Based Interfaces.* 6
SDN *Software-Defined Networking.* 5

TMSI *Temporary Mobile Subscriber Identity.* 6

UE *User-Equipment.* 3

1

Introduction

5G is the new generation of mobile networks. 5G will improve the efficiency and performance of regular smartphone users, and enable new technologies such as autonomous vehicles, and increase the potential of IoT. Furthermore, Ericsson expects that mobile data traffic will expand by a factor of eight by 2023 [9], which will require mobile networks to enable lower latency and at the same time higher capacity, allowing for more network traffic. In the 5G standard, the packet core, 5GC, is migrated from the previous generation's monolithic architecture to a cloud-native *Service-Based Architecture* (SBA), consisting of decoupled applications called *Network Functions* (NF). Each NF can be implemented as several *microservices*. The microservices that make up an NF need to communicate with each other, which introduces extra delay compared to the previous generation of networks. The microservices architecture also introduces many *Application Programming Interfaces* (API)s, and the need for maintaining these can quickly become cumbersome. Furthermore, features such as upgradability, scalability, and backward-compatibility are essential for microservices applications and need to be handled efficiently.

For ease of development, it could be beneficial to adopt a general third-party communication framework for the *inter-service communication* of the NFs rather than to use a legacy solution or to develop a framework from scratch. Moreover, a third-party framework built for use in a cloud-native environment could bring relevant technologies needed to fulfill many of the requirements set on 5G. Although it could potentially increase the productivity of developing microservices, the framework might not have been built with the strict performance requirements of the 5G domain in mind, as they are generally built for the web-scale domain.

This thesis consists of quantitative and qualitative research methods to evaluate third-party RPC frameworks as inter-service communication in the NFs of the 5GC. This thesis aims to assess whether a third-party framework can comply with the strict requirements of 5G. We have chosen two RPC frameworks to evaluate, gRPC, and Apache Thrift.

Since the area of cloud-native applications in 5GC is novel, there is not yet any standard for inter-service communication within an NF. Besides, there is limited research available, and preliminary results are not always validated thoroughly. Therefore, our results could potentially be of great interest to anyone integrating inter-service communication between microservices for applications with similarly strict require-

ments on performance.

1.1 Problem description

This thesis is a collaboration with Ericsson, who is migrating the packet core to the cloud and upgrading it to follow 5G standards. One crucial design principle for cloud-native applications, according to Ericsson, is to follow a microservices architecture [3]. Microservices need to communicate with each other through inter-service communication, which introduces additional delay and overhead to the application. Adopting a microservices architecture also introduces new APIs that need to be maintained. Furthermore, deployed microservices can be upgraded independently of each other, meaning communication needs to be backward-compatible. Due to the nature of microservices, software needs to be scalable, and allow for more throughput than before. As 5GC becomes cloud-native, there are even more demands in place. Mainly, the inter-service communication for 5GC NFs has strict requirements on latency.

This thesis aims to evaluate if a third-party RPC framework is suitable as inter-service communication in a 5GC NF, taking into account the high demands presented above. The full list of evaluation requirements are described in Section 4.1. We are investigating this subject on behalf of Ericsson, who wants to find a simpler solution than writing a custom communication interface, while at the same time not losing too much performance. There is currently no standard for inter-service communication in 5GC NFs, and there is little or no research on the subject.

1.2 Novelty

Several studies investigate inter-service communication, some also in a 5G setting, such as in the papers of Kempf et al. [22], Zhang et al. [34], and Buyakar et al. [12]. However, no previous work has consisted of a qualitative and quantitative evaluation of different RPC frameworks in 5G. Our work evaluates RPC frameworks as inter-service communication and compares the frameworks to each other and a reference solution based on TCP. Our thesis contributes with a thorough evaluation of the performance of single-request gRPC and Apache Thrift, as well as bidirectional streaming gRPC. This thesis also provides a qualitative comparison of design styles and the design complexity of the frameworks.

1.3 Limitations

Due to the limited scope of the thesis, RPC is the only type of communication explored, even though many different protocols are potentially usable for this use case. This limitation is also due to several qualities of RPC, which we believe make it very suitable for fulfilling the requirements.

This thesis only evaluates two RPC frameworks due to time constraints, gRPC and

Apache Thrift. Furthermore, the evaluation performed in this thesis focus on a specific use-case in the 5GC, namely the process of allocating a 5G *Globally Unique Temporary Identifier* (GUTI) for a *User-Equipment* (UE).

1.4 Research questions

This thesis attempts to answer the following research questions:

1. Is it possible to fill all the demands on communication between microservices in a cloud-native 5GC NF using a general RPC framework?
2. Is the performance of a cloud-native 5GC NF high enough if implementing inter-service communication using a general RPC framework?

1.5 Organization of the thesis

The remainder of the thesis is organized as follows: Chapter 2 introduces background information such as cloud-native applications, state-of-the-art RPC frameworks, as well as the 5GC and its enabling techniques. Chapter 3 describes related work to this thesis. Chapter 4 describes in detail the assessment criteria and system used in the evaluation as well as the implementation needed to integrate RPC frameworks into a prototype application. The results are presented in Chapter 5, and discussed in Chapter 6. Finally, Chapter 7 consists of concluding remarks of the thesis.

2

Background

This chapter contains the necessary background knowledge needed to comprehend the work presented in this thesis. Section 2.1 includes an overview of the control plane and NFs in 5G. Section 2.2 and Section 2.2.1 describe cloud-native applications and microservices respectively. Furthermore, Section 2.3 includes information on asynchronous and synchronous communication, and finally, Section 2.4 consists of background on RPC, as well as the RPC frameworks evaluated in the thesis.

2.1 5GC

The 5GC is the packet core of the 5G system. Packet core is the core network that connects the *Radio Access Network* (RAN) and external access network, i.e., the Internet. Some of the packet core's functions include mobility as well as session management. Mobility management is a responsibility of the *Access and Mobility management Function* (AMF) NF. It handles the connection of a geographically moving UE, e.g., a smartphone, connecting to different radio base stations in the RAN. In contrast, session management maintains a session towards the UE. Moreover, the 5GC has functions for networking, such as packet-forwarding rules and deep packet inspection.

Two critical enablers for the 5GC, moving from the previous generation's packet core, are *Software-Defined Networking* (SDN) and *Network Function Virtualization* (NFV). SDN is used for the separation of control-signaling functions (control plane) and packet-processing functions (user plane) of the packet core, allowing them to deploy, and thus scale separately [22, 30]. NFV is used for the virtualization of NFs, allowing them to migrate from expensive dedicated hardware to *Commercial-Off-The-Shelf* (COTS) hardware in the cloud [30]. Using these technologies, SDN and NFV, together with cloud-native technologies, *Third Generation Partnership Project* (3GPP) has defined the next generation packet core, the 5GC, to be of a cloud-native SBA [33]. SBA is a type of software architecture which focuses on the use of services.

5GC Control Plane

The 5GC's control plane's function is to manage all the control signaling required for serving UEs. The control plane consists of several NFs, which are defined and

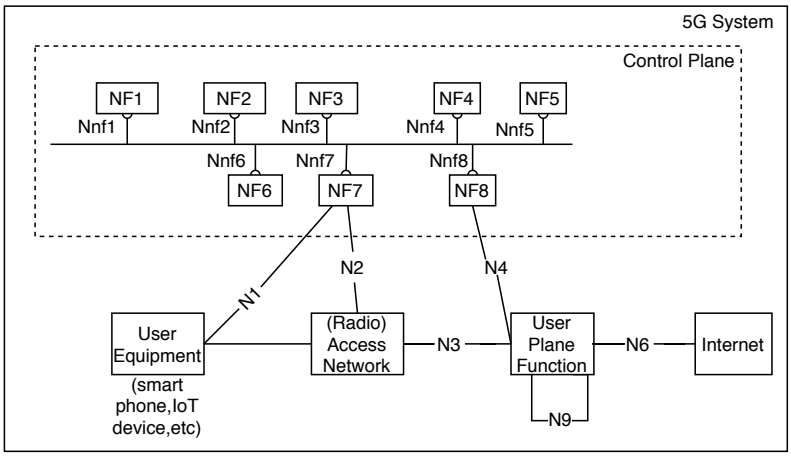


Figure 2.1: The 5GC and its control plane.

standardized by the 3GPP. Figure 2.1 gives an overview of the 5GC control plane's architecture. All NFs are crucial to have a working control plane. *Service-Based Interfaces* (SBI) are defined and standardized for how NFs communicate with each other [33] and are referenced to as Nnfx (e.g., Nnf1) in Figure 2.1. Each NF is, in turn, implemented as microservices that provide the functionality of the NF, see Figure 2.2.

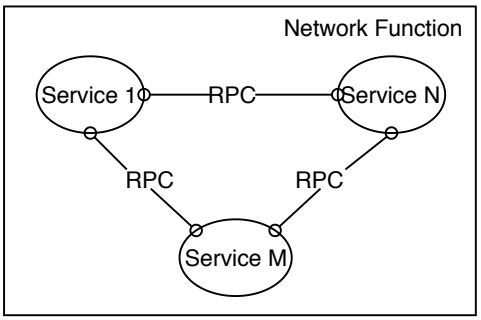


Figure 2.2: Network Function architecture, the circles are microservices.

5G GUTI

A 5G GUTI consists of two parts, *Globally Unique AMF Identifier* (GUAMI), and *Temporary Mobile Subscriber Identity* (TMSI). The GUAMI identifies one or several AMFs from a set, while the TMSI identifies the UE within the AMF. The GUTI's purpose is to provide a UE with a unique identity in the network. The AMF NF is responsible for allocating the GUTI [33].

2.2 Cloud-Native Applications

The *Cloud Native Computing Foundation* (CNCF) defines cloud-native as: "technologies [that] empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach" [4].

There are several benefits of adopting a cloud-native architecture: better performance, higher efficiency, and scalability features such as load-balancing and automatic scaling [23]. Automatic scaling means that more resources are allocated to a process when needed. Automatic scaling can ensure that applications keep on running when suddenly experiencing a substantial increase in traffic. Load balancing means that workload is split over machines so that one or a few machines are not overloaded with work.

2.2.1 Microservices

Microservices are both a type of software architecture, and a term meaning services, usually running in a cloud-native environment. Dragoni et al. define a microservice as *"a cohesive, independent process interacting via messages"* [13]. Microservices architecture can be very beneficial in sizable applications, for example, when it comes to upgrading or scaling. When having several small services, one or a few pieces of software can be upgraded at a time, which reduces, or entirely removes downtime for the application [28]. Furthermore, developers can usually deploy two copies of a service of different versions simultaneously, to test out new features. Microservices also give developers the possibility of adjusting requirements on each microservice, rather than for the entire application. This could mean that different technologies are used for different parts of the application, potentially improving performance.

A common way to run microservices is to use containers, such as Docker [5]. Container orchestration systems such as Kubernetes [7] are used to manage them. In Kubernetes, a smaller group of containers is called a pod [8]. Orchestration systems provide many different services, such as health monitoring and scheduling. These containers communicate with each other and internally through inter-service communication. Inter-service communication comes in many different forms, where some of the most common ones are *Representational State Transfer* (REST), RPC, and message queues.

2.3 Asynchronous Function Calls

Asynchronous function calls are function calls that are performed without the calling thread waiting for the function to complete its execution. As a regular function call executes in the calling thread, a function called asynchronously must be executed in a separate thread, allowing the calling thread to continue its program execution.

Asynchronous function calls can be achieved on the level of the programming language using a keyword to the function signature or similar, on the level of a library, for example wrapping the function in an asynchronous object, or on the level of the function itself, implementing it in a way to facilitate an asynchronous behavior.

In the typical case where the calling thread eventually relies on the result of the asynchronously executed function, some synchronization mechanism is needed for the calling thread to access the function's result. One such mechanism is a `promise`

connected to a `future`. The promise and future together form a shared state between the calling thread and the asynchronous function [24]. The function provides the calling thread with a promise that a value in the shared state will be set eventually. The calling thread can initiate a future object from the promise provided by the asynchronous function, creating a data channel between the function and the calling thread. When the calling thread needs the promised result, it will wait on the future object, sleeping, until the function sets the promised value as well as wakes up the calling thread, and the calling thread can access the promised value. Instead of sleeping, it can also check the state of the future object, and continue doing other work while waiting.

2.4 RPC

Bruno Nelson defined RPC in his dissertation on the subject as *"the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel"* [27]. Essentially, RPC is a mechanism that enables a program to invoke a function(procedure/method) in another program. The goal of RPC is for a call to have the same semantics as if it was a local function call [27]. Since Nelson's dissertation on RPC, the definition of RPC has relaxed to exclude the requirement on both the type of underlying communication medium and that of RPC calls to be synchronous [32].

Figure 2.3 illustrates the general process of an RPC call. The client application invokes an RPC method available in the client stub with input parameters (1). The stub *marshalls* (packs) the method name and parameters into a request message and passes it to the RPC library run-time (2). The run-time performs some internal bookkeeping before writing it to the underlying transport (3). The client sends the message over the wire to the server (4), where the server's RPC library reads it from the transport and reconstructs the message (5). The message is *unmarshalled* (unpacked) and passed on to the server stub (6), which invokes the method, implemented in the server application, with input parameters from the request (7). The method's return value is *marshalled* into a response message (8) by the server stub, which the server stub passes down to the run-time (9). The response message is then transferred to the client stub (10-13) in the same way as the request message was transferred to the server-stub. The client stub unmarshalls the response message into the return value of the RPC method and returns it to the client application (14), finishing the RPC.

