Performance Evaluation of REST and GraphQL APIs Searching Nested Objects

Marlon Diego Casagranda França
ASAAS Gestão Financeira SA
Joinville, SC, Brasil
marlondiego407@gmail.com

Eduardo da Silva
Instituto Federal Catarinense
Araquari, SC, Brasil
eduardo.silva@ifc.edu.br

ABSTRACT

With the growth of mobile devices claiming an increasing share of internet traffic, optimizing data search performance becomes important. REST architecture has been the most common solution to develop Web APIs, but GraphQL is becoming, recently, an attractive alternative. This paper discusses the REST and GraphQL techniques for data communication between Web applications. An experiment is also performed, to evaluate the performance of APIs implementing REST and GraphQL when requesting nested objects. Prototypes of each API were implemented to perform measurements of each technique's performance. Results show that GraphQL performed better in the most evaluated scenarios.

KEYWORDS

Information, GraphQL, Rest, Queries, Neasted

1 INTRODUCTION

In an increasingly interconnected world, users require increased availability, with low access latency, of information accessed on the Web. With the expansion of the Internet to heterogeneous devices, such as smartphones, tablets and IoT devices, issues concerning load time and bandwidth consumption are becoming more relevant. A recent study shows that the Internet access from mobile devices grew by 63% in 2016. Also, mobile data traffic grew from 4.4 exabytes per month in 2015 to 7.2 exabytes per month in 2016. This increase is the result of joining about 429 million new mobile devices to the network, being the smartphones responsible for most of this growth [3].

In addition, it is known that users consider more important the speed they receive information than its aesthetics [12]. The load time is a decisive factor to the users permanance in a web site, since the most users are willing to wait from 6 to 10 seconds before abandon any web site. In fact, every second delay can result in a reduction of up to 7% in conversion rates. If considering an e-commerce portal with $100,000.00 mensal invoicing, such delay may imply a loss of $84,000.00 along a year.

In this context, in recent years there has been a shift to an Internet data computing model called client / server, that addresses the failures of the centralized computing. Recently, there has been an advance in the number of public Application Programming Interfaces (APIs) driven by the transition in the distributed application communication model, which has now made extensive use of the HTTP protocol for web-based information exchange [6]. Considering this new web application development model, the adoption of REpresentational State Transfer (REST) as the predominant method to build APIs has obfuscated any other technology or approach [8]. Although several alternatives (mainly SOAP) are yet in the developement market, adepts of the SOA model to build applications have choosed the REST as the communication model and the JSON as the message format [5].

However, the increased use of REST has exposed some limitations, that harmed the performance of such APIs in crucial aspects. In general, clients with complex routines require nested object searches with multiple relationships. Due to the characteristics of a REST API, that exposes resouces exclusively, it is necessary to perform several searches on the server before some routines be processed by the client, since not all information is sent in a single reply message. Also, most of these calls will return unnecessary data to the routine context that executed it, which is known as over-fetching. Thus, multiple solutions have been proposed to increase the efficiency of data search, ones considering the requests and replies formats, while other ones are optimizing the number of requests in the network. A recent trend involves a declarative data query model, in which client applications specify data they need. Then, such models optimized the communication with server to get data in a efficient way. This is the purpose of GraphQL, which aims to mitigate some chronic problems of REST design, such as APIs versioning, multiple round trips and excessive data traffic on the network.

This paper aims to identify the differences, in terms of load time, amount of data traffic, and resource consumption, between REST and GraphQL applications. To achieve this, two prototypes of API are created, one implementing the REST design and the other one using GraphQL as a mechanism to respond to queries. The performance of both APIs is measured based on quantitative metrics and the analysis of the obtained results is discussed.

The remainder of this paper is organized as follows: Section 2 presents the main protocols for communication between Web applications, from traditional Remote Procedure Call (RPC) to current

REST and GraphQL; Section 3 describes how the prototype for comparing REST and GraphQL APIs was developed; Section 4 describes the metrics used and the results; Finally, section 5 presents the final considerations.

2 INTER-APPLICATION COMMUNICATION

In this section some of the main protocols for communication between applications are presented. Initially, some RPC and Simple Object Access Protocol (SOAP) approaches, which today are losing the development market, but played a very important role in the evolution of Service-Oriented Architecture (SOA)-based systems. Then, is presented a description of the REST and GraphQL communication models, which are the approaches of the present work.

