FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 48 (2023) No. 1
ISSN 0867-6356
DOI: 10.2478/fcds-2023-0003 e-ISSN 2300-3405

Comparative Review of Selected Internet Communication Protocols

Łukasz Kamiński, Maciej Kozłowski, Daniel Sporysz, Katarzyna Wolska, Patryk Zaniewski, Radosław Roszczyk *

Abstract. With a large variety of communication methods and protocols, many software architects face the problem of choosing the best way for services to share information. For communication technology to be functional and practical, it should enable developers to define a complete set of CRUD methods for the processed data. The research team compared this paper's most commonly used data transfer protocols and concepts: REST, WebSocket, gRPC GraphQL and SOAP. A set of web servers was implemented in Python, each using one of the examined technologies. Then, the team performed an automated benchmark measuring time and data transfer overhead for a set of defined operations: creating an entity, retrieving a list of 100 entities and fetching details of one entity. Tests were designed to avoid the results being interfered with by database connection or docker-compose environment characteristics. The research team has concluded that gRPC was the most efficient and reliable data transfer method. On the other hand, GraphQL turned out to be the slowest communication method of all. Moreover, its server and client libraries caused the most problems with proper usage in a web server. SOAP did not participate in benchmarking due to limited compatibility with Python and a lack of popularity in modern web solutions.

Keywords: REST, gRPC, websockets, GraphQL, Internet communication protocol

1. Introduction

In the modern world of technology, there are many tools that developers can choose to exchange data between web clients, such as web pages and a server. These tools are wary of speed and the amount of sent data. Therefore, deciding which tool to use in a project takes work. Our team recognized that problem and decided to check which

*Faculty of Electrical Engineering, Warsaw University of Technology, Warsaw, Poland

standard web protocol is the fastest and the most memory efficient. We decided to test gRPC, REST architectural style, the WebSockets Protocol and GraphQL. We skipped SOAP protocol because of its lack of popularity in modern web applications.

This paper is organized as follows. The next subsections briefly describe tested protocols. Section two presents related articles. Section three describes testing methods and testing environment. Section four presents test results. Finally, section five discusses them ad provides concluding remarks.

1.1. REST

REST (*Representational State Transfer*) is an architectural style for distributed systems that Roy T. Fielding described in 2000. It is defined by several constraints that will be related shortly.

The idea of this approach is based on widely known client-server architecture. REST allows the client and server to work independently by separating the user interface on the client side and business logic from data storage on the server side. Thanks to that, REST achieves the portability of the user interface across multiple different platforms.

Another constraint of this architecture is that the communication between the system parts must be stateless. It means that each request from the client to the server must contain all the information necessary to understand and complete the request. Because of that, only the client side is responsible for keeping the session state.

The following principle is the cacheability of responses. It states that every response must inform the client if it is cacheable or noncacheable because if it is, the client can reuse that data for later, similar requests.

After that, there is a uniform interface constraint, which suggests that by "applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved". This rule also describes more requirements for REST interface to work correctly, such as hyperlinks or unique identification of each resource.

REST architecture also specifies that it requires the system to be layered. The layered system style allows an architecture composed of hierarchical layers by constraining component behaviour.

The last constraint is Code-On-Demand. It means that REST can also allow the clients to download and execute code in the form of applets or scripts.

In REST architecture, the sample of information is referred to as a resource. Each resource has its identifier, which is the URL pointing to the specified information. To manage this data, REST applications provide RESTful APIs, where resources are exposed through endpoints. The endpoints, in general, are the URLs that identify resources.

REST is strictly related to HTTP protocol. It is caused by the fact that to interact with resources in a REST system, we use four basic HTTP methods. These are:

- `GET` – used to retrieve list of resources or aspecific resource by its ID,

- `POST` – used to create a new resource,
- `PUT` – used to update a resource by its ID,
- `DELETE` – used to delete a resource by its ID.

Some data serialization formats are used in REST communication. The most popular are JSON and XML. Using the specified system format, we send resources to create or update, and also the server provides us with information on the resources we requested in that format.

Summing up, the use of REST architecture can result in better performance and scalability in simple, lightweight applications [9, 10].

1.2. The WebSocket Protocol

The WebSocket Protocol is a protocol which enables two-way communication. This means that the server does not have to wait for the client to make a request and can send data independently. This is achieved by using a single TCP connection which is established between client and server [8].

