Performance and Energy Evaluation of Restful Web Services in Raspberry Pi

Luiz H. Nunes, Luis H. V. Nakamura, Heitor de F. Vieira,
Rafael M. de O. Libardi, Edvard M. de Oliveira, Lucas J. Adami, Julio C. Estrella
University of São Paulo (USP)
Institute of Mathematics and Computer Science (ICMC), São Carlos-SP, Brazil
Email: {lnunes, nakamura, heitorfv, mira, edvard, ljadami, jcezar}@icmc.usp.br

Abstract—This paper analyzes the execution behavior of web services on devices with limited resources. To conduct the experiments, web services are available for both Axis2 and CXF framework. To determine which framework is better suited for service provision, a performance and energy evaluation between them is presented. In this context, services developed with CXF framework proved to be more suitable for these devices, since its abstract implementation details both services provider and clients applications, also it has shorter times to process large amount of data.

I. INTRODUCTION

The Raspberry Pi is a low-cost computer similar to a motherboard of a personal computer. It consists of a set of microchips similar to those found in mobile phones, so it has the size of a credit card. Furthermore, it is available in two variations: model A and model B. The latter being provided with more features. This device consists of a microprocessor System on Chip (SoC) of ARM11 architecture and it has a graphic processor unit (GPU) with at least 256MB of RAM memory. Moreover, depending on the model, it may contains some I/O ports such as Fast Ethernet, USB and HDMI [8].

The Raspberry Pi lacks operation system by default, it must be installed into a SD memory card. However, this facilitates maintenance and helps system recovery in case of problems.

The original application of the Raspberry Pi was for teaching Computer Science in schools [26]. Though, due to its configuration and price, it also allows users to use it for games, Internet access, multimedia applications and personal projects [12]. Currently, some of these personal projects use the Raspberry Pi to photograph the person who opens the refrigerator, monitor the temperature in a freezer and also take air photos from party balloons [12].

The Raspberry Pi is also used in commercial projects, for example, the German company All For Accounting put it on a black and red box to sell it as a cheap computer for small commercial tasks (inventory management and accounting) [15]. Also, the Raspberry Pi was used to create a computer cluster to perform parallel processing [25].

Although there are numerous projects for Raspberry Pi, the limited resources are still a problem, especially related to the processing capacity and memory. On the one hand this limits the execution of applications that consume a large amount of data and require high computing capacity [10], but on the other hand, it makes the device cheap and saves energy.

Because service-oriented applications use other resources to perform tasks, they can overcome these limitations and improve the performance of embedded devices, including the Raspberry Pi. For example, a project presented in [12] used Raspberry Pi to monitor heart rate and body temperature. However, the use of service-oriented applications in embedded devices still need to be investigated with other types of information, such as larger messages that demand more resources. This study is relevant because services use large amounts of data to create additional information in the message body, which increases the size and processing time of those messages [17]. Another important factor to investigate is the energy consumption to perform those process in embedded devices.

The service-oriented applications are implemented as web services that can be classified into two categories: RESTful and SOAP. REST is the acronym for Representational State Transfer. In RESTful services the resources are identified by an URI (Uniform Resource Identifier) and manipulated using HTTP methods (i.e: GET, PUT, POST, DELETE). Furthermore, SOAP is an acronym for Simple Object Access Protocol and services based on this protocol are described by a Web Services Description Language (WSDL) and manipulated by XML messages. Finally, both RESTful and SOAP services enable the exchange of information through XML messages [24], [30].

SOAP web services uses additional XML data unnecessary for embedded and mobile clients, which consumes more bandwidth and increase the complexity to parse the message. Therefore, REST is more suitable than SOAP for embedded applications because it is based on the Web, which was created to be scalable. Thus, the advantages of RESTful services over SOAP services are simplicity, flexible interfaces and scalability [31]. However, it lacks important features provided by SOAP, such as security and QoS support standards.

In this paper, we conducted a performance and energy consumption evaluation to study the behavior of RESTful web services using the Raspberry Pi. To perform this evaluation, services were developed with the same features using both Axis2 and CXF frameworks developed by the Apache Software Foundation. Besides, it was analyzed the times to marshal/unmarshal messages with different sizes. Tests were made in real devices with normal CPU clock and overclocking configurations.

