An Empirical Study about the Adoption of Multi-language Technique in Computation Offloading in a Mobile Cloud Computing Scenario

Filipe Fernandes S. B. de Matos[1], Paulo A. L. Rego[2] and Fernando A. M. Trinta[2]
[1]*GOHaN Research Group, Federal University of Ceará, Crateús, Brazil*
[2]*GREat Research Group, Federal University of Ceará, Fortaleza, Brazil*

Keywords: Multi-language, gRPC, Offloading, Performance Evaluation, Mobile Cloud Computing.

Abstract: Low processing capabilities and limited energy autonomy are common restrictions faced by most mobile devices. In order to address these issues, the computation offloading technique has been proposed to transfer tasks from low processing devices to other machines with higher computing capability. This paper presents an empirical study on the performance of multi-language techniques in offloading procedures. Our experiments evaluate the processing time and the energy consumed by a mobile device when executing methods of two applications locally (on a mobile phone) and remotely (via offloading) on a server process developed using distinct programming languages (Go, C++, Java, and Python). Google's gRPC and Protocol Buffers were used as a data serialization mechanism to allow offloading between client and server processes. The results show that using a multi-language approach for offloading can reduce the processing time by up to 39 times and the mobile device's energy consumption by up to 96% approximately.

1 INTRODUCTION

It is widely well-known the increase in the number of mobile devices (for example, smartphones or tablets) available in the global market in the last decade (O'Dea, 2021). Sales growth motivated the emergence of new applications and constant improvements in the hardware of these devices. However, despite all this technological progress, these devices still have severe computational restrictions (enhanced by the emergence of increasingly complex and demanding applications in processing and storage), and issues related to power consumption (Coulouris et al., 2011).

A recent study (Cisco, 2020) presents impressive projections about mobile devices and applications' demand until 2023. It reports a global increase of 5% and 8% in the number of mobile users and connections, respectively. The same document also estimates an increment in the number of application downloads in the range of hundreds of billions and a more extensive exploration of machine-learning techniques and other computationally expensive procedures. Studies like this indicate that the restrictive problems of mobile devices could become more critical soon.

Computation offloading is a possible solution to mitigate such a problem. This method allows a more restrictive device (especially concerning computational and energetic aspects) to submit a task/method to be processed in another equipment (less restrictive) via network and then receive the processing result. The offloading technique can also be applied for storage purposes, where a more restrictive device can send data to be persisted remotely in another machine with a larger storage capacity. The decision on whether the offloading will be enforced locally or remotely depends on factors like network quality and task complexity, for example.

There are too many programming languages available in the market. Each has peculiar features that vary from simple technical aspects (such as the data types and code structures available) to more complex differences (portability or parallelism support, for instance). These differences make each language better suited to specific activities, and their programs have different computational performances when running the same task (Sebesta, 2012). Due to all these distinctions, developers can create systems (or part of them) using multiple programming languages, making communication between components a challenge. One possible solution for this problem is the marshaling/unmarshaling multi-language technique, which consists of standardizing data representation and allows programs written with different programming languages to communicate with each other (Coulouris

B. de Matos, F., Rego, P. and Trinta, F.
An Empirical Study about the Adoption of Multi-language Technique in Computation Offloading in a Mobile Cloud Computing Scenario.
DOI: 10.5220/0010437802070214
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 207-214
ISBN: 978-989-758-510-4

et al., 2011).

This paper presents an empirical study about the performance of offloading methods when performed in a multi-language approach. In this method, processes developed with different programming languages can perform computation offloading with each other. We aim to investigate the impact caused on the offloading performance by adopting different programming languages on the server-side. We conducted several experiments using two applications that performed the offloading of execution methods in processes implemented in four different languages. The results indicate that it is possible to reduce the processing time and the energy consumption of computationally complex tasks when submitted to processes implemented with compiled or partially compiled languages. The processing time in scenarios with these types of languages was 39% lower, while the mobile device's energy consumption decreases up to 96% caused by the offloading performance. The main contributions of this work are:

1. Adapt two applications in the literature and develop eight server processes (four for each application) able to performing offloading via gRPC;
2. Contribute with new experiments and results that indicate gRPC as a promising framework for computation offloading solutions;
3. The results obtained in this paper indicate that the adoption of a server process developed with a non-Java programming language can provide significant gains in computation offloading.

