Computing and Informatics, Vol. 42, 2023, 1–36, doi: 10.31577/cai_2023_1_1

EVALUATION OF MICROSERVICE COMMUNICATION WHILE DECOMPOSING MONOLITHS

Justas Kazanavičius, Dalius Mažeika

Faculty of Fundamental Sciences
Vilnius Gediminas Technical University
Sauletekio al. 11
LT-10223 Vilnius, Lithuania
e-mail: {justas.kazanavicius, dalius.mazeika}@vilniustech.lt

Abstract. One of the biggest challenges while migrating from a monolith architecture to a microservice architecture is to define a proper communication technology. In monolith applications, communication between components is performed using the in-process method or function calls, while different communication methods have to be established to achieve the same functionality in a microservice architecture. A microservices-based application is a distributed system running on multiple processes or services. Therefore, microservices must interact using interprocess communication technologies. This research aims to evaluate synchronous and asynchronous communication technologies and determine particular cases for their application while decomposing monolith into cloud-native applications. Five communication technologies, such as HTTP Rest, RabbitMQ, Kafka, gRPC, and GraphQL, have been evaluated and compared by proposed evaluation criteria. The advantages and disadvantages of each communication technology were identified in the context of microservices architecture.

Keywords: Microservice communication, cloud computing, software engineering, HTTP rest, RabbitMQ, Kafka, gRPC, GraphQL

Mathematics Subject Classification 2010: 68-04

1 INTRODUCTION

A microservice architectural style is an approach to developing an application as a suite of small services where every service communicates with other services via light-weight mechanisms such as Hypertext Transfer Protocol (HTTP) Application Programming Interface (API). Services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of services that may be written in different programming languages and use diverse data storage technologies [1].

International Data Corporation predicts that 90 % of all new applications in 2022 will be developed based on microservices architectures. Microservice architecture, as well as software development and IT operations (DevOps) practice, improve software development agility and flexibility. Enterprises can bring their digital products and services to a very competitive market faster [2, 3, 4]. Microservice architecture is becoming a design standard for modern cloud-based software systems because it helps develop a cloud-native application [5, 6]. Using microservices and embracing cloud-native technologies is the way to reduce development time and increase deployment speed [2]. Each microservice is an independent process and could be developed and deployed independently to a container or virtual machine in the cloud. Many different solutions are used to support microservices in a cloud environment, such as Docker or Rocket containers, Docker Swarm, Mesos and Kubernetes orchestration tools, etc. [5].

This research aims to analyze and evaluate different communication technologies between microservices and determine particular cases for their application while decomposing monolith into cloud-native applications. This article is part of our research on legacy monolith software migration to microservices architecture. Literature review, monolith decomposition into microservices methods and database decomposition were analyzed in previous papers (studies) [7, 8, 9]. This paper provides an analysis of how proper communication between decomposed microservices could be established. A set of criteria that is important while decomposing monoliths to microservices was identified. The benefits and drawbacks of communication technologies as well as the impact on communication between microservices, were evaluated based on these criteria.

Five technologies were chosen for analysis, i.e., HTTP Representational State (Rest) API, RabbitMQ, Kafka, gRPC, and GraphQL. Rest API represents asynchronous communication style and has become a de facto standard synchronous communication technology. RabbitMQ and Kafka represent asynchronous communication based on message broker. GraphQL and gRPC have been selected for the investigation because of the rapidly growing popularity. GraphQL provides client-side applications functionality to query databases at server-side applications, while gRPC is a technology to implement remote procedure call (RPC) API. It uses HTTP 2.0 as its underlying transport protocol and is provided as a data structure. Various criteria were taken into account while analyzing selected communication technologies, including influence on microservice topology, the performance of re-

mote procedure call, message size, memory consumption, storage usage, boot time, and availability of the corresponding libraries.

The main contribution of this work is a unique set of criteria used to compare five communication technologies and evaluate their advantages and disadvantages in the context of monolith decomposition to microservices. The key findings identified during this research are provided as a guideline for the researchers and industry that can help to speed up legacy monolith decomposition to microservices and make this complex procedure more obvious.

The rest of this paper is organized as follows: Section 2 gives a review of microservice communication technologies and patterns. Section 3 describes the research methodology. Section 4 reports investigation results of HTTP Rest, RabbitMQ, Kafka, gRPC, and GraphQL communication technologies. Section 5 provides an analysis and comparison of the results. Finally, Section 6 concludes this work.

2 RELATED WORKS

There are many different studies performed to determine which communication technology or pattern is faster, more secure, more robust, etc. But none of them have investigated communication technologies from the perspective of legacy monolith decomposition into microservices and a cloud-native application. This section reviews the research performed on communication technologies, orchestration architecture patterns, streaming, performance, distributed cache, and differences between microservices and service-oriented architecture (SOA) communication. A research plan was created based on literature findings, and previous work in this domain and a set of criteria for communication evaluation was introduced.

2.1 Communication Technologies

Communication between components in a monolithic application is implemented using in-process based methods or function calls. Microservices-based application is a distributed system running multiple processes and services, so different communication technologies must be used. Design of communication between microservices is one of the most significant challenge while migrating from a monolithic software to a microservices architecture [10].

Microservices can communicate using different ways, but all of them can be classified into two groups – synchronous and asynchronous. The client sends a request and waits for a response from the service in a synchronous communication style. It results in tight runtime coupling because both the client and service must be available for the duration of the request. Usually, HTTP/HTTPS protocols are used for synchronous communication. The main advantage of this communication is that system is simple and easily implemented. Also, there is no intermediate component, such as a message broker. In asynchronous communication, microservices communicate by exchanging messages over messaging channels based on advanced message

queuing protocol (AMQP). All counter parties can send messages, and senders do not wait for the response message. There are several different asynchronous communication patterns, such as request-response, publish-subscribe, and notification. Asynchronous communication has the following benefits as loose runtime coupling and improved availability. However, it has a more complex implementation. The following message-based technologies as RabbitMQ, Apache Kafka, etc., are the implementation of asynchronous communication between microservices [9, 10]. It must be noted that the most popular communications technologies used for microservices are based on HTTP protocol and asynchronous message patterns [1, 10, 11, 12].

gRPC is an open-source Remote Procedure Call (RPC) framework developed by Google. It enables to establish communication between server and client applications transparently in any environment. Before gRPC became open source in March 2015, it had been used as a single general-purpose RPC infrastructure to connect the large number of microservices running within and across Google data centers for over a decade [13, 14].

GraphQL is a query language for APIs and a runtime for fulfilling those queries with existing data. GraphQL was developed internally by Facebook in 2012 and was published to the community in 2015. The key functionality of the GraphQL framework is a query language that allows clients to define the structure of the data required, and the same structure of the data is returned from the server [15, 16, 17].

It must be noted that it is common practice to use several communication technologies to develop the microservices-based application.

2.2 Architecture Patterns

Taibi et al. conducted a systematic literature review and identified three microservice orchestration architecture patterns that also include communication and coordination of the microservices. Patterns were classified as API Gateway, service discovery, and hybrid. A summary of the advantages, disadvantages of each architecture pattern was presented in the paper as well.

API Gateway operates as the entry point of the system that routes the requests to the appropriate microservices. This pattern is technology agnostic but usually is implemented using the HTTP protocol. API Gateway has the following advantages as ease of extension, market-centric architecture, and backward compatibility. The high complexity of implementation, low reusability, and low scalability can be mentioned as disadvantages of the pattern [18, 19].

The service discovery pattern uses a different approach, i.e., the client can communicate with each service directly without an intermediate layer. Domain name system (DNS) address resolution into internet protocol (IP) address must be supported to achieve end-to-end communication between services. Pattern relies on Service Register service that performs similarly as DNS. Advantages of service discovery patterns are ease of development, maintainability, migration, communication, health management. As pattern disadvantages, it can be mentioned high coupling

between the client and the service registry, high complexity of Service Registry, high complexity of distributed system [18, 19].

The Hybrid pattern combines Service Registry and API-Gateway and replaces the API-gateway with the message bus. Clients communicate only with the message bus that operates as a registry and gateway. Services communicate with each other via message bus, and direct communication between microservices is not used. Advantages of the pattern are ease of migration while disadvantages are high coupling between services and message bus [18].

2.3 Streaming and Distributed Cache

Smid et al. discussed the balance between the performance and coupling and pointed out situations where suggested architectures were appropriate. The authors introduced a streaming platform based on the message bus (Kafka) and data change capture platform (Debezium) to synchronize data between different databases effectively. Streaming is a totally different approach than orchestration and communication patterns mentioned in the previous chapter. Service generating event notifies other services by using streaming events to the message bus. Therefore, almost all communication is performed by consuming events from the message bus or database. The proposed solution has the following limitations as overhead for deployment and maintenance for applying the streaming platform. The microservices need to be synchronized under a similar data model with the master system, and additional source code must be introduced [19]. A distributed cache was introduced to improve communication performance. The advantages of using a distributed cache are performance, scalability, and ease of migration, while a disadvantage is high complexity. The communication performance decreases significantly when data frequently changes. The authors concluded that the message broker is an efficient way of communication between microservices, and the publish/subscribe model is very flexible and provides a faster mechanism than HTTP request with the benefit of persistent messages [20].

2.4 Microservices and SOA Communication

Černy et al. performed a detailed study analyzing differences between microservices architecture and SOA. Microservices provide decomposition preferring smart services while considering simple routing mechanisms without the global governance notable in SOA. This leads to higher service autonomy and decoupling since services do not need to make agreements on the global level [21]. In general, there are two well-defined approaches used to coordinate services, i.e., using a central orchestrator or decentralized distributed way. The centrally orchestrated approach is the typical SOA pattern, while the distributed approach is dominant for microservices-based applications. These approaches are named orchestration and choreography, respectively. Service orchestration works as a centralized business process, coordinating activities over different services and combining the outcomes. The choreography

works without a centralized element. The control logic is described by message exchanges and rules of interactions as well as agreements among interacting services. Both SOA and microservice use the same communication technologies, i.e., HTTP (Rest) and messaging [21].

2.5 Communication Security

Yarygina et al. analyzed security challenges in a microservice architecture. Potential threats in microservice communication were identified that are attacks on the network stack and protocols, attacks against protocols specific to the service integration style (SOAP, RESTful Web Services). Security threat mitigation techniques were proposed. The authors highlighted the leading microservice security industry practices such as Mutual Authentication of Services using Mutual Transport Layer Security, Principal Propagation via Security Tokens. The authors proposed the method that combined both techniques and presented proof-of-concept evaluation results [22]. Walsh et al. introduced new comprehensive, automated, and fine-grained mutual authentication mechanisms. To ensure a secure connection between microservices, the authors suggested using a combination of authentication and attestation. The proposed attestation mechanisms were built on top of standard transport layer security channels and certificates [23].