This paper is organized as follows: Section II presents a literature review of embedded web services performance
evaluations studies and methodologies. Section III highlights the characteristics of hardware, frameworks and technologies used in this study. Section IV describes the methodology and configurations used for the experiments. The results are then discussed in Section V. Finally, the conclusions and future work are presented in Section VI.

II. RELATED WORK

The literature review followed a strict methodology, it was used the following databases: IEEE Xplore\(^1\), ACM Digital Library\(^2\), Web Of Science\(^3\), Scopus\(^4\) and the meta-searcher Google Scholar\(^5\). The research query used was: “((performance AND (evaluation OR analysis) AND (mobile OR embedded) AND ("web services" OR SOAP OR RESTful))”. Relevant research papers and their references were reviewed and categorized into a synthesis matrix regarding two subjects: performance evaluation methodologies for embedded/mobile devices; and previous web services performance evaluation for mobile devices. Then, it was realized that the literature lacks studies comparing performance of RESTful frameworks in embedded devices, which is the focus of this study.

Previous web services performance evaluation methodologies for embedded and mobile environments were categorized into two approaches: real device experiments; and mathematical models with simulation. Real experiments were found in [27], [29], [21] and [17] and is the most used. In this approach, real devices and prototypes are used to measure response variables in a real environment using replications for non-deterministic variables. Although this is the widely used approach, it is more expensive, needs to follow a rigid methodology and requires deep statistical analysis.

Another methodology found in [32] and [18] uses mathematical models and simulation tools to evaluate their proposed architecture. In this methodology there is the need to emulate or create mathematical models that represent the system to evaluate. This approach is cheaper than real experiments and often misrepresent the environment, but it is useful when it is hard or expensive to evaluate using a real device or prototype. The approach used in this study was based on real device experiments using a testbed environment.

It was also researched previous web services performance evaluations for embedded environments. The work of [29] reviewed some web services approaches and concluded that they are suited for mobile devices. Although it was an extensive study, it lacks RESTful web services experiments. The study in [17] evaluated two SOAP frameworks (ksoap and ws4d) for embedded devices. They used a real device and measured the overhead using distinct metrics, such as response time and energy consumption. However, this work lacks the evaluation of REST architectural concepts in embedded devices.

Some studies [27], [18], [11], [9], [19] compared SOAP against RESTful for mobile environments and concluded that RESTful is more suited for these devices because it generates less overhead in data transfers and spend less time to pack/unpack the request and response messages. However these studies performance evaluation are superficial because they lack at least one of the following elements for performance evaluations: confidence intervals (or other deviance estimators), replication and statistical analysis [20]. In [14] it was also discussed the usage of RESTful web services in embedded devices, concluding that they save a lot of resources and energy because they use small messages by removing their headers. But it lacks a performance evaluation and replication tests to support this conclusion.

Thus, this study differs from the other studies in the literature in at least three points. It presents an energy consumption and performance analysis of RESTful applications developed using different frameworks in embedded devices, represented by the Raspberry PI.

III. TOOLS AND TECHNOLOGIES

A. Raspberry PI

Raspberry Pi model B is a computer based on a SoC Broadcom BCM2835, which has a 700 MHz ARM1176JZF-S processor integrated with a VideoCore IV GPU and 512 MB of RAM memory. Besides it has a SD card slot for data storage, although it does not include a non-volatile memory such as a hard drive [13].

Figure 1 shows the architectural model of a Raspberry Pi, which is composed by the following components:

- **Two video outputs**: An RCA and HDMI output, which can not be used simultaneously;
- **Two audio outputs**: TRS (3.5mm) and HDMI audio output, which can not be used simultaneously either;
- **Two USB 2.0 inputs** to communicate with peripherals;
- **One Ethernet 10/100MBps input** to allow the communication between device and network;
- **One micro-USB input** for energy supply. For this, it is essential to use a power supply with 5V and 700mA for appropriate operation of the device.

The Raspberry Pi can use a customized version of Debian GNU/Linux called Raspbian (http://www.raspbian.org/). This operating system contains a set of basic programs and utilities that make the Raspberry Pi usable. It also includes over 35,000

---

software packages pre-compiled and ready for installation, and it has an active community focused on improving stability and performance for each new version released [6].