This work is organized as follows: Section 2 addresses related works. Section 3 contextualizes the main concepts related to conventional offloading and gRPC. Section 4 defines the offloading multi-language model and presents the configurations of the tests performed. Section 5 presents the results obtained and analyzes them. Finally, Section 6 presents the obtained conclusions and future works.

2 RELATED WORKS

There are several works in the literature related to the offloading technique in Mobile Cloud Computing (MCC) contexts. However, few of them on adopting gRPC as a support technology for communication between devices. To the best of our knowledge, (Chamas et al., 2017) was the first work to conduct a study in this direction. The authors compared the performance of REST, SOAP, Socket, and gRPC, analyzing energy consumption and total processing time as a communication resource during offloading procedures. To this end, the authors used a data sorting application in their tests. This application consisted of sorting data (integers, floats, or objects) using well-known sorting algorithms (Bubble Sort, Heap Sort, or Selection Sort). Such an application can compute tasks locally or send them to a server process developed in Java for processing using one of the communication resources. Although this work had adopted a wider variety of data types, it was limited to analyzing the offloading performance using only one application and one programming language.

(Araújo et al., 2020) conducted a specific study about the performance of gRPC in the Internet of Things (IoT) context. The authors evaluated the total time spent in offloading methods in a scenario with two applications developed in Kotlin. The first application performs the multiplication of random matrices with order *NxN*, while the second one applies different filters to images of different resolutions. The mobile device offloads its tasks to a server process developed in C++, Python, Java, Go, or Ruby and hosted in a BeagleBone Black device. The communication between the mobile application and the server process used gRPC. Their study performed experiments with more than one application and more than one programming language on the server-side. However, the proposed test environment is somewhat atypical since the mobile device has better configurations than the server-side (for instance, processor and primary memory resources). In typical offloading scenarios, mobile clients offload their tasks to more robust server devices with more significant computational resources and energy capabilities.

(Georgiou and Spinellis, 2019) conducted a comparative study involving three widely known interprocess communication mechanisms: gRPC, RPC, and REST. The authors developed client and server applications using some of the most popular languages, such as Java, Python, and C#. The scenario for the experiments involved machines with both Intel (two Desktops) and ARM architectures (two Raspberry Pi), interconnected by a wired local network (LAN). The authors analyzed the processing time of the tasks and the energy consumption on both devices. The results showed that Go and JavaScript's implementations had the fastest execution time and the lowest energy consumption. Despite using more programming languages in the development of server and client programs, the application proposed in the experiments was a "toy application" (a *Hello World* in a Request/Response fashion). The adopted environment is also not consistent with a typical MCC scenario because it does not use cell phones and inter-

Table 1: Comparison between this work and related works.

Work	Apps Adopted	Server Languages	Metrics Evaluated		
			Time	Energy	Network
(Chamas et al., 2017)	Sorting	Java	✓	✓	✗
(Georgiou and Spinellis, 2019)	Request/Response (HelloWorld)	Go, Java, JavaScript, PHP, Python, Ruby, and C#	✓	✓	✗
(Araújo et al., 2020)	Image filtering and Matrix multiplication	C++, Python, Java, Kotlin, Go, and Ruby	✓	✗	✗
This work	Image filtering and Matrix multiplication	C++, Go, Java, and Python	✓	✓	✓

connects the devices through a wired network, which tends to simplify the network's issues.

Table 1 summarizes the comparison between the works. This paper presents the evaluation of gRPC as a computation offloading tool (including using a multi-language approach) as its main contribution. None of the related works carried out tests, including all the following aspects simultaneously: 1) Adopt recurrent application types in offloading tests by the literature (Silva et al., 2016); 2) Use different types of languages in the development of the server process (compiled, partially compiled and interpreted); 3) Evaluate processing time, energy consumption, and network time as performance metrics.