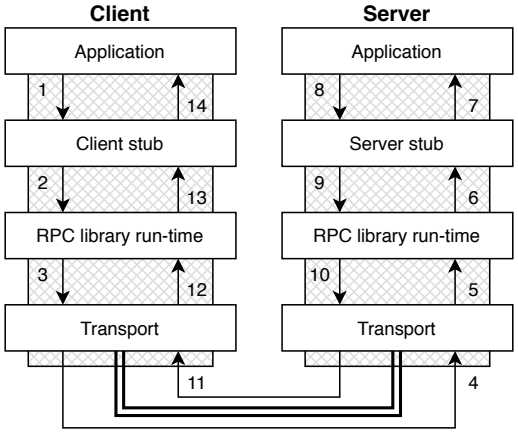


Figure 2.3: The process of an RPC.

An RPC function can be either synchronous or asynchronous. The standard procedure is synchronous RPC, which means that the client is blocked during the RPC call, waiting for the RPC return. However, this is not feasible in many cases, as the latency of an RPC call is in orders of magnitude larger than a local function call, leaving the client blocked for a very long time. By having an asynchronous RPC, the client can invoke the RPC call, without getting blocked, and retrieve the return value at a later point in time, when the value is needed. During the time of the RPC call, it can do other processing. Asynchronous RPC can be naively implemented on top of a synchronous RPC method, using asynchronous primitives, as described in Section 2.3. However, this would not scale very well with many concurrent RPC calls, as each RPC call would spawn a new thread. Therefore, it is more beneficial to have an RPC system with proper built-in functionality for asynchronous RPCs.

Motivation for RPC

Using RPC has several advantages. The main reason for investigating RPC for this thesis is that RPC abstracts many underlying mechanics behind communication, meaning that a developer can instead focus on the functionality of the application rather than the communication itself [11]. Furthermore, the API design philosophy of RPC is well suited for communication between microservices. The simple implementation of RPC could also make development more efficient and code less complicated. Moreover, the simplicity of RPC makes it very efficient [11].

Comparison of RPC and REST

There are several options for inter-service communication, and all alternatives have their advantages and disadvantages, and no solution will be optimal for all microservices applications. An alternative inter-service communication protocol could be REST. REST is widely adopted an API, where the operations on a resource are limited to the HTTP verbs such as GET, PUT, and DELETE. This API style may become a limitation for an inter-service communication where the purpose of the communication is to access another microservice's function, rather than its re-

sources. Some benchmarks also point to the conclusion that RPC could be more efficient than REST [12].

An essential principle of REST is that each request shall contain all information regarding the session, making the server stateless towards the client session. At this stage, we cannot tell if this is tolerable in all aspects of the 5GC control plane. Therefore, it might be safer to go for an RPC solution that does not set this requirement on the server. Inter-service communication is generally used by a microservice to access another microservice's functionality, which suits an operations-focused communication protocol such as RPC well. Since this thesis considers inter-microservice communication within a product's internal architecture, we also believe that RPC design-wise is more suitable since an RPC call aligns well with the flow of the program. If we instead would have an application that is available for external entities, one might choose REST for an external API due to its well-defined API principles.

2.5 RPC Frameworks

An RPC framework is a set of tools that together implement RPC and enables developers to build RPC services. Using an RPC framework, a developer will commonly define services and messages they want to use with an *Interface Definition Language* (IDL). The RPC framework generates code based on the IDL definitions that the developer can use to define new clients and servers, which sends and receives RPC calls. We explore two different RPC frameworks in this thesis, gRPC, and Apache Thrift, which we describe in this section

2.5.1 gRPC

One of the most widely used RPC frameworks today is gRPC [16]. This framework is a former Google project which is currently hosted by the CNCF as an incubating project. gRPC provides low latency and is very well suited for developing cloud-native applications by design [15].

The protocol stack used in gRPC is HTTP/2 on top of TCP for transport, and *Protocol Buffers* (protobuf) for data serialization. gRPC uses protobuf's IDL for defining services and messages. It has built-in support for secure communication with authentication and encryption using TLS, as well as client-side load balancing policies. Furthermore, gRPC has support for integrating health checking into the server. Health checking means that the client can query the server for its health or status via a well-defined API. For example, this mechanism could be used by an external monitoring service to check the status of the server [18].

Developers define the API in a *.proto* file using protobuf as IDL, from which the gRPC compiler generates code for client and server stubs. The API consists of services and messages. A service is composed of a set of RPC methods as API endpoints, and a message is an entity of data structured in strongly typed numbered fields. A field can also be another message, creating nested messages. Protobuf provides backward-and forward-compatibility for the messages and services, how-

ever, with some inevitable limitations. This backward and forward compatibility is beneficial when a system runs clients and servers of different versions, which can occur when updates roll out gradually. When updating a message, some precautions are necessary in order to maintain back-compatibility. A new field cannot reuse the field number of a previously removed field. A field can change type if the new type is compatible with the current type. When an RPC endpoint reads a message and does not recognize some field, the RPC endpoint ignores the field. All fields are optional, so if an expected field is missing during serialization or deserialization, they are either set to zero or a specified default value.

gRPC implements streaming by using HTTP/2 streams. A bidirectional stream allows a single RPC method to consist of an arbitrary number of request messages and response messages, sent and received in any order.

The run-time of gRPC uses a `completion queue` to convey the state of the RPCs to the application to provide asynchronous RPC calls. When the application invokes an RPC operation, it needs to provide the operation with a unique tag. The tag is pushed to the completion queue when the gRPC run-time has completed the operation. The application queries the completion queue for a tag using the `Next` or `AsyncNext` method on the completion queue and thus know when an operation finishes. `Next` is a blocking method, and the gRPC borrows the calling thread for the processing of RPCs until a tag is pushed to the completion queue. With `AsyncNext`, a timeout can be set to limit the time that gRPC may borrow the thread. If no thread is performing `Next` or `AsyncNext`, the RPCs will not be processed, and no progress will be made.

With asynchronous single-request mode, gRPC offers an API for sending the request and receiving the response. The application invokes an RPC with the completion queue and the request as input parameters. The application informs the gRPC run-time where to store the response, and what to tag the completed operation (RPC) with. When the response is received, the gRPC run-time pushes the tag to the completion queue, from which the application can read the tag.

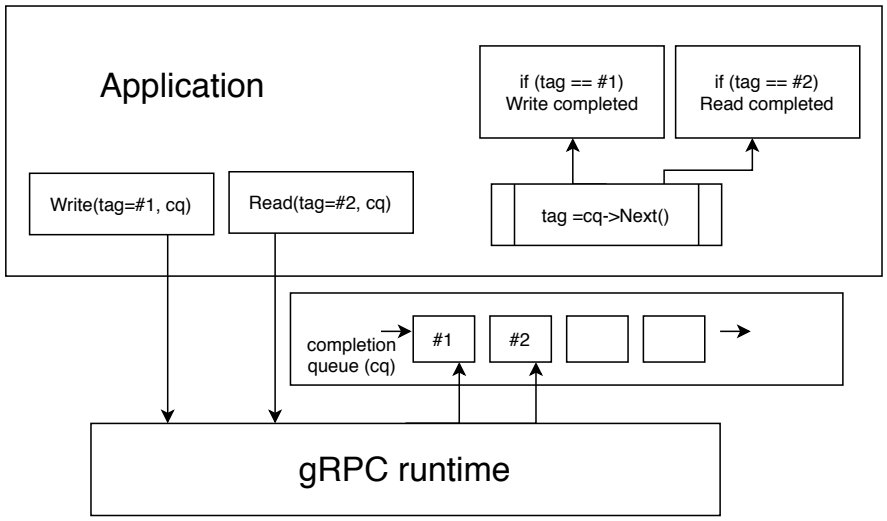


Figure 2.4: Example of asynchronous bidirectional gRPC.

gRPC's API for asynchronous bidirectional streaming is similar to, but more complex than the asynchronous single-request API, as illustrated in Figure 2.4. The application sends or receives a message by invoking a read or write operation with a unique tag on a stream. The gRPC run-time will notify the application on completion of the operation by adding the unique tag to the completion queue. The application continuously polls the completion queue for a tag, which returns when a tag is added to the completion queue by the gRPC run-time. A limitation set on the completion queue by gRPC is that there may only ever be at most one read and one write per stream issued by the application at any point in time.

When using the synchronous mode, the completion queue is unexposed to the application, as opposed to asynchronous mode. There is no need for this since the RPC call blocks the application while waiting for a response.

2.5.2 Apache Thrift

Apache Thrift, henceforth referred to simply as Thrift, is a framework that combines serialization and code generation for RPC [1]. Thrift generates API code by using an IDL to define data types and services in a *.thrift* file. The IDL used for Thrift is heavily influenced by C in its syntax and uses types such as structs to define messages and objects [31]. To enable upgradability and backward-compatibility, Thrift supports reading data from clients that are of older versions than the server. Thrift handles versioning by using field identifiers, which encodes field headers in Thrift structs.

Thrift has both single-threaded servers and multi-threaded servers. The simplest kind of server is the `TSimpleServer` which uses one thread all in all, but can only serve a single client at a time. `TThreadedServer`, `TThreadpoolServer` and `TNonblockingServer` can all use multiple threads. `TThreadedServer` and `TThreadpoolServer` use one thread per client. The `TThreadpoolServer` has a fixed-size pool of threads and reuses threads for new clients, while `TThreadedServer` destroys threads when clients disconnects and creates new threads for new clients [10]. The `TNonblockingServer` uses one or several threads dedicated to IO and can use a thread pool for the processing of incoming RPCs. If a thread pool is used, the IO threads distribute incoming RPCs among these threads for processing. If not, then the IO thread itself serves the RPC. A single IO thread can serve multiple client connections. It uses the library `libevent` to receive notification of incoming data on multiple client connections' file descriptors simultaneously.

Thrift servers use a `TProcessor` that reads and writes data from the wire. Processors can be either synchronous and asynchronous. Thrift supports several different transport protocols for data transport, not only TCP sockets [10]. Furthermore, Thrift also supports several different serialization protocols. Binary serialization can be utilized to gain speed, while compact serialization can be used to instead get as compact a message as possible.

Thrift provides an asynchronous `TEvhttpServer`, asynchronous `TEvhttpClientChannel` and `TAsyncChannel` for some languages. The TEvhttpServer

needs to be initialized with an asynchronous `TProcessor`, which is generated by the Thrift compiler. The asynchronous client uses `TEvhttpClientChannel`, which extends the `TAsyncChannel` class with the use of the libevent library's `evhttp` API to make HTTP requests to the server. The client assigns a callback function to each RPC, which is called when the response has returned from the server. The client is reliant on the application to provide the client with an `event_base` from the libevent library, and to run libevent's `event_base_loop` on the `event_base`. Each RPC call registers an event on the `event_base` which gets processed in the `event_base_loop`.

3

Related Work

This chapter covers previous work in areas related to this thesis, such as mobile networking and cloud-native architecture. Furthermore, this section highlights the differences between the previous work and this thesis.

Kempf et al. describe how to theoretically move the evolved packet core to the cloud using SDN in their work Moving the mobile evolved packet core to the cloud [22]. This solution includes modifying OpenFlow to separate the control plane and the user plane, making it possible to deploy control plane in the cloud, separate from the packet-processing functions in the user plane. This work does not implement or evaluate RPC framework but theorizes on the potential in using RPC as a candidate for communication in the proposed architecture.

In Performance evaluation of candidate protocol stack for service-based interfaces in 5G core network [34], Zhang et al. propose a protocol stack for the SBIs of the 5GC by individually comparing several different properties, both quantitatively and qualitatively. These properties include API design styles, as well as data serialization formats. The work of Zhang et al. is similar to the work presented in our thesis, as it also concerns cloud-native 5G and RPC communication, however the SBIs are external interfaces towards other NFs, which results in other requirements on the API design compared to the interfaces used for internal communication within an NF. Zhang et al. provide a qualitative comparison of RPC to REST, based on API design style. However, Zhang et al. only consider RPC as communication between network functions, and not as inter-service communication. Moreover, their study does not compare or mention RPC frameworks or perform benchmarks on performance of any RPC framework.

Buyakar et al. have built a prototype of 5G SBI and SBA and evaluated it with regards to latency and CPU usage in their work Prototyping and Load Balancing the service based architecture of 5G core using NFV [12]. Buyakar et al. used open-source tools to prototype SBA and deployed it in a network function virtualization environment. Their work compares the latency and CPU usage of gRPC and REST and, based on the evaluation, they chose to implement gRPC as SBI for the prototype. Their work differs from ours in that gRPC is used as SBI rather than inter-service communication. Buyakar et al. compare gRPC to REST, while we compare gRPC to TCP and Thrift. Our work also provides a more rigorous evaluation.

Nguyen et al. evaluate the performance of gRPC and Thrift as communication between microservices in Benchmarking performance of data serialization and RPC frameworks in microservices architecture: gRPC vs. Apache Thrift vs. Apache Avro [29], similar to our thesis. This work, however, does not involve 5G and thus concerns very different requirements.

Manso et al. demonstrate a cloud-native SDN controller for control of transport network in their work Cloud-native SDN controller based on micro-services for transport Networks [26]. The SDN controller is implemented as multiple microservices communicating via RPC, namely gRPC. While it is not strictly within the telecommunications domain, it is still architecturally similar to the control plane of the 5GC, with the similar requirements from the cloud-native domain. Manso et al. chose gRPC due to it being a modern framework built for the cloud-native domain. However, Manso et al. do not present any comparison to alternative RPC frameworks, nor do they perform any further evaluation on the performance impact of using gRPC compared to other candidates.

Hawilo et al. describe the challenges of a microservices architecture as the platform for NFV in Exploring microservices as the architecture of choice for network function virtualization platforms [20]. This article brings up communication in virtualized network functions, but on another level than in our thesis, and in this regard mainly focuses on reducing latency while also fulfilling demands on placement of virtual network function components. While our thesis investigates the inter-service communication to find the impact that RPC frameworks have on communication, Hawilo et al. target the same problem, inter-service communication, but on platform rather than application level. By minimizing the network path delay between communicating entities, they lower the latency of inter-service communication.