The protocol consists of two parts. The first one is a handshake. In order to make it client sends a HTTP request. This request must have several headers set (`Connection`, `Upgrade`). Example client request:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Server replies with its own handshake response which has HTTP code set to 101. Example response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

When the handshake is successfully finished further communication is done via a single TCP connection using The Websocket Protocol. The client and server can send data independently in chunks named as messages. Messages are made of frames which are transferred across sides.

In order to close the connection, a unique frame is sent by one of the peers. This frame indicates the will of closure. When another peer receives it, it responds with a Close frame. Thanks to that, the first peer can safely close the connection, knowing that no more data will be transferred.

1.3. gRPC

gRPC (*Google Remote Procedure Call*) is an open source system that implements remote procedure call concept. It was developed in 2015 by Google as a successor of previous RPC system called Stubby.

The power of gRPC comes from many things but there are two which are major. It is HTTP/2 communication protocol and Protocol Buffers.

HTTP/2 as the newest version of popular HTTP protocol provides a lot of useful features to this type of communication. Thus, gRPC provides the mechanisms of deadlines, timeouts, terminations and cancelling calls. Moreover, thanks to this protocol, there are 4 types of communication that can be implemented [1]:

- Unary Call – the simplest type of communication where there is only one for one request,
- Client Stream – in some way it is very similar to Unary Call, but here a client sends a stream of messages instead of a single one,
- Server Stream – the opposite to Client Stream, in which client sends only one request and expects a stream of messages,
- Bidirectional Stream – combination of two previous methods, in which both sides of a stream send messages at the same time.

Protocol Buffers is a cross-platform mechanism developed by Google which allows to define data structures and interfaces in `*.proto` files. Then, developed file can be used to generate a file in targeted language (i.e., Java, Python, Go). The benefit of this is that one Protobuf file can generate an API for different language that is implementing server side, and other which is implementing client side. What is more, Protocol Buffer is a binary format what means that is faster and smaller than other formats (i.e., JSON) [2].

1.4. SOAP

Simple Object Access Protocol (SOAP) defines a mechanism for exchanging structured and typed information in a decentralized, distributed environment. It possible to express application semantics with packaging model and encoding rules for data within modules. It is important to note that SOAP does not define any application semantics itself but a mechanism for expressing it, so it has many use cases ranging from messaging systems to RPC.

Each SOAP message consists of an envelope, encoding rules and RPC representation. SOAP envelope represents a message framework describing what a message is, who its receiver is and which data is mandatory or optional. SOAP encoding rules explain a serialization and de-serialization mechanism for application data types. SOAP RPC representation constructs remote procedure calls and responses.

SOAP data model was designed with language-independent abstraction for data types. Two data types are supported: a simple XSD and a compound one. For example, simple XSD types include `int`, `boolean` and `string`. Compound types are used to exchange structures and arrays. Simple Object Access Protocol is XML-based and each message is serialized into XML file.

1.5. GraphQL

GraphQL is an open-source solution for querying and mutating data via remote API. The solution was first developed internally by Facebook, but it became publicly released in 2015. Now it is developed by Linux Foundation.

The core concept of GraphQL relies on an extensive query language that enables API consumers to not only apply filters but also choose fields of the returned structure. This approach is more flexible than REST APIs exposing a set of endpoints, each returning a consistent set of data. The responsibility for defining the presence of individual fields in a returned structure is moved from an API server to an API consumer, which results in an optimization of data traffic and an ability to eliminate API versioning. GraphQL user HTTP protocol for network communication.

GraphQL allows the API developers to declare an interface for querying data on SQL and NoSQL database solutions. Since GraphQL allows for nested structures, a NoSQL database (e.g., MongoDB) fits the technology well. GraphQL also improves development flexibility by providing client-side and server-side API libraries for many popular programming languages (e.g., Python, as used in this work).

2. Related work

We have searched for similar works, that intended to compare REST, gRPC, Websockets and GraphQL protocols, but we have found none that would compare all of them in terms of performance. However, there are studies that set some of those protocols side by side, for example, contrasting the time it takes to implement a query or analysing an energy costs of specified protocols.

Many of the found papers focus on testing REST and GraphQL protocols. In the first work, authors lead a performance comparative study with the stated two technologies. They analysed three applications and each of them was developed using two protocols – REST and GraphQL. The researched was based on measuring the response time and the average transfer rate between the requests. The study describes, that in two applications migrating from REST to GraphQL resulted in higher performance in the aspects of average number of requests per second and transfer rate of data. However, above workloads of 3000 requests, REST protocol performed better than GraphQL [16].