2.6 Performance Evaluation

Hong et al. provided a detailed study on the performance evaluation of RESTful API and RabbitMQ for Microservice Web Application. Experimental results showed that when a large number of users sent requests to the web application in parallel, RabbitMQ as the Message-oriented middleware provided more stable results compared to the RESTful API. On the other hand, the RESTful API has shown better request-response performance results [24].

Fernandes et al. performed a comparison study between a RESTful Web service and the AMQP protocol for exchanging messages between clients and servers. The final results showed that for applications that exchange a large amount of data, the best approach is to use the RabbitMQ server and Back-End Service to consume the messages, process them, and send them to the database. As a result, fewer messages per second were sent, time for exchange increased, and even more resources were used evaluating RESTful Web service [25].

It can be summarized that different factors like request load, IT environment and network technologies determine communication performance between microservices. It cannot be unambiguously defined which of the communication technology is faster. It depends on the specific application. It can be stated that asynchronous communication is a more robust and stable communication mechanism than HTTP (Rest) and enforces the microservice autonomy.

3 RESEARCH METHODOLOGY

This section describes the research methodology used to evaluate and compare chosen communication technologies:

- The experimental design is provided in Section 3.1.
- Criteria and metrics used for the evaluation and comparison are described in Section 3.2.
- Topologies used for the evaluation and comparison are described in Section 3.3.
- Tools, libraries, and IT equipment used to perform experiments are listed in Section 3.4.

3.1 Experimental Design

A set of five microservices were created and connected in a line topology to evaluate and compare communication technologies (Figure 1). RPC technique was used for communication between microservices. Only pure server and client functionality were implemented in each microservice, and the server component exposes API, and the client component is used for executing RPC. The experiment aimed to evaluate and compare communication based on the remote procedure call (RPC). RPC technique was chosen because it supports the same functionally as a function call and in-process-based communication.

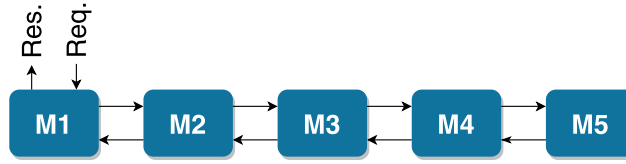

Figure 1. The topology of microservices used for the experiment. Where: Req. is request, Res. is response and Mi is microservice.

The full flow of message processing in the experiment is defined as follows:

$$t = \left(\sum_{i=1}^{n=4} M_i \rightarrow M_{i+1} \right) + \left(\sum_{i=0}^{n=3} M_{5-i} \rightarrow M_{5-i-1} \right), \tag{1}$$

where: t is the time used to process the message, M_i is microservice with index i, and arrow ($\rightarrow$) is request/response operation. Different size and complexity messages were sent to evaluate and compare the impact of message size, message complexity and request load on the latency and throughput of each technology. The time duration between requests sent from M1 to M5 and the response received from M5 to M1 was measured and used to calculate latency and throughput.

Different data models were used (Figure 2) for messages to measure message size and complexity impact on latency and throughput. The *TestModelOnlyText* data model was used to measure impact on message size, the *TextField* value was set to 10, 1 000, 100 000, 1 000 000 characters. The *TestModelAllTypes* data model was used to measure impact on message complexity, especially on serialization. Messages with 10, 100, 1 000 and 10 000 properties were used.

```
public class TestModelOnlyText
{
    public string TextField { get; set; }
}

public class TestModelAllTypes
{
    public string TextField { get; set; }
    public bool BoolField { get; set; }
    public byte ByteField { get; set; }
    public DateTime DateTimeField { get; set; }
    public decimal DecimalField { get; set; }
    public double DoubleField { get; set; }
    public float FloatField { get; set; }
    public int IntField { get; set; }
    public short ShortField { get; set; }
    public long LongField { get; set; }
}
```

Figure 2. Data models used in experiment

The latency was measured by processing different size and complexity messages while requesting with 1 client. The throughput was measured by processing the same messages as it was processed in latency tests, but with the increased request load. During the experiment the request load started with 10 clients and was constantly increased by 10 clients each 30 seconds until it reached 200 clients.

3.2 Criteria

This section provides information about criteria that were taken into account while analyzing different communication technologies. Previous research performed by different authors was mainly focused on performance evaluations and comparison. In order to cover more communication aspects that can potentially be a challenge during legacy monolith application decomposition to microservices, a set of new criteria was introduced. These criteria were chosen to compare each communication technology in the context of communication between microservices decomposed from monolith application.

Performance: communication technology performance is measured and analyzed by latency and throughput. Latency was measured by time in milliseconds since the request was sent till the response was received. Throughput was measured

by number successful request per second (RPS). The successful request was considered if response was received within 1 second.

Messages size: to determine the potential technology impact on network load requests and response size in bytes were measured.

Memory size: to evaluate how much memory is needed to run an application with each communication technology, application memory usage in bytes was measured.

Storage size: to evaluate how much storage is needed to store an application with each communication technology, storage usage in bytes was measured.

Boot time: application boot time in seconds was measured to determine how much time is needed to start the application.

Architecture: to highlight the specific impact of each technology regarding application architecture.

Topology: technology impact on the topology of microservices. More details about the topology used in the experiment are provided in Section 3.3.

Used applications and libraries: to analyze the availability of the particular library.

3.3 Tools

All microservices were written using C Sharp and .Net Core [26]. All coding and testing were done using Microsoft Visual Studio 2022 IDE [27]. All libraries used in the research were downloaded from NuGet gallery [28]. Latency tests were conducted using BenchmarkDotNet library [29]. Throughput tests were executed by using NBomber library [30]. Network data was analyzed by Wireshark application [31].

All experiments were performed on a computer with the following specification: CPU – Core i7 9850H, memory – 30 GB RAM, storage – 512 GB SSD, and OS – Windows 10 Enterprise (20H2). All applications were run on a computer, no external devices or networks were used.

The experiment can be reproduced on a computer with Visual Studio 2022 IDE, RabbitMQ (3.10.0 version) and Kafka (3.2.0) installed. The source code used in the experiment and experiment results are freely accessible and can be found under the following link `https://bitbucket.org/justas_kazanavicius/communicationexperiment`.

3.4 Topology

Three different topologies of microservices were chosen to analyze how communication technology influences topology criteria defined in the previous section (Figure 2). *Linear (single receiver) topology* – request processing flow has only one way in, and each microservice is involved in request processing. *Tree type topology* – request processing flow has a few ways. Middleware microservices work as gateways. *Star*

type topology (multiple receivers) – first microservice works as a gateway and routes request to specific microservice. Those topologies were chosen because each of them represents a different way how data can be processed and communication between microservices can be established.

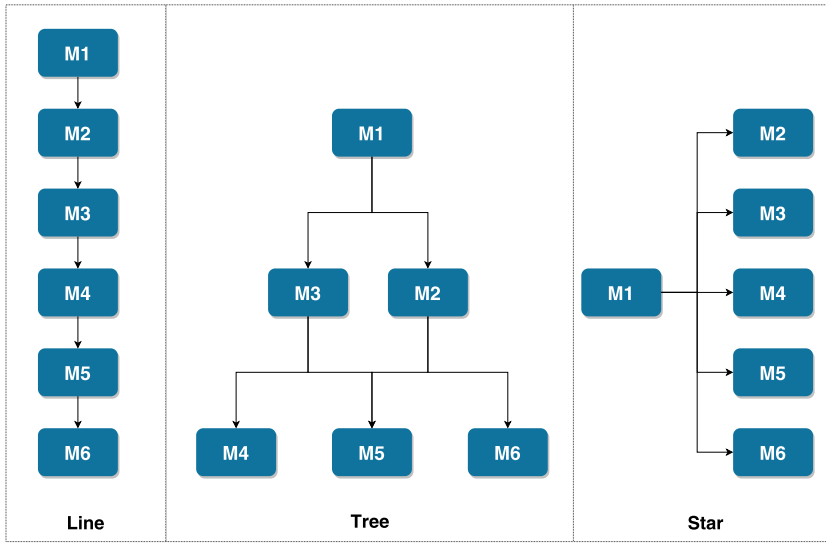

Figure 3. Topologies used in the research

4 RESULTS OF EXPERIMENT

This section provides results obtained during the evaluation of five communication technologies: HTTP (Rest API), RabbitMQ, Kafka, gRPC, and GraphQL. Deeper discussions on results are provided in Section 5. Each technology subsection is divided into four subsections to provide more details in terms of experiment results:

- Latency test results are provided in table.
- Throughput test results are provided in chart.
- Results of other metrics : Request/Response size, Microservice application size, Memory usage size, Boot time.
- Architecture – technology and libraries impact to the architecture.
- Topology – technology and libraries impact to the topology.
- Libraries – a list of libraries that were used in the experiment to establish a connection between microservices via particular technology.

4.1 HTTP (Rest API)

Latency results: Results of the latency test are shown in Table 1. The best results, 7.265 ms, were achieved processing 1 000 characters messages. The worst results, 31.410 ms, were achieved processing 1 000 000 characters messages.

Message Size	Mean	Median	Min	Max
10 characters	7.527 ms	7.404 ms	5.801 ms	9.923 ms
1 000 characters	7.265 ms	7.149 ms	5.685 ms	9.459 ms
100 000 characters	11.745 ms	11.356 ms	9.543 ms	15.875 ms
1 000 000 characters	31.410 ms	30.563 ms	25.304 ms	44.212 ms
10 properties	8.236 ms	8.055 ms	6.465 ms	11.516 ms
100 properties	8.459 ms	8.408 ms	6.396 ms	10.940 ms
1 000 properties	9.826 ms	9.726 ms	7.567 ms	13.284 ms
10 000 properties	21.779 ms	21.096 ms	19.010 ms	26.546 ms

Table 1. Latency test results for message processing with HTTP Rest

Throughput results: Throughput results of the load test are shown in Figure 4. The best average results, 99.7 RPS, were achieved processing 10 properties messages. The worst average results, 4.7 RPS, were achieved processing 1 000 000 characters messages.

