Performance analysis of computational offloading on embedded platforms using the gRPC framework

Mateus Araújo, Marcio E F Maia, Paulo A. L. Rego, Jose N de Souza

Performance analysis of computational offloading on embedded platforms using the gRPC framework

Mateus S. Araújo[1], Marcio E. F. Maia[1], Paulo A. L. Rego[2], and José N. de Souza[2]

[1] Campus Quixadá – Universidade Federal do Ceará
Quixadá, Ceará, Brasil
mateuseng_ec@alu.ufc.br, marcioefmaia@ufc.br
[2] Departamento de Computação – Universidade Federal do Ceará
Fortaleza, Ceará, Brasil
pauloalr@ufc.br, neuman@ufc.br

Abstract

Embedded systems are becoming increasingly accessible to the Internet allowing the creation of new services and applications. Such systems need to communicate in a structured form in a way that uses standardized technologies for better results. Mobile systems also have a number of limited features like battery life, internal storage and processing performance. Such restrictions can be mitigated by the use of computational offloading since algorithms or applications can be executed in the cloud or other networked devices. This article is intended for the analysis of emerging technologies in cross-process communication between Linux and Android-based multiplatforms using the gRPC framework. Applications have been developed in various object-oriented programming languages for performing remote procedure calls between a single-board computer and a personal-use smartphone for processing higher order arrays and applying filters to images. Then, a series of analyzes were performed on the transferred data and the computational offloading performance of the algorithms in each platform.

1 Introduction

The Internet and the advance of hardware platforms is reshaping the mechanisms used to create networked embedded applications, with new opportunities to improve our lives rising on a daily basis. In that direction, forecasts made by hardware manufacturers indicate an order of 56 billions connected devices in 2020 [13]. That is a result of a reduction in the cost to create new embedded solutions, an improvement in the processing capability of these devices and a reduction in energy consumption [5]. Another dimension going under improvement is the communication using wireless and application protocols to permit embedded devices to interact using the Internet. Hence, it is now possible to create a device running a Linux distribution for embedded devices and use most of the open-source communication protocols designed for unix-based operating systems on a resource-constrained embedded device [14]. The evolution of embedded technologies and communication protocols to connect devices using the Internet is known as the Internet of Things (IoT).

The IoT is providing mechanisms for novel solutions to emerge, based on algorithms created using several research domains, such as Artificial Intelligence, Computing Vision, Machine Learning, Image and Natural Language Processing, among others. One common issue in the IoT is the low response time requirements imposed on these algorithms [10]. Response time is usually a function of the input data and the processing time, and it is affected by the processing capabilities of the devices and the communication protocols.

In IoT systems, aiming to achieve the shortest response time possible, developers usually have to decide between processing everything locally in the embedded devices, or send all the collected data to another (more capable) device to process it remotely. However, the time to send all the input data to remote devices may reduce or cancel out the benefits of processing it in a more capable device. In that direction, several work are studying the benefits and challenges of using Computation Offload to reduce the burden on embedded devices and improve the overall execution time [11].

Computational offloading is dealt with in this paper with a performance analysis on embedded platforms using the gRPC framework. Here, two processing-intensive applications were implemented, and the execution time compared against with and without offloading. Moreover, these applications were developed using 5 different programming languages, executed on three different operating systems for embedded devices and run on two different devices. The goal was to analyze their impact on the overall performance on embedded systems.

The remainder of this paper is composed of section 2 with a discussion on RPC for embedded systems. Section 3 presents the related work. Section 4 presents the experiment design to evaluate computational offloading using gRPC. Section 5 highlights the main results from the experiments carried out. Finally, section 6 presents some conclusions and future work.

2 RPC on Embedded systems

Remote Procedure Call (RPC) permits system modules running on different address spaces to interact. Using RPC, one module is capable of invoking procedures from another module, receiving the consequent return of the result data. RPC can be achieved by specifying procedures able to be called remotely and the individual input and output parameters in each procedure. Thus, when different parts of an application need to communicate, they do so by serializing data and sending them throughout the network. One common feature used by RPC implementations to enforce interoperability is the use of Data Description Languages (DDL), which is a language to specifiy data structures, to permit interaction between remote modules, regardless of the platform and language used in the procedure implementation.