In 5G enhanced service-based core design [25], Lu et al. propose a new SBA design which is called Not-only-stack. In the Not-Only-stack design, each NF consists of a server-processing entity and Sidecar. The server-processing entity handles logic while the Sidecar handles communication and cloud-native functionality such as load balancing. The Not-only-stack design was created to simplify inter-service communication in network functions, and is presenting a different solution to the issue at hand in our thesis.

Gal and Delimitrou highlight the impact that a microservice architecture has on the ratio between application and communication processing, compared to a monolithic architecture. Their evaluation shows that up to 70% of time is used for communication processing in an application implemented in a microservice architecture, compared to 41% for a monolithic implementation [14]. The inter-service communication in the application based on microservices uses RPC, and their results highlight the need for efficient communication when moving from a monolithic architecture towards a microservice architecture. Their findings are interesting as our thesis investigates the inter-service communication of an NF based on a microservice architecture, software that has been migrated from monolithic architecture in earlier generations of mobile networking.

4

Methods

In this thesis, we use a prototype application that simulates a 5GC application to measure the performance and other aspects of two different RPC frameworks. We do this by integrating and evaluating several different communication *adapters* into a 5G prototype application. In this case, we define an adapter as a client and server that integrates a specific framework or transport protocol, and mode of communication. The prototype application used initially has a simple asynchronous communication solution that uses TCP sockets to send and receive data. We refer henceforth to this adapter as the *TCP-adapter*. We have added new server and client classes to the prototype application, which use gRPC and Thrift RPC frameworks.

The rest of this chapter is organized in the following way: Section 4.1 describes the assessment criteria and system used for the evaluation, which includes a description of the prototype application on which the benchmarks are run. This section also contains a description of the communication of the prototype application at the start of the thesis, the TCP-adapter. Section 4.2 describes how we chose the RPC frameworks used in this thesis. Section 4.3 consists of a description of the RPC frameworks.

4.1 Assessment Criteria and System Model

This section describes the assessment criteria and system used for the evaluation. The evaluation consists of a qualitative as well as a quantitative evaluation. Subsection 4.1.1 describes the qualitative properties, and subsection 4.1.2 describes the quantitative properties of the assessment criteria. Finally, subsection 4.1.3 describes the system used for the quantitative evaluation.

4.1.1 Assessment Criteria for Qualitative evaluation

The qualitative evaluation evaluates the adapters based on the properties listed below. The properties were evaluated based on available features of the RPC frameworks and TCP-adapter.

1. **High-availability features:** features that can benefit applications in a cloud-native setting on being continuously available.

2. **Backward-compatibility:** interoperability between services of different version.
3. **Cross-language support:** support for multiple languages.
4. **Bidirectional streaming RPC:** allow a single RPC method to contain several RPC requests and responses.
5. **Asynchronous communication:** sending and receiving messages without the calling thread blocking until a response is received.
6. **Secure communication through TLS:** authentication and encryption using TLS.

4.1.2 Assessment Criteria for Quantitative Evaluation

The following properties were evaluated in the quantitative evaluation:

1. **Latency:** the time it takes for a request registered on the client to reach the server and get a response.
2. **Tail latency:** the 99th percentile of latency, i.e., the 1 % of messages with the highest latency.
3. **Throughput:** measured as both the number of successful *Queries Per Second* (QPS) and bytes per second.
4. **CPU usage:** measured for the server during single-client evaluation.

4.1.3 System Model

To run benchmarks in an environment that simulates cloud-native 5G, we used a system consisting of a client-server architecture with the server being a 5GC prototype application. Henceforth we refer to this 5GC prototype application as the *GUTI-prototype*. The GUTI-prototype simulates the process of allocating a 5G GUTI while not being actual production code used in a real 5GC NF. The prototype has 2 API-endpoints, `Allocate` and `Deallocate`, however only Allocate is used in the evaluation. The former is for allocating a GUTI. Allocate takes an `AllocateRequest` object as input parameter, and the server returns an `AllocateResponse` object which contains a GUTI object. Deallocate performs the opposite operation; it takes a `DeallocateRequest` object containing the GUTI object that is to be deallocated and returns a `DeallocateResponse` object. For evaluation, the AllocateRequest and AllocateResponse objects also contain a field named `payload` that is a variable-length byte-array. Henceforth, the term **payload** refer to this field.

We integrated two different modes of generating requests for clients. The client can either send a large number of messages as fast as possible and measure the time it takes to send and receive all messages. We henceforth refer to this mode of operation as **no-rate** mode. The client can also use a rate, which means that a benchmark

runs for a predefined amount of time, in which it tries to send a specific amount of messages per second. We henceforth refer to this mode as **rate** mode. For example, a client using a rate of 10 and a set duration of 60 seconds will attempt to send ten messages per second for 600 requests. The rate mode runs in one of seven different rates: 10, 20, 50, 100, 200, 500, and 1000 requests per second. Different amounts of payload can be used in the benchmarks, from 0 B to 100 kB. The different payload sizes evaluated are 0, 10, 100, 1k, 10k, and 100k. When altering the payload, the `payload` field of the request and response messages are altered. With 0 B payload, a GUTI is still returned from the server.

The purpose of running benchmarks with different rates and payloads is to evaluate if adapters perform well for different use cases. It is interesting to know if a particular adapter performs very poorly in one specific use case. For example, if an adapter performs well for some use cases, but has a significant drop in performance for other use cases, it might not be the best option.

The client measures the latency of each RPC, i.e each request-response pair, using the `std::chrono::steady_clock` primitive of *C++*. It records the **maximum**, **minimum** and **mean latency** over the course of the benchmark. Moreover, it records the **throughput** and a histogram of the latency. We calculate mean latency by dividing the sum of the latencies of the requests by the number of requests. We calculate throughput as the total number of requests divided by the total duration of the benchmark. The histogram has 400 bins with a granularity of 50 μs. The 400th bin contains the number of recordings of latency that are 20 ms and above. In post-processing of the data, **median latency**, **tail latency** and **standard deviation** are calculated from the latency histogram. We calculate the standard deviation as the square root of the variance. CPU usage of the server process is also measured, using a *Bash* script running *top* in a loop on the server. Moreover, the underlying **mean network latency** between client and server is measured using *netperf* on the client and *netserver* on the server, illustrated in Figure 4.1.

The benchmarks are performed in a Kubernetes environment, running on a cluster of virtual nodes. The nodes are virtual machines running on the same hardware. We compile the server program into a docker image. The docker image runs in a docker container deployed in a Kubernetes pod on a node in the cluster, as can be seen in Figure 4.1. The client program is compiled in the same way as the server and deployed in a pod on another virtual node. We initialize the benchmarks with a warm-up phase. This means that the client starts sending requests in no-rate mode for a specified time before the actual benchmarking starts. It is possible to run benchmarks with several clients for all adapters, and all clients run in the same docker container. The hardware on which the virtual nodes run has 12 CPU cores. The server container is limited to use one core via Kubernetes, while the client container does not have such restriction. However, it is in practice limited to the capacity of the server.

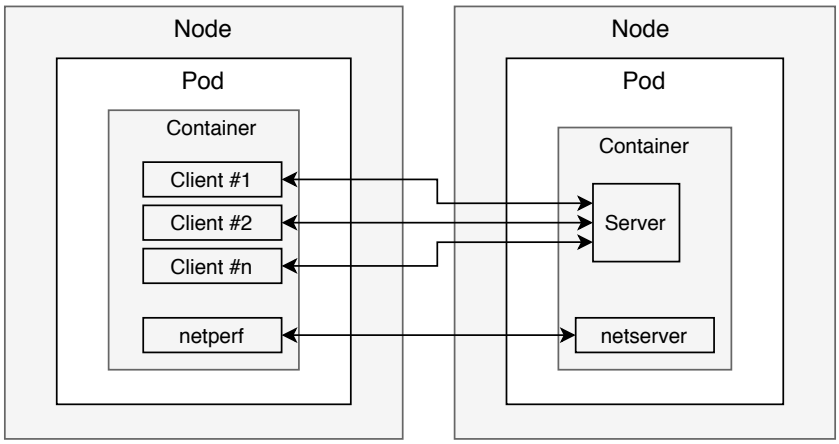


Figure 4.1: Overview of the system.

We evaluate the adapters with a single client as well as multiple clients. For the single-client benchmarks, we evaluate the adapters with every combination of rate and payload. In *rate* mode, the benchmarks run for 120 seconds and in *no-rate* mode, one million requests are performed. Multi-client benchmarks are run in *no-rate* mode with payloads of 0, 10 k and 100 kB. The amount of clients evaluated is 2, 4, 6, 8, 10, 12, 14, and 16 clients. The benchmark runs for 120 seconds, and each client starts at the same time. The recorded results of each client is combined to obtain a result that includes all clients. When running benchmarks with multiple clients, throughput can be measured with the server running at 100 % CPU utilization. Moreover, we can observe how the adapters scale with multiple clients.

TCP-adapter

The TCP-adapter has an asynchronous client, a blocking server, and uses TCP as the transport protocol. It uses a custom data serialization method that marshalls a message into a byte array containing a fixed-length header and a variable-length body. The header consists of a message type, a request tag, and the message length. The body consists of the message, i.e., the payload in case of a request message. The body also contains a GUTI and the payload in case of a response message. We represent the payload as a byte array and the GUTI as a C struct.

We implement the client-side of the adapter with two threads: the main thread from which the application sends a request to the server, and another thread for reading responses from the TCP socket. A promise-future channel is generated for the request. The promise is registered as the tag in the request header, and the main thread holds on to the future. When the thread reading responses receives a response, it identifies the promise from the tag and sets the promised value. Thus, the main thread gets notified on a returned response.

The server implements a fixed-size pool of worker threads (thread pool) where each worker thread handles a client connection exclusively, i.e., it is blocked while serving a client. When the client connection is closed, the worker thread is unblocked and returned to the thread pool. The main thread uses a listening socket to listen for

incoming client connections. An incoming client connection is accepted on a new file descriptor, which the client hands to an available worker of the thread pool. If there are no available workers, it hangs until a worker becomes available, i.e., the size of the thread pool limits the number of concurrent client connections.

To optimize TCP performance, `TCP_NODELAY` option is set on the sockets, which disables Nagle's algorithm, whose purpose is to reduce the number of TCP packets. In the case of the *TCP-adapter*, `TCP_NODELAY` makes sure that the requests and responses are sent over the wire immediately, which potentially improves latency.

4.2 Choosing RPC Frameworks

We researched several different RPC frameworks to find suitable candidates for the thesis. The first requirement was that the frameworks had to be suitable for inter-service communication between microservices in a cloud-native environment. Therefore we considered frameworks from the RPC frameworks listed on the Cloud-Native Computing Foundation (CNCF) landscape, which lists several open-source tools suitable for cloud-native applications. These frameworks are gRPC, Thrift, Apache Avro, Tars, SOFARPC, and DUBBO.

Ultimately, we chose two RPC frameworks for the evaluation, gRPC, and Thrift. A reason for choosing these two is that they are widely used, which means that there exists a decent amount of documentation. These frameworks are also compatible with several programming languages, unlike the other frameworks considered. Both gRPC and Thrift are compatible with C++, which is widely used in the telecom industry. Furthermore, both of these frameworks include data serialization, backward compatibility, and security through TLS.

4.3 Integration of frameworks

We integrated two different frameworks into the system described in subsection 4.1.3 with several different modes of operation per framework. This section describes how we integrated Thrift and gRPC, which includes developing the clients and servers.

4.3.1 Adapters

Table 4.1 displays the different adapters and the names used to refer to them. There are four synchronous and four asynchronous adapters. The asynchronous adapters are TCP-AS, GRPC-AS, GRPC-ASBI, and THRIFT-AS. The synchronous adapters are GRPC-S, GRPC-BI, THRIFT-NB and THRIFT-S.

GRPC-BI and GRPC-ASBI are adapters with bidirectional streaming RPC, while all other adapters are single-request adapters. Single-request mode is the trivial request/response protocol where the client makes a request and receives a response from the server.

Abbreviation	Adapter
GRPC-S	Synchronous single-request gRPC
GRPC-AS	Asynchronous single-request gRPC
GRPC-BI	Synchronous bidirectional streaming gRPC
GRPC-ASBI	Asynchronous bidirectional streaming gRPC
TCP-AS	TCP-adapter
THRIFT-S	Synchronous Thrift
THRIFT-AS	Asynchronous Thrift
THRIFT-NB	Synchronous Thrift with non-blocking IO on server

Table 4.1: Mapping of legend name and communication adapter.

4.3.2 gRPC

We integrated four different gRPC adapters. Two **synchronous** adapters, one of which uses **single-request** RPC and the other uses **bidirectional streaming** RPC. There are also two **asynchronous** adapters, one with single-request RPC, and one with bidirectional streaming RPC.

Synchronous gRPC

The synchronous gRPC adapters are GRPC-S and GRPC-BI. Neither of these adapters require much in terms of implementation to get them operational. Other than implementing the Allocate function in the server stub, only the client and server's initialization are needed. For GRPC-S, invoking the Allocate function invokes the RPC call. In the case of GRPC-BI, the Allocate function instead opens a stream to the server and returns a stream object. The stream object is used to write `AllocateRequest` to and read `AllocateResponse` messages from it.

Asynchronous gRPC

Implementing an asynchronous gRPC client or server is non-trivial compared to a synchronous one. It requires much more logic put into the handling of an RPC call. While gRPC provides an API for making asynchronous RPC calls, managing the calls during their lifetime is not within the scope of gRPC, nor is the threading model of the application.

We implement the clients of the asynchronous gRPC adapters with an event loop that drives the progress of the RPC in the gRPC runtime by continuously reading completion tags from the completion queue. A tag is a pointer to an object instance that encapsulates the context of the actual RPC. The event loop is deployed in a separate thread from the application. It communicates the receipt of a response to the application thread using a promise-future channel accessible via the tag.