3 BACKGROUND

This section presents the main concepts associated with the computation offloading and the gRPC framework. Both topics are essential for a better understanding of the paper's proposal and the tests.

3.1 Computation Offloading

Computation offloading is an approach where devices with low computational power can migrate tasks for processing in a remote execution environment (RRE) with better computational resources (Fernando et al., 2013). RREs vary from virtual machines hosted by public cloud providers to traditional devices on the same local network where the weaker device is running. However, the effective performance of offloading depends typically on a decision process to evaluate whether migrating the task to another device with higher computational power is advantageous or not. In general, this decision making involves criteria such as (*i*) performance, (*ii*) energy savings, or (*iii*) both (Kumar et al., 2013). If performing offloading is not advantageous, the task is executed locally by the weaker device itself.

In general, the performance has been the most evaluated criteria by offloading solutions (Silva et al., 2016). At this point, it is crucial to highlight that the total time of offloading comprises, in addition to the process execution time remotely, the time related to sending data to the offloading target and receiving the response from it. Thus, factors such as the quality of the connection between the client device and the remote server, the amount of data sent, and the response's size significantly impact the advantage or not of the offloading (De, 2016).

3.2 gRPC Framework

gRPC is a framework developed by Google to simplify and potentialize Inter-Process Communication (IPC) in a distributed system (Indrasiri and Kuruppu, 2020). gRPC adopts the Client/Server model and supports advanced features such as load balancing, data streaming treatment, multiplexing, cryptography, among others. Another advantage of gRPC is the extensive support for several programming languages such as Java, PHP, Ruby, C#, Dart, and Objective-C.

The development with gRPC is similar to the development of other traditional RPC tools. Initially, the developer must create the interaction interfaces between the processes and define the messages exchanged between them using an Interface Definition Language (IDL) with syntax based on another Google tool: the Protocol Buffers. The gRPC also adopts Protocol Buffers as a standard mechanism for representing its messages, as well as marshaling and unmarshaling them. This standardization of messages allows programs developed with different programming languages to talk to each other. So, the main objective of gRPC is to abstract from the developer how the remote invocation procedure is performed.

Once defined the interfaces and the messages, the generated code is submitted to a compiler (specific for each language) that produces a standard code in the target language. With the generated code, the developer must implement the client and server processes and the server's services. Finally, the developer must inform the server process IP address and port to the client process, and the system is ready for execution.

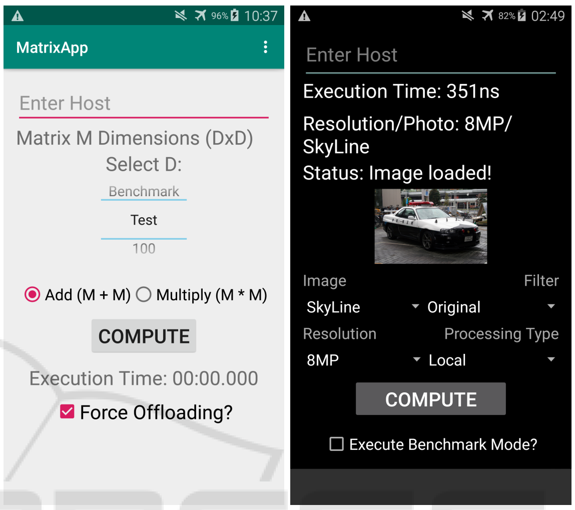
4 MULTI-LANGUAGE OFFLOADING EVALUATION

Our work conducted a performance evaluation of the computation offloading between an Android mobile device and a server process developed with different programming languages (C++, Java, Python, and Go) when performed using multi-language techniques. Initially, we chose the languages based on their type and popularity: compiled (C++), partially compiled (Java), and interpreted (Python). Later, we selected Go because it has a different architecture compared to other languages in gRPC (Mastrangelo, 2018). Like traditional offloading techniques, the idea is to allow devices with computational or power restrictions to submit part of their processing to remote servers with better computing resources or unlimited power sources (i.e., directly connected to a power supply). However, the possibility that a process developed in a programming language can submit tasks to a remote process developed in another language differentiates the multi-language approach from the traditional ones. By adopting the multi-language approach, developers may take advantage of features, optimizations, and other languages' facilities.