B. Arduino

Arduino is an open-source electronics prototyping platform used by developers and engineers to create personal and commercial projects. A major use of the Arduino is to control electronics like lights, motors, and other components. Arduino can also sense environments by receiving input information from sensors, hardware extensions (known as Shields) can be attached to provide more features such as Ethernet and WiFi access [4]. There are several types of Arduinos boards (products) like Arduino Uno, Arduino Due, Arduino Robot, etc. The board used for the experiments in this paper was an Arduino Nano 3.0, based on the ATmega328 microcontroller running at 16MHz with 1Kb SRAM, 32KB Flash memory and Mini-B USB interface (Figure 2).

Fig. 2: Arduino Nano 3.0

C. RESTful Web Services

REST provides a set of standards that, when applied properly, emphasizes the scalability of component interactions, generalization of interfaces and encapsulate legacy systems [16]. Besides, clients and service providers exchange resources representations through a standard interface using free state protocols for communication, typically HTTP. These resources are identified by an universal resource identifier (URI) and manipulated through four HTTP methods: GET, PUT, POST and DELETE. WSDL 2.0 and Web Application Description Language (WADL) files can be used to describe RESTFul services, although there is no need, once they are usually described by their own URLs. The main advantages of RESTful services according to [24], [28] are:

- **Unique addresses** - resources are accessed through an URI, eliminating the use of a separate resource to discover services;
- **Free state** - client requests has all data required to perform an operation in service provider. In addition they are not related to previous requests;
- **Several supports for data access** - information can be accessed through multiple formats of files such as plain text, JavaScript Object Notation (JSON) or XML.

Figure 3 shows the common flow of a RESTFul application. The client application makes a request to a server through an HTTP method, where a message (XML, JSON, plain text, etc.) is directed to a specific URI. The GET and POST methods are most used by these requests.

When this message arrives at the server side, it starts its process seeking information of which service it should run, which parameters will be used and it checks errors in the message. Therefore, when this message is processed, the desired service runs, then the response message is produced and forwarded to the client application by HTTP methods.

According to [30], choosing the REST protocol removes the need to make a number of architectural decisions related to SOAP protocol layers, introducing a simple and lightweight alternative to services development based on a set of operations already defined in the HTTP protocol. However, for advanced functionality, such as security, it is complex to extend RESTful web services to support them and the use of SOAP protocol is more appropriate.

D. Axis2 Framework

Apache Axis2 is a project implemented in Java language that facilitates the implementation of web services for both client applications and service providers. Besides, it offers a completely object-oriented approach and it is built upon a modular architecture. Its core processing is done only to process SOAP messages and every message that arrives to the system is transformed into a SOAP message before handled by Axis2 [22]. The messages used in this framework are built from the Apache Axiom library, which provides a set of XML information for the creation and organization of objects into a tree [7].

Sending and receiving messages is one of the key tasks of web services. The Axis2 architecture provides two pipes (or flows) to perform these two operations. The flow to receive a message is known as Inflow, while the flow that sends the message is the Outflow. Therefore, the complex message exchange (MEPs) are performed through the combination of these two flows [23].

The flow concept consists of a series of stages, in which each stage is described as a set of handlers. These handlers process parts of the message and provide functions to ensure quality of service. If the message goes through this flow without any problem, it is sent to a message receiver to be handled by the application. Otherwise, if it is needed to return any data to the application that made the request, the Outflow is initialized and the message is sent. Once the client application receives the message, the Inflow will start to process the returned data.

Figure 4, illustrates the message flow described in Axis2, where InFlow, Outflow and execution of the requested service are represented.
Phase N
Service Model
ends

be summarized as [3]:

as presented in Figure 5. The function of each component can
allow the reuse of existing databinds, facilitating integration
with other specifications, for example JAXB [2].

E. CXF Framework

Apache CXF is an open source framework that provides
easy development of web services in Java, for SOAP protocol
and RESTful architectural concepts. This is done through
standard-based programming model for the development of
web services. Several patterns are provided by CXF to develop
web services, including: Java API for XML Web Services
(JAX-WS) and Java API for XML Binding (JAXB) to the
SOAP protocol and Java API for RESTful Services (JAX-RS)
to RESTful architectural concepts, which is the aim of this
paper.