2.1 RPC and SOAP communications

Remote Procedure Call (RPC) is a mechanism in which an application requests service from another application that is on another computer, usually connected through a network. A RPC requires a given *X* application to send one or more messages to another *Y* application in order to invoke a procedure from the *Y* application, that responds with one or more messages [7]. The fundamental idea of RPC is to be transparent, so that the client is not aware that the called procedure is executed on a different computer or vice-versa [10]. This flow of information can be seen in the Fig. 1.

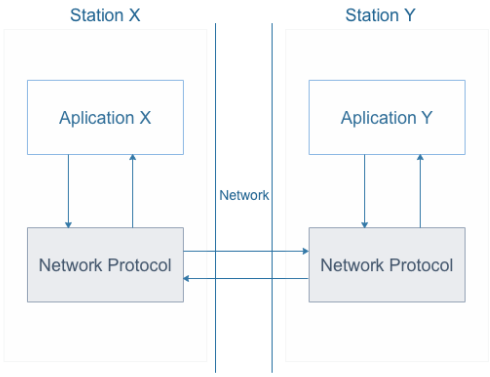

Figure 1: RPC's information flow

However, despite being a widely used technique in many distributed systems, RPC has several limitations. For example, different RPC implementations are, in general, incompatible. In addition, such an approach has other technical and performance limitations, such as different parameter structures, lack of parallelism, and lack of standardization.

An advance to RPC is the use of Simple Object Access Protocol (SOAP), a protocol for message exchange in decentralized and distributed environments. It is XML-based, and consists of three parts: a header that describes the content of transmitted messages and how to process them, a set of coding rules for expressing instances of data types defined by an application, and a convention for representing remote calls and their answers [2]. In general, SOAP eliminates some of the complexities found in RPC implementations. For example, the use of SOAP, further the standardization of the structure of message structures using XML, allows communication between protected systems without opening additional ports, reduces errors since XML format is more easily read and understood, is cross-platform, and can be used with various data transport protocols, such as HTTP, SMTP, and FTP. On the other hand, while XML is extremely robust, its use can degradate the performance in terms of coding and decoding speed, and increase the message size.

Altough this standard of messages format had advantages when compared with previous technologies, it also brought a number of limitations as its popularity emerged. Thus, the need to search for alternatives to overcome these problems increased, and new formats were emerging, and among then, the JavaScript Object Notation (JSON) format stands out.

2.2 REpresentational State Transfer

The REpresentational State Transfer (REST) is an architecture design based on a set of principles that describe how network resources are defined and addressed. The adoption of REST as the predominant method for building public APIs has obfuscated any other technology or approach in the recent years. The simplicity of REST, which deals only with data structure and transport, along with its natural fit over the HTTP, has made it the choosed method for Web 2.0 applications to expose their data [1].

The Richardson Maturity Model describes the requirements to develop a well-structured REST API and compliant with the constraints defined by the architecture. The better the API adheres to the constraints – client/server, stateless, cache, uniform interface, layered systems, or code on demand – the better it will be scored [4]. Richardson's model describes 4 levels (0-3), in which the level 3 designates a truly RESTful API.

However, with the evolving and increasing complexity of APIs, communication via the REST protocol has often proved unfeasible. This is consequence of the REST implementation, that requires the execution of multiple requests, between clients and servers, to get complex objects with nested attributes. Also, REST implementations can encourage the practice of overfetching, that is, when the client fetches some information from the server and the response contains more information than the client needs. Another difficulty is the API versioning, which occurs when there are significant changes to API, subject to code breaking in consumer clients.

2.3 JSON/Graphs-based architectures

Recently, a new architectural design has been gaining attention of developers, filling some gaps that previous architectures have left. GraphQL is a query language, created by Facebook in 2012, that provides a common client-server interface for data manipulation and fetching. It uses a system called client-specified queries, in which the response format of a request is defined by the client. Thus, since the data structure is not coded, the server data query becomes more efficient for the client [9].

Also, queries using GraphQL always return only what was predefined by the request. So its answers are predictable. GraphQL queries not only access the properties of a single resource, but also follow the references between them. As a consequence, while

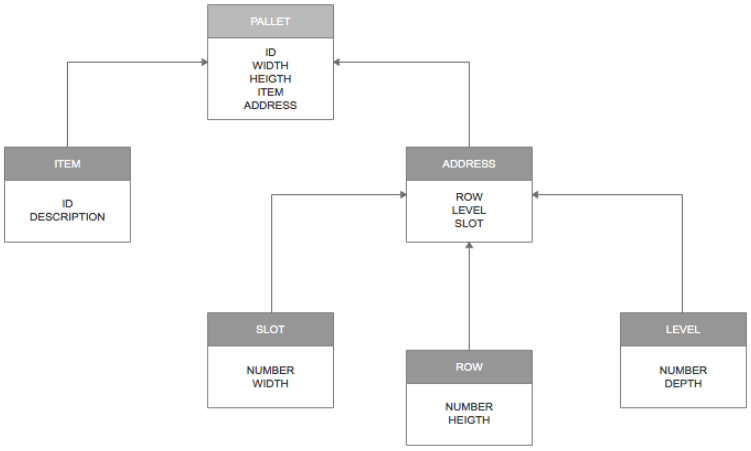

Figure 2: WMS modelling

typical REST APIs require loading multiple URLs, GraphQL APIs get all the data they need in a single request.