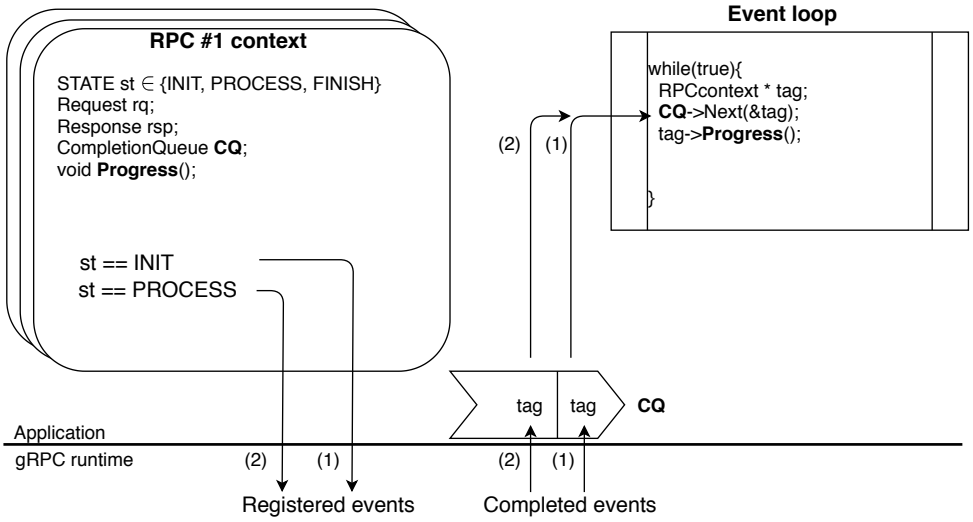


Figure 4.2: Event loop of the GRPC-AS's server.

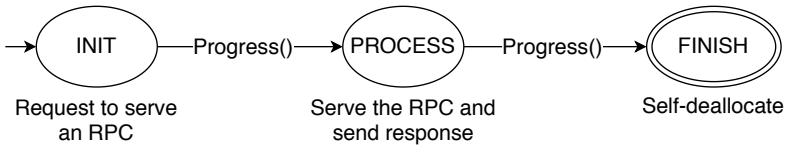


Figure 4.3: Finite state machine of GRPC-AS's server.

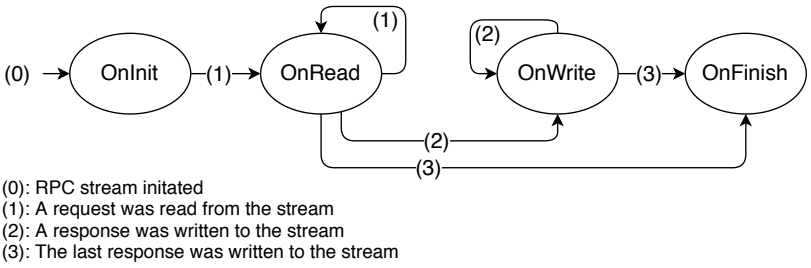


Figure 4.4: Callbacks of GRPC-ASBI's server.

We also implemented the servers of the asynchronous gRPC with an event loop. The adapter accepts incoming client connections and processes the incoming RPCs. Each RPC is encapsulated in an *RPC-context* object containing context variables and state. In the case of GRPC-AS, each RPC-context is a finite state machine that the event loop progresses, as can be seen in Figure 4.2. The *tag* in the figure is, in fact,a pointer to the instance of an RPC-context. The finite state machine is detailed in Figure 4.3. The implementation of GRPC-ASBI is similar to that of GRPC-AS. However, instead of a finite state machine, it operates solely based on callback functions, which we detail in Figure 4.4.

Since the server is limited to one CPU core, the adapters are single-threaded. The recommendation from the gRPC team for the best performance is to have one completion queue per thread and one thread per CPU core. Multiple threads per CPU core would result in extra context switches, which are costly.

4.3.3 Thrift

Three different Thrift versions are implemented, two **synchronous** versions and one **asynchronous** version.

Synchronous Thrift

We integrate two different synchronous servers into the system. The first synchronous Thrift adapter, THRIFT-S, uses a `TThreadedServer`, which spawns a new thread for each client. The other synchronous Thrift adapter, THRIFT-NB, uses a `TNonblockingServer` with only one thread serving all incoming clients concurrently. The synchronous adapters use the same client.

Asynchronous Thrift

The asynchronous Thrift adapter, THRIFT-AS, uses a `TEvhttpServer` and `TEvhttpClientChannel`, and promises and futures to asynchronously receive data from the server after processing of an RPC call.

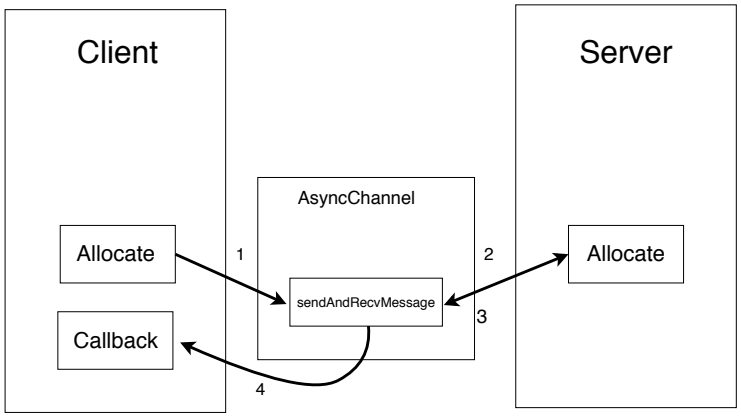


Figure 4.5: The asynchronous Thrift adapter.

As seen in Figure 4.5, the client's call to Allocate passes through an `TAsyncChannel` to the server (1). When the server has processed the RPC request, it sends a response, again through the channel (2,3). Furthermore, the event loop triggers a callback function on the client (4). In the callback function, the client calls on an `recv_Allocate` function, which deserializes the RPC response.

5

Results

In this chapter, we present the results of a qualitative evaluation of gRPC and Thrift, and the TCP-adapter, and the results of the evaluation of the RPC frameworks based on the criteria and system detailed in section 4.1

5.1 Evaluation of qualitative properties of adapters

This section contains the results of the evaluation of gRPC and Thrift. We also provide results of an evaluation of the TCP-adapter. The requirements surveyed are features for obtaining high availability in a cloud-native environment, backward-compatibility of the API, cross-language support, streaming RPC support, asynchronous RPC calls, and secure communication using TLS. Moreover, this section also presents the TCP-adapter's and the frameworks' compatibility with a cloud-native setting. In addition, we also present an evaluation of the ease of development. Further discussion regarding these results and a comparison between the frameworks and the TCP-adapter are done in Chapter 6.

Table 5.1 summarizes what the two frameworks and the TCP-adapter offer from the requirements set. We base these results on standard features, without any modifications of, or additions to the framework. The signs corresponding to each adapter and property aligns with how well the adapter fulfills the property. A minus sign means that the adapter does not fulfill the property at all, while double plus signs mean that it fulfills the property well. We provide more details of each framework later in this section.

Requirements	**gRPC**	**Thrift**	**TCP-adapter**
High availability	++	-	-
Back-compatibility	+	+	-
Cross-language support	++	++	-
Bidirectional Streaming RPC	+	-	-
Asynchronous RPC calls	++	+	+
TLS	++	++	-

Table 5.1: Fulfillment of requirements. Scale ++ > + > -.

gRPC

Table 5.2 summarizes how gRPC fulfills the requirements set for the framework.

Requirements	Features
High Availability	Client-side load-balancing policies Support for integrating health checking
Back-compatibility	Add fields to Protobuf messages Remove fields from Protobuf messages
Cross-language support	Core implementation in C, Java and Go Language-bindings for 10+ languages
Bidirectional Streaming RPC	Yes, using HTTP/2 streams
Asynchronous RPC calls	Asynchronous API
TLS	Yes

Table 5.2: gRPC features.

gRPC has support for integrating health checking into the server and client-side and load-balancing between multiple back-end servers to provide high-level features for high availability. Load-balancing can be achieved via the DNS records received when looking up the server name, or an external load-balancer can be used to provide the client with a list of servers [19, 18]. Back-compatibility is provided not by gRPC but Protocol Buffers, which gRPC uses by default. This enables compatibility between old clients and new servers and vice versa when updating a message. There are some restrictions on how a message can be updated, as detailed in Subsection 2.5.1.

The gRPC core is implemented in the languages C, Java, and Go, and there are official bindings for ten additional languages built on top of a core implementation. There are many other unofficial language bindings. An unofficial bindings' language also needs to have support in Protocol Buffers, unless using another serialization protocol. As stated in Subsection 2.5.1, gRPC supports streaming RPC by the use of HTTP/2 streams multiplexed over a single TCP connection. Streams can be unidirectional or bidirectional. In the case of a unidirectional stream, the client sends a single request, and the server responds with a stream of responses or vice versa with the client streaming requests and the server responding with a single response. In a bidirectional stream, either side can send as many messages as needed in any order.

For asynchronous RPC, gRPC provides an asynchronous API that can build asynchronous clients and servers. The API has the *completion queue* as a central construct. An event loop is constructed by polling the completion queue for completed operations. GRPC has built-in full support for secure communication using TLS. Considering that gRPC is an incubating project at the CNCF whose primary focus is to push the development of cloud-native software, one can assume that gRPC has been implemented with cloud-native constructs in mind with a focus on more than the single RPC protocol.

Thrift

Table 5.3 summarizes how Thrift fulfills the requirements of the qualitative evaluation.

Requirements	Features
High availability	None
Backward-compatibility	Add fields to messages Ignore unrecognized fields
Cross-language support	Implementations in 28 languages
Bidirectional Streaming RPC	None
Asynchronous RPC calls	Yes, but limited
TLS	Yes

Table 5.3: Thrift features.

Thrift is not explicitly adapted for running in a cloud-native environment. Moreover, Thrift does not provide any features such as load-balancing or health checking. Thrift does, however, offer backward-compatibility by allowing servers to read data from clients with an older version than themselves, and vice versa. Furthermore, to allow for backward-compatibility, empty or mismatched field identifiers can be ignored.

Thrift is compatible with 28 programming languages, yet several features are only available for certain languages [2]. For example, several languages, including C, only supports `TSimpleServer` while most features are available for C++. Thrift does currently not support streaming RPC for any language. Furthermore, Thrift only offers limited support for asynchronous RPC requests with the `TEvhttpServer`, `TEvhttpClientChannel` and `TAsyncChannel`. These features are not available for all languages, however.

Furthermore, there is very little documentation on how to implement asynchronous Thrift clients and servers. Moreover, the event loop used for asynchronous communication in the client cannot run on a separate thread, which means that the application and the event loop must run in the same thread. TLS is available and easy to use for RPC clients and servers for many programming languages.

TCP-adapter

Table 5.4 summarizes the qualitative evaluation of the *TCP-adapter*. The TCP-adapter is not built using a framework but is developed specifically for the application use case; hence, it is missing all higher-level features sought after moving towards a microservice architecture. The API and serialization scheme is hard-coded into the adapter, and thus there are no guarantees that two different versions would be compatible with one another. The adapter is implemented in just one programming language, so there is no cross-language support either. Besides, the *GUTI* is sent over the wire in the form of its *C++* struct's memory representation, making it even less compatible for another programming language to parse.

The client is by design, communicating asynchronously with the server. Bidirectional streaming is not an available concept. The adapter only has a single request-response scheme. The adapter does not provide any form of authentication or encryption using TLS.

Requirements	Features
High availability	None
Back-compatibility	None
Cross-language support	None
Bidirectional Streaming	None
Asynchronous communication	Yes
TLS	None

Table 5.4: *TCP-adapter* features.

5.2 Evaluation of quantitative properties of adapters

This section contains the results from a quantitative evaluation of the RPC frameworks based on the assessment criteria detailed in subsection 4.1.2 and using the system described in subsection 4.1.3. We present a summary of the results in Subsection 5.2.3

5.2.1 Single-client evaluation results

Figure 5.1 shows the adapters' mean latency relative to that of TCP-AS, at different rates with zero payload. The reason for displaying the mean latency relative to that of TCP-AS is due to the implementation of rate-mode. For gRPC in general, the streaming RPC adapters have lower latency than the single-request adapters, and the asynchronous adapters have lower latency than their synchronous counterpart.

Figure 5.2 displays the mean latency for adapters when running benchmarks with different payload sizes in *no-rate* mode. The white fogged areas are the mean network latency, as measured by *netperf* at the beginning of each benchmark. The black vertical segment at the top of each bar is the standard deviation for that adapter. Figure 5.3 shows the tail latency of the adapters. The tail latency is the 99th percentile of messages in terms of high latency.

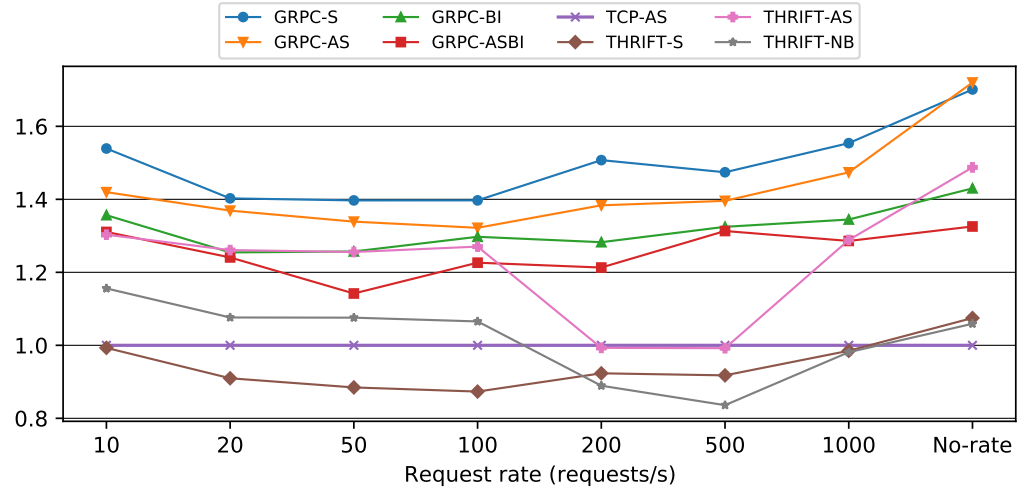


Figure 5.1: Mean latency for rates in rate mode with 0 payload.