Next study compares REST and GraphQL in terms of convenience of use. It answers the question "*How much time do developers spend when implementing queries in REST and GraphQL?*". Results of the experiment presented in that paper show that implementing remote service queries was more quick, when using GraphQL technology than REST technology. Implementing GraphQL queries took less effort compared to REST, among participants with no or little knowledge of those protocols, as well as among experienced developers [4].

Another paper examines the performance of REST and GraphQL protocols. A series of experiments was also executed that showed that the choice of better protocol for given purpose depends highly on type of query and requested data. With the same amount of used data, REST protocol is obvious better choice as it stands out with better response times, CPU and memory consumption. However, in test cases where REST was forced to under-fetch data, GraphQL performed better in terms of response time and memory usage but still needed more of the CPU resources. Overall the result showed that GraphQL should be considered when under- and over-fetching or request smaller subsets of data is likely to happen and REST protocol is the best option in other, more basic cases [11].

The following work contrasts REST, SOAP, Socket and gRPC in computation offloading of mobile applications and it analysis energy costs of those protocols. The experiment carried out in that study was about evaluating the energy consumption of stated protocols using algorithms of different complexities and different input sizes and types. Results show that, when executing more simple algorithms with small input data, local execution is way more economic. Regarding remote execution, the best option is REST architecture, that is followed by Socket. The paper also states, that computation offloading can be responsible of saving up to 10 times as much energy when compared to local execution [5].

After that, there is also a paper that analyzes the efficiency of REST and gRPC protocols in microservice-based ecosystems. In order to perform tests, the authors created implementations of REST and gRPC services which were developed using .NET 5 platform. The main parameter that was tested, was the response time of the performed operations. The explored communication tasks was based on: text cloning, fetching maximum value of an integer, fetching an array of consecutive integers, fetching a text file and downloading a PDF file. Each test was performed with the use of both encrypted and not encrypted data. For most of the tasks that was tested with not encrypted data, the results was better for the REST protocol. Only in the large file transfer gRPC performed better. On the other hand, with the usage of encrypted data, both protocols got similar results. REST performed better during transmission of numerical data and gRPC was faster for file transfer operations [3].

Next work describes performance comparison between GraphQL, REST and SOAP protocols. The method used to evaluate the differences between those three protocols, was based on data fetching operation. For the test purpose, authors created systems for each protocol using .NET technologies. There were several test cases that included fetching elements from database, as well as using simple and more complex joins. These operations was tested for a single row and also for 100 rows. For the analysis, the authors also used two types of databases: MySQL and MongoDB. The experiment was performed using two computers – server and client – that were connected through a local network. The results showed that GraphQL was characterized by the worst performance in all test cases. Another conclusion is the fact that the packet size is the largest with the usage of SOAP protocol. It is caused by the XML format used in SOAP message passing. The performance results of the REST protocol were the best among tested technologies [7].

Another study compares the performance of REST API, GraphQL and gRPC. For the research, the authors developed three applications that contained the same functionalities, but with the use of different protocols. The systems was created using .NET 5 platform. The experiments measured the execution time, performance and volume of the data, that was processed during display and adding operations. The exact testing methods relied on fetching small, medium and large amount of data, as well as inserting new data. The result showed, that the best protocol in terms of performance is REST. However, considering the smallest data package size, the better option is gRPC protocol. Overall, the choice of selecting one specific protocol for a given task is very complicated. During selecting a technology, several factors should be considered such as: data size, system performance and number of users [17].

The following work contrasts WebSocket and HTTP protocol performance. For the testing purposes the authors used machines working in the same LAN network and developed a special application. The methods used for the analysis depended on sending and receiving texts of the length of 100 characters. The main conclusion from the research is the fact that with the transmission of over 100 data copies using the WebSocket protocol can result in over 100 times better performance over HTTP protocol. It was also proved that the usage of WebSockets is a good way to transfer a big number of small data packages in the period of one second, because in other scenarios basic HTTP requests are better option. The authors also noticed that the TLS encryption has no effect on the performance of both protocols [12].

After that, there is also a paper that compares the performance of web services using Symfony, Spring, and Rails technologies. Using each of the frameworks, REST and SOAP application were developed. In research the authors focused on measuring the request execution time. The tests included select, insert and update operations. Results showed that performance of the REST and SOAP protocols highly depends on technology in which the application was developed [14].

Last study presents the performance and usage comparison of REST and SOAP web services. Results show that REST outperforms SOAP in terms of bandwidth usage and message processing performance. Authors also stated that REST is a good option in basic, most common cases, while SOAP should be considered if particular functionality, such as security options, is required. REST is also a more simple, easier to develop technology than SOAP protocol [15].