A data description language commonly used by cloud applications today is Protocol Buffer. According to [7], Protocol Buffer is a neutral feature that enables data structuring for multi-level communication across the network. Protocol Buffer is an alternative to JSON (Javascript Object Notation), a widely used notation to represent data in web applications[16]. Although JSON has advantages such as good human-readability, it has several limitations as well. Data transferred using JSON is not encoded by browsers and is transferred as clear text, which presents important performance and energy consumption restrictions in embedded systems. Alternatively, Protocol Buffer is designed to have smaller memory footprint (as opposed to JSON), making it more efficient to exchange messages across cross-platform and embedded applications [7].

Using Protocol Buffers as a DDL to enforce interoperability, gRPC[1] is a general purpose framework to permit RPC interactions based on HTTP/2 as transport to traverse proxies and firewalls. There are a number of advantages of using the GRPC framework on various devices such as desktops and mobile devices. Among the highlighted advantages, the low-memory footprint can be cited. Applications using Protocol Buffer serialize the data in a structure called proto file, regardless of the language used. Additionally, the ease with which applications can be developed just by defining the proto file interface stands out. Once data structuring is

[1] https://grpc.io

defined, it is transparent to the developer how communication between different platforms will take place, since GRPC and Protocol Buffer together handle data parsing between the most diverse types of object oriented languages such as Java, C ++, Python, among others.

Embedded systems are traditionally defined as resource-limited platforms with specific processing purposes. Most of these systems require light programming languages for optimal performance. In that direction, Protocol Buffer is an interesting solution, since it is optimized for low-memory footprint and reduced communication latency [15]. In addition, Protocol Buffer makes interoperability between distributed applications much simpler [7], depending only on a common file for defining the data to be transferred.

To improve the overall execution performance in terms of processing time, memory and energy consumption, developers usually have to decide between processing everything locally or to send all the collected data to another device to process it remotely. However, the time to send all code and data may reduce or cancel out the benefits of processing it in a more capable device. In summary, computation offloading is the set of techniques to permit algorithms to be offloaded to remote devices, trying to improve performance metrics in resource-constrained devices.

3 Related Work

The authors in [4] present performance and efficiency information for an embedded system using protocols for structured information exchange based on remote procedure calls. Such a feature, according to the paper, is an interoperable mechanism to exchange information between networked embedded platforms. They compare the use of SOAP and XML-RPC applications through data processing in embedded systems and mobile devices. For this, an engine monitoring system was developed through a Controller Area Network (CAN) in which the server was developed in C/C ++ language using XML-RPC. As a result, a Windows-based computer operates as a remote machine that receives all information through data serialization via the DS80C400 microcontroller, which in turn has CAN support and 1-wire communication by transmitting data over the network between vehicle control modules. The system proved to be efficient when RPC was used instead of SOAP for communication and information exchange between the computer and the embedded integrated chip.

RPC has proven to be an useful protocol for embedded systems. However, there is also the challenge of monitoring and controlling applications using wireless networks. The Marionette system implemented in [15] demonstrates the use of RPC to facilitate the development of applications with data being transferred between multiple devices wirelessly. It demonstrates how RPC can be used in applications needing a better distribution of their services and consequently a more efficient serialized data structure. The proposed application demonstrates the use of an RPC-based framework to permit the communication between a desktop computer and mobile devices through programming languages, such as Java, Python or Bash Script. The networked computer allows sensor data acquisition as well as sensor configuration through remote methods found on the server. In addition, the system is also capable of obtaining sensory network information through client/server applications implemented in other specific languages such as nesC for network programming using embedded systems and operating systems specific to such applications such as TinyOS.

The Any Run Computing (ARC)[6] architecture proposes a dynamic offloading model, demonstrating energy efficiency and latency reduction using resources available on cloud devices. According to the authors, computational offloading is considered advantageous compared to local execution, as internal resources such as CPU and memory usage can be used efficiently

while data is processed and organized in external services. The ARC architecture is implemented in Java with SCAMPI framework support and works both on Dalvik Android-based virtual machines and on Oracle's Hotspot virtual machines, used on desktop computers. Their performance evaluation showed how the ARC architecture was able to reduce offloading time by transferring applications that used high computational resources to the cloud. A more detailed analysis is required to understand the offloading impact in other parameters such as battery or network.