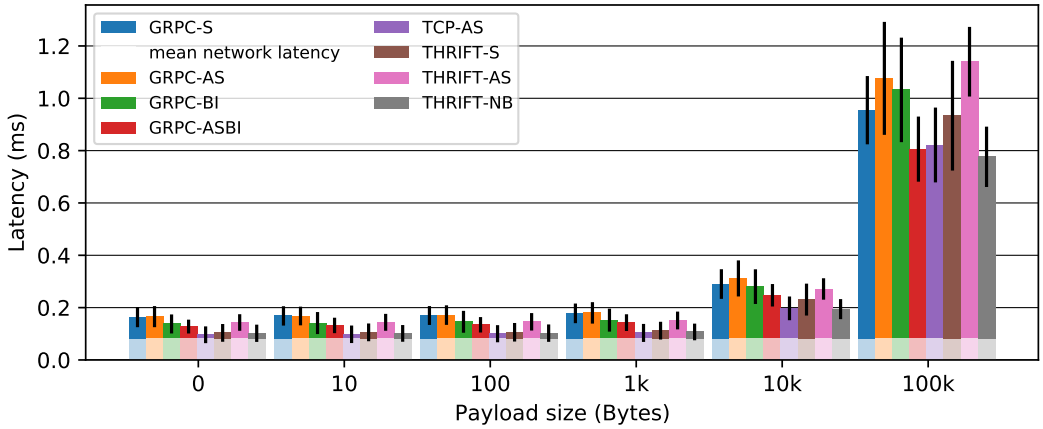


Figure 5.2: Mean latency for payload sizes in no-rate mode.

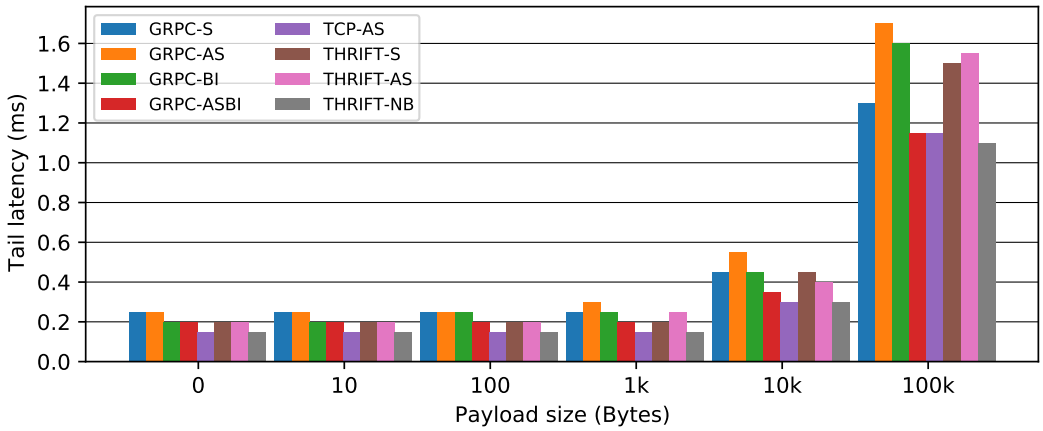
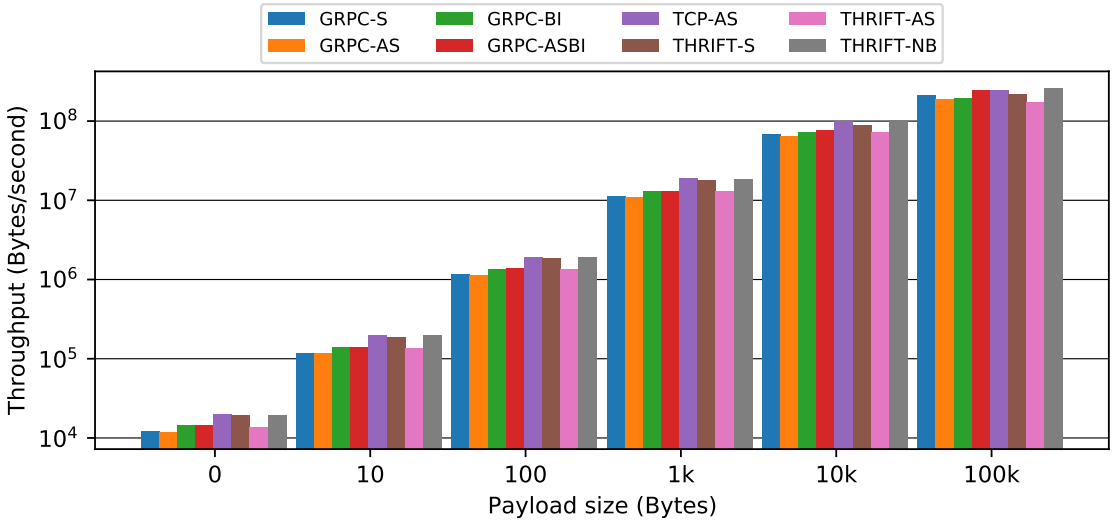
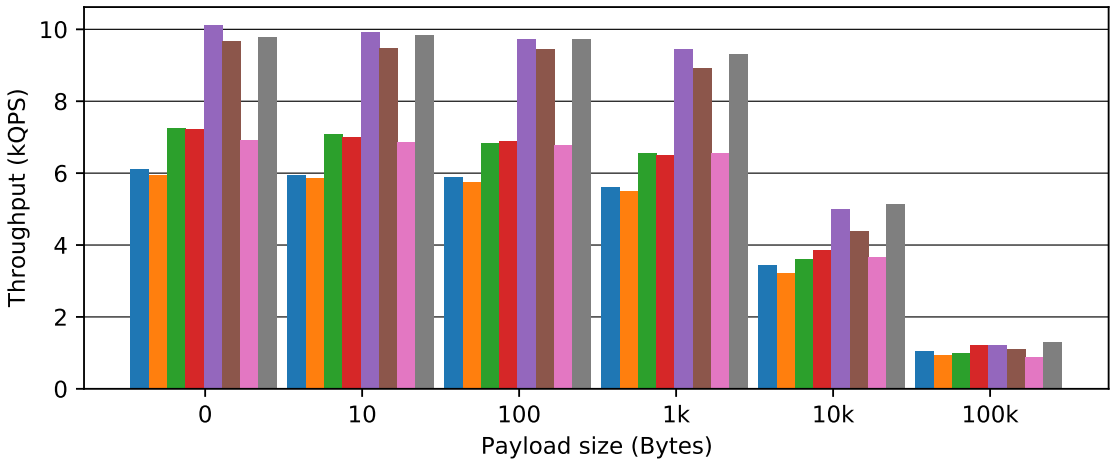


Figure 5.3: Tail latency for payload sizes in no-rate mode. 99th percentile.



(a) Throughput measured in Bytes per second.



(b) Throughput measured in 10^3(k) QPS.

Figure 5.4: Throughput for different payload sizes in no-rate mode with a single client, higher results are preferable.

Figures 5.4a and 5.4b display the throughput of different adapters measured in bytes per second and QPS, respectively. Figure 5.4a uses a logarithmic scale. While Figure 5.4b shows a decrease with increased payload, Figure 5.4b shows that in terms of bytes per second, throughput increases rather than decreases.

Figure 5.5 displays the CPU usage of the adapters. The adapters are run in *no-rate* mode with all payload sizes. The CPU usage is that of the server process.

Figures 5.6a and 5.6b display the throughput divided by the CPU usage for different adapters to take into account the resources utilized for the achieved throughput.

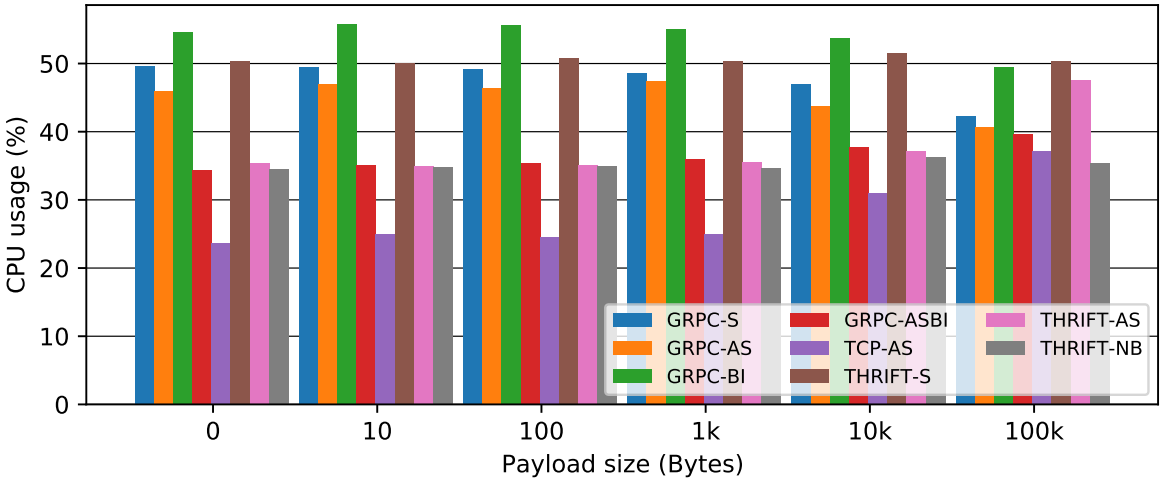
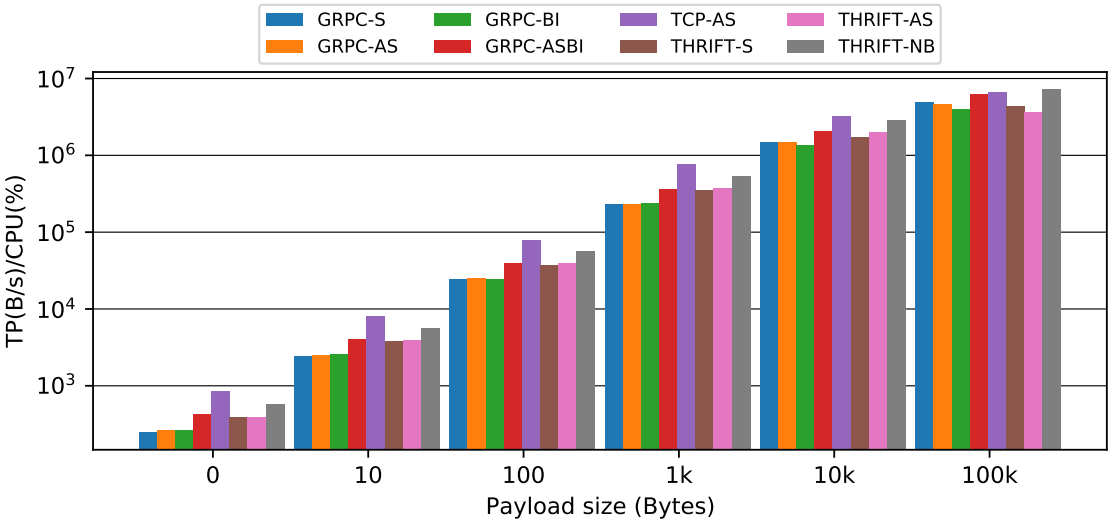
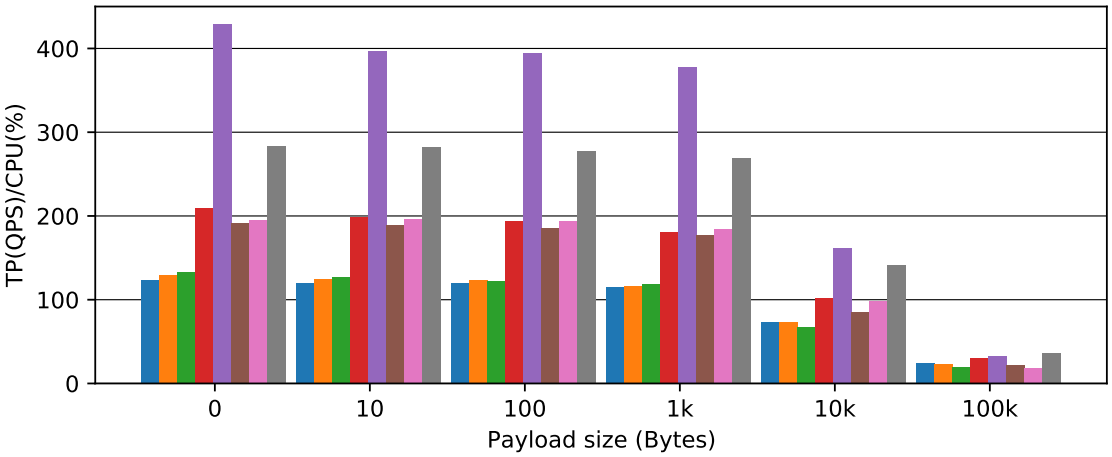


Figure 5.5: Mean CPU usage for payload sizes in no-rate mode.



(a) Throughput measured in Bytes/s.



(b) Throughput measured in QPS.

Figure 5.6: Throughput/CPU usage for payload sizes in no-rate mode. Higher results are preferable.

5.2.2 Multi-client evaluation results

This section presents the results of evaluating adapters while running multiple clients.

Figures 5.7, 5.8 and 5.9 present the mean and median latency of the adapters in the multi-client evaluation with 0 B, 10 kB, and 100 kB payload sizes, respectively. These graphs are box plots which show the distribution of data for the adapters. The box stretches from Q1, the 25th percentile of latency results (bottom of the box) to Q3, the 75th percentile (top of the box). The *whiskers*, the vertical lines coming out of the box, stretches from $Q1 - 1.5 * (Q3 - Q1)$ from the bottom and from $Q3 + 1.5 * (Q3 - Q1)$ from the top. The white line on the box plots mark the median latency while the slightly transparent black line marks the mean latency. For payload zero, some adapters have a very compact distribution of values of latency such as TCP-AS, which makes the box plots look completely flat.