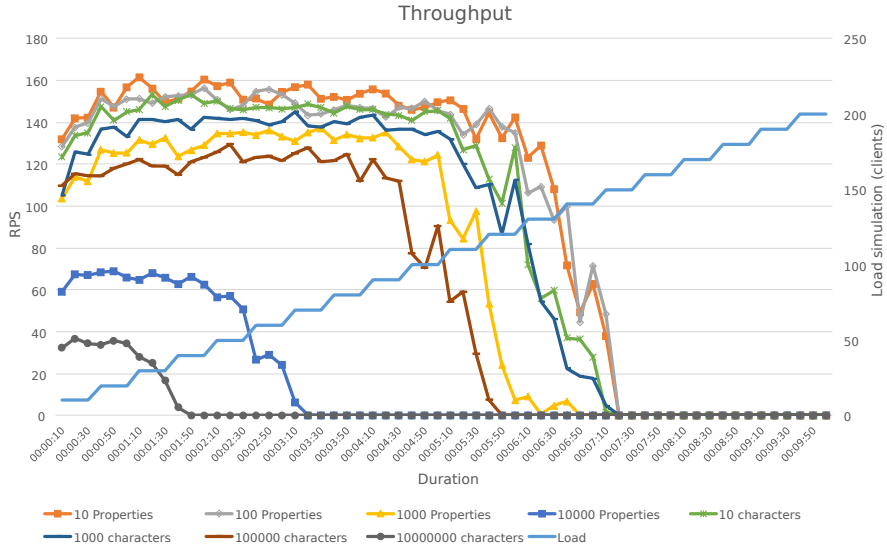

Figure 4. Load test results for message processing with HTTP Rest

Results of other metrics: Other results obtained during the experiment are presented in Table 2.

Metric	Result
Request/Response size	172 B/185 B (payload 26 B)
Microservice application size	4.71 MB (empty 159 kB)
Memory usage size	69 MB (empty 9 MB)
Boot time	3.1 s

Table 2. Results of HTTP Rest experiment measurements

Architecture: In order to communicate via Rest API, microservice has to have at least 3 additional components: Rest API, Controller, and Rest Client (Figure 5). Rest API component is exposing the HTTP server and routes requests to the Controller component, which operates as a facade for business logic. Rest Client is needed to make requests to Rest APIs exposed by other microservices.

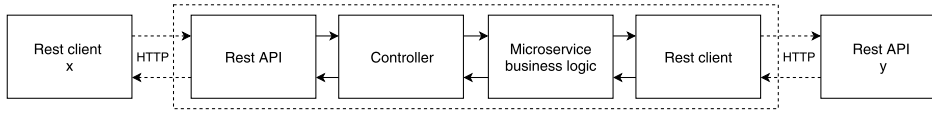

Figure 5. The architecture of Rest API in microservice

Topology: Microservices M1–M5 have to know how to reach the next microservice (M1 → M2, M2 → M3 etc.) when a linear topology is used. Microservice M6 only exposes Rest API. The tree type topology shows that microservices M1, M2, and M3 each have two dependencies (M1 should know URLs of M2 and M3). M4, M5, and M6 only expose the Rest API. In the star type topology, the M1 microservice has to know the URLs of all micro-services.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via HTTP Rest technology is provided bellow:

- Microsoft.AspNetCore.App (Version 6.0.7),
- Microsoft.NETCore.App (Version 6.0.7),
- Swashbuckle.AspNetCore (Version 6.2.3),
- System.Net.Http.Json (Version 6.0.0).

4.2 RabbitMQ

Latency results: Results of the latency test are shown in Table 3. The best results, 2.976 ms, were achieved processing 1 000 characters messages. The worst results, 118.657 ms, were achieved processing 1 000 000 characters messages.

Throughput results: Throughput results of the load test are shown in Figure 6. The best average results, 231.5 RPS, were achieved processing 10 characters messages. The worst average results, 0.01 RPS, were achieved processing 1 000 000 characters messages.

Message Size	Mean	Median	Min	Max
10 characters	2.982 ms	2.946 ms	2.551 ms	3.491 ms
1 000 characters	2.976 ms	2.939 ms	2.721 ms	3.712 ms
100 000 characters	5.166 ms	5.023 ms	4.674 ms	6.360 ms
1 000 000 characters	118.657 ms	116.824 ms	73.740 ms	157.821 ms
10 properties	4.354 ms	4.265 ms	3.059 ms	6.605 ms
100 properties	3.197 ms	3.108 ms	2.843 ms	4.387 ms
1 000 properties	4.752 ms	4.670 ms	4.278 ms	5.875 ms
10 000 properties	20.310 ms	19.974 ms	19.529 ms	23.098 ms

Table 3. Latency test results for message processing with RabbitMQ

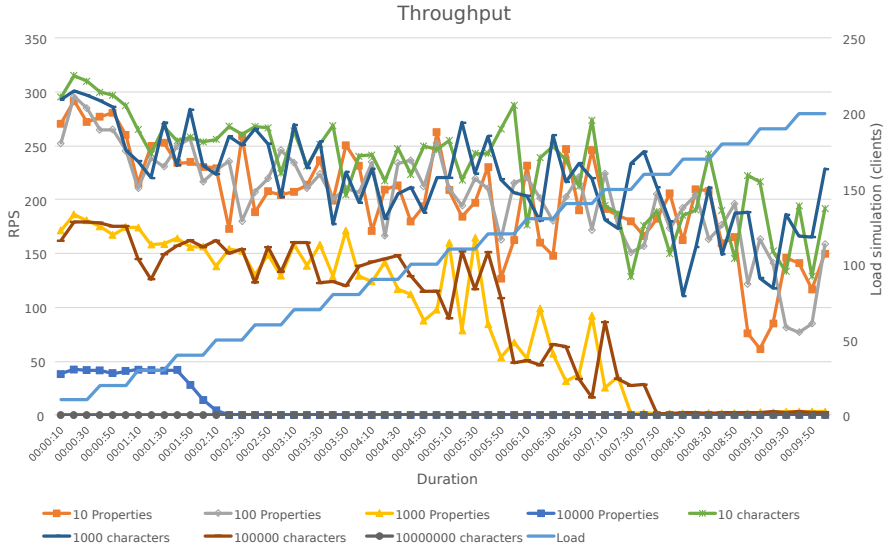

Figure 6. Load test results for message processing with RabbitMQ

Results of other metrics: Other results obtained during the experiment are presented in Table 4.

Metric	Result
Request/Response size	206 B/225 B (payload 26 B)
Microservice application size	2.26 MB (empty 159 kB)
Memory usage size	23 MB (empty 9 MB)
Boot time	3.8 s

Table 4. Results of RabbitMQ experiment measurements

Architecture: In order to utilize RabbitMQ as RPC, microservices have to contain two additional components: Rabbit server and Rabbit client (Figure 7). Rabbit server consumes messages from queue x1 and routes them to business logic where messages are processed and moved to rabbit client to publish them to queue y1. After pushing messages to queue y1 Rabbit client starts listening to queue y2 for a response. A message which is consumed from queue y2 goes from Rabbit client through business logic to Rabbit server where it is published to queue x2.

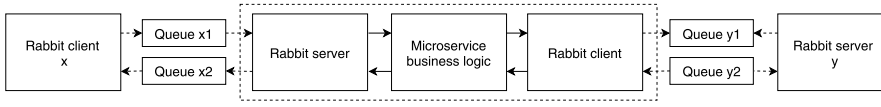

Figure 7. The architecture of RabbitMQ in microservice

Topology: Similar to HTTP communication Rabbit server component is not needed for those microservices which are only used as clients, and the client component is not needed for those microservices which are only used as servers. The most significant difference using RabbitMQ is that there is no need for microservices to know about each other's endpoints, such as IP address or hostname. Instead of communicating directly with each other microservices are communicating through RabbitMQ, which acts as a router. Clients are producers and produce messages to the RabbitMQ queue while servers are consumers and consume messages from the same RabbitMQ queue.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via RabbitMQ technology is provided bellow:

- Microsoft.NETCore.App (Version 6.0.7),
- RabbitMQ.Client (Version 6.3.0),
- Nito.AsyncEx (Version 5.1.2).

4.3 Kafka

Latency results: Results of the latency test are shown in Table 5. The best results, 7.191 ms, were achieved processing 10 characters messages. The worst results, 42.600 ms, were achieved processing 1 000 000 characters messages.

Throughput results: Throughput results of the load test are shown in Figure 8. The best average results, 93.3 RPS, were achieved processing 10 characters messages. The worst average results, 1.6 RPS, were achieved processing 1 000 000 characters messages.

Results of other metrics: Other results obtained during the experiment are presented in Table 6.

Architecture: In order to utilize Kafka as RPC, microservices have to contain two additional components: Kafka server and Kafka client (Figure 9). Kafka

Message Size	Mean	Median	Min	Max
10 characters	7.191 ms	7.130 ms	6.836 ms	8.023 ms
1 000 characters	8.073 ms	8.016 ms	5.398 ms	11.428 ms
100 000 characters	11.643 ms	11.397 ms	8.811 ms	15.241 ms
1 000 000 characters	42.600 ms	42.187 ms	35.172 ms	54.572 ms
10 properties	8.183 ms	8.115 ms	6.009 ms	11.441 ms
100 properties	7.761 ms	7.605 ms	5.782 ms	10.627 ms
1 000 properties	12.116 ms	11.566 ms	8.704 ms	16.905 ms
10 000 properties	28.612 ms	28.366 ms	24.451 ms	34.667 ms

Table 5. Latency test results for message processing with Kafka

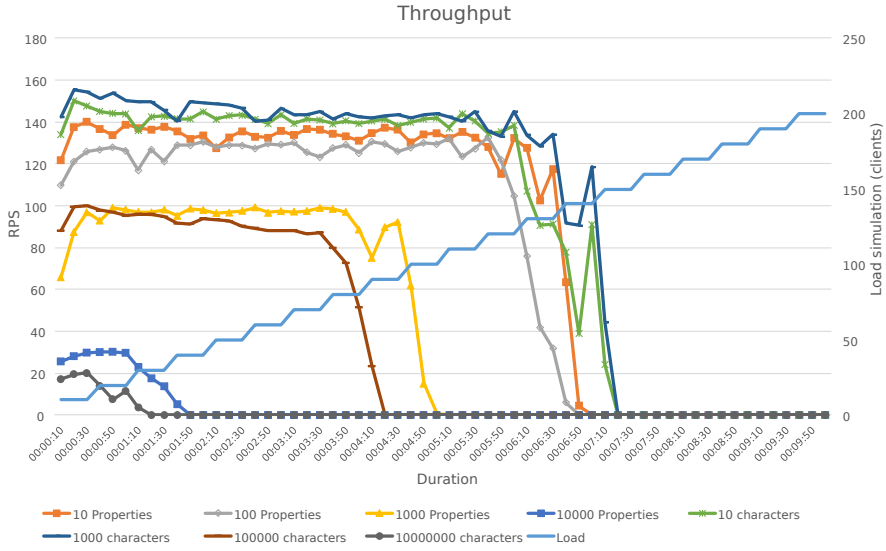

Figure 8. Load test results for message processing with Kafka

server consumes messages from topic x1 and routes them to business logic where messages are processed and moved to Kafka client to publish them to topic y1. After pushing messages to topic y1 Kafka client starts listening to topic y2 for a response. A message which is consumed from topic y2 goes from Kafka client through business logic to Kafka server where it is published to topic x2.