3. Measurement methods

In this paper, we discuss the differences in features and limitations of communication protocols for web services to help engineers and architects choose a protocol that suits their needs the most. Although the knowledge of their functionalities is often sufficient to select one, some solutions are required to handle large requests. We need insight into protocol performance to make an informed decision. We want to perform benchmarks to find the fastest protocols and protocols with the most miniature network data footprint.

Nowadays, web services are usually deployed in a containerized environment. We are interested in knowing how much performance is allocated for virtualization thus we will perform benchmarks comparing the performance of applications deployed in a bare metal environment and a virtualized environment.

3.1. Performance comparison

For each protocol, we implemented a web service providing an interface to CRUD operations on a database using libraries listed in the Table 1. The main factors for choosing these libraries are the popularity and deployment web frameworks. Ith popular projects, we are less likely to encounter unusual performance issues and bugs. Also, notice how Flask is used across three out of four implementations. Using the same web framework should provide results that are better comparable.

To evaluate the performance of protocols, we measured total time that elapsed between making a request and fully receiving a response. We implemented clients in Python for each service using libraries from Table 2 to achieve this. The clients, the servers and a database were hosted on the same machine. Although this architecture could result in them competing for resources, we believe the benefits of reduced network delay outweigh a solution in which we could deploy these projects on separate host machines.

We used MongoDB 5.0.5 database and Steam Games DataSet [6]. NoSQL database was chosen because we want to minimize the impact of database operations on our measurements, and NoSQL databases are generally faster than SQL databases [13].

Three scenarios were considered in test cases: (1) inserting an entry, (2) fetching an entry, and (3) fetching several entries. First we measured how performant protocols are with inserting a single game entry into a database. The test was performed on an empty database with indexing turned off. Before proceeding to another protocol, the database was restored to an initial state to avoid performance degradation over time. The same game entry data was used for the testing of every service. Then indexing was turned on and the database was populated with 100 entries from Steam Game DataSet. We used just 100 entries to keep a table index short and make queries to the database as quickly as possible. The other two test cases aim to measure how protocols behave with small and large outbound data transfers. The second and the third test cases involve fetching a single game entry and 100 game entries simultaneously. Same as for the first test, we ensured that the same data was used to test each protocol.

Before any benchmark took place, we generated a load on each service to address the cold start issue.

Table 1. API libraries and versions

Project	Used libraries and versions
REST	Flask 2.0.3, requests 2.27.1, pymongo 4.0.2
GraphQL	Flask 2.0.3, graphql-core 2.3.2, graphql-relay 2.0.1, graphql-server-core 2.0.0, graphene 2.1.9, graphene-mongo 0.2.13
WebSockets	Flask 2.0.3, simple-websocket 0.5.1, pymongo 4.0.2
gRPC	pymongo 4.0.2, grpcio 1.45.0, grpcio-tools 1.45.0

Table 2. Client libraries and versions

Project	Used client libraries and versions
REST client	requests 2.27.1
GraphQL client	python-graphql-client 0.4.3
WebSockets client	simple-websocket 0.5.1
gRPC client	grpc 1.45.0

3.2. Network load comparison

To evaluate the impact on network, we measured how much data is sent and received to operate on a web service. We captured and recorded every network packet sent and received during a single request. Then the data consumption was evaluated by assessing the length of every packet and summing them up. Two operations were tested against data consummations: add one entry and get one entry. All tests were performed on the same entry data.

3.3. Test environment

All benchmarks were performed on a platform with Windows 10 OS with hardware configuration specified in a Table 3. To minimize the impact of a real-time system on the results of benchmarks, we ensured no background tasks were running on a test machine. We used Docker Desktop 4.6.1 as a virtualization platform to test the performance of a vitalised environment. The same Python interpreter in version 3.9.12 was used across all implementations to make results better comparable.

Table 3. Hardware configuration

Component	Model
CPU	AMD Ryzen 5600X
OS	Windows 10 Pro
RAM	DDR4 16GB 3200Mhz

4. Results

This section presents the results of conducted tests. We divided them into several subsections. First subsection covers the performance comparison. It shows the tests of inserting one element into the database, getting one element and a hundred elements. We ran tests on the native OS and Docker. The second subsection presents how much data needs to be transferred between the client and server to achieve those operations.

4.1. Performance comparison

4.1.1. Inserting one element to database

As presented in Figure 1, the times needed to insert one element into the database is shorter. We decided to remove outliers from the chart in order to maintain readability. We achieved the lowest mean times on Docker when using REST style. Surprisingly, gRPC turned out to be slower on this platform.