Like traditional offloading, multi-language offloading is based on Client-Server architecture too. In architectural terms, the difference between the models is how the communication occurs among the devices. As the communication involves processes developed with different programming languages (i.e., with different resources and forms of information representation), the messages exchanged must follow a pre-defined pattern and compatible with adopted languages. In this respect, the gRPC is perfect because, in addition to standardizing remote service invocation (and the messages involved), it also supports a wide variety of programming languages. It is essential to highlight that we could have chosen any other multi-language communication tool for this purpose, as long as the above requirements are satisfied.

We choose two applications to analyze the performance of the multi-language offloading proposed in this work: MatrixGRPC and BenchImageGRPC. We choose these applications because they are the most adopted by literature in tests with offloading (Silva et al., 2016). Both applications are modified versions of two applications already used in the literature (Rego et al., 2017). The modifications consist of adapting them to work together with gRPC. Figure 1 shows the initial screenshot of the two applications mentioned above. The objectives with the application choices are as follows:

- **MatrixGRPC.** This application computes multiplication between quadratic matrices. Our objective is to analyze offloading performance with a task that requires high computational power and large input and output parameters;
- **BenchImageGRPC.** This application applies a filter in images with different resolutions. Our objective is to analyze the performance of offloading with a common task in users' daily lives.



Matrix M Dimensions (DxD)
Select D:
Test
100
[✓ Add (M + M)] Multiply (M * M)
Execution Time: 00:00.000
[✓] Force Offloading?

Execution Time: 351ns
Resolution/Photo: 8MP/ SkyLine
Status: Image loaded!
Image: SkyLine
Filter: Original
Resolution: 8MP
Processing Type: Local
[ ] Execute Benchmark Mode?

(a) MatrixGRPC (b) BenchImageGRPC

Figure 1: Main screens of the applications adopted.

In MatrixGRPC, the user must choose the dimension of the matrices, the operation to be performed, and whether the processing must be done remotely or locally. The matrices are quadratic and also formed by integers. Upon receiving the response, the application displays the task's execution time. In BenchImageGRPC, the user must choose the image, its resolution, the filter to be applied, and whether the processing must be done remotely or locally (choosing the Local or Cloudlet option, respectively). Both applications have a benchmark mode where it is possible to perform all tests automatically.

On the server-side, it is required to highlight the libraries and resources adopted to perform mobile devices' offloading. In addition to gRPC, we used the following libraries: in MatrixGRPC, Gonum, Numpy, Eigen, and Apache Math in Go, Python, C++, and Java implementations respectively; in BenchImageGRPC, we used OpenCV4 as a base library for manipulating the images. In Python, Java, and Go languages, we used wrappers from the base library developed in C++, respectively, opencv-python, opencv-java e GoCV. The adoption of libraries can skew (positively or negatively) the implementations' performance since they tend to execute the tasks effi-

Table 2: Details of the experiment.

System (used devices)	Smartphone with Qualcomm Snapdragon 600 Processor (1.9GHz, Quad Core), 2GB RAM and Android 5.0.2, Notebook with Intel Core i5-5200U Processor (2.20GHz, Quad Core), 8GB RAM and Ubuntu 19.04 64 bits, A Netgear WGR612 router was used to build an exclusive 2.4 GHz wireless network between the devices. PowerMonitor connected to the smartphone to collect energy consumption data from it.
Factors/ Levels	Applications (MatrizGRPC and BenchImageGRPC), Processing Type (Local or Remote), Programming Languages (Go 1.14.6, Java Openjdk 8, Python 2.7.16, and C++ 5.1.0), Matrix dimension of MatrixGRPC (400x400, 700x700, and 1000x1000) and image resolution of BenchImageGPRC (0.3 MP, 4 MP, and 8 MP).
Iterations	Each experiment was performed 33 times for each combination of factors/levels. Totalizing 990 iterations (2 applications $\times$ 5 processing types $\times$ 6 matrix dimensions or image resolution $\times$ 33 iterations).
Response Metrics	Total time of processing, Time spent in network operations (sending and receiving data), and Energy consumption of the mobile device.