JAX-RS provides the semantics for the creation of RESTful
web services and abstracts implementation details of their
clients. These clients can exchange data in different formats
such as JSON and XML. For the XML format, CXF JAXB allows
the reuse of existing databinds, facilitating integration
with other specifications, for example JAXB [2].

Apache CXF Services Framework

Fig. 4: Message flow in Axis2 [22].

Fig. 5: Architecture Layers of the Apache CXF [3].

The CXF architecture composed by layers and components
as presented in Figure 5. The function of each component can
be summarized as [3]:

- **Bus**: It is considered the backbone of CXF, which the
default implementation is based on Spring Framework. Its
target is to provide shared resources (extensions register,
interceptors and properties) during CXF execution;
- **Front-End**: Provides a programming model that makes
available front-ends APIs (JAX-WS, JAX-RS, Simple and Javascript)
to developers. Furthermore, the front-ends are capable of providing functionality to services and clients
using interceptors;
- **Messaging and Interceptors**: This component provides
a generic low-level message layer and pipelines or Inter-
ceptors upon which most of the functionality of CXF are
built;
- **Service Model**: provides a service representation within
CXF. Besides, it adopts a WSDL service model with
their operations, connections, terminals and layout. Others
information like data-bindings, interceptors and service
properties are also available in this module;
- **Protocol Bindings**: allows the use and interpretation
of various types of binding protocols (SOAP 1.1, SOAP
1.2, REST/HTTP, pure XML and CORBA). Thus, it
makes possible to exchange the message according to the
specified protocol;
- **Pluggable Data Bindings**: let other Data Binding con-
nect to CXF;
- **Transports**: Provides a transport abstraction layer that
abstracts the specific binding process and front-end layer
details from the developer.

Thus, although complexity of the components and various
APIs usages, Apache CXF framework is a powerful tool rich
in features and capabilities ensuring agile and practical develop-
ment of web services (SOAP and RESTful). The messages
flow in CXF is similar to the process discussed previously
for Axis2, in which the messages are handled in several
phases. However, in Apache CXF, the messages are handled by
interceptors that perform a particular functionality. They can be
added to an interceptor chain, which are grouped and arranged
in stages. Thus, the interceptor chain manages the resources
and information from others CXF components (Front-End,
Protocol Bindings, Transports) to handle the message and
establish communication among client and service [1].

IV. PERFORMANCE EVALUATION

The methodology used for the performance and energy
evaluation is explained in this section. First of all, it is impor-
tant to understand how the experiments were conducted. Thus,
Figure 6 illustrates the messages flows among the applications
used in this paper.

Fig. 6: Messages flow in the experiments.

The client device starts the application by generating a
sequence of random numbers and then performs the serial-
ization of this sequence in an XML message. Once built, this
message can be sent to a web service provided by the server
device. When the message arrives, the server first deserializes
the message, and then process its content sorting the numbers
that was received from the client, so a new message with the
ordered numbers is created, serialized in XML format and sent
back to the client device. Finally, the client deserializes the
response from the server and ends its execution.

Energy consumption was monitored using the Arduinos
Nanos. To obtain the electric current a 1Ω (Ohm) resistor
was connected in series with the Raspberry Pi (Figure 7). By
measuring the voltage drop in this resistor is possible to infer
the electric current of the circuit.

The Arduino was responsible to collect data like time,
voltage and current. These data are collected 170 times per
second from both client and server devices through a shield
Joule (J) is a unit commonly used to measure mechanical energy (work). It is also used to measure thermal energy (heat). In the International System of Units (SI), all work or energy are measured in joules. The energy consumption, in Joules, is calculated according with Formula 1, where \( V(t) \) is voltage, \( I(t) \) is electrical current collected in the time \( t \).

\[
\text{Joules} = \int_{0}^{x} V(t) I(t) \, dt. \tag{1}
\]

A. Experiment Environment

The experiments environment configurations are described in Table I. Two Raspberry PI devices model B were used with the settings modified for each experiment. The first configuration uses the standard CPU clock (700Mhz) and the second one uses an overclock configuration (950MHz). Moreover, the maximum amount of memory configured in the JVM (Java Virtual Machine) was set to 128Mb and the rest was used by the system.