The comparison between running tests via Docker and direct on Windows is worth mentioning. Figure 2 shows shorter times for three protocols. The inserting operation is faster when run on a native OS than in Docker. This time gRPC protocol was the fastest of all. Only GraphQL achieved worse performance on native OS than on Docker. What is more, Docker turned out to be less stable than Windows. We noticed several tests which took abnormally long. It is evident in the maximum value for WebSockets tests. The value is more than ten times bigger than on a similar test on Windows.

Table 4 presents collected values for Docker and Windows tests.

4.1.2. Fetching one element

Our second test was to measure times needed to fetch one element from database. Figure 3 shows result chart of tests done on Docker platform, while Figure 4 presents results from Windows. Once more times measured during tests conducted on native OS were shorter.

GraphQL protocol was the slowest both on Docker and Windows. REST and

Table 4. Times measured for inserting one value to database

		Docker [μs]	Windows [μs]
REST	mean	2425	2278
	min	1563	1818
	max	3479	7779
	σ	280	395
gRPC	mean	2695	1512
	min	1565	999
	max	6560	4822
	σ	667	253
GraphQL	mean	4161	4834
	min	3605	4128
	max	8337	7151
	σ	318	429
WebSockets	mean	2950	2424
	min	1999	1248
	max	90272	6632
	σ	2789	775

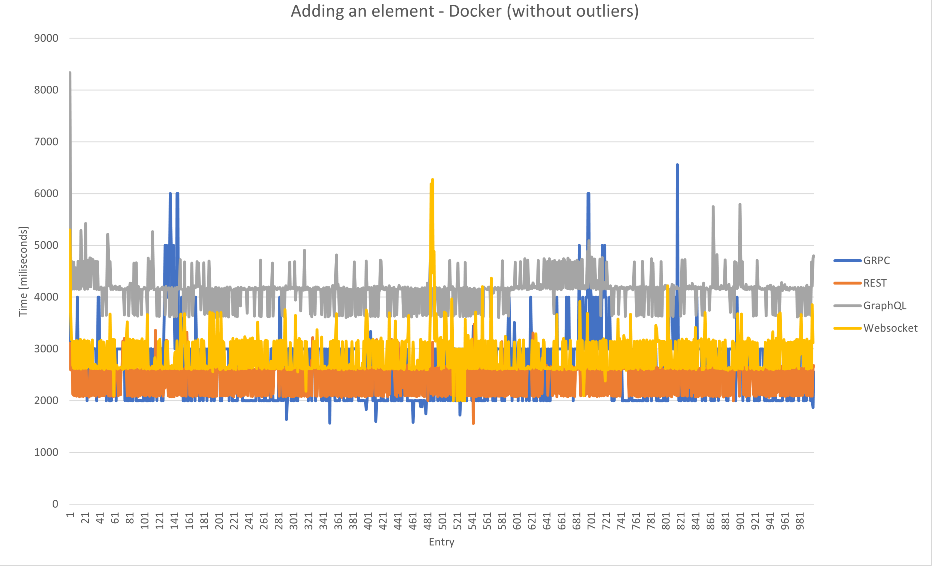

Figure 1. Inserting one element to database on Docker

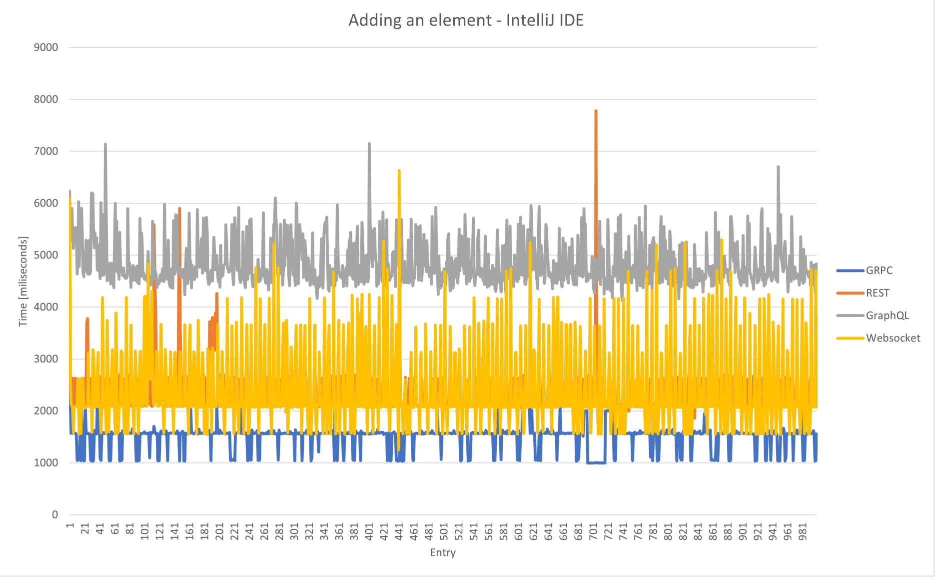

Figure 2. Inserting one element to database on Windows

gRPC achieved very similar results, but gRPC turned out to be the fastest of all. Its speed is especially visible when testing on native OS. WebSockets were the least stable again. Their maximum test value was several times bigger than other protocols. Table 5 shows test times.