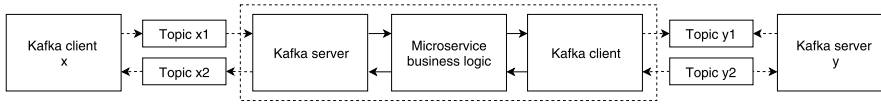

Figure 9. The architecture of Kafka in microservice

Metric	Result
Request/Response size	219 B/252 B (payload 26 B)
Microservice application size	2.18 MB (empty 159 kB)
Memory usage size	40 MB (empty 9 MB)
Boot time	2.6 s

Table 6. Results of Kafka experiment measurements

Topology: Kafka server component is not needed for those microservices which are only used as clients, and the client component is not needed for those microservices which are only used as servers. Similar to the RabbitMQ, the most significant difference comparing to HTTP Rest, gRPC and GraphQL is that there is no need for microservices to know about each other's endpoints, such as IP address or hostname. Instead of communicating directly with each other microservices are communicating through Kafka, which acts as a router. Clients are producers and produce messages to the Kafka topic while servers are consumers and consume messages from the same Kafka topic.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via Kafka technology is provided bellow:

- Microsoft.NETCore.App (Version 6.0.7),
- Simple.Kafka.Rpc (Version 1.8.3).

4.4 gRPC

Latency results: Results of the latency test are shown in Table 7. The best results, 6.761 ms, were achieved processing 1 000 characters messages. The worst results, 35.384 ms, were achieved processing 1 000 000 characters messages.

Message Size	Mean	Median	Min	Max
10 characters	7.004 ms	6.787 ms	5.336 ms	9.455 ms
1 000 characters	6.716 ms	6.729 ms	5.396 ms	8.136 ms
100 000 characters	10.188 ms	10.021 ms	7.976 ms	13.537 ms
1 000 000 characters	35.384 ms	34.262 ms	25.406 ms	52.120 ms
10 properties	8.022 ms	7.929 ms	6.651 ms	9.874 ms
100 properties	8.183 ms	8.211 ms	6.692 ms	10.243 ms
1 000 properties	8.501 ms	8.487 ms	7.354 ms	10.228 ms
10 000 properties	14.855 ms	14.562 ms	12.778 ms	18.263 ms

Table 7. Latency test results for message processing with gRPC

Throughput results: Throughput results of the load test are shown in Figure 10. The best average results, 170.1 RPS, were achieved processing 1 000 characters messages. The worst average results, 5.0 RPS, were achieved processing 1 000 000 characters messages.

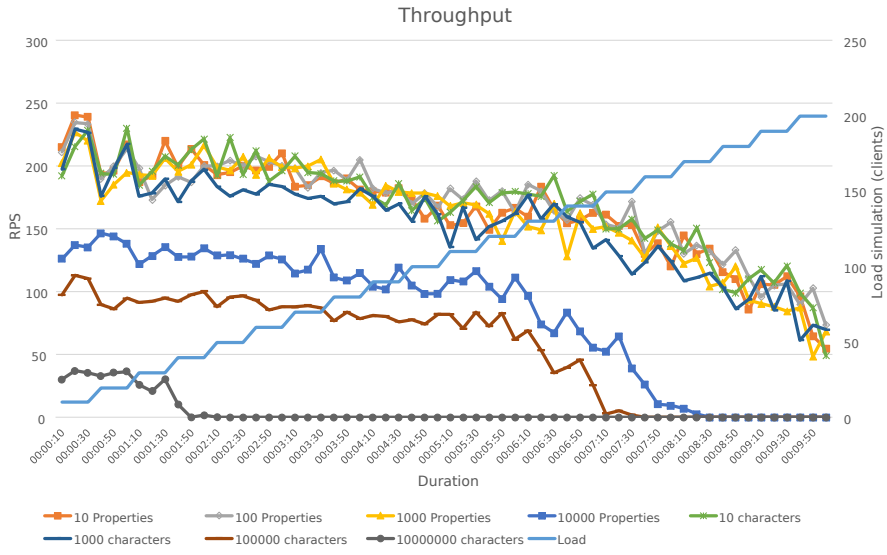

Figure 10. Load test results for message processing with gRPC

Results of other metrics: Other results obtained during the experiment are presented in Table 8.

Metric	Result
Request/Response size	363 B/162 B (payload 12 B)
Microservice application size	1.85 MB (empty 159 kB)
Memory usage size	70 MB (empty 9 MB)
Boot time	3.4 s

Table 8. Results of gRPC experiment measurements

Architecture: In order to communicate via gRPC microservice has to have at least three additional components: gRPC server, Service, and gRPC Client (Figure 11). gRPC server component is exposing gRPC server and sends requests to Service component which acts as a facade for business logic. gRPC Client sends a request to gRPC server y. The components and flow are very similar to the Rest API case.

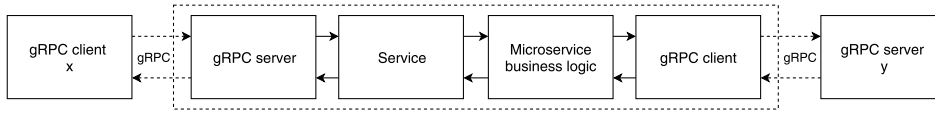

Figure 11. The architecture of gRPC in microservice

Topology: In terms of topology, gRPC and Rest API have no difference. Microservices M1–M5 have to know how to reach the next microservice when a linear topology is used. Micro-service M6 only exposes the gRPC server. Microservices M1, M2, and M3 have two dependencies in the tree type topology. Microservices M4, M5, and M6 only expose the gRPC server. In the star type topology, the M1 microservice has to know all microservices URLs.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via gRPC technology is provided bellow:

- Microsoft.NETCore.App (Version 6.0.7),
- protobuf-net.Grpc (Version 1.0.152),
- protobuf-net.Grpc.AspNetCore (Version 1.0.152),
- Grpc.Net.Client (Version 2.45.0).

4.5 GraphQL

Latency results: Results of the latency test are shown in Table 9. The best results, 7.711 ms, were achieved processing 1 000 characters messages. The worst results, 51.170 ms, were achieved processing 10 000 properties messages.

Message Size	Mean	Median	Min	Max
10 characters	7.755 ms	7.718 ms	5.945 ms	10.69 ms
1 000 characters	7.711 ms	7.376 ms	5.846 ms	12.02 ms
100 000 characters	12.349 ms	11.392 ms	9.083 ms	18.83 ms
1 000 000 characters	29.575 ms	29.137 ms	24.780 ms	38.70 ms
10 properties	10.498 ms	10.302 ms	7.652 ms	14.67 ms
100 properties	9.860 ms	9.624 ms	8.383 ms	12.63 ms
1 000 properties	13.262 ms	13.261 ms	10.921 ms	15.73 ms
10 000 properties	51.170 ms	49.828 ms	44.979 ms	65.10 ms

Table 9. Latency test results for message processing with GraphQL

Throughput results: Throughput results of the load test are shown in Figure 12. The best average results, 185.5 RPS, were achieved processing 10 properties messages. The worst average results, 4.8 RPS, were achieved processing 1 000 000 characters messages.

Results of other metrics: Other results obtained during the experiment are presented in Table 10.

Topology: GraphQL, gRPC, and Rest API have no big difference in terms of topology. All technologies use client/server synchronous communication model. In order to establish communication, a client has to know the server endpoints such as IP address or host name.

GraphQL is also a query language for APIs – a client has the ability to request very specific data from the server. Queries in GraphQL can be written in

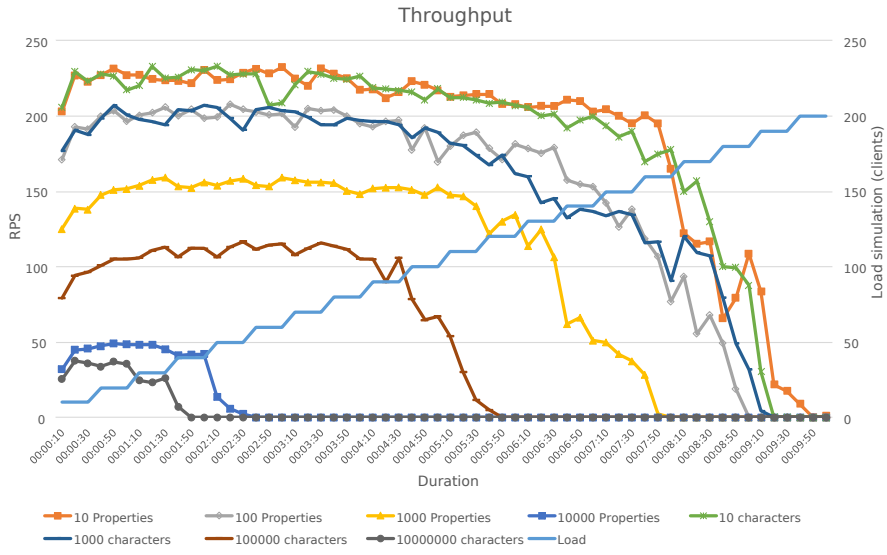

Figure 12. Load test results for message processing with GraphQL

Metric	Result
Request/Response size	390 B/843 B (payload 49 B)
Microservice application size	5.53 MB (empty 159 kB)
Memory usage size	65 MB (empty 9 MB)
Boot time	4.4 s

Table 10. Results of GraphQL experiment measurements

such a manner that would not only access separate properties but also follow references between them. Star type topology best utilizes this GraphQL feature.

Architecture: GraphQL flow is quite similar to REST API. Three additional components are needed to communicate via GraphQL: GraphQL Server, GraphQL abstraction layer, and GraphQL client (Figure 13). GraphQL is transport-layer agnostic, but the most common technology used for transport is HTML.

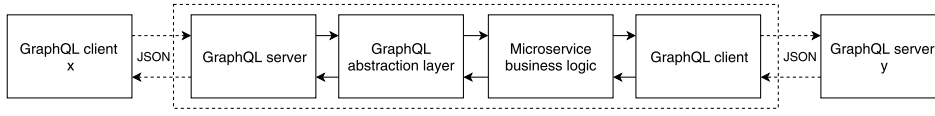

Figure 13. The architecture of GraphQL in microservice

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via GraphQL technology is provided bellow:

- Microsoft.NETCore.App (Version 6.0.7),
- GraphQL (Version 4.6.1),
- GraphQL.Client (Version 4.0.2),
- GraphQL.Client.Serializer.SystemTextJson (Version 4.2.2).

5 COMPARISON OF COMMUNICATION TECHNOLOGIES

This section compares communication technologies in different aspects based on the obtained results of the executed experiments. Section 5.1 provides details about available libraries for each technology. Section 5.2 gives an overview of the components used for each technology and highlights specific requirements for some technologies. Section 5.3 analyses the impact of the communication technology on the topology. Performance evaluation is presented in Section 5.4 using different aspects. The last section evaluates different metrics of each technology.