ciently. Thus, implementations that use libraries tend to perform better than those that do not. As a consequence, there is also no guarantee that different libraries adopt the same optimizations. It is also possible to find variations in performance results between implementations that used different libraries.

Table 2 summarizes the main configurations and parameters of the experiments performed. The test environment consisted of a wireless network with a star topology and exclusive to both devices involved in the tests. Unfortunately, because it is a widespread frequency of use (Dolińska et al., 2017), even though it is a dedicated network, it was not possible to isolate it from other networks and devices that operate on the same frequency. Thus, data transmissions were not free from external interference and collisions, which can delay the effective delivery of packets and generate unwanted variations in the time spent on network operations and, consequently, in the total offloading time. We monitored the mobile device's power consumption during offloading using the Monsoon PowerMonitor equipment[1].

Each application consisted of two parts: an Android client hosted in the smartphone and one server developed in Go, C++, Java, or Python hosted in the notebook. About MatrixGRPC, the tests consisted of multiplying matrices of integers with dimensions 400x400, 700x700, and 1000x1000. About BenchImageGRPC, the tests consisted of applying the gray filter in the same image in different resolutions (0.3 MP, 4 MP, and 8 MP). When executing the client part, it is possible to choose whether the processing is local (on the smartphone itself) or remote (on the notebook). We conducted the tests 33 times for each combination of factors. For example, we repeated 33 times the experiment in a scenario where the multiplication of two $400x400$ matrices was processed remotely by a Go server process. The number of repetitions was chosen according to (Jain, 1991).

We choose as metrics: 1) total processing time; 2) time spent with the network; and 3) the mobile device's energy consumption; The total processing time is the period between the beginning and the end of the task execution. The task starts when the button generating the request is pressed and ends the moment before the result display. Only one connection was opened for each remote processing performed. The time spent with the network consists only of the period that involves uploading the request and downloading the response. It is clear that this last metric only exists in offloading scenarios since the task's local processing does not use network resources.

[1]https://www.msoon.com/high-voltage-power-monitor

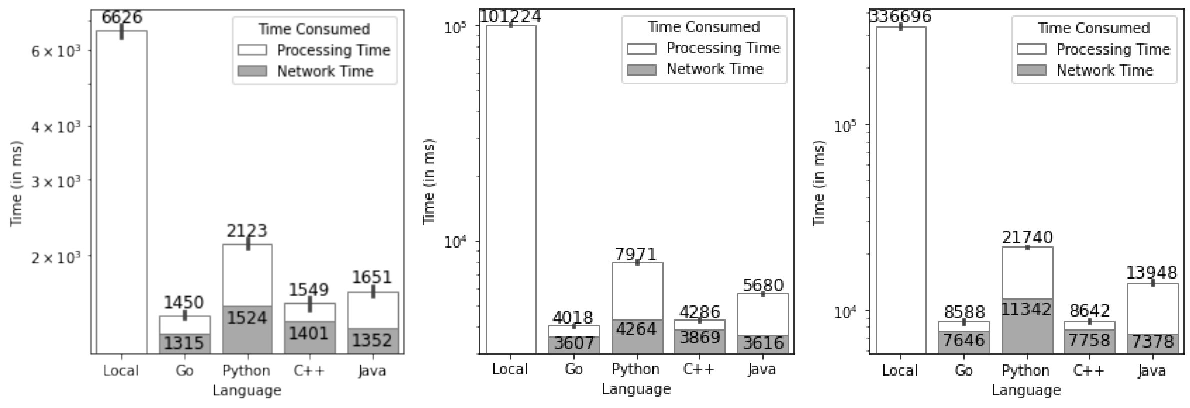
5 RESULTS

This section presents the results of the experiments performed with the devices and applications described in Section 4. The results are displayed in tables and bar charts with a 95% confidence interval.