Table 5. Times measured for fetching one value from database

		Docker [μs]	Windows [μs]
REST	mean	2631	2289
	min	2066	1837
	max	7683	5134
	σ	428	280
gRPC	mean	2495	1456
	min	1537	1029
	max	7001	5101
	σ	616	258
GraphQL	mean	6022	6271
	min	4209	4989
	max	9656	10000
	σ	588	578
WebSockets	mean	3069	2439
	min	1999	1031
	max	30320	9491
	σ	1071	843

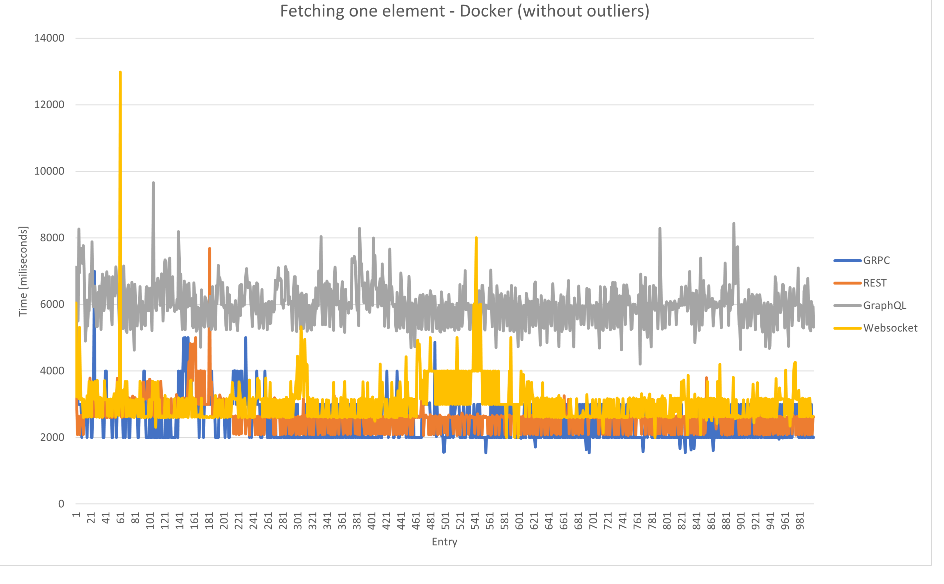

Figure 3. Fetching one element on Docker

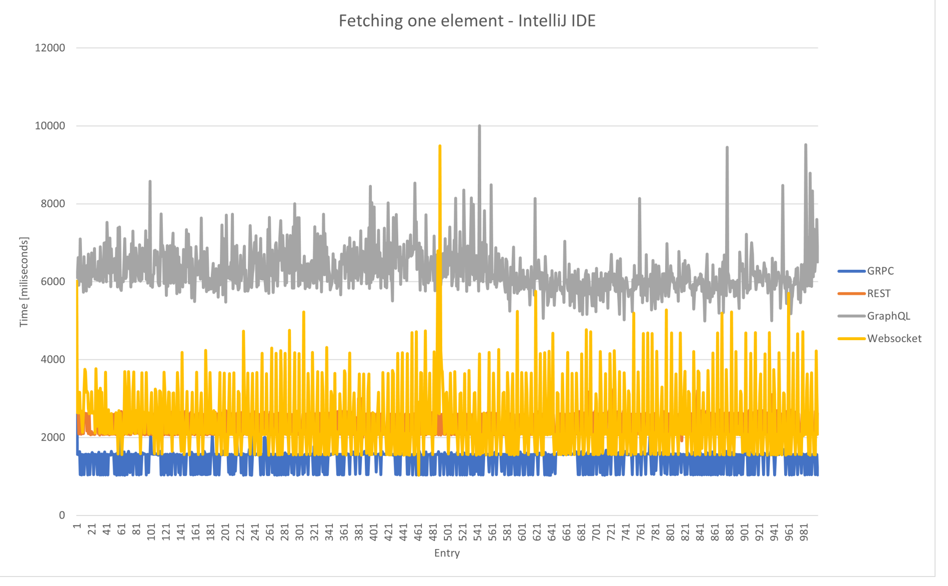

Figure 4. Fetching one element on native OS

4.1.3. Fetching a hundred elements

The last conducted test measured how long it takes to fetch a hundred elements from database. Figure 5 present results from Docker platform, and Figure 6 shows results from Windows.