On the other hand, a major threat that GraphQL facilitates is the denial of service attacks [11]. A GraphQL server can be attacked with an excessive number of complex queries that may to consume all server resources. This type of attack is not GraphQL-specific, but in such case extra caution is needed to avoid them. There are, however, some procedures that, if implemented, can mitigate the threat of denial of service. You can previously perform a cost analysis of the query and impose a threshold of the amount of data a request can consume. You can also implement a timeout, in which the requests that take a long time to resolve are deleted from the execution queue.

3 CASE STUDY

For a real-world example, a case study was designed based on a Warehouse Management System (WMS) application. In order to perform relevant queries, a schema containing six entities was modeled, representing real queries and producing information for a performance analysis. The modeling of figure 2 represents the management of a warehouse with capacity to store several items. Items are placed in pallets and allocated at addresses within the warehouse.

Consider a code item 22B12, for example, that represents a given product X. This product is placed in a pallet that can hold 30 units of item 22B12. The warehouse consists of 26 shelves, sequenced from "A" to "Z". Each shelf consists of two depth lines and three height levels. A pallet with code 001 contains 30 units of item 22B12, and needs to be allocated into the warehouse. For this, a code system involving the three dimensions of the warehouse is employed.

Thus, 60 units of item 22B12 are placed on two pallets (001, 002) in the warehouse, and each pallet will be assigned to an address. The pallet 001 will be allocated to the third shelf, the second level and the first row. After address formation, pallet 001 will be located at address $C0201$, shelf C, level 02 and line 01.

Based on a WMS system, two APIs prototypes are implemented. The first is implemented using the best practices of REST design, while the second one is on GraphQL. The performance of such prototypes is compared to each other.

3.1 Assumptions and hypotheses

The assumptions on the performance difference between the APIs derived from the theoretical foundations. They are based on the understanding that the protocols allow the implementation of a combination of techniques to positively affect the performance of the API. Therefore, both implementations must have the same properties, following their best practices, maturity models, and documentation.

The hypotheses of this work are:

(1) Response size will be smaller using GraphQL;
(2) Response time will be shorter using GraphQL;
(3) CPU utilization time will be shorter using REST;
(4) Memory consumption will be lower using REST;

In order to validate the defined hypotheses, two questions involving all entities were determined. For each question there is only one correct answer and its logic is based on fields of the return data structures.

Question 1: Which item occupies the largest amount of pallets allocated in the warehouse?

Question 2: Item code 22B12 is stored in which addresses?

Thus, searches will be performed in both the REST and the GraphQL API in order to retrieve the information needed to formulate the answers. To perform these searches, measure the performance of such APIs processing their responses, and compare them, some tools were used during this work. These tools are described in the section 3.2.

3.2 Used tools

Choosing of the tools to be used for implementation is one of the most important parts of the case study planning. The APIs are

written in JavaScript using the ECMAScript 5 specification, which allows the access to tools for building web services, such as Node.js, used in this study. MongoDB, which adopts an object-oriented data model, is used for data persistence.

In order to simplify the construction of web server applications, the Express.js framework is used. In the developed prototypes, Express.js acts as a middleware that manages routes and delegates responsibility for interpreting requests to the Controllers in the REST API and to the Resolvers in the GraphQL API.

The listing 1 illustrates how Controllers send request information to the model, which is responsible, in the REST API, for executing the queries in the database. In the example, the code snippet returns an "Item" based on the received id as a request parameter. Note that in line 2 the "Item" entity model is imported for use in the Controller, and in line 4, the get method, whose logic is implemented within the model, is invoked by passing the id as parameter.

Listing 1: Controller to get an item

```
1  //item.controller.js
2  import Item from '../models/item.model';
3
4  function load(req, res, next, id) {
5    Item.get(id)
6      .then((item) => {
7        req.item = item;
8        return next();
9      })
10     .finally(e => next(e));
11 }
```

In turn, listing 2 shows how Express.js manages the request received via the GET method, and delegates responsibility for schema.js, where is the logic for interpretation of request parameters, and returns a JSON object with the appropriate response.