5.1 MatrixGRPC Application

The results about processing time and time spent with the network are compiled in Figure 2. Note that, for all matrix dimensions, all multi-language solutions were faster than the respective local approach. The Go language presented the best results among all other languages. For instance, offloading to a process implemented with Go reduced the multiplication time of the 1000x1000 matrices around 39 times compared to the local execution. In the same scenario, even the process developed with Python language (the one with the worst performance) reduced around 15 times the local processing time.

In all scenarios, it is possible to notice that network operations consumed most of the 50% time spent on computation offloading, which indicates a strong influence of the network on multi-language offloading performance. For all languages, we believe that these delays caused by the network are due to



(a) 400x400 (b) 700x700 (c) 1000x1000

Figure 2: Total processing time and network time spent during the MatrixGRPC's experiments. Note there is no Network time when the filter is executed locally, and the chart uses a logarithmic scale.

Table 3: Average energy gain of the mobile device during MatrixGRPC's experiments.

Matrix Dimension	**400x400**		**700x700**		**1000x1000**	
Method (Language)	**Energy Cons (in uAh)**	**% Gain**	**Energy Cons (in uAh)**	**% Gain**	**Energy Cons (in uAh)**	**% Gain**
Local (Android)	23908.68	NA	383591.64	NA	1173827.79	NA
Remote (Go)	6578.56	72.49%	19273.02	94.98%	41568.77	96.46%
Remote (C++)	6938,14	70.98%	19980,85	94.79%	42606,12	95.97%
Remote (Java)	7435,55	68.90%	23848,72	93.78%	55925,93	95.24%
Remote (Python)	8347.34	65.09%	28277.73	92.63%	67494.83	94.25%

external interference caused by other devices and/or networks that operate on the 2.4 GHz frequency. This fact indicates that multi-language offloading becomes even more attractive if the network's transmission quality is improved.

Table 3 presents the results related to the mobile device's power consumption during the experiments with the MatrixGRPC application. It also presented the percentage values of each multi-language offloading solution with the local approach's power consumption as a reference (baseline). In general, we observe that all offloading multi-language solutions obtained better performances than the local approach in this metric. We observed that the multi-language solutions reduced between 65% and 72% the energy consumptions of the local approach in the worst case (1MP images). In the best case (8MP images), the improvement was between 94% and 96%. We also highlight the Go language results that showed the most significant reductions in power consumption in all scenarios, as well as the task's total processing time (Figure 2). We believe that this behavior is mainly because the local approach is significantly slower than the others. In this way, the task processing consumes more time, resources, and power from the device to be concluded.

5.2 BenchImageGRPC Application

Figure 3 shows the results about processing time and time spent with the network of the experiments with the BenchImageGRPC application. The results indicate that all multi-language offloading solutions have reduced local processing time by up to 5 times in scenarios with images with 4 MP or 8 MP resolutions. In contrast to MatrixGRPC, for the simplest data (0.3MP images), local processing proved to be more advantageous than any multi-language solutions. Thus, the results discourage the use of offloading for small images. Just like in the MatrixGRPC app, the time spent with the network consumes most of the time dedicated to offloading in all scenarios, especially where the amount of data to be processed is relatively small.