5.1 Libraries

Many different libraries can be chosen for HTTP Rest implementation mainly because it is the oldest and relatively simple technology. RabbitMQ and Kafka are also very popular technologies, so it also have quite a few libraries. GraphQL and gRPC are quite new technologies, and not so many libraries exist in the market. Microsoft .Net framework has built-in support and provides libraries for HTTP Rest and gRPC communication technologies.

5.2 Architecture

HTTP Rest, gRPC, and GraphQL communication technologies have very similar architecture: one component is used to expose a server, the second one to translate from technology-specific message to business-specific, and the last component is used to send a message.

Communication models and methods have to be defined in *proto files and shared between microservices in order to use gRPC communication technology. Similar to gRPC *proto files, GraphQL has a schema. GraphQL schema contains information about server methods and data types.

RabbitMQ and Kafka are a message based technologies, and they are different than others used in the research. Communication between microservices is not point-to-point like in HTTP Rest, gRPC, and GraphQL. All communication in RabbitMQ is implemented via queues: microservices can publish to and consume from the queue. Similar to RabbitMQ, Kafka uses topics to implement communication. Two queues, or two topics in Kafka case, have to be created in order to implement RPC call between microservices: one for a request and the second for a response.

5.3 Topology

HTTP Rest, gRPC, and GraphQL technologies are independent of topology. Microservice has to know how to reach other microservice in order to establish communication i.e. it has to know the addresses of other microservices. It is a known problem, and there are many solutions how to solve it, but all of them increase the complexity of the solution, especially if scalability is needed.

RabbitMQ and Kafka technologies do not have this challenge because it works as an intermediary communication layer and all communication between microservices happens through it. Communication in RabbitMQ and Kafka is based on queues and topics. A microservice has to know only the name of the queue, or topic name in Kafka case, in order to communicate with other microservice. A few microservices can publish and consume the same queue or topic. It is a powerful feature to support scalability.

GraphQL best utilizes its features in star-type topology where one microservice acts as a gateway and others as data sources. Powerful GraphQL query language allows creating a specific request in such a way that it can fetch data from multiple data sources in one API call. This feature can potentially reduce the number of calls between microservices needed to implement the functionality.

5.4 Performance

Performance tests were executed to compare latency and throughput in the case of RPC calls between five microservices. No performance optimizations were applied to any technology during this experiment. Latency results based on message size in characters are shown in Figure 14. Latency results based on number of properties are shown in Figure 15.

The lowest latency results for string up to 1 000 000 characters were obtained by RabbitMQ technology. RabbitMQ RPC calls were 2 times faster than other technologies. It showed best results processing smallest messages (10 and 1 000 characters), the results were 2 times better than processing 100 000 character's messages. HTTP Rest, Kafka, gRPC and GraphQL showed similar latency results, however results obtained by gRPC were slightly better.

On the other hand, the RabbitMQ had the highest latency results while processing messages which consisted of 10 000 000 characters. It was from 3 to 4 times slower than the others. The best latency results for 10 000 000 characters messages were obtained by GraphQL and HTTP Rest technologies. Kafka was 40 % and gRPC was 16 % and slower than GraphQL and HTTP Rest technologies.

The lowest latency results for messages containing up to 1 000 properties were also obtained by RabbitMQ technology. RabbitMQ RPC calls were from 2 to 3 times faster than other technologies. It showed best results processing messages containing 100 properties, the results were 37 % better than processing messages containing 1 000 properties and 47 % better than processing messages containing 10 properties.

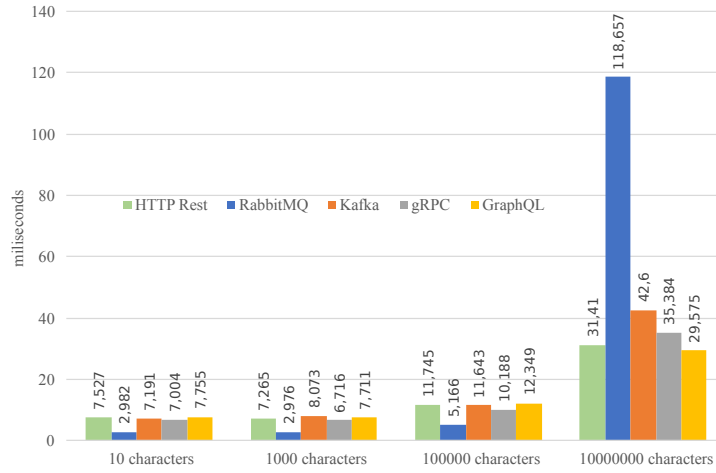

Figure 14. Latency tests results based on string size

HTTP Rest, Kafka and gRPC showed similar results for messages containing 10 and 100 properties.

The best results for communicating via messages containing 10 000 properties were obtained by gRPC technology. The binary serialization used by gRPC technology is faster than JSON serialization, which has been used by other technologies during the experiment, hence the more properties the message contains, the greater advantage gRPC has.

The GraphQL showed worst latency results for messages containing at least 10 properties. The more properties the message contained, the greater difference was comparing to other technologies. It was from 2 to 4 times slower than others while communicating via messages containing 10 000 properties.

Analyzing results it can be seen that the best RPC call latency results were achieved by RabbitMQ in 6 of 8 cases. However, the RabbitMQ was the slowest technology processing 10 000 000 characters messages. It can be summarized that the RabbitMQ has the lowest latency in case the message size is not bigger than 0.1 MB and data model contains up to 1 000 properties.

Throughput results for 10 characters size message are shown in Figure 16. The best throughput results were obtained by RabbitMQ technology, with average 231.6 RPS. The maximum result, 315.1 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by HTTP Rest technology with average 89.8 RPS and 140 clients limit.

Throughput results for 1 000 characters size message are shown in Figure 17. The best throughput results were obtained by RabbitMQ technology, with average 219.5 RPS. The maximum result, 300.1 RPS, was reached while requesting with

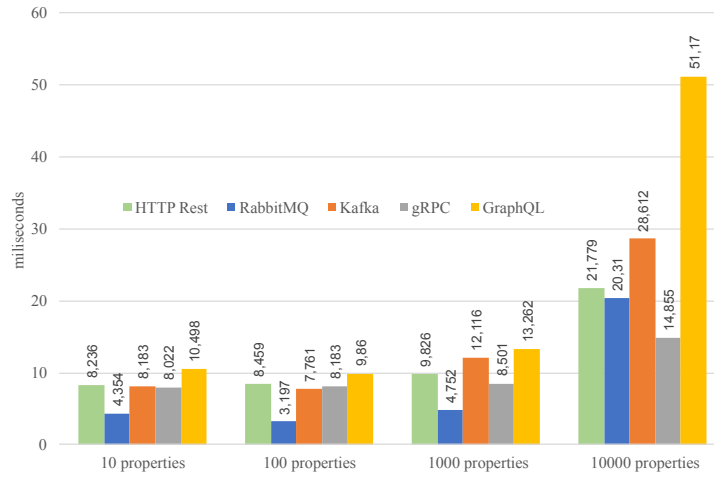

Figure 15. Latency tests results based on number of properties

10 clients. The worst RPC throughput test results were obtained by HTTP Rest technology with average 89.9 RPS and 140 clients limit.

Throughput results for 100 000 characters size message are shown in Figure 18. The best throughput results were obtained by RabbitMQ technology, with average 93.3 RPS. The maximum result, 179.3 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by Kafka technology with average 36.2 RPS and 80 clients limit.

Throughput results for 10 000 000 characters size message are shown in Figure 19. The best throughput results were obtained by gRPC technology, with average 5.0 RPS and 40 clients limit. The maximum result, 37.1 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by RabbitMQ technology with average 0.01 RPS.

Throughput results for 10 properties size message are shown in Figure 20. The best throughput results were obtained by RabbitMQ technology, with average 200.4 RPS. The maximum result, 291.5 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by Kafka technology with average 87.0 RPS and 140 clients limit.

Throughput results for 100 properties size message are shown in Figure 21. The best throughput results were obtained by RabbitMQ technology, with average 203.5 RPS. The maximum result, 295.5 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by Kafka technology with average 72.2 RPS and 130 clients limit.

Throughput results for 1 000 properties size message are shown in Figure 22. The best throughput results were obtained by gRPC technology, with average 161.9 RPS.