Listing 2: Express handling routes to GraphQL

```
1  //server.js
2  import express from 'express';
3  let app = express();
4
5  import schema from './schema.js';
6
7  app.get('/', (req, res) => {
8      graphql(schema, req.query.query)
9      .then((result) => {
10         res.send(result);
11     });
12 });
```

Some tools are used as clients of the built APIs, and execute queries. Postman software is primarily used for performing API searches. Along with the query response, Postman provides information such as the time required for the response and its size, in bytes. These features, combined with the option to perform a customizable number of iterations for each request, are the basis for evaluating the performance of both the REST and GraphQL APIs.

3.3 Scenario

The APIs were built to respond to requests, returning responses in the JSON format to generate a performance measurement factor of the test execution. For the execution of the experiment, only the necessary applications are kept active. Therefore, when performing the searches, either the REST or the GraphQL server will be active, as well as the MongoDB database server and the Postman client application.

The REST server consists of a HTTP server written in JavaScript an running on a Node.js server that receives HTTP requests. Depending on the method and URL of the request, the server routes it to the corresponding controller. Then, controller queries the MongoDB database, and records the appropriate latency data. After the processing, the response is sent to the customer. For measurements, the client application sends, for example, a request to retrieve all registered items. As can be seen from Table 1, it must perform a /item search, which is interpreted by the REST server. The server queries the database by returning a response in JSON format, with all items registered in the API.

URI	Descrição
/item	Query items list
/item/:id	Query an item by id
/pallet	Query pallets list
/pallet/:id	Query palley by id
/address	Query addresses list
/address/:id	Query address by id
/slot	Query shelves list
/slot/:id	Query shelf by id
/row	Query rows list
/row/:id	Query row by id
/level	Query levels list
/level/:id	Query level by id

Table 1: REST server

The GraphQL server was also implemented using the Node.js. The difference compared to the REST server implementation is that the GraphQL server sends all requests to its core, rather than routing incoming requests to many different controllers. GraphQL parses the query and sends the parameters to the responsible resolvers located in the schemas. These functions are performed when the corresponding fields are queried and the results are returned in the response. To retrieve some information in the GraphQL API, it must make a request using the HTTP GET method with the desired query. The API processes the query and returns an JSON object. The query illustrated in the code snippet of listing 3 request the list of all items stored in the database. Note, that the answer contains only the attributes the query requested – the items id description.

3.4 Metrics

Four metrics were defined to compare the performance measures of APIs developed in REST and GraphQL. The metrics are: CPU utilization time, memory consumption, response time and response size. Note that each metric was measured separately so that logs and outputs related to a specific metric do not interfere on the results of the other ones.

```
query RootQuery {
 items {
  id
  description
 }
}
{
 "data": [
  {
   "id": 22B12,
   "description": "Flat screen"
  },
  {
   "id": 21C44,
   "description": "Computer screen"
  },
  {
   "id": 43F12,
   "description": "Smartphone screen"
  },
 ]
}
```

Listing 3: Items resquest and response

The definition of each metric is detailed as follows:

- **CPU utilization time**: time, in miliseconds (*ms*), in which a CPU processed the instructions. It is extracted through *process* module of Node.js *core*. This metric can be defined as

$$CPU = \frac{\sum \Delta cpu}{n},$$

in which n is the number of CPUs and Δcpu is the CPU use time by the application.

- **Memory consumption**: memory, in megabytes (*MB*), used by the API in each search. It can be defined as

$$Mem = \frac{m}{M} * 100,$$

in which m is the amount of memory used by the application and M is the total amount of memory.

- **Response time**: time gap, in miliseconds (*ms*), between each resquest and its respective response. In case of REST API, this metric considers all the needed requests. It can be defined as

$$\Delta t = T2 - T1,$$

in which $T1$ and $T2$ respresent, respectively, the request and the response time.

- **Response size**: size, in *bytes*, of the response. In case of the REST API it considers the mean of the sum of all searches. It can be defined as:

$$Size = \sum^{n} ti,$$

in which ti is the size of each response and n is the total number of requests.

Figure 3 shows how the metrics will be extracted. CPU utilization and memory consumption will be measured using Node.js tools, via logs in the prototype source code. Response size and response time will be extracted using the Postman tool at the end of queries.