This time the WebSockets protocol turned out to be the slowest and the least stable. There are several tests which took much longer than they normally last. gRPC protocol was definitely the fastest in this task.

Table 6 shows mean test times.

Table 6. Times measured for fetching a hundred elements from database

		Docker [μs]	Windows [μs]
REST	mean	24301	19728
	min	22420	17758
	max	90189	38277
	σ	3030	1600
gRPC	mean	11730	11611
	min	8002	9893
	max	38763	20067
	σ	5403	1189
GraphQL	mean	23107	22176
	min	20820	19266
	max	48715	44608
	σ	2567	2565
WebSockets	mean	25327	23533
	min	22962	20888
	max	189695	51930
	σ	9444	2244

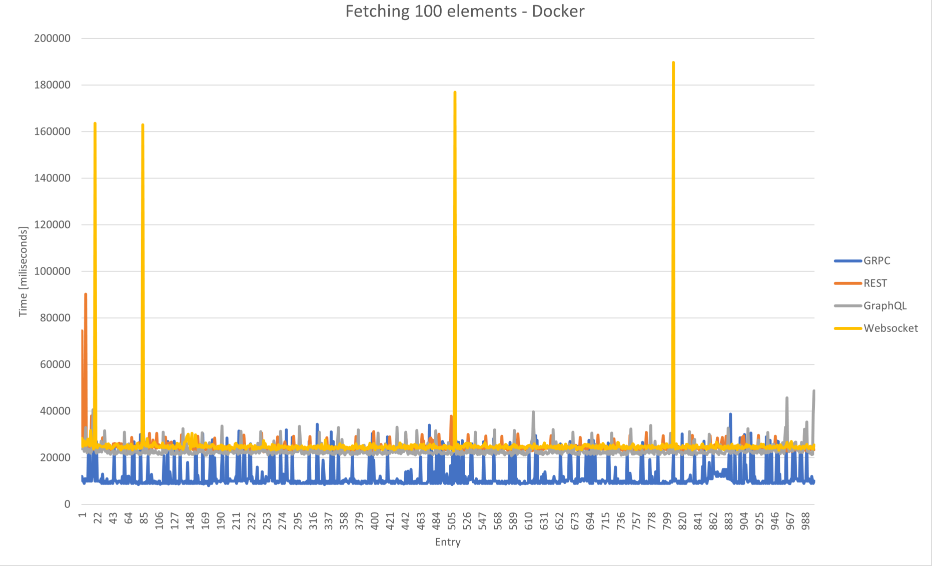

Figure 5. Fetching a hundred elements on Docker

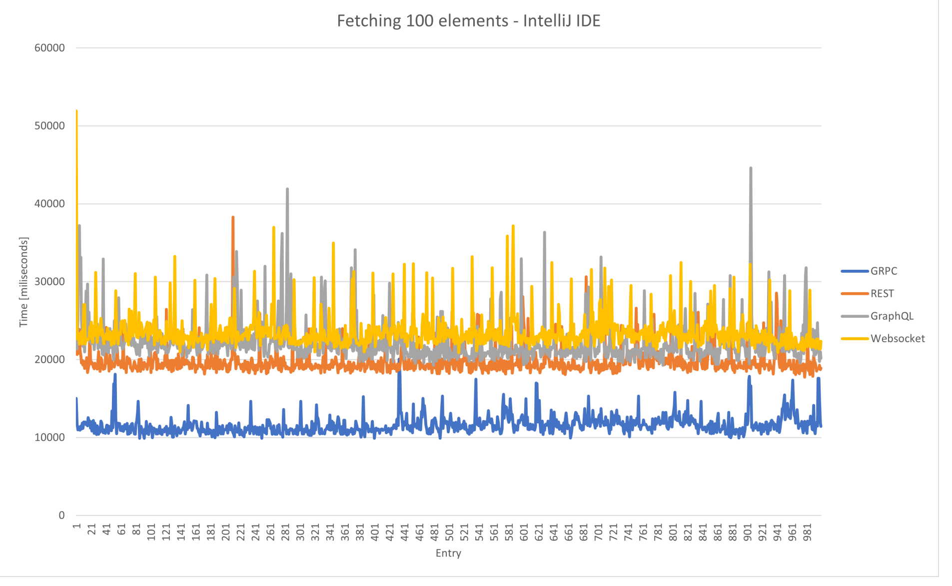

Figure 6. Fetching a hundred elements on native OS

4.2. Memory comparison

4.2.1. Insert one element to database

Table 7 presents how many bytes have been transferred in each protocol and operation. The measurements have been collected with WireShark.