The authors on [9] demonstrate several results executing specific algorithms and techniques on mobile devices. Depending on the programming language used and the hardware architecture of the device, the offloading time may vary sharply. A Java application was developed to simulate an agent for resource access on a remote Linux machine. As more resources were requested, the slower the offloading computation time was perceived. However, if the agent was implemented in C/C++ using native libraries from the embedded device, the more efficient the overall offloading process was carried out.

The authors in [2] conducted experiments to evaluate the energy consumption of Android devices when using different communication protocols and architectural styles, such as REST, SOAP, Socket and gRPC, and executing classical sorting algorithms of different complexities and different types and input sizes. Their results show that local execution is more economic with less complex algorithms and small input data. When it comes to remote execution, REST is the most economic choice followed by Socket, they show that computation offloading can save up to 10 time as much energy when compared to local execution for some executions configurations. Surprisingly, gRPC did no achieved good results in their experiments. It is not easy to explain such a result, but we guess it might be caused by the type of application used in the experiment: sorting algorithms, in which data serialization is quite simple, decreasing the benefits gained from using gRPC.

Several studies [2, 4, 6, 9, 15] have been using remote invocation techniques to perform computation offloading between various systems to improve application's performance and to reduce device's energy consumption. However, to the best of our knowledge, few of them consider different communication protocols and none of them have evaluated the influence of different programming languages and embedded systems in this context.

4 Evaluating computational offloading using gRPC

Computational offloading permits application developers to migrate the execution of processing- and memory-intensive algorithms from embedded devices to more capable devices. The goal is to reduce execution time or to reduce energy consumption. Moreover, the decision to migrate it is based on the type of algorithm, network latency and size of the input parameters. In that direction, gRPC is an useful underlying technology to implement the migration of code and data.

We have implemented applications in various object-oriented programming languages that leverage the gRPC framework for performing computation offloading. Our goal is to assess the offloading performance on heterogeneous environments, using different programming languages as well as the following Linux and Android-based devices:

BeagleBone: a development kit based on 32-bit ARM processor, capable of performing over 3 million Dhrystone and floating-point vector arithmetic operations per second [3]. It is widely used and has support for various Linux distributions like Debian, Angstron and Ubuntu and can also run Android for various applications. In addition, BeagleBone allows

the inclusion of image processing libraries such as OpenCV and OpenNI for object recognition in automation and robotics projects. Its main specification is a Cortex-A8 ARM AM335x processor with 1GHz clock and 512MB of DDR3 RAM. Besides, BeagleBone contains 4GB internal storage with 3D graphics accelerator and two 32-bit microcontrollers dedicated for real-time processing.

Zenfone 5: an Android-based smartphone running an Intel 32 bits processor, different from most of ARM-based smartphones running Qualcomm's Snapdragon processors. It uses an Intel Atom Z2560 dual-core processor, 2GB RAM and 8GB storage [8].

Two case studies were used to measure the overall execution time with and without computational offloading. The first case study uses an application to perform multiplication of 1000x1000 matrices. The application client was implemented in Kotlin while its server version running the same algorithm was implemented in C++, Python, Java, Go, and Ruby programming languages.

In the second case study, an image processing application transfers images between a Kotlin client and Python and C++ servers in order to apply several filters using the OpenCV library. The client serializes and transfers one image to the server, which applies Cartoon, Bilateral and Gray filters [1] and sends back the three processed images.

Depending on the type of the applied filter, the final image size may vary as processing methods and parameters vary between filtering types and languages used. Figure 1 exemplifies the process of offloading and processing an image on the BeagleBone platform using the OpenCV library. The image was sent to the server, all three filters were applied and the resulting images were sent back to the client.