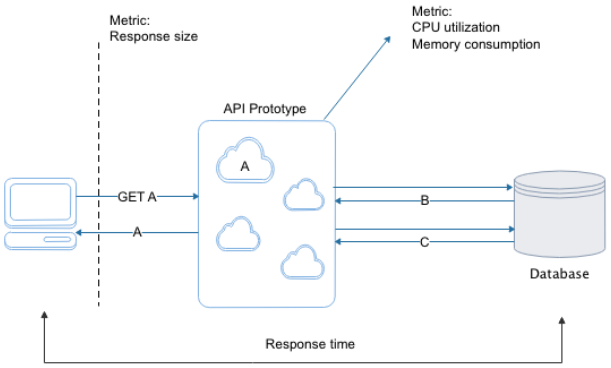

Figure 3: APIs architecture and the different measure points

4 RESULTS

The results obtained in the validation tests present the average of the values collected after executing 30 iterations for each request. Three scenarios were also set up for better analysis of the results. These scenarios have different record quantities for the Item and Pallet entities, as they are the ones that most significantly affect how efficient is the performance of the APIs. The three scenarios are described in the table 2.

Resource	Scenario 1 (S1)	Scenario 2 (S2)	Scenario 3 (S3)
Item	1000	10000	30000
Pallet	1000	10000	30000
Address	156	156	156
Slot	26	26	26
Row	2	2	2
Level	3	3	3

Table 2: Analyzed scenarios

Queries were based on a real use case from a WMS system and aimed to answer the following question: What are the addresses containing Item *22B12*. Results obtained in the validation tests present the average of the values collected after several executions of the scenarios in a sequential way. By analyzing the results, it is possible to identify the difference in the performance of APIs, highlighting the difference in the obtained size of the response .

4.1 Question 1

Although it needs simpler searches, through Question 1 is already possible to see the performance differences between REST and GraphQL applications. Question 1 looks for the item with the largest amount of pallets allocated in the warehouse. To answer it is required two steps: the first one searches all pallets registered in the system, and after identifying the most common item in the pallets, the second step details such an item.

4.1.1 CPU utilization time. By analyzing the CPU utilization time, illustrated in the figure 4, it can be seen that both API, REST and

Request	Result	# of requests
/items	Item 22B12 ID	1
/items/:id	Item 22B12 details	1
/pallets	Pallets with item 22B12	1
/pallets/:id	Details of the Pallet withitem 22B12	5
/addresses/:id	Detais of address with item 22B12	5
/levels/:id	Level with item 22B12	5
/slots/:id	Shelf with item 22B12	5
/rows/:id	Row with item 22B12	5

Table 3: Data flow to get the results

GraphQL, have similar performance in scenarios S1 and S2. On the other hand, queries from scenario S3 demonstrate that REST API is less efficient, requiring more CPU time.

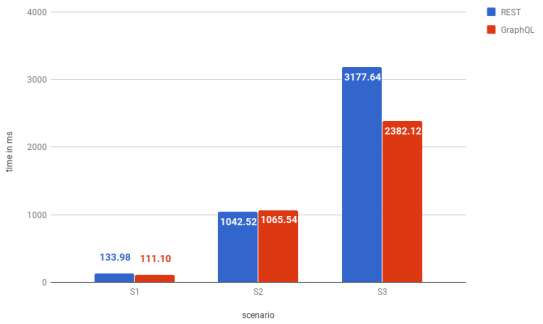

Figure 4: Comparison of CPU utilization time

Note that REST API required 133.80 ms of CPU to perform S1 queries, while the GraphQL API required 111.10 ms. When considering the scenario S2, the REST API was processed at 1042.52 ms while the GraphQL API took 1065.45 ms. We noted that in scenario S2, both APIs presented the most similar CPU utilization time. Finally, when analyzing the S3 scenario results, there is a major disadvantage for REST API, which used 3177.60 ms of CPU while the GraphQL API used only 2382.10 ms, a difference of about 25%.

4.1.2 Memory consumption. Results of memory consumption comparison show that the REST API is also less efficient than the GraphQL API. The results can be seen in the figure 5.

When comparing the results of S1 and S3 scenarios, the REST API proved to be about 15% less efficient than the GraphQL API. The difference is more relevante when comparing the S2 results, in which the GraphQL API consumed 127.71 megabytes of memory and the REST API consumed 178.01 megabytes, a difference of approximately 30%.

4.1.3 Response time. As expected, the API implemented by using GraphQL actually responded to queries in a shorter time than the REST API. Figure 6 shows the difference in the APIs response time to execute the queries of the first question.