<table>
<thead>
<tr>
<th>Raspberry Pi (model B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Hardware Rev.</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>SD Card</td>
</tr>
<tr>
<td>Operational System</td>
</tr>
<tr>
<td>Others</td>
</tr>
</tbody>
</table>

B. Experiment Design

After setting up the environment, it is necessary to plan the performance evaluation experiments to extract relevant information about the system.

The experiments were designed to gather as much information as possible to compare the performance difference between the Devices CPU Frequency Configurations (700Mhz and 950Mhz) using different development Frameworks for web services (Axis2 and CXF). It was designed eight experiments involving combinations of factors (Device CPU Frequency, Framework and Message Size) with different configurations or levels, as shown in Table II.

<table>
<thead>
<tr>
<th>Exp</th>
<th>Device CPU Frequency</th>
<th>Framework</th>
<th>Message Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>700Mhz</td>
<td>Axis2</td>
<td>100Kb</td>
</tr>
<tr>
<td>2</td>
<td>700Mhz</td>
<td>Axis2</td>
<td>500Kb</td>
</tr>
<tr>
<td>3</td>
<td>700Mhz</td>
<td>CXF</td>
<td>100Kb</td>
</tr>
<tr>
<td>4</td>
<td>700Mhz</td>
<td>CXF</td>
<td>500Kb</td>
</tr>
<tr>
<td>5</td>
<td>950Mhz</td>
<td>Axis2</td>
<td>100Kb</td>
</tr>
<tr>
<td>6</td>
<td>950Mhz</td>
<td>Axis2</td>
<td>500Kb</td>
</tr>
<tr>
<td>7</td>
<td>950Mhz</td>
<td>CXF</td>
<td>100Kb</td>
</tr>
<tr>
<td>8</td>
<td>950Mhz</td>
<td>CXF</td>
<td>500Kb</td>
</tr>
</tbody>
</table>

The web services developed and their respective frameworks were instrumented to collect the time of serialization and deserialization in both client and service. The total time of both were also recorded. Each experiment of Table II was repeated 50 times in order to calculate the average time and guarantee a statistically correct result. Besides, the standard deviation and confidence intervals (assuming a rate of 95% of confidence level) were also calculated for each of the average times collected. All results of the experiments and some statistical information are presented in the next section.

V. RESULTS

This section presents the results of experiments using bar graphs. Subsections divide these results by server and client devices, where it is possible to observe the behavior of both frameworks considering different sizes of messages. In the last subsection it is made a general analysis of the results.

A. Server Results

1) Server Serialization: Figure 8 shows the time in milliseconds for the message serialization on the device server, where both Axis2 and CXF frameworks had an increased time for serializing 500Kb messages when compared to 100Kb messages.
This behavior occurs because both frameworks use a "flow" of methods to process the data and perform the message serialization, making its time proportional to the amount of serialized data. Axis2 framework was faster than CXF to process both messages. This difference occurs because during the message deserialization all information on how serialize that message is obtained. On the other hand, JAX-RS library is less optimized in relation to Axis2 because it uses methods of generic classes to serialize the message.

Figure 9 shows energy consumption during the server message serialization. It is possible to notice that energy results reflect the same behavior of time results, because the longer the processing time is greater will be the energy consumed. An interesting observation was noticed when the Raspberry Pi is working with 950Mhz (Overclock), it consumes fewer energy than when working with 700Mhz. It can be explained because when it is using 950Mhz the processing time is shorter. Thus, the total consumption is lower although it consumes more Joules per unit time, the amount of this time is shorter. More explanation of this behavior is discussed in section V-C.

**2) Server Deserialization:** The graph in Figure 10, presents the deserialization message results from the server device.

**Fig. 9: Server Serialization Energy Consumption.**

It is noted that due to the use of libraries (JAX-RS), the CXF framework had deserialization times lower in messages of 100Kb and 500Kb. On the other hand, the framework Axis2 showed larger times to deserialize messages of 100Kb and 500Kb. This behavior occurs because before starting the deserialization, the Axis2 libraries needs to read a WSDL file, which contains the format of input/output messages and services descriptions.