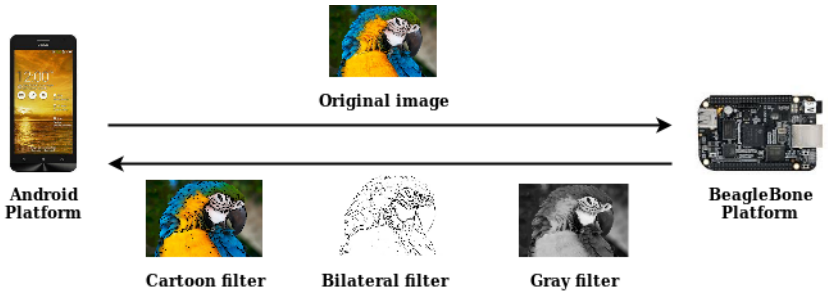

Figure 1: Overview of the image processing application workflow

The implementation of both case studies used gRPC and Protocol Buffers (Protobuf) as the underlying structure to serialize/deserialize data before sending it through the network. Using the Interface Description Language (IDL) present in Protocol Buffer, we have defined the messages to be exchanged during RPC and the Protobuf compiler generated predefined structures to each of the programming languages used. Table 1 presents the experiment design and details such as languages and operating systems used to implement the algorithms.

It is important to highlight that a Debian Linux and two Android distributions (KitKat 4.0 and Vanilla) were used to execute the applications on the BeagleBone. Android Vanilla is an OS version intended for specific use on Embedded Systems, unlike KitKat, which has features indented for smartphones [12]. For the smartphone, the application was executed on Android 5.0 (Lollipop).

Factors	server device operating system, server application programming language, and application.
Parameters	server device operating system (Debian, Android 5.0/KitKat and Vanilla), server application programming language (C++, Python, Java, Kotlin, Go, and Ruby); application (matrix multiplication and image processing).
Design	The experiment was executed 100 times for the matrix multiplication algorithm using 1000x1000 matrices and 50 times for the image processing algorithm using a 216 kB image.
Response	Offloading time for the aforementioned algorithms running on each platform using gRPC (seconds).

Table 1: Experiment Design

Figure 2: BeagleBone Black running Android OS during the performance evaluation.

5 Experiment results

This section presents the results of the experiments using the two case studies (matrix multiplication and image processing). Table 2 shows the mean and standard deviation of the method execution time for the matrix multiplication application (case study one). We can see that the best result was achieved using the C++ version of the application running on BeagleBone with the Debian Linux operating system. The second-best result was achieved with the Python version of the application, while the execution times are approximately the same for Java, Go and Ruby. In fact, we can see a difference of approximately 44% between the best and worst results, which shows the impact of platform choice on offloading performance. Another interesting result is that the Android Vanilla operating system presented a better performance than the Android Kitkat, which can be explained by the fact that Vanilla is an optimized version of Android for embedded devices [12].

The fact that different operating systems and languages present distinct characteristics, such as C and C++ running on top of the OS, while others run on top on a virtual machine, is another parameter to be considered, since it may present important impact in the execution performance.

As a baseline, the Local column of Table 2 shows the average execution time in seconds and the standard deviation for matrix multiplication algorithm without offloading, running on the smartphone. Here, the same pattern is present, with C++ outperforming all other languages. Additionally, we can see the benefits of offloading when we compare the execution time of the same languages running locally and remotely on a device with better processing capabilities.

Table 3 shows the mean and standard deviation of the method execution time for the image processing application (case study two). Again, as expected, the combination C++/Linux Debian outperforms the other combination, mainly because of the optimization of the Linux OS

	Local (baseline)		Linux Debian		Android KitKat		Android Vanilla	
Languages	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**
C++	16.07	0.47	10.36	0.57	13.62	0.64	12.74	0.26
Python	18.21	0.55	12.94	1.89	16.72	0.69	14.88	0.60
Java	20.07	0.47	17.41	0.77	18.07	1.09	18.07	0.49
Go	19.97	0.57	17.39	0.78	17.50	0.65	16.67	0.25
Ruby	19.11	0.45	18.19	0.56	18.39	0.47	17.59	0.30

Table 2: Total execution time for the matrix multiplication application with and without offloading

for embedded platforms. The Android platforms presented statistically the same performance.