In the requests of scenario S1, the REST API resulted in a response time of 147.23 ms, while the GraphQL API answered the query in 115.63 ms, representing a difference of 21%. When analysing the

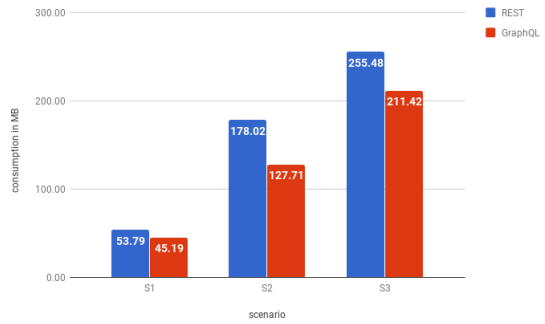

Figure 5: Memory consumption comparison

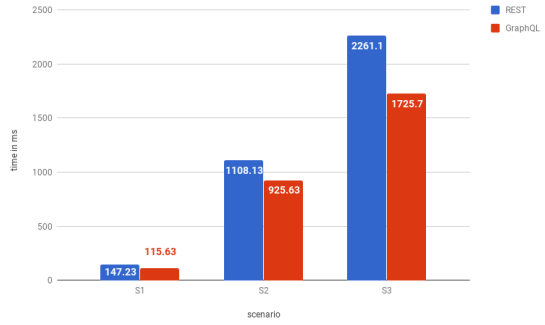

Figure 6: Response time comparison

queries of scenario S2, the REST API answered such queries at 1108.13 ms and the GraphQL API returned the results at 925.63 ms, a difference of 16%. Finally, queries of scenario S 3were answered at 2261.10 ms in the REST API and 1725.70 ms in the GraphQL API, which represents a difference of 23%.

For the response time we extracted another graph, which can be seen in the figure 7, which illustrates the response time for each of the 30 requests in both the APIs. This graph is based on the queries for scenario S1, and helps to explain the response time behavior of each API. Note that in both the prototypes the first request takes a much longer time than the average time. This happens since in the first request the API needs a warmup time to be at full performance. This warmup period occurs only on the first request, and the next requests are already much faster.

4.1.4 Response size. Another expected result was that the GraphQL API response size was smaller than the REST API response size. This hypothesis was confirmed as can be seen in the figure 8.

The REST API answered the requests from Question 1 with a response size of 174.04 kilobytes, 1740.17 kilobytes, and 4980 kilobytes for scenarios S1, S2, and S3, respectively. Similarly, the GraphQL API resulted in responses of 31.68 kilobytes, 322.53 kilobytes, and 967.35 kilobytes. Comparing the three scenarios shows a constant difference of about 80% between the REST API and the GraphQL API.

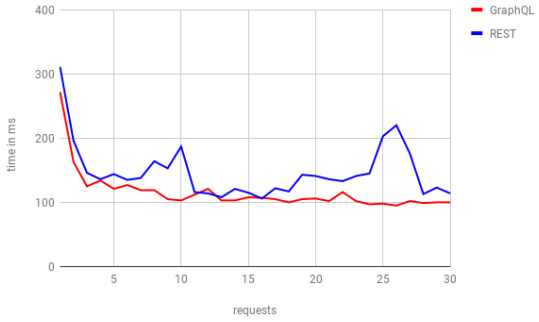

Figure 7: Response time

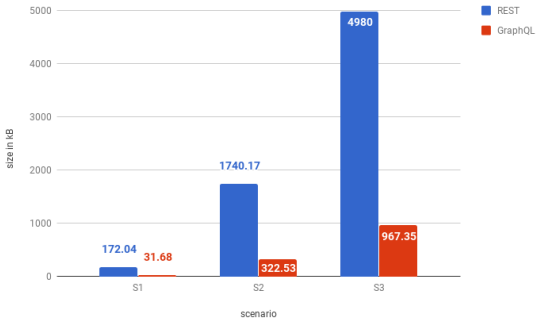

Figure 8: Response size comparison

4.2 Question 2

For the answers to question 2, the queries were more complex in the GraphQL API and more numerous in the REST API. These queries show which addresses contain Item 22B12, and the results allow us to identify the biggest performance difference of the APIs, highlighting the size of the response used.

4.2.1 CPU utilization time. The graphic in figure 9 illustrates the CPU utilization time results. Note that GraphQL API uses this feature more efficiently than REST API. In the evaluated scenarios, the difference in the obtained results is more clearly noted as the number of records increases. Analyzing scenario S1, REST API required 244.41 ms of CPU time to process, while the GraphQL API required only 178.22 ms. In queries of scenario S2, REST API required 1787.74 ms to process, and the GraphQL API required 1199.53 ms. However, the biggest difference is in scenario S3, in which REST API required 5383, 40 ms to be processed by the CPU and the GraphQL API only 3132, 98 ms, a difference of more than 40%.