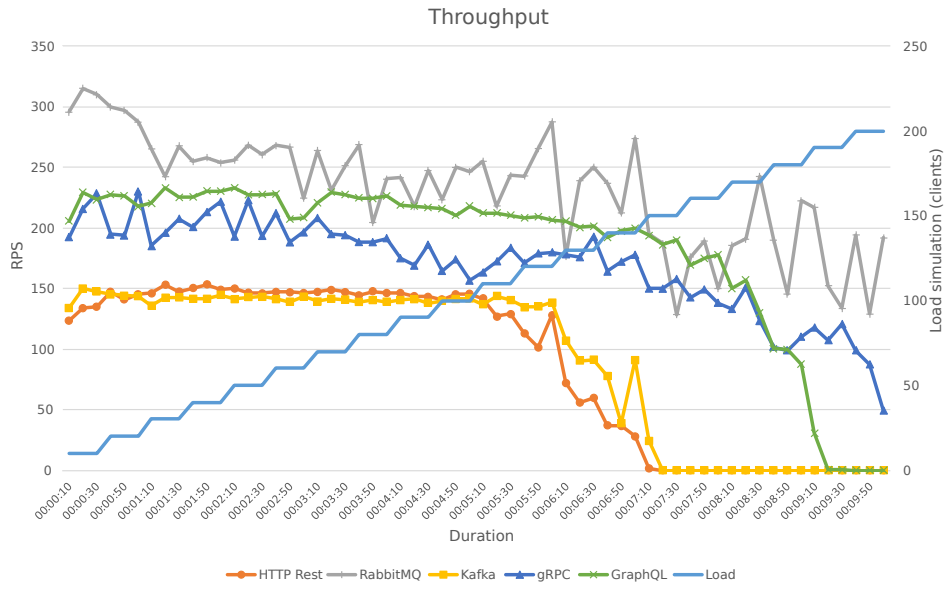

Figure 16. Throughput tests results for 10 characters size messages

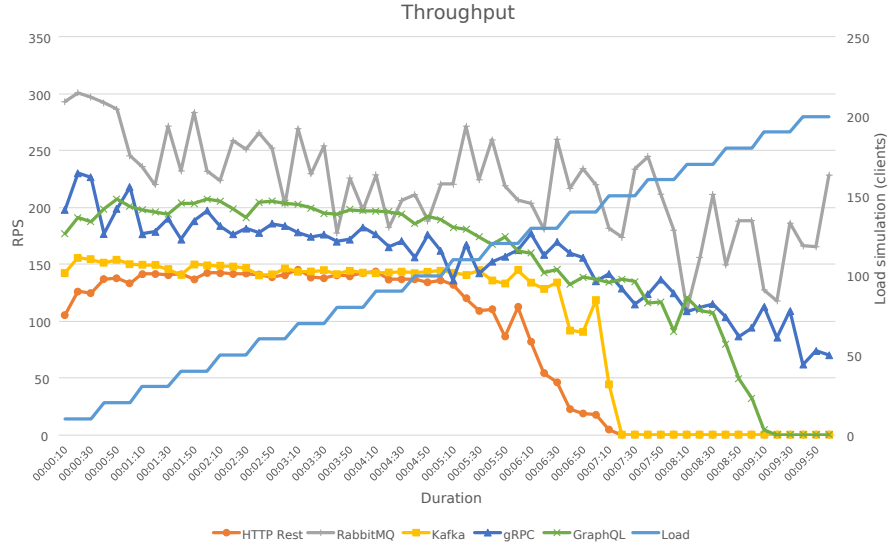

Figure 17. Throughput tests results for 1 000 characters size messages

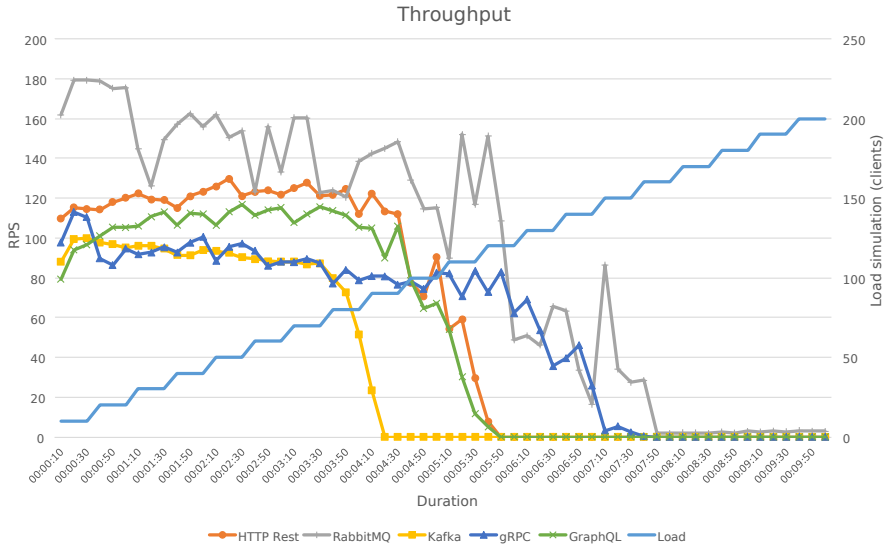

Figure 18. Throughput tests results for 100 000 characters size messages

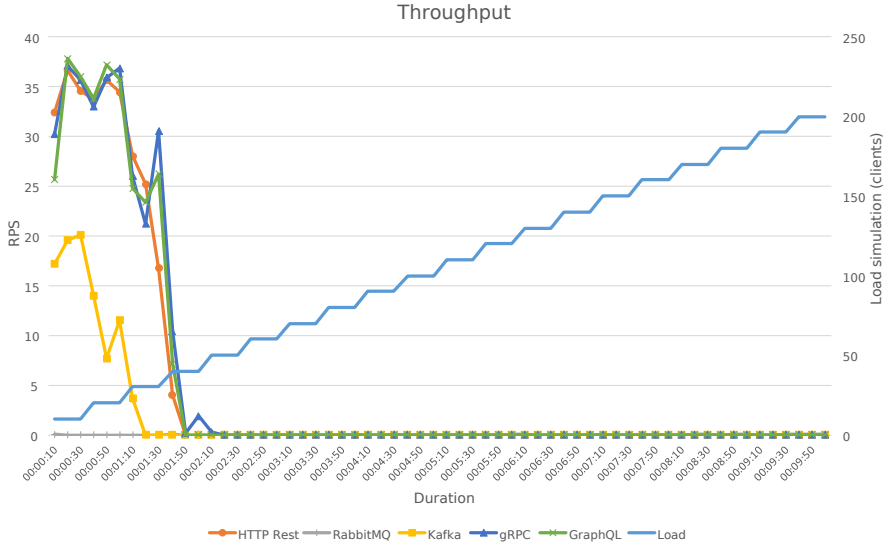

Figure 19. Throughput tests results for 10 000 000 characters size messages

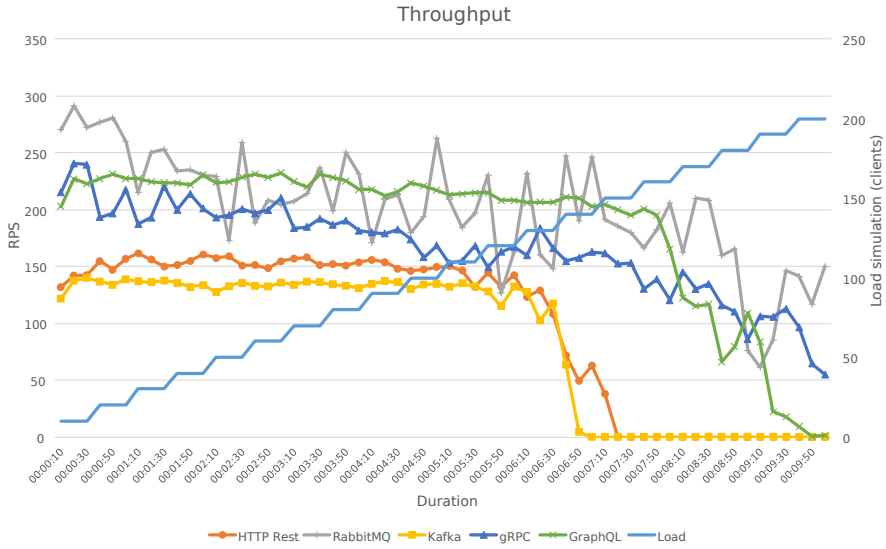

Figure 20. Throughput tests results for 10 properties size messages

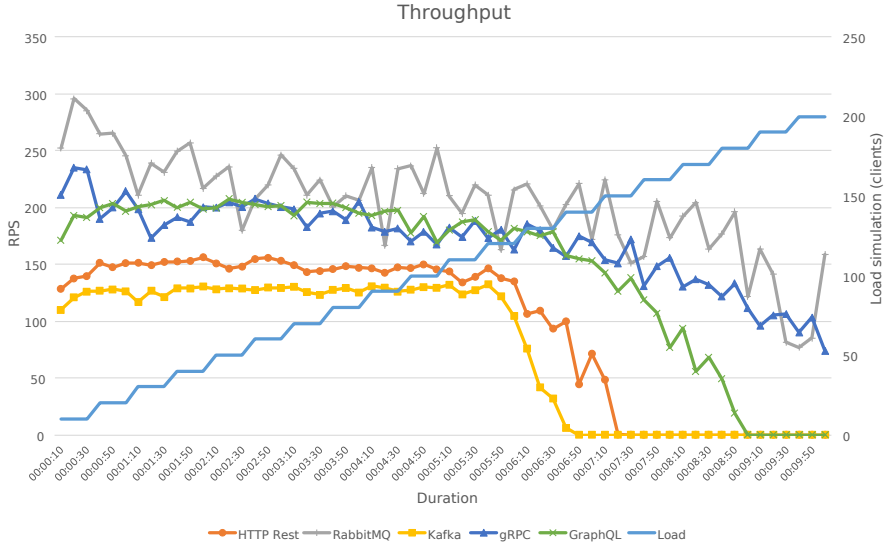

Figure 21. Throughput tests results for 100 properties size messages

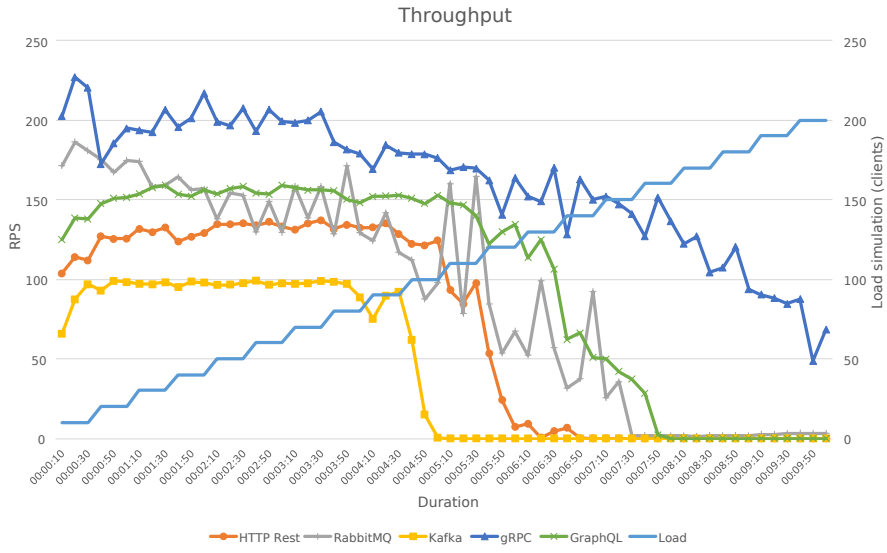

Figure 22. Throughput tests results for 1 000 properties size messages

The maximum result, 227.0 RPS, was reached while requesting with 10 clients. The worst RPC throughput test results were obtained by Kafka technology with average 43.7 RPS and 100 clients limit.

Throughput results for 10 000 properties size message are shown in Figure 23. The best throughput results were obtained by gRPC technology, with average 83.3 RPS. The maximum result, 146.6 RPS, was reached while requesting with 20 clients. The worst RPC throughput test results were obtained by Kafka technology with average 3.9 RPS and 30 clients limit.

It can be summarized that the best RPC call throughput results for smaller messages, up to 0.1 MB and up to 100 properties, were achieved by RabbitMQ technology. The best RPC call throughput results for bigger messages were achieved by gRPC communication technology. The worst throughput results in 5 of 8 cases were achieved by Kafka. The slowest technology processing biggest messages, 1 000 000 characters, was RabbitMQ.

However, if we compare latency distribution results (Figures 24, 25, 26, 27 28) we will see, that both Kafka and RabbitMQ can process more messages (with latency higher than 1 second) and works more stable when dealing with more then 50 clients load, comparing to HTTP Rest, gRPC and GraphQL technologies.