Another point in these results is how libraries, even wrappers from other libraries, can improve offloading performance. As already mentioned, all languages have adopted the same library for manipulating images: OpenCV4. This decision provided greater equality in the performance of languages. The results seen with Python and Java are examples of this performance improvement. Unlike the MatrixGRPC (Figure 2) where such languages (mainly Python) presented the worst results, in BenchImageGRPC (Fig-

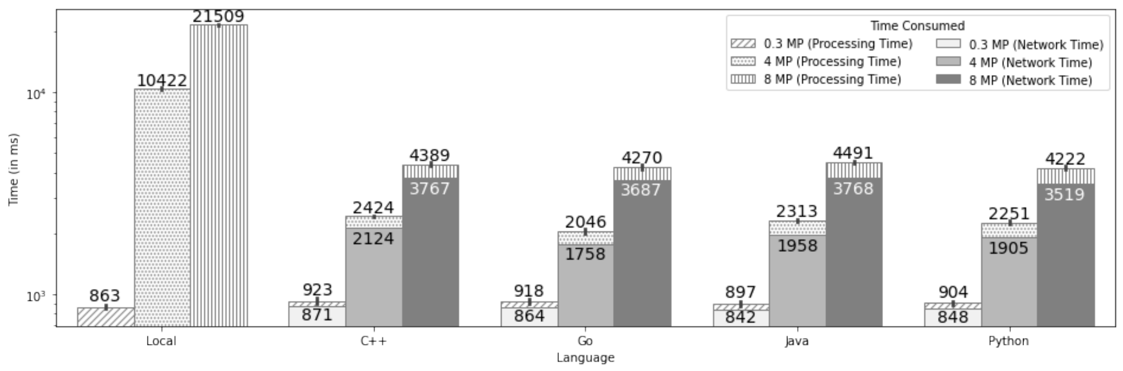


Figure 3: Total processing time and network time spent during the BenchImageGRPC's experiment. Note there is no Network time when the filter is executed locally, and the chart uses a logarithmic scale.

Table 4: Average energy consumption (in percentages) of the mobile device during BenchImageGRPC's experiment.

Image Resolution	**0.3MP**		**4MP**		**8MP**	
Method (Language)	**Energy Cons (in uAh)**	**% Gain**	**Energy Cons (in uAh)**	**% Gain**	**Energy Cons (in uAh)**	**% Gain**
Local (Android)	3796.35	NA	40786.24	NA	84483.09	NA
Remote (Go)	4177.95	110.05%	9401.84	76.95%	18813.86	77.73%
Remote (C++)	4331.52	114.09%	10156.79	75.10%	19847.76	76.51%
Remote (Java)	4599.55	121.15%	10382.93	74.55%	21269.56	74.83%
Remote (Python)	4506.77	118.71%	9794.64	75.90%	18810.57	77.74%

ure 3), their results were very close to other languages and, in some cases, even better (for example, those related to 0.3MP images). By allowing a greater variety of languages on the server-side, the multi-language approach increases the number of libraries available for processing tasks. So, this more generous offer enables even more significant performance improvements than those seen in traditional offloading solutions, where many of them are limited to only libraries and resources supported by mobile devices.

The results of energy consumption obtained in the experiments with the application BenchImageGRPC are presented in Table 4. In general, it is important to highlight the exception that occurred with 0.3MP images. In this scenario, the energy consumption when processing the task on the mobile device was slightly lower than processing it remotely using any of the multi-language solutions. In the worst case, the Java server consumed 21.15% more energy than the local approach for this kind of image. When we compare this result with the one presented in Figure 3, we realize that the local processing time was lower than the time of any of the multi-language approaches. The speed when computing the task justifies this low power consumption on the device. For larger images (4MP and 8MP), all multi-language offloading solutions showed significantly better results than the local approach. These solutions saved between 75% and 77% energy of the device.

6 CONCLUSION AND FUTURE WORKS

This work evaluated the performance of multi-language techniques applied in computation offloading in MCC scenarios. We have conducted experiments with four server processes (developed in C++, Go, Java and Python) that receive offloaded tasks via gRPC, from two Android applications: multiplication of quadratic integer matrices (MatrixGRPC) and an application that applies different filters in images (BenchImageGRPC). In these experiments, we have evaluated the overall processing time, the device's energy consumption, and the elapsed network time.