4.2.2 Memory consumption. Memory consumption results are illustrated in the graphic of figure 10, which indicates that REST API makes a less efficient use of this feature. In requests of scenario S1, the REST API proves to be more efficient even though the difference is only 9.88 megabytes (about 12%) from the GraphQL API. The GraphQL API consumed 76.19 megabytes of memory, compared to 66.27 consumed by the REST API. However, this better efficiency is no longer identified in the scenario S2, in which the REST API

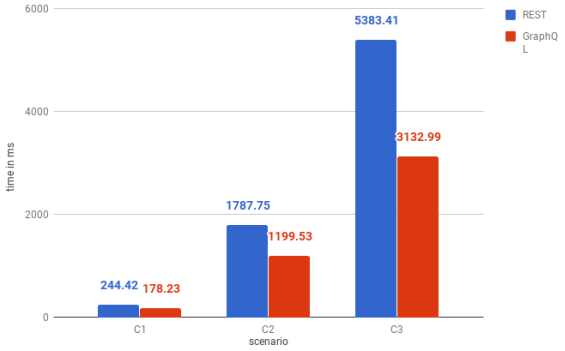

Figure 9: Comparison of CPU utilization time

consumed 181.02 megabytes of memory, and the GraphQL API consumed 26% less, 133.52 megabytes. In scenario S3, there is a more significant difference, with the REST API consuming 300.30 megabytes of memory while the GraphQL API consummed 206.02 megabytes, a difference of almost 30%.

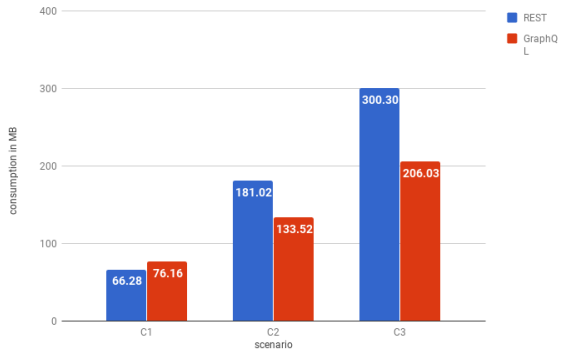

Figure 10: Memory consumption comparison

4.2.3 Response time. As the graph in figure 11 illustrates, the REST API takes a longer time to respond all requests when compared with the GraphQL API. Considering the queries of scenario S1, the REST API required a response time of 254.56 ms, while the GraphQL API answered the queries at 148.50 ms, a difference up to40%. When analyzing the requests of scenario S2, the REST API answered the queries at 2072.03 ms and the GraphQL API at 1201.30 ms, a difference of 42%. Finally, queries of scenario C3 were answered at 4770.20 ms in the REST API and at 2291.10 ms in the GraphQL API, which represents a difference higher than 50%.

The graphic in figure 12 shows the response time for each of the 30 requests. This result considers the scenario S1, and details the response time behavior. You can see that the first request takes a longer time than average, since the API needs a warmup time needed to be at full performance. The REST API answers the first request in almost 500 ms, while the GraphQL API in approximately 400 ms. This difference of approximately 100 ms is maintained over the course of requests, and the GraphQL API is slightly more stable than the GraphQL API, with smaller oscillations.