5.5 Metrics

The smallest size of request/response was obtained by HTTP Rest technology with total size 357 B. The GraphQL request/response was approximately 2–3 times big-

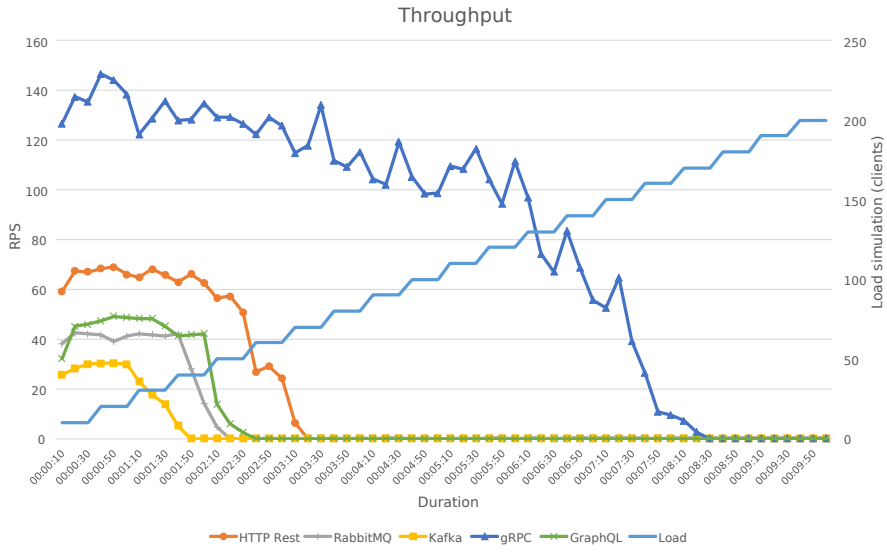

Figure 23. Throughput tests results for 10 000 properties size messages

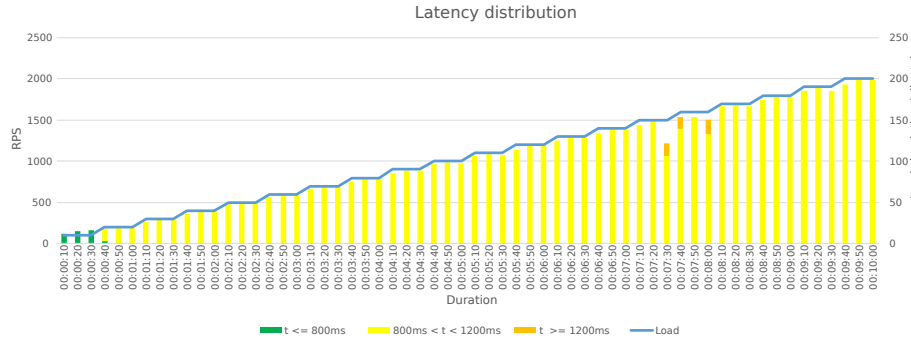

Figure 24. Kafka latency distribution for 1000 000 characters size messages

ger than others (Figure 29). If the message size is important criteria for choosing communication technology, then HTTP Rest is a recommended technology. On the other hand, GraphQL supports remote querying, so potentially, one GraphQL request/response could transfer as much information as a few requests/responses using other technologies.

A comparison of application size is presented in Figure 30. It can be seen that the biggest application size of 5 530 kB was obtained when GraphQL libraries were used for microservices. The smallest application size of 1 850 kB was when GraphQL libraries were included. Application size is independent of communication

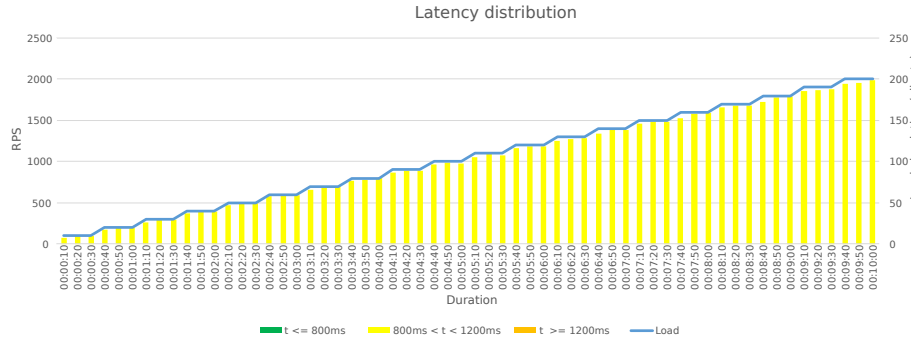

Figure 25. RabbitMQ latency distribution for 1000 000 characters size messages

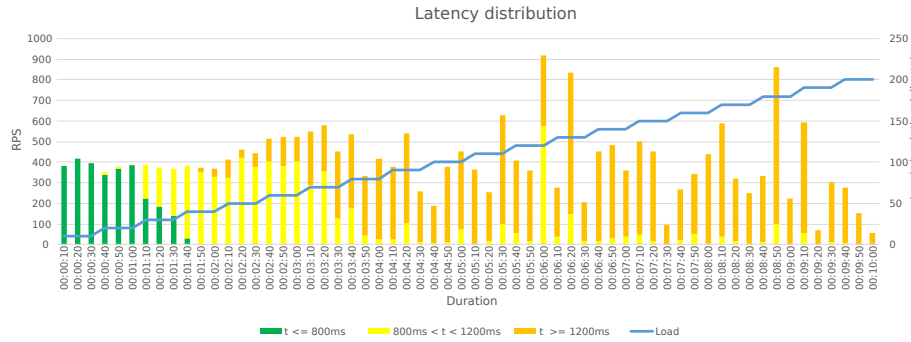

Figure 26. HTTP Rest latency distribution for 1 000 000 characters size messages

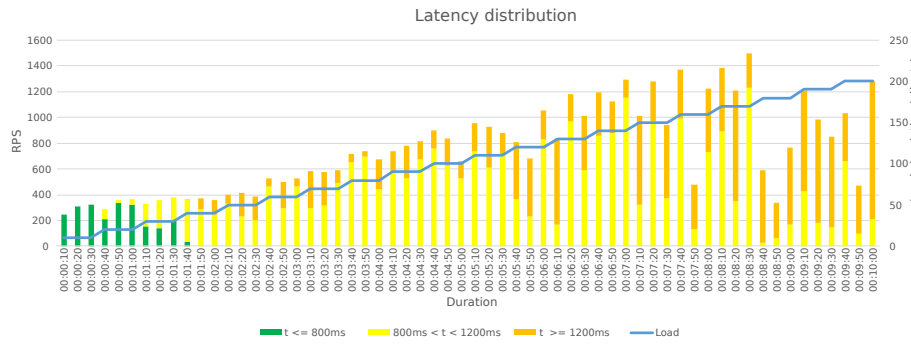

Figure 27. gRPC latency distribution for 1 000 000 characters size messages

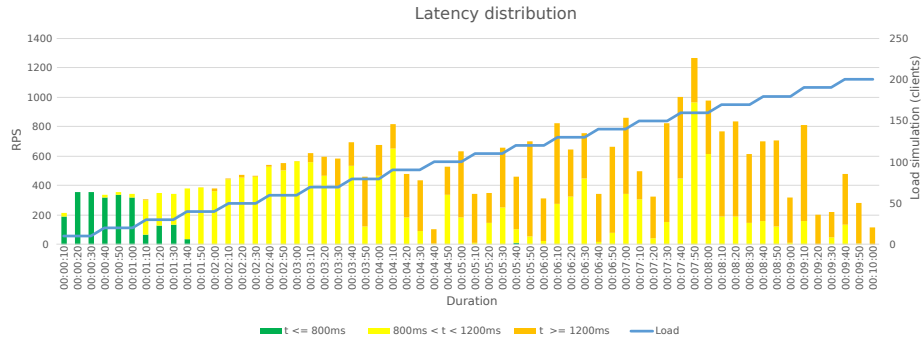

Figure 28. GraphQL latency distribution for 1 000 000 characters size messages

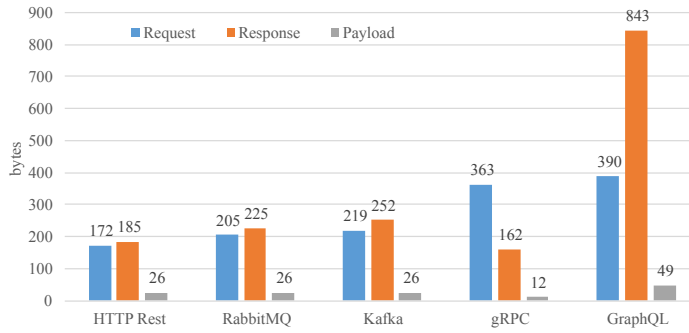

Figure 29. Request/Response size measured during experiment

technology. It depends on the way how it was implemented in the library. If the library size is too big, then the microservice developer can implement it by him selves.

The smallest amount of memory of 23 MB was allocated using RabbitMQ libraries, while gRPC used 70 MB of memory and it is almost three times more than RabbitMQ (Figure 31). It can be noted that if an application is running in an environment where memory is limited, then the best solution for implementing communication is between RabbitMQ and Kafka. Also, it must be pointed out that RabbitMQ and Kafka do require additional applications installed compared to other communication technologies.

A comparison of microservice boot time is shown in Figure 32. The longest boot time was spotted using GraphQL technology and it took 4.4 seconds while the shortest boot time of 2.6 seconds was obtained using Kafka technology. Boot time as well as the microservice size mostly depend on implementation, but not on communi-

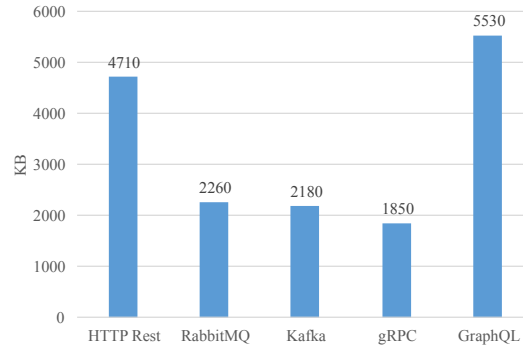

Figure 30. Application size measured during experiment

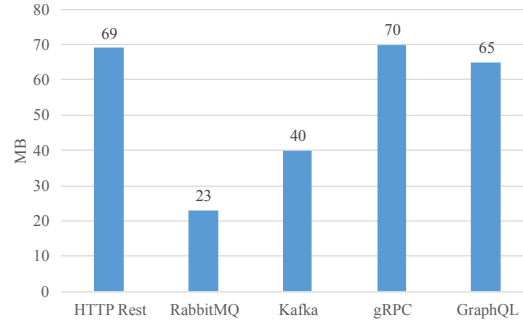

Figure 31. Memory consumption measured during experiment

cation technology itself and can be potentially improved by tuning implementation details.

6 DISCUSSION

One of the most significant challenge during monolith application transition into microservice architecture is data communication management. How to migrate from in-process method or function calls to inter-process communication? The high complexity, variety of architectural aspects, technological stack, and business objects make every application different and create challenges during monolith application decomposition to microservices. The introduced criteria allowed us to evaluate various aspects of communication technologies that are important while designing microservices. The key findings discovered in this study are provided below.