Table 7. Message size in bytes

Operation	REST	gRPC	GraphQL	WebSockets
Fetching element	3536	3811	3766	3469
Inserting element	3298	3769	3746	3069

Results show that, indisputably, WebSockets offers the best memory usage out of 4 presented communication protocols. In both fetching and inserting it achieved the best result. Other protocols use noticeably more resources, especially gRPC which is the most memory-consuming mechanism of communication in this comparison.

5. Conclusion

Out tests showed that gRPC protocol is the fastest in transferring data between client and server. The WebSockets protocol achieved similar results to REST when transferred data was small (inserting and fetching one element). When data was larger (test with a hundred elements) it turned out to be the slowest. REST style was moderately fast. It turned to be slower than gRPC, but was never the worst in any category. GraphQL had some troubles with small data. It was the worst in inserting and fetching one element from database. It was slightly faster than the WebSockets in fetching a hundred elements.

What can easily be visible is that Docker platform is less stable than native OS. There are several tests (in particular, the WebSockets protocol tests) which show outlier times. This means that Docker might have an impact on a web app performance.

What was also noticed is that the memory usage is in some way associated with protocols performance. WebSockets scored poorly in performance comparison but used the littlest amount of memory to transfer data. The opposite is with gRPC, it has noticed the best performance but was the most memory-consuming at the same time.

Above conclusions and the overall authors feelings lead to another conclusions about proper use cases of each protocol. If a programmer is looking for the fastest way to transfer data and does not care about message size the relevant option will be gRPC. On the other hand, if delivery time is not crucial and there is need of low memory usage the right option will be WebSocket. REST protocol is also an interesting way to communicate. It provides decent memory usage, time performance and what is the most important it is easily accessible, which means that every new client can easily connect to server. There is also GraphQL which provides the worst memory usage to time performance ratio especially in single element operations. This means that this protocol should be used in other way, e.g., nested data structures where data is fetched with queries.

References

[1] gRPC official documentation. https://grpc.io/docs/what-is-grpc/core-concepts/. Accessed on May 17, 2022.

[2] Protocol buffers official documentation. https://developers.google.com/protocol-buffers/docs/overview. Accessed on May 17, 2022.

[3] Bolanowski M., Żak K., Paszkiewicz A., Ganzha M., Paprzycki M., Sowiński P., Lacalle I., and Palau C. E. Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems. 2022.

[4] Brito G. and Valente M. T. REST vs GraphQL: A controlled experiment. In *2020 IEEE international conference on software architecture (ICSA)*, pages 81–91. IEEE, 2020.

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

[6] Deepan N. and Narayanan S. 80000 steam games dataset. https://kaggle.com/datasets/deepann/80000-steam-games-dataset/metadata. Accessed in April, 2022.

[7] Erlandsson P. and Remes J. Performance comparison: Between GraphQL, REST & SOAP (bachelor's thesis), 2020.

[8] Fette I. and Melnikov A. The WebSocket Protocol. RFC 6455, IETF Trust, December 2011.

[9] Fielding R. and Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and content. Technical report, 2014.

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

[11] Frigård E. GraphQL vs. REST: A comparison of runtime performance (bachelor's thesis), 2022.

[12] Łasocha W. P. and Badurowicz M. Porównanie wydajności protokołu WebSocket i HTTP. *Journal of Computer Sciences Institute*, 19:67–74, 2021.

[13] Li Y. and Manoharan S. A performance comparison of SQL and NoSQL databases. pages 15–19, 08 2013.

[14] Lubartowicz P. and Pańczyk B. Performance comparison of web services using Symfony, Spring, and Rails examples. *Journal of Computer Sciences Institute*, 17:384–389, 2020.

[15] Makkonen J. et al. Performance and usage comparison between REST and SOAP web service. 2017.

[16] Seabra M., Nazário M. F., and Pinto G. REST or GraphQL? a performance comparative study. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 123–132, 2019.

[17] Śliwa M. and Pańczyk B. Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute*, 21:356–361, 2021.

Received 14.09.2022, Accepted 02.12.2022