The results obtained were promising for the proposed technique. Regarding the task processing time metric, the results showed a reduction in the local processing time of up to 5 times in BenchImageGRPC and up to 39 times in MatrixGRPC. The only exception was for the scenario that involved applying a filter to 0.3 MP images, where the local processing was slightly better than remotely. We believe that was a consequence of the small size of the images used, making it more advantageous to process the task locally. Considering MatrixGRPC, solutions with servers developed with compiled languages obtained better results, while in BenchImageGRPC, the languages' performance was very close. We believe that the choice of the adopted libraries justifies this behavior. Regarding the metrics related to the network, we observed that the network has a strong influence on multi-language offloading performance. In extreme cases, 97% of the offloading time was dedicated only to network operations.

Regarding the results of the mobile device's energy consumption, we observed that, in general, energy consumption is proportional to the task processing time. All multi-language solutions showed lower energy consumption than processing tasks locally, except in the scenario with 0.3 MP images in BenchImageGRPC, where remote processing performed worse than local processing. In general, compiled languages obtained more significant gains with MatrixGRPC (between 70% and 96%) and similar gains with BenchImageGRPC (close to 76%).

As future work, we plan to expand the current tests to new applications, particularly those that use machine learning techniques (such as facial detection apps). We also plan to conduct multi-language offloading experiments in D2D (Device-to-Device) scenarios by performing, through gRPC, computation offloading between Android and iOS smartphones. We consider comparing the performance of gRPC with other consolidated offloading frameworks from the literature and other multi-language frameworks (for example, Apache Thrift and Cap'n Proto). Finally, we also intend to explore the native support that gRPC offers to handle data streaming and evaluate its performance in applications of this type.

ACKNOWLEDGEMENTS

The authors would like to thank The Ceará State Foundation for the Support of Scientific and Technological Development (FUNCAP) for the financial support (6945087/2019) and the National Institute of Science and Technology for Software Engineering (INES).

REFERENCES

Araújo, M., Maia, M. E. F., Rego, P. A. L., and De Souza, J. N. (2020). Performance analysis of computational offloading on embedded platforms using the gRPC framework. In *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, pages 1–8.

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

Cisco (2020). Cisco annual internet report (2018–2023) white paper. Available in: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Access in: 11-02-2020.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition.

De, D. (2016). *Mobile Cloud Computing: Architectures, Algorithms and Applications*. CRC Press, 1st edition.

Dolińska, I., Jakubowski, M., and Masiukiewicz, A. (2017). Interference comparison in wi-fi 2.4 ghz and 5 ghz bands. In *2017 International Conference on Information and Digital Technologies (IDT)*, pages 106–112.

Fernando, N., Loke, S., and Rahayu, W. (2013). Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29:84–106.

Georgiou, S. and Spinellis, D. (2019). Energy-Delay Investigation of Remote Inter-Process Communication Technologies. *Journal of Systems and Software*.

Indrasiri, K. and Kuruppu, D. (2020). *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 1st edition.

Jain, R. (1991). *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley professional computing. Wiley.

Kumar, K., Liu, J., Lu, Y.-H., and Bhargava, B. (2013). A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18.

Mastrangelo, C. (2018). Visualizing grpc language stacks. Available in: https://grpc.io/blog/grpc-stacks/. Access in: 11-02-2021.

O'Dea, S. (2021). Number of smartphones sold to end users worldwide from 2007 to 2021. Available in: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/. Access in: 11-02-2021.

Rego, P. A., Costa, P. B., Coutinho, E. F., Rocha, L. S., Trinta, F. A., and Souza, J. N. d. (2017). Performing computation offloading on multiple platforms. *Computer Communications*, 105(C):1–13.

Sebesta, R. W. (2012). *Concepts of Programming Languages*. Pearson, 10th edition.

Silva, F. A., Zaicaner, G., Quesado, E., Dornelas, M., Silva, B., and Maciel, P. (2016). Benchmark applications used in mobile cloud computing research: a systematic mapping study. *The Journal of Supercomputing*, 72(4):1431–1452.