- If latency and throughput are main criteria during the transition from a monolith architecture to a microservice architecture, then RabbitMQ and gRPC are the

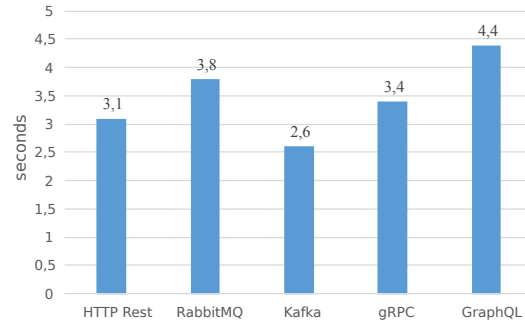

Figure 32. Boot time measured during experiment

most suitable technologies. RabbitMQ showed best results in RPC latency and throughput tests for small messages (up to 0.1 MB and data model up to 100 properties), while gRPC showed best results in RPC latency and throughput tests for big messages. The worst result were obtained by HTTP Rest and Kafka technologies.

- Kafka and RabbitMQ showed best throughput results in the most loaded conditions: requested by more than 100 clients at the same time and processing 1 000 000 characters messages. However, latency of RPC was high, more than 1 second.
- If horizontal scalability is important aspect, Kafka and RabbitMQ are the best candidates as they have built in cluster functionality. It must be noted that others technologies can be scaled horizontally as well, but it requires additional tools and effort.
- HTTP Rest has the smallest request and response message size. If the message size is important criteria for choosing communication technology, then HTTP Rest is a recommended technology. On the other hand, gRPC has the smallest payload as it uses binary serialization. Theoretically, at some point of complexity, for complex data models with many properties, gRPC request and response message size should become smaller than HTTP Rest. A deeper research is needed to determine exact complexity threshold.
- gRPC library is using the least amount of storage. If microservices are running in an environment with limited storage, then gRPC must be used. The maximum amount of storage is allocated for GraphQL libraries. It must be pointed out that storage size weakly depends on technology. It mostly depends on how it was implemented in the particular library. If the library size is too big, then microservice developers can implement it by themselves, but there is no guarantee that the new library will be smaller.
- RabbitMQ and Kafka consume the smallest amount of memory. Therefore, if memory size is one of the essential criteria, then RabbitMQ and Kafka must

be used for implementation. On the other hand, HTTP Rest consumes the largest amount of memory. Memory size and storage usage depend on library implementation so that a similar recommendation can be provided as in the previous list item.

- Microservice implemented using Kafka library boots up in the fastest way, while using GraphQL library boots up in the slowest way. If the boot time or restart time of microservice is essential, then Kafka must be used for microservice communication.
- HTTP Rest and RabbitMQ are prevalent communication technologies, and many different libraries exist in the market to choose from, while GraphQL and gRPC are relatively new and rapidly growing communication technologies with fewer libraries to choose from.
- Synchronous communication style communication technologies gRPC, HTTP Rest, and GraphQL do not require any additional components to communicate, while asynchronous communication technologies RabbitMQ and Kafka requires service as interim communication layer. Hence, additional components increase solution complexity and maintenance cost. On the other hand, if a solution contains many microservices and scalability is a challenge, RabbitMQ and Kafka as an interim layer can provide centralized communication routing functionality.

Known limitations and threats to validity of the research are provided below:

- Experiment was conducted using programming language: C Sharp. Measured results can be different using other programming languages and libraries.
- Experiment was conducted using computer with Windows OS. Measured results can be different using different environment such as Linux, Docker, OpenShift, public cloud, and etc., due to their specifics and implementation details of the libraries.

7 CONCLUSIONS

The aim of this paper was to analyze and evaluate different communication technologies for communication between microservices. A set of new criteria including influence on microservice topology, the performance of remote procedure call, message size, memory consumption, storage usage, boot time, and availability of the corresponding libraries was introduced to evaluate a variety of aspects that can potentially be a challenge during legacy monolith application decomposition to microservices. Five communication technologies, such as HTTP Rest, RabbitMQ, Kafka, gRPC, and GraphQL, have been evaluated and compared by proposed evaluation criteria.

The key advantages and disadvantages of each communication technology have been identified and provided in Section 6.

The results found in this research will be used for algorithm development of automatic monolith decomposition to the microservices.

REFERENCES

[1] Fowler, M.—Lewis, J.: Microservices. 2014, http://martinfowler.com/articles/microservices.html.

[2] Newman, S.: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, 2019.

[3] Columbus, L.: IDC Top 10 Predictions for Worldwide IT, 2019. 2018, https://www.forbes.com/sites/louiscolumbus/2018/11/04/idc-top-10-predictions-for-worldwide-it-2019/.

[4] International Data Corporation (IDC). 2023, https://www.idc.com/.

[5] Pozdniakova, O.—Mazeika, D.: Systematic Literature Review of the Cloud-Ready Software Architecture. Baltic Journal of Modern Computing, Vol. 5, 2017, No. 1, pp. 124–135, doi: 10.22364/bjmc.2017.5.1.08.

[6] Pozdniakova, O.—Mazeika, D.: A Cloud Software Isolation and Cross-Platform Portability Methods. 2017 Open Conference of Electrical, Electronic and Information Sciences (eStream), IEEE, 2017, pp. 1–6, doi: 10.1109/eStream.2017.7950315.

[7] Kazanavičius, J.—Mažeika, D.: Migrating Legacy Software to Microservices Architecture. 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream), IEEE, 2019, pp. 1–5, doi: 10.1109/eStream.2019.8732170.

[8] Kazanavičius, J.—Mažeika, D.: Analysis of Legacy Monolithic Software Decomposition into Microservices. In: Matulevičius, R., Robal, T., Haav, H. M., Maigre, R., Petlenkov, E. (Eds.): Conference Forum and Doctoral Consortium of Baltic DB & IS 2020 (Baltic-DB & IS-Forum-DC 2020). CEUR Workshop Proceedings, Vol. 2620, 2020, pp. 25–32.

[9] Kazanavičius, J.—Mažeika, D.—Kalibatienė, D.: An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database. Applied Sciences, Vol. 12, 2022, No. 12, Art. No. 6189, doi: 10.3390/app12126189.

[10] Microsoft: Communication in a Microservice Architecture. 2020, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[11] Bandhamneni, N.: Inter-Service Communication in Microservices. 2018, https://walkingtree.tech/inter-service-communication-in-microservices.

[12] Galbraith, K.: 3 Methods for Microservice Communication. 2019, https://blog.logrocket.com/methods-for-microservice-communication/.

[13] Biswas, R.—Lu, X.—Panda, D. K.: Designing a Micro-Benchmark Suite to Evaluate gRPC for TensorFlow: Early Experiences. 2018, doi: 10.48550/arXiv.1804.01138.

[14] gRPC: gRPC – A High-Performance, Open-Source Universal RPC Framework. 2023, https://www.grpc.io.

[15] Hartig, O.—Pérez, J.: An Initial Analysis of Facebook's GraphQL Language. In: Reutter, J., Srivastava, D. (Eds.): Alberto Mendelzon Workshop on Foundations of Data Management and the Web (AMW 2017). CEUR Workshop Proceedings, Vol. 1912, 2017.

[16] Brito, G.—Valente, M. T.: REST vs GraphQL: A Controlled Experiment. 2020 IEEE International Conference on Software Architecture (ICSA), 2020, pp. 81–91, doi: 10.1109/ICSA47634.2020.00016.

[17] GraphQL: GraphQL – A Query Language for Your API. 2023, https://www.graphql.org.

[18] Taibi, D.—Lenarduzzi, V.—Pahl, C.: Architectural Patterns for Microservices: A Systematic Mapping Study. Proceedings of the 8th International Conference on Cloud Computing and Services Science – Closer, SciTePress, 2018, pp. 221–232, doi: 10.5220/0006798302210232.

[19] Montesi, F.—Weber, J.: Circuit Breakers, Discovery, and API Gateways in Microservices. 2016, doi: 10.48550/arXiv.1609.05830.

[20] Smid, A.—Wang, R.—Cerny, T.: Case Study on Data Communication in Microservice Architecture. Proceedings of the Conference on Research in Adaptive and Convergent Systems (RACS '19), 2019, pp. 261–267, doi: 10.1145/3338840.3355659.

[21] Cerny, T.—Donahoo, M. J.—Trnka, M.: Contextual Understanding of Microservice Architecture: Current and Future Directions. ACM SIGAPP Applied Computing Review, Vol. 17, 2018, No. 4, pp. 29–45, doi: 10.1145/3183628.3183631.

[22] Yarygina, T.—Bagge, A. H.: Overcoming Security Challenges in Microservice Architectures. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018, pp. 11–20, doi: 10.1109/SOSE.2018.00011.

[23] Walsh, K.—Manferdelli, J.: Mechanisms for Mutual Attested Microservice Communication. Companion Proceedings of the 10th International Conference on Utility and Cloud Computing (UCC '17 Companion), ACM, 2017, pp. 59–64, doi: 10.1145/3147234.3148102.

[24] Hong, X. J.—Yang, H. S.—Kim, Y. H.: Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. 2018 International Conference on Information and Communication Technology Convergence (ICTC), IEEE, 2018, pp. 257–259, doi: 10.1109/ICTC.2018.8539409.

[25] Fernandes, J. L.—Lopes, I. C.—Rodrigues, J. J. P. C.—Ullah, S.: Performance Evaluation of RESTful Web Services and AMQP Protocol. 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2013, pp. 810–815, doi: 10.1109/ICUFN.2013.6614932.

[26] Microsoft: C# Documentation. 2023, https://docs.microsoft.com/en-us/dotnet/csharp.

[27] Microsoft: Visual Studio. 2023, https://visualstudio.microsoft.com.

[28] Microsoft: NuGet. 2023, https://www.nuget.org.

[29] BenchmarkDotNet. 2023, https://benchmarkdotnet.org/articles/overview.html.

[30] NBomber. 2023, https://nbomber.com/docs/getting-started/overview/.

[31] Wireshark. 2023, https://www.wireshark.org/.

Justas Kazanavičius is currently Ph.D. candidate at the Vilnius Gediminas Technical University, Lithuania. He received his Master's degree from the Vilnius Gediminas Technical University, Lithuania, in 2011. His current research interest is migration from monolith architecture to microservices architecture. He has more than 10 years experience as a software developer in different business domains.

Dalius Mažeika received his Ph.D. degree from the Vilnius Gediminas Technical University in 2000. He is Professor at the Information Systems Department, Vilnius Gediminas Technical University, Lithuania. He has published over 100 research papers in international journals and conference proceedings. He led two projects funded by the Research Council of Lithuania and holds 6 Lithuanian patents. His research interests include cloud computing, microservice architecture, adaptive cloud resource provisioning, predictive auto-scaling methods, legacy software migration to clouds.