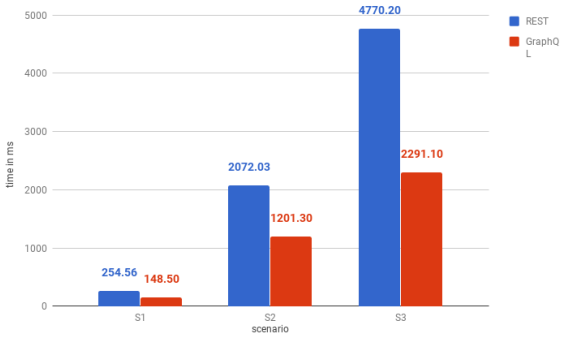

Figure 11: Response time comparison

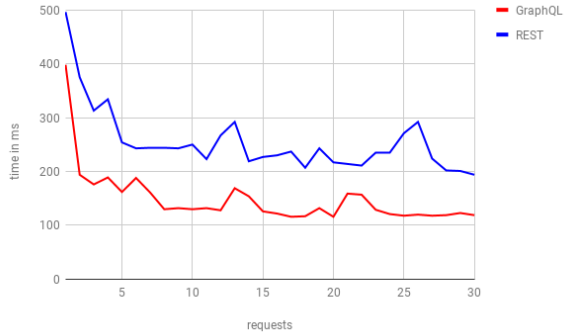

Figure 12: Response time

4.2.4 Response size. The graphic in figure 13 shows that the API developed by using GraphQL also proved to be more efficient in response size, which directly affects the network cost. The REST API responded to requests with an average response size of 259.73 kilobytes, 2522.17 kilobytes, and 7221.00 kilobytes for scenarios S1, S2, and S3 respectively. The GraphQL API had responses with 101.73 kilobytes, 1005.23 kilobytes, and 2850.00 kilobytes. The performance difference between the APIs was about 60% in all the three scenarios, making it clear that GraphQL API is more efficiency.

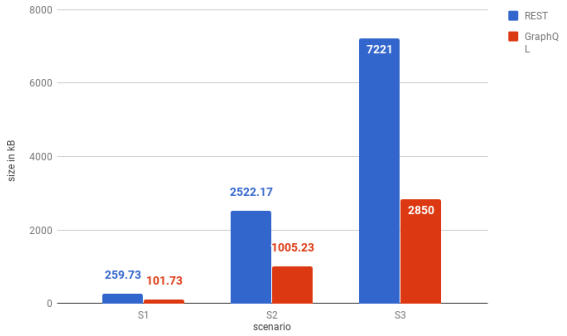

Figure 13: Response size comparison

5 CONCLUSION

With more interconnected devices, efficient communication between applications is an issue that is constantly debated and evolving. Computational resources such as memories and processors, although gradually more accessible, still demand concern regarding their sustainable use. This paper presented concepts and models of communication between applications, namely REST and GraphQL. Experiments were performed to compare them in terms of response time, amount of used bandwidth and computational resource consumption.

At the end of the work, we concluded that both technologies offer a practical and efficient solution to the same problem, however GraphQL presents itself as a great alternative as a communication mechanism between applications. Ease of use, lower bandwidth and computational resources contribute to GraphQL being a tool widely used in the future by organizations and developers. In addition to the performance presented criteria, GraphQL helps to maximize application development productivity, resulting in improved Developer Experience (DX), an area that is increasingly gaining dedication and importance.

However, the maturity of the GraphQL ecosystem is still much discussed by the community. Future work could address how using JSON / Graphs-based architectures can optimize software development not only in a quantitative but also qualitative metric, considering the Developer Experience. Other work may also address the advantages of strongly typed communication, such as GraphQL, or whether categories theory can be applied with the GraphQL.

REFERENCES

[1] Robert Battle and Edward Benson. 2008. *Web Semantics: Science, Services and Agents on the World Wide Web.* BBN Technologies, Arlington, VA, USA.
[2] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. 2000. *Simple Object Access Protocol (SOAP) 1.1.* W3C Consortium.
[3] Cisco. 2017. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper.* Technical Report. Cisco CO.
[4] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation. University of California, Irvine. AAI9980887.
[5] Ole Lensmar. 2013. Is REST losing its flair - REST API Alternatives. https://goo.gl/bPm1ez
[6] Mateus Maso. 2016. *Um Modelo de Comunicação para Automação na Execução de Consultas de Dados sobre APIs Web.* Ph.D. Dissertation. Universidade Federal de Santa Catarina.
[7] P. Merrick, S. Allen, and J. Lapp. 2006. *XML remote procedure call (XML-RPC).* Patent. https://www.google.com/patents/US7028312 US Patent nr. 7,028,312.
[8] A. Neumann, N. Laranjeiro, and J. Bernardino. 2018. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing* Early Access (2018). https://doi.org/10.1109/TSC.2018.2847344
[9] Nick Schrock. 2015. GraphQL Overview. https://goo.gl/wX2VJ8
[10] Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed Systems: Principles and Paradigms* (2 ed.). Pearson Prentice Hall, Upper Saddle River, NJ.
[11] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. 2018. Experiences on Migrating RESTful Web Services to GraphQL. In *Service-Oriented Computing – ICSOC 2017 Workshops*, Lars Braubach, Juan M. Murillo, Nima Kaviani, Manuel Lama, Loli Burgueño, Naouel Moha, and Marc Oriol (Eds.). Springer International Publishing, Cham, 283–295.
[12] Sean Work. 2018. *How Loading Time Affects Your Bottom Line.* Disponível em https://goo.gl/xUpdJ8. Acessado em out de 2018.