	Local (baseline)		Linux Debian		Android KitKat		Android Vanilla	
Languages	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**	**Avg.(s)**	**Dev.**
Python	6.89	0.58	7.24	0.97	8.15	0.6058	7.98	0.55
C++	6.81	0.59	5.66	0.79	5.88	0.599	5.89	0.53

Table 3: Total execution time for the image processing application with and without offloading

As a baseline, the Local column of Table 3 shows the average execution time in seconds and the standard deviation for running the image processing application on the smartphone. Comparing with the offloading times presented on Table 3, we can see that the C++ total offloading time still outperformed the local execution, even considering that the image had to be transferred from the smartphone to the Beaglebone. However, considering the Python implementation, we can see that it is worth executing the method locally because the benefits of offloading the image processing do not pay off, and the time to send the image through the network outweighs the offloading benefits.

6 Conclusion and Future Work

Despite using the same embedded device as a server, the choice of platform impacts overall offloading performance, causing total offloading time to vary by up to 44%. This result reveals the importance of choosing the technology stack when using embedded devices with limited resources. It is also important to highlight the advantages obtained by using gRPC as interprocess communication technology, to permit the communication between processes implemented with different programming languages. Hence, the developer can implement their components and services on the platform that suits them or perform better, which is essential in computational offloading scenarios.

The results set forth in this study indicate a wide range of possibilities and research associated with computational offloading in RPC-based embedded multiplatforms. It can be seen from the overall execution time how the programming languages and operating systems used are also relevant factors for a reduction in offloading time in most applications. Associated with this reduction, we also have the increased performance and energy savings that these analyzes can lead to overall mobile devices, as well as the use of tools that enable data transfer across the network using remote procedure calls using the open source GRPC framework.

As future work, we plan to investigate alternatives for performing offloading on embedded devices and evaluate other platforms such as Raspberry Pi and Jetson, investigate other technologies such as REST/JSON, and investigate the energy consumption of these devices.

Acknowledgments

The authors would like to thank The Ceará State Foundation for the Support of Scientific and Technological Development (FUNCAP) for the financial support (grant number 6945087/2019).

References

[1] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[2] C. L. Chamas, D. Cordeiro, and M. M. Eler. 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, Nov 2017.

[3] Gerald Coley. Beaglebone black system reference manual. *Texas Instruments, Dallas*, 2013.

[4] Suru Dissanaike, Pierre Wijkman, and Mitra Wijkman. Utilizing XML-RPC or SOAP on an embedded system. In *24th International Conference on Distributed Computing Systems Workshops (ICDCS 2004 Workshops), 23-24 March 2004, Hachioji, Tokyo, Japan*, pages 438–440, 2004.

[5] Gabriel B dos Santos, Fernando AM Trinta, and Paulo AL Rego. Impactos do offloading de processamento no tempo de execução e consumo energético de dispositivos m óveis. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, volume 36. SBC, 2018.

[6] Alan Ferrari, Silvia Giordano, and Daniele Puccinelli. Reducing your local footprint with anyrun computing. *Computer Communications*, 81:1–11, 2016.

[7] Gurpreet Kaur and Mohammad Muztaba Fuad. An evaluation of protocol buffer. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pages 459–462. IEEE, 2010.

[8] Yoon-young Kim and Min Oh. Cover for mobile phone, May 26 2015. US Patent D730,342.

[9] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[10] Rainer Leupers, Miguel Angel Aguilar, Jeronimo Castrillon, and Weihua Sheng. Software compilation techniques for heterogeneous embedded multi-core systems. In *Handbook of Signal Processing Systems*, pages 1021–1062. Springer, 2019.

[11] Yuchuan Liu, Cong Liu, Xia Zhang, Wei Gao, Liang He, and Yu Gu. A computation offloading framework for soft real-time embedded systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 129–138. IEEE, 2015.

[12] Anand Nayyar and Vikram Puri. A review of beaglebone smart board's-a linux/android powered low cost development platform based on arm technology. In *2015 9th International Conference on Future Generation Communication and Networking (FGCN)*, pages 55–63. IEEE, 2015.

[13] D Pavithra and Ranjith Balakrishnan. Iot based monitoring and control system for home automation. In *2015 global conference on communication technologies (GCCT)*, pages 169–173. IEEE, 2015.

[14] Richard D Snyder. A cross-language remote procedure call framework. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 3822, 2017.

[15] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *2006 5th International Conference on Information Processing in Sensor Networks*, pages 416–423. IEEE, 2006.

[16] Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 222–233. ACM, 2019.