As can be seen in Figures 5.7, 5.8 and 5.9, some adapters have significant differences between mean and median latency. This corresponds to large amount of tail latency for these adapters.

Figures 5.10a, 5.10b, and 5.10c display the throughput in QPS for multiple clients with payloads 0 B, 10 kB and 100 kB respectively. Note that we do not plot the throughput divided by the CPU usage for multiple clients. The reason for this is that in general, when running benchmarks with multiple clients, the CPU usage is at 100 %.

Figures 5.11a, 5.11b and 5.11c present the tail latency of the adapters in the multi-client evaluation with 0, 10 and 100 kB payload respectively.

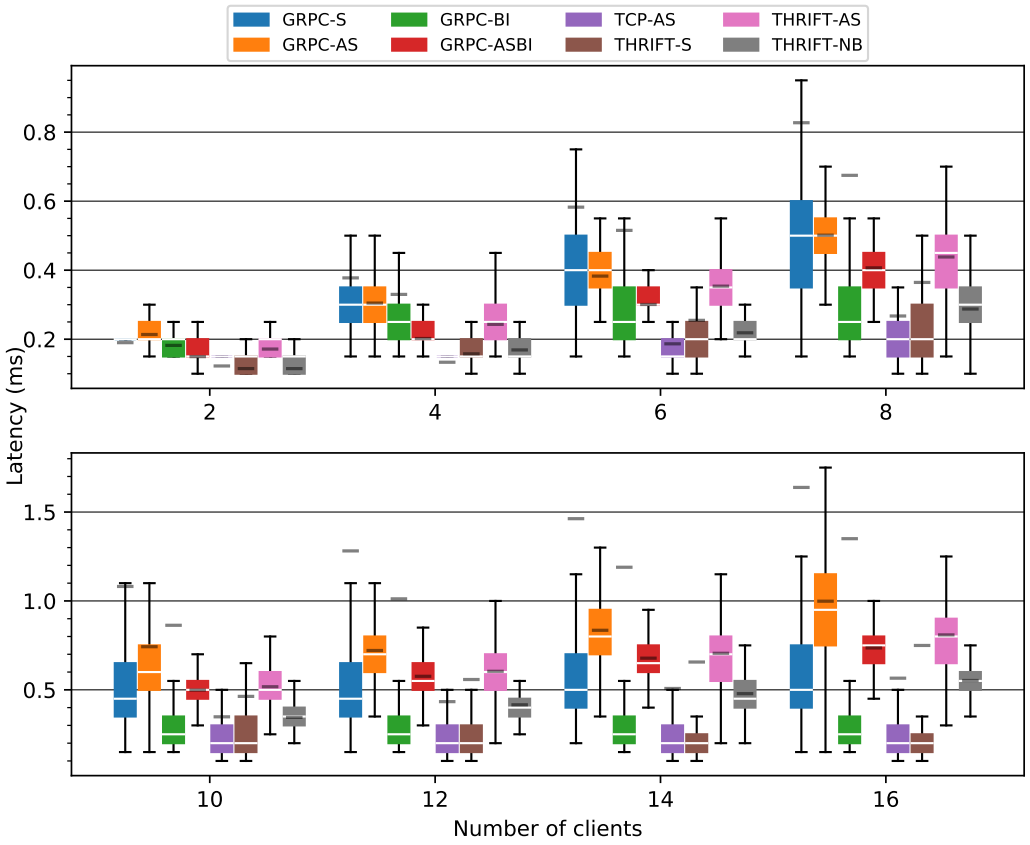


Figure 5.7: Mean latency with multiple concurrent clients, running 0 B payload.

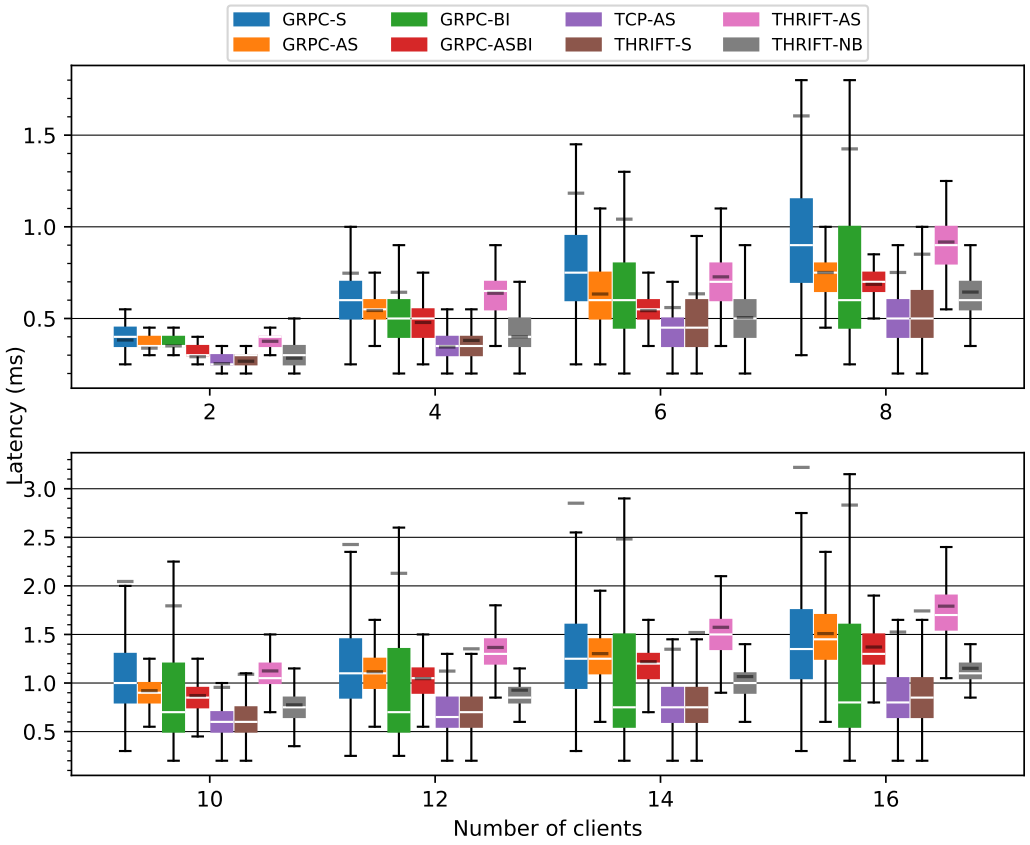


Figure 5.8: Mean latency with multiple concurrent clients, running 10 kB payload.

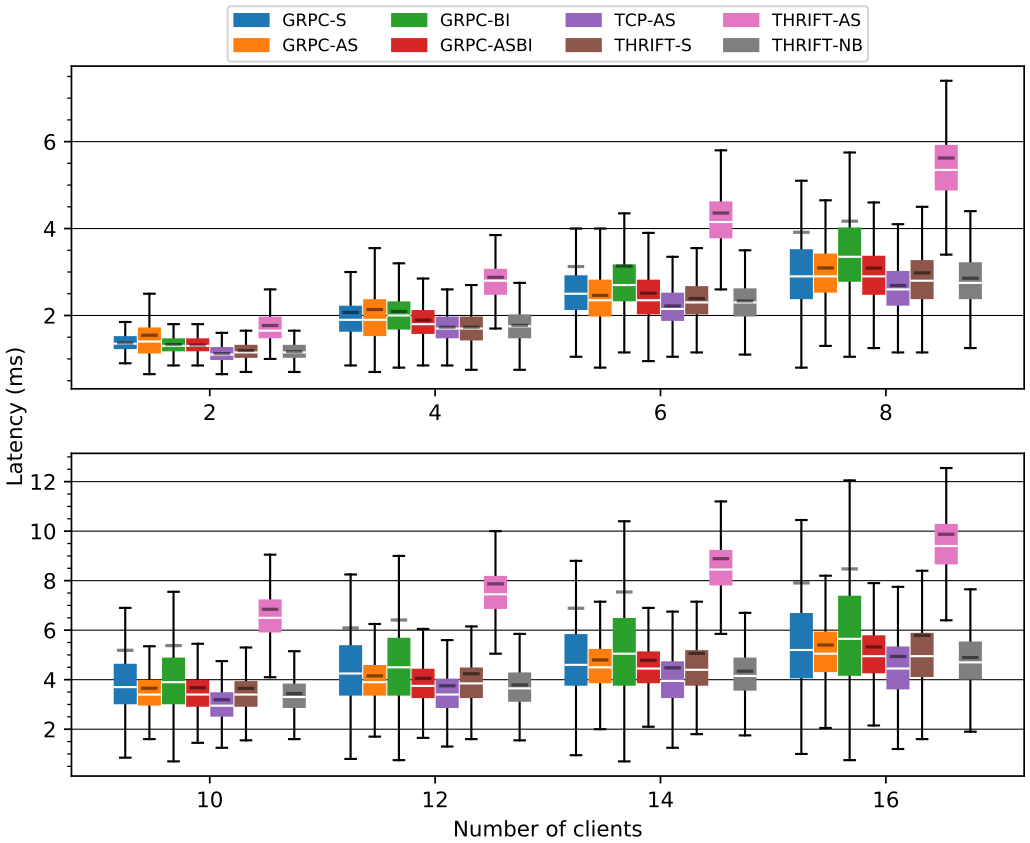
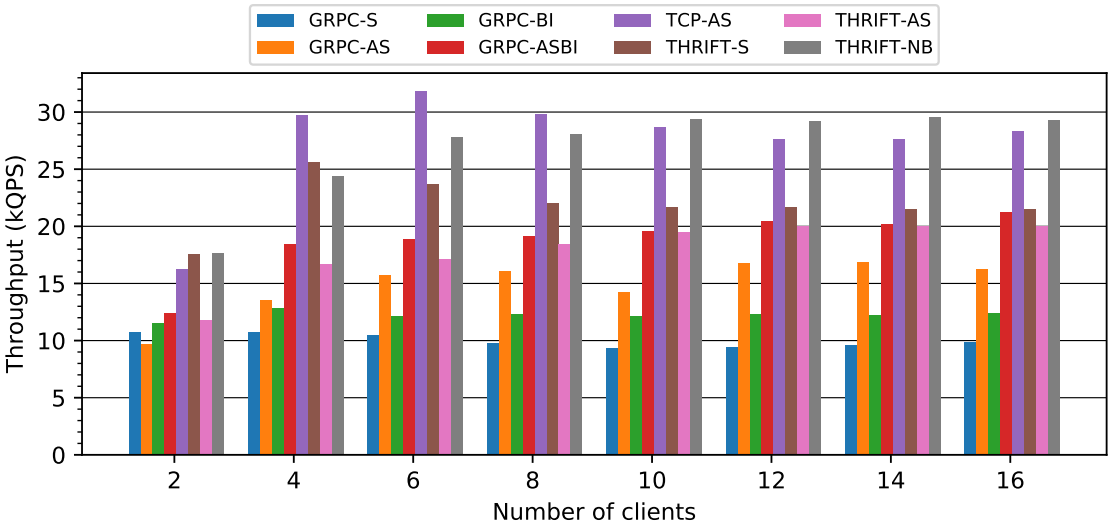
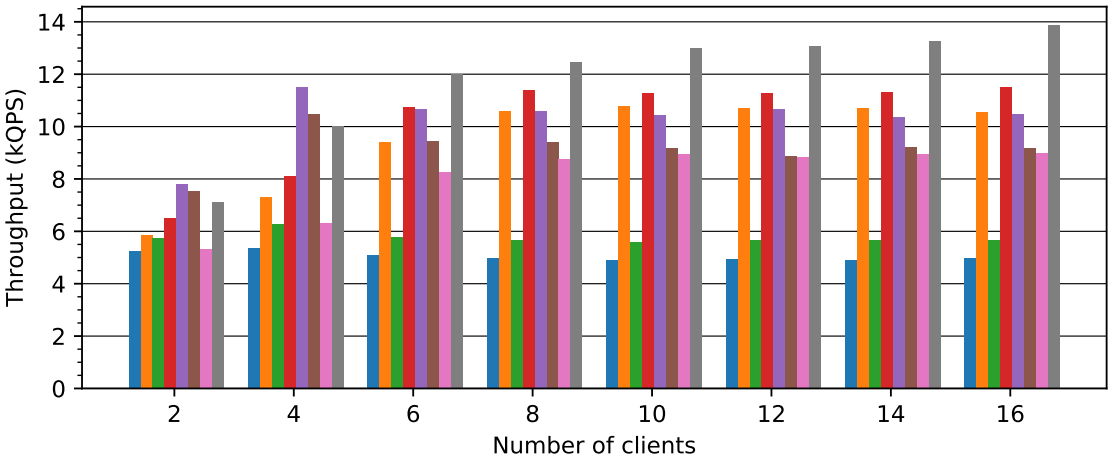


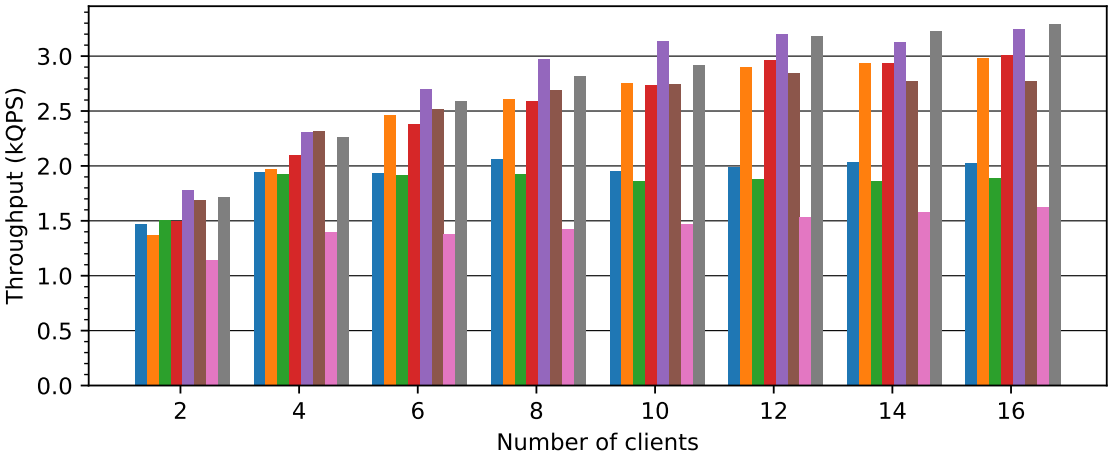
Figure 5.9: Mean latency with multiple concurrent clients, running 100 kB payload.