**Fig. 10: Server Deserialization Time.**

It is possible to notice that the CXF framework requires less time than the Axis2 framework to execute services with 500Kb messages. This reduction is explained by the use of JAX-RS library, which handles with real RESTful messages, whereas Axis2 has an overhead to handle with SOAP messages during service execution. CXF also was faster to messages with 100Kb when the server device was configured to 950Mhz.

**Fig. 11: Server Deserialization Energy Consumption.**

Figure 11 shows the energy consumption during the server message deserialization. Again, the energy results were proportional to the time results. Unlike the serialization, the deserialization results shows lower energy consumption when using CXF.

**3) Server Total:** The graph in Figure 12, shows total time in seconds to execute the service on the server device. As expected, both frameworks had increased service execution time with the increase the size of message, since the processing time is directly proportional to the amount of data.

It is possible to notice that the CXF framework requires less time than the Axis2 framework to execute services with 500Kb messages. This reduction is explained by the use of JAX-RS library, which handles with real RESTful messages, whereas Axis2 has an overhead to handle with SOAP messages during service execution. CXF also was faster to messages with 100Kb when the server device was configured to 950Mhz.

**Fig. 12: Server Total Execution Time.**

Figure 13 shows the energy consumption during the server execution (Serialization + Service Processing + Deserialization). In all results the CXF saves more energy than Axis2, except in 100Kb messages and 700Mhz of CPU clock. Thus, in general CXF framework presents better performance and energy results than Axis2 for those experiments executed on the server side.
B. Client Results

1) Client Serialization: The graph in Figure 14 presents the message serialization time in the client device where the Axis2 framework showed a large difference in time while serializing 100Kb and 500Kb messages. This difference is due to the serialization step implemented by the developer in Axis2 framework, which two points should be highlighted.

Firstly, in Axis2 client a ‘fake’ message serialization is done directly by the developer through AXIOM library, where no pre-processing of data is accomplished, constructing the message to be sent directly. When this messages comes to the transport level, it is converted into a SOAP message before it is sent to the server provider. The second point is that Axis2 uniform times showed in Figure 14 occurs because the message only has one field for the payload, which is already in the client device memory. This is the procedure adopted by the developer to send REST messages using the Axis2 framework.

Figure 15 shows the energy results obtained from the client serialization experiments. Once the client serialization time in Axis2 was shorter than CXF, so Axis2 had lower energy consumption. Furthermore, it is possible to notice that the CXF energy results with 500Kb messages are statistically equals using 700MHz and 950Mhz.

2) Client Deserialization: The Figure 16 presents the results of the deserialization time in the client device. In these results, the CXF framework has shorter times to deserialize 500Kb messages. This difference is due to the deserialization performed through CXF framework libraries that are optimized to deserialize the messages efficiently.

The Axis2 framework was more efficient to deserialize 100Kb than 500Kb messages, once they showed a considerable increase in the time of deserialization. This increase is due to Axis2 deserialization be executed sequentially by the Axiom libraries, where the information is extracted from the object tree received in the message. Thus, the larger is the size of the information contained in the message, the longer will be the time to extract the information required by their deserialization.

Therefore, unlike the serialization time, the deserialization time in Axis2 framework was faster to 100Kb messages,
once this deserialization is implemented sequentially by the developer. Otherwise, the CXF framework was more efficient to deserialize messages 500Kb, once its deserialization is performed through the JAX-RS library, which handles messages with generics classes before starting the extraction of information, and consequently improving the deserialization of larger messages.

Figure 17 presents the client energy consumption results in the deserialization step. When considering 500Kb messages this process is faster with CXF and consequently the average energy consumption is lower. But, this deserialization process for 100Kb messages saves energy when using Axis2 framework.

3) Client Total: The Figure 18, shows the total time for the client application execution. This time includes the client serialization and deserialization times, the network traffic, and the service execution time on the server side (Server Total Time). As expected, both frameworks have increased the client execution time according to the increasing size of the messages. This occurs because the service execution time is proportional to the amount of data to be processed and this time has great influence on the final client time.

Fig. 18: Client Total Execution Time.

Its also possible to notice that Axis2 framework had a better performance for 100Kb messages while CXF framework had a better performance for 500Kb messages. JAX-RS libraries uses methods of generics classes to handle with RESTful messages which has better performance for larger messages. In the other hand, Axis2 framework handles REST messages as SOAP messages in its core that causes an overhead proportional to the quantity of converted data.