(a) 0 B payload.

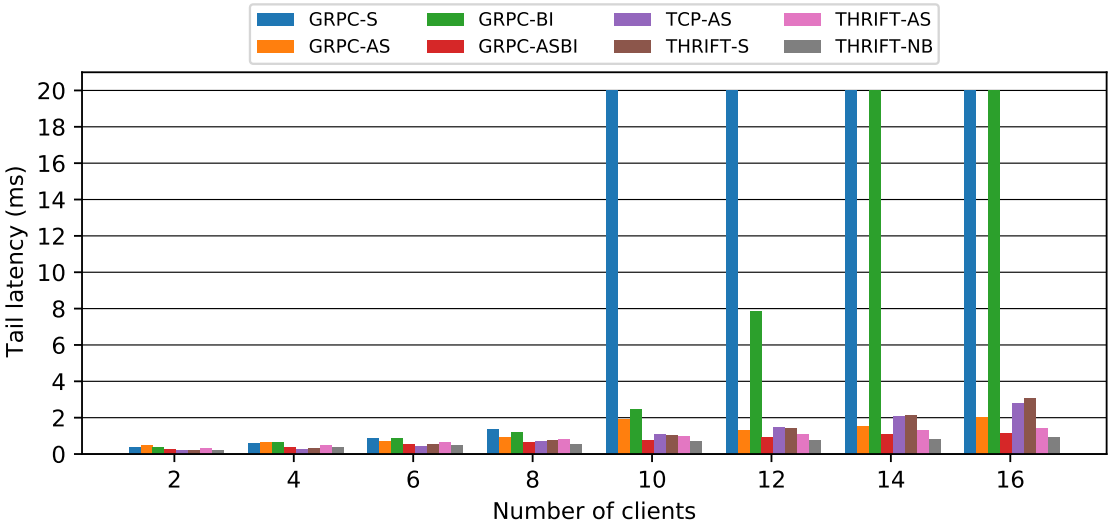


(b) 10 kB payload.

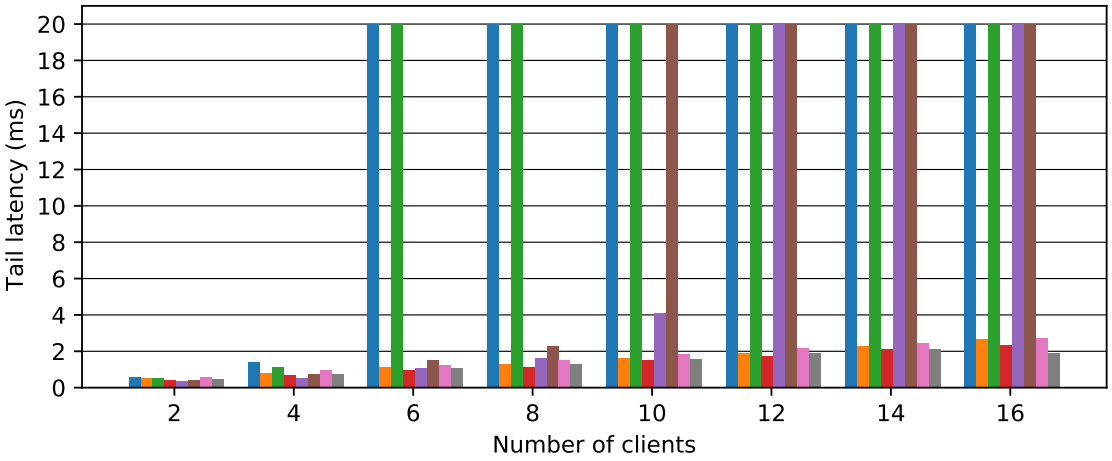


(c) 100 kB payload.

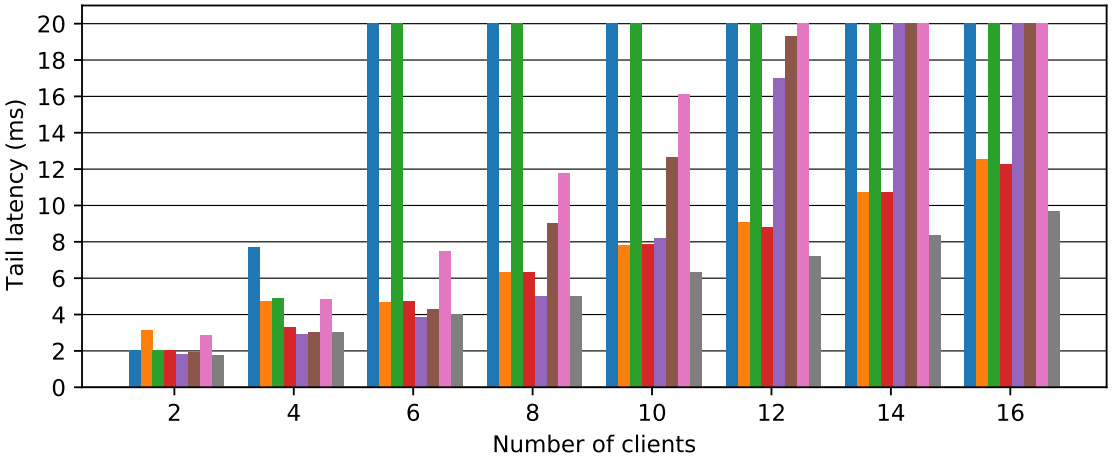
Figure 5.10: Throughput(QPS) with multiple clients and 0-100 kB payload in no-rate mode. Higher results are preferable.



(a) 0 B payload.



(b) 10 kB payload.



(c) 100 kB payload.

Figure 5.11: 99th percentile tail latency with multiple concurrent clients, running 0-100 kB payload in no-rate mode. Lower results are preferable.

5.2.3 Summary of quantitative results

Table 5.5, 5.6 and 5.7 provide a summary of the quantitative results of mean latency, tail latency and throughput. As can be seen from these tables, three adapters constantly perform the best in these areas, TCP-AS, THRIFT-NB, and GRPC-ASBI. In general, we can see that TCP-AS has a larger amount of tail latency with multiple clients. It also seems like THRIFT-NB performs the best of the three adapters for benchmarks with zero payload and many clients. GRPC-ASBI has the best results in these categories for a payload of 10 kB.

Client/Payload	0	10k	100k
1	TCP-AS	THRIFT-NB	THRIFT-NB
2	THRIFT-S THRIFT-NB	TCP-AS	TCP-AS
4	TCP-AS	TCP-AS	TCP-AS THRIFT-S
6	TCP-AS	THRIFT-NB	TCP-AS
8	TCP-AS	THRIFT-NB	TCP-AS
10	TCP-AS THRIFT-NB	THRIFT-NB	TCP-AS
12	THRIFT-NB	THRIFT-NB	THRIFT-NB
14	THRIFT-NB	THRIFT-NB	THRIFT-NB
16	THRIFT-NB	THRIFT-NB	THRIFT-NB

Table 5.5: Adapter with lowest mean latency for different amount of clients for payloads 0 B, 10 kB and 100 kB.

Client/Payload	0	10k	100k
1	TCP-AS THRIFT-NB	TCP-AS THRIFT-NB	THRIFT-NB
2	TCP-AS THRIFT-S THRIFT-NB	TCP-AS	TCP-AS
4	TCP-AS	TCP-AS	TCP-AS
6	TCP-AS	GRPC-ASBI	TCP-AS
8	THRIFT-NB	GRPC-ASBI	THRIFT-NB
10	THRIFT-NB	GRPC-ASBI	THRIFT-NB
12	THRIFT-NB	GRPC-ASBI	THRIFT-NB
14	THRIFT-NB	GRPC-ASBI	THRIFT-NB
16	THRIFT-NB	THRIFT-NB	THRIFT-NB

Table 5.6: Adapter with lowest amount of tail latency for different amount of clients for payloads 0 B, 10 kB and 100 kB.

Client/Payload	0	10k	100k
1	TCP-AS	THRIFT-NB	THRIFT-NB
2	THRIFT-NB	TCP-AS	TCP-AS
4	TCP-AS	TCP-AS	TCP-AS
6	TCP-AS	THRIFT-NB	TCP-AS
8	TCP-AS	THRIFT-NB	TCP-AS
10	THRIFT-NB	THRIFT-NB	TCP-AS
12	THRIFT-NB	THRIFT-NB	TCP-AS
14	THRIFT-NB	THRIFT-NB	THRIFT-NB
16	THRIFT-NB	THRIFT-AS	THRIFT-NB

Table 5.7: Adapter with highest throughput for different amount of clients for payloads 0 B, 10 kB and 100 kB.

6

Discussion

This chapter contains an evaluation of the results gathered in this thesis. Section 6.1 discusses the results of the different gRPC adapters, while Section 6.2 compares results for the Thrift adapters. Section 6.3 contains a comparison of the RPC frameworks and the TCP-adapter. Section 6.4 consists of a comparison of Thrift and gRPC based on the results presented in Chapter 5.

6.1 gRPC adapters

Among gRPC adapters, GRPC-ASBI has the lowest latency results overall. These results hold for all different latency evaluations, with different amounts of clients, payloads, and rates. GRPC-ASBI also has lower tail latency than the other gRPC-adapters. GRPC-S and GRPC-BI show the best median latency with multiple clients. However, they also show a significantly wider distribution and a higher mean latency, probably due to the tail latency they both suffer from. Regarding throughput, GRPC-ASBI and GRPC-BI have the highest throughput of the gRPC-adapters. However, GRPC-AS has better throughput than GRPC-BI for multiple clients.

Regarding ease of development, all adapters provide an abstraction of all the networking and data serialization. The adapters GRPC-S and GRPC-BI are very easy to use and require close to no additional logic to function. GRPC-ASBI and GRPC-AS take the most effort to implement. The reason for this is that the threading model, as well as an event loop for processing the RPCs, need to be implemented in the adapter. However, for an advanced user, this allows for more fine-grained tuning of performance and other aspects such as handling external blocking IO without blocking the application. Comparing these adapters, the streaming one requires some more implementation since the handling of a streaming RPC requires more logic than that of a single-request RPC, as detailed in Section 4.3.

Considering that GRPC-ASBI outperforms its counterpart GRPC-BI combined with the strict requirements for performance in the 5GC, going the extra mile to implement the needed functionality for GRPC-ASBI, is deemed worth it, especially since asynchronous RPCs are desirable. If streaming is unnecessary for a specific use case, then GRPC-AS would be worthwhile to implement in the case of several clients. If only one client were to be connected and asynchronous RPCs are not needed, then GRPC-S would do just fine, as compared to GRPC-AS.

6.2 Thrift-adapters

The thrift adapters vary in transport protocol, threading as well as IO-model. THRIFT-S and THRIFT-NB have a very similar mean latency up until about six clients across all payloads. After six clients, THRIFT-NB performs the best across the board. THRIFT-NB shows the lowest CPU usage, the best mean and tail latency, and the highest throughput. The reason why THRIFT-NB performs better than THRIFT-AS could be that the latter uses HTTP. HTTP probably induces more overhead to the RPC in terms of transport and processing to pack and unpack the request and response. Moreover, the uncertain performance of `libevent`'s `evhttp` interface for constructing and parsing HTTP requests and responses is another factor that might affect THRIFT-AS.

THRIFT-S uses the same client as THRIFT-NB, so the differences in performance between them are due to the server. Both adapters use `TFramedTransport` and the same `TProcessor`. However THRIFT-NB uses `libevent` to handle the IO, which in turn uses `event poll` (epoll) for efficient event notification. THRIFT-S uses regular blocking `read` and `write` syscalls on the client connection socket. While THRIFT-NB is not built with the use case of a single client connection in mind, it interestingly, still performs very similar to THRIFT-S, which, with its simplistic design we thought would excel at low amounts of concurrent clients. However, as the number of clients increases, the efficiency and consistency of THRIFT-NB become clear. THRIFT-S displays a low median latency; however, the tail latency is quite severe. If the server would be allowed to use as many cores as clients connected, then the THRIFT-S would be able to process the clients in parallel, increasing efficiency at the cost of CPU. However, in a cloud-native setting, horizontal scaling would be preferred over vertical. With THRIFT-NB, the optimal performance is theoretically obtained with as many processing threads as cores available. So in our case, an increasing number of threads would not result in improved performance as we only have one core available to the server.

Secure communication using TLS is available for THRIFT-NB and THRIFT-S; however, not for THRIFT-AS.

Concerning ease of development, the framework severely lacks documentation on how to use it practically. Much time is necessary to understand how to initiate a client and server and how to tweak it for certain use cases. Often, the only way is to dig into the source code. Once a client and server are up and running, using them are easy, and they abstract all underlying data serialization and networking for the developer. The hardest adapter to use is THRIFT-AS, as the developer needs to provide and run the client in an event loop using `libevent's event_base_loop` and provide a callback function with each RPC call. Moreover, the `TEvhttpClient` does not allow the event loop to run in a thread separate from the thread that the RPC is from, typically the application's main thread, as doing this causes synchronization problems in the `TEvhttpClient`. I.e. the `TEvhttpClient` is not thread-safe.