In Figure 19, the energy consumption results are presented. They represent the total energy consumption in the client side including the time to create the message, serialize it, sending it through the network, waiting for a server response, deserialize the message, and having it as a Java object. In resume, the results show that CXF framework saves energy with 500Kb messages while Axis2 saves energy with 100Kb messages. Moreover, the overclocking configuration had influence in the time results and consequently in the energy consumption. In a controlled environmient (air conditioning at 23°C), we reached better results using 950Mhz for both performance and energy consumption.

C. Results Analysis

A general analysis of the results that were obtained from the client and servers devices indicate that Axis2 framework was more effective for messages with 100Kb while the CXF framework is most suitable for 500Kb. However, the differences in the execution time among the frameworks is shorter.

Table III shows the percentages occupied by the serialization, deserialization and execution time in the service on the client device. It can be seen that for both frequencies the serialization and deserialization times practically have no influence in total time. It is also noticeable that the average time of client applications is primarily influenced by the service time followed by other times that include the manipulation of data and network times.

TABLE III: Average Client Times (%)

<table>
<thead>
<tr>
<th>Client Times</th>
<th>700MHz</th>
<th>950MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Time(%)</td>
<td>0.002</td>
<td>0.001</td>
</tr>
<tr>
<td>Deserialization Time(%)</td>
<td>0.004</td>
<td>0.003</td>
</tr>
<tr>
<td>Service Times(%)</td>
<td>62.946</td>
<td>73.495</td>
</tr>
<tr>
<td>Others(%)</td>
<td>37.048</td>
<td>26.501</td>
</tr>
<tr>
<td>Total Time (s)</td>
<td>30.00</td>
<td>124.91</td>
</tr>
<tr>
<td>Frequency</td>
<td>700MHz</td>
<td>950MHz</td>
</tr>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Time(%)</td>
<td>0.002</td>
<td>0.001</td>
</tr>
<tr>
<td>Deserialization Time(%)</td>
<td>0.004</td>
<td>0.003</td>
</tr>
<tr>
<td>Service Times(%)</td>
<td>61.190</td>
<td>71.324</td>
</tr>
<tr>
<td>Others(%)</td>
<td>38.804</td>
<td>28.672</td>
</tr>
<tr>
<td>Total Time (s)</td>
<td>28.54</td>
<td>106.71</td>
</tr>
</tbody>
</table>

As this table shows, Axis2 framework spend more time to receive the response message, once the server side handles with a SOAP message. While CXF framework spend more time doing others tasks like in preparing and recovery content phases with generics classes. Furthermore, the frequency increase reduces the total time needed to conclude the operation.

Table IV shows the percentages occupied by serialization, deserialization and execution energy consumption on client device. Through this table, it can be seen for both frequencies the major factors were the same as in Table III. But the serialization and deserialization consumption have more influence than that showed in this table, because more processing is
in this scenario, once the time to perform these steps is less than 7% in all cases. Likewise, in Table III, higher frequencies reduces the total time of service execution.

Table VII shows the percentages of serialization, deserialization, and execution consumption in the server device. For both frequencies the major factors were the same as in Table VI. But the serialization and deserialization consumption have more influence than that showed in this table, because in these phases more processing is necessary which raise the energy consumption. It is important to highlight that higher frequencies also reduces the total energy consumption needed to conclude the operation.

Like Table V, Table VIII shows that the relation between frequency and power is proportional. Therefore the shortest time achieved by higher frequencies also leads to lower total energy consumption rather than higher power levels.

Finally, when Raspberry Pi is working as a server, it increased the final time of the client application, although the serialization and deserialization times have few influence on the final result, once this device has low CPU resources. Also working at higher frequencies Raspberry Pi can achieve a better performance and energy consumption rather than higher power levels.

VI. CONCLUSION

A wide variety of projects explore the resources of mobile and embedded devices that, although their hardware are limited, they are able to perform personal and commercial tasks. The Raspberry Pi offers a computational architecture with a general purpose, low cost, and low power consumption. In this paper, the performance and energy results showed that Raspberry Pi resources are insufficient for the execution of tasks that depend mostly on the processor. However, these devices can be used