With the performance that THRIFT-NB show in comparison to THRIFT-AS, it would

be the preferred adapter. However, as it does not support asynchronous RPC invocations, it would not suit all use cases, and thus THRIFT-AS would be needed.

6.3 Comparison of RPC frameworks and TCP-adapter

Looking at the summary of results in Subsection 5.2.3, the TCP-adapter has the best results for several of the evaluation categories, and stands out in mean latency and throughput, while not showing as good results in tail latency.

Before performing the quantitative evaluation, it seemed probable that the TCP-adapter would perform best since all other adapters have TCP as transport protocol, but with added overhead, meaning that the purest form of TCP is probably the most efficient. However, this theory seems to be untrue, and it seems like the overhead is not always enough to give TCP an advantage. Instead, using features such as HTTP/2 streaming, or non-blocking server features, seems to compensate in many cases.

When looking at the qualitative evaluation, it is clear the TCP-adapter cannot compete with gRPC. While TCP is available for almost any programming language, TCP has no built-in mechanisms for high availability or backward compatibility. Furthermore, TLS is not a built-in feature for TCP sockets as they are for gRPC and Thrift, so a lot more effort is required to integrate them.

To conclude, we judge that RPC seems to be a better option than using the TCP-adapter for the use case investigated in this thesis.

6.4 Comparison of Thrift and gRPC

This section contains a comparison of Thrift and RPC based on the results in Chapter 5.

6.4.1 Comparison of quantitative results

The most essential criteria of this evaluation is low latency. In this regard, the Thrift-adapters outperform the gRPC adapters in every evaluation. The synchronous Thrift adapters, THRIFT-S, and THRIFT-NB perform the best of all adapters except the TCP-adapter when comparing mean latency for different payloads for single clients. We see similar results when evaluating multiple clients at a time. An exception to these results is for payload 100 kB when the gRPC asynchronous and Thrift synchronous adapters perform very similarly. This result may be because gRPC's overhead from using HTTP/2 becomes negligible at large payloads. When comparing results in *rate-mode*, we can see similar results, where THRIFT-S and THRIFT-NB have the lowest mean latency of the RPC-adapters.

Until a payload of 1 kB, results are very similar for tail latency for single-client

evaluation with different payloads. GRPC-S and GRPC-AS have slightly higher tail latency, while TCP-AS and THRIFT-NB have the lowest tail latency. Tail latency for both GRPC-AS and THRIFT-AS increases slightly more than the other adapters for higher payloads.

GRPC-ASBI, THRIFT-AS and THRIFT-NB have the lowest CPU usage of all RPC adapters in the evaluation. Furthermore, the CPU usages are entirely consistent until 100 kB payload, for which the Thrift adapters seem to be affected the most by the increased payload.

Looking at throughput/CPU usage for single clients, THRIFT-NB, THRIFT-S, and GRPC-ASBI are the adapters with the lowest CPU usage, and have very similar results. Comparing throughput through CPU usage however, THRIFT-NB sticks out from the rest of the RPC adapters as having the best results.

Throughput increases a lot when utilizing four clients instead of two for all adapters except for GRPC-S, GRPC-AS, and GRPC-BI. Looking at these results, it seems like Thrift, in general, handles multiple clients better, and therefore potentially scales better than gRPC.

When only comparing asynchronous frameworks, the differences between gRPC and Thrift are not as substantial. For example, when comparing mean latency for clients in no-rate mode, THRIFT-AS is about on par with the asynchronous gRPC adapters and performs worse than GRPC-ASBI in some cases. Since only the synchronous Thrift adapters stand out in performance, Thrift is probably only preferable for use cases where synchronous, non-streaming communication is appropriate. The reason for this could be the use of HTTP protocols in all of the asynchronous RPC frameworks. While HTTP/2 brings features such as multiplexed streaming, the added overhead from HTTP and HTTP/2 seems to increase latency and reduce performance in general.

With multiple clients, there is a clear distinction as to which adapters handle multiple clients better than the others. At 10 kB payload, GRPC-S, GRPC-BI, TCP-AS and THRIFT-S manage tail latency very poorly, having a tail latency of highest measurable value 20 ms. GRPC-S and GRPC-BI have this tail latency already at six clients, while THRIFT-S and TCP-AS show it at 10 and 12 clients. The common denominator between these four adapters is that they require one thread per client connection, which results in many thread switches that impact the performance. The other group of adapters, those that perform well while handling multiple clients, are GRPC-ASBI, GRPC-AS, and THRIFT-NB. Their common denominator is that they all use an IO event notification scheme based on asynchronously watching several client connections simultaneously. This means that a thread avoids being blocked on one client connection but can serve multiple client connections per thread, reducing the number of thread switches and thus increases performance in a use case of many times more clients than CPU cores. The main costs are increased complexity of implementation and that an extended processing time of a request would block incoming requests.

6.4.2 Comparison of qualitative results

When considering the qualitative evaluation, gRPC is superior in most aspects. Unlike Thrift, it has several features which ensure high availability, as described in Section 5. Furthermore, gRPC fully supports streaming, which is nonexistent in Thrift. Moreover, while Thrift technically supports asynchronous communication, it is not as simple to implement as in gRPC due to the almost nonexistent documentation on the area. Furthermore, the developer has much more freedom when implementing asynchronous communication for gRPC rather than Thrift. Moreover, TLS is not available for asynchronous communication in Thrift.

One main difference between Thrift and gRPC is that, while Thrift is twice as old as gRPC, gRPC is adopted on a larger scale than Thrift, and has a broader community of developers. The CNCF landscape lists over 500 contributors for gRPC, while approximately 300 are listed for Thrift. This might be because gRPC supports more features than Thrift. Moreover, Thrift updates are few and far between, around two each year. gRPC, is updated more often than every second month according to their release schedule [17].

The only feature of Thrift that stands out is cross-language support, as Thrift currently supports 28 languages, and gRPC officially supports only 11. Due to the frequent updating of gRPC, together with a larger adoption rate, this will likely change in the future. If integrating Thrift rather than gRPC, one might have to write more thorough documentation. Furthermore, streaming would need to be added somehow. Even though Thrift outperforms gRPC, at present and in the foreseeable future, it is probably better to instead integrate gRPC for this particular use case of inter-service communication in a 5G environment. Especially since gRPC is being updated regularly, and new benchmarks are performed often, meaning that performance is a critical property for gRPC, and will probably improve in the future [21]. To conclude, when integrating an RPC framework as inter-service communication in 5GC NFs, we judge that is that it is probably more beneficial to integrate gRPC than Thrift.

7

Concluding remarks

This chapter consists of a conclusion in Section 7.1 and future work in Section 7.2

7.1 Conclusion

This thesis aims to investigate whether RPC frameworks are suitable as inter-service communication in 5GC NFs. This evaluation was necessary due to the high demands on, for example, latency in 5G, coupled with a wish of making development more manageable by using third-party frameworks.

We have evaluated gRPC and Thrift by implementing several *adapters* using different frameworks and modes. We have evaluated these adapters quantitatively and qualitatively. The quantitative evaluation consisted of benchmarks, while the qualitative evaluation consisted of comparing the adapters against each other based on a set of properties deemed essential for the use case.

We can see from our results that the RPC frameworks, in general, have slightly worse results in terms of mean latency, tail latency, and throughput than a TCP-adapter. The results also point towards Thrift-frameworks performing better than gRPC frameworks in the quantitative evaluation. When considering only qualitative properties, however, gRPC is superior. Not only does gRPC provide many useful features such as support for bidirectional streaming RPC, but it also comes with excellent documentation and a large community.

Considering all results and the context of the thesis, we believe that RPC frameworks seem to be suitable for use as inter-service communication in a 5GC NF, and gRPC specifically, seems to be a preferable RPC framework in its current state.

7.2 Future work

This thesis provides an evaluation of only two RPC frameworks due to time limitations. Potential future research could include more frameworks to evaluate and compare. One particular framework that could be interesting to research is the Facebook branch of Thrift [6]. This framework is built on regular Thrift, but has further support for asynchronous communication, among other features. Furthermore, new

RPC frameworks could be developed with this specific use case to properly fulfill the requirements presented in this thesis. Moreover, it could be interesting to further develop the TCP adapter so that it has all the requested features. This comparison would probably be on more fairgrounds.

Another aspect to consider could be to try to increase the performance of gRPC and Thrift by changing the source code to accommodate this particular use case. Another change in the integration of the frameworks could be to increase security by adding mutual authentication through TLS.

To further evaluate the gRPC and Thrift, one could run additional benchmarks. As discussed in 6, the difference in performance between TCP and the synchronous Thrift adapters, and the gRPC adapters and asynchronous Thrift, could be caused by the use of HTTP in the latter adapters. Therefore, it could be interesting to run benchmarks with HTTP to measure the overhead of HTTP. Other types of evaluation that could be interesting are more evaluation on the usage of TLS and investigate how the use of authentication through TLS affects the frameworks differently.

Bibliography

[1] Apache thrift. https://thrift.apache.org/. Accessed: 2020-03-26.

[2] Apache thrift language support. http://thrift.apache.org/docs/Languages. Accessed: 2020-06-10.

[3] Cloud native design for telecom applications. https://www.ericsson.com/assets/local/digital-services/doc/2101_cloud-native-design-pa4.pdf. Accessed: 2020-06-16.

[4] Cncf cloud native definition v1.0. https://github.com/cncf/toc/blob/master/DEFINITION.md. Accessed: 2020-06-16.

[5] Docker. https://docker.com/. Accessed: 2020-03-26.

[6] fbthrift. https://github.com/facebook/fbthrift. Accessed: 2020-06-09.

[7] Kubernetes. https://kubernetes.io/. Accessed: 2020-03-26.

[8] Pods. https://kubernetes.io/docs/concepts/workloads/pods/pod/. Accessed: 2020-03-26.

[9] This is 5g. https://www.ericsson.com/4a3114/assets/local/newsroom/media-kits/5g/doc/ericsson_this-is-5g_pdf_v4.pdf. Accessed: 2020-05-18.

[10] Randy Abernethy. *Programmer's Guide to Apache Thrift*. Manning Publications, 2019. Accessed: 2020-07-1.

[11] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. Accessed: 2020-07-1.

[12] Tulja Vamshi Kiran Buyakar, Harsh Agarwal, Bheemarjuna Reddy Tamma, et al. Prototyping and load balancing the service based architecture of 5g core using nfv. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 228–232. IEEE, 2019. Accessed: 2020-07-1.

[13] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing,

Cham, 2017. Accessed: 2020-07-1.

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

[15] gRPC contributors. Faq. `https://grpc.io/faq/`. Accessed: 2020-06-17.

[16] gRPC contributors. Grpc. `https://grpc.io/`. Accessed: 2020-03-11.

[17] gRPC contributors. grpc release schedule. `https://github.com/grpc/grpc/blob/master/doc/grpc_release_schedule.md`. Accessed: 2020-06-22.

[18] gRPC contributors. Grpc health checking protocol. `https://github.com/grpc/grpc/blob/master/doc/health-checking.md`, 2019. Accessed: 2020-05-29.

[19] gRPC contributors. Load balancing in grpc. `https://github.com/grpc/grpc/blob/master/doc/load-balancing.md`, 2019. Accessed: 2020-05-29.

[20] Hassan Hawilo, Manar Jammal, and Abdallah Shami. Exploring microservices as the architecture of choice for network function virtualization platforms. *IEEE Network*, 33(2):202–210, 2019.

[21] K. Indrasiri and D. Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020. Accessed: 2020-07-1.

[22] James Kempf, Bengt Johansson, Sten Pettersson, Harald Lüning, and Tord Nilsson. Moving the mobile evolved packet core to the cloud. In *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 784–791. IEEE, 2012. Accessed: 2020-07-1.

[23] David S Linthicum. Cloud-native applications and cloud migration: The good, the bad, and the points between. *IEEE Cloud Computing*, 4(5):12–14, 2017. Accessed: 2020-07-1.

[24] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, June 1988. Accessed: 2020-07-1.

[25] J. Lu, L. Xiao, Z. Tian, M. Zhao, and W. Wang. 5g enhanced service-based core design. In *2019 28th Wireless and Optical Communications Conference (WOCC)*, pages 1–5, 2019.

[26] C. Manso, R. Vilalta, R. Casellas, R. Martínez, and R. Muñoz. Cloud-native sdn controller based on micro-services for transport networks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 365–367, 2020.

[27] Bruce Jay Nelson. Remote procedure call. 1981. Accessed: 2020-07-1.

[28] Sam Newman. *Building microservices: designing fine-grained systems*. "

O'Reilly Media, Inc.", 2015. Accessed: 2020-07-1.

[29] Thuy Nguyen et al. Benchmarking performance of data serialization and rpc frameworks in microservices architecture: grpc vs. apache thrift vs. apache avro, 2016. Accessed: 2020-07-1.

[30] Van-Giang Nguyen, Anna Brunstrom, Karl-Johan Grinnemo, and Javid Taheri. Sdn/nfv-based mobile packet core network architectures: A survey. *IEEE Communications Surveys & Tutorials*, 19(3):1567–1602, 2017. Accessed: 2020-07-1.

[31] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007. Accessed: 2020-07-1.

[32] R. Thurlow. Rpc: Remote procedure call protocol specification version 2. RFC 5531, RFC Editor, 05 2009. Accessed: 2020-07-1.

[33] 3GPP TS 23.501 v16.3.0. 3rd generation partnership project; technical specification group services and system aspects; system architecture for the 5g system (5gs); stage 2 (release 16). Technical report, 3rd Generation Partnership Project, 2019. Accessed: 2020-07-1.

[34] Cheng Zhang, Xiangming Wen, Luhan Wang, Zhaoming Lu, and Lu Ma. Performance evaluation of candidate protocol stack for service-based interfaces in 5g core network. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2018. Accessed: 2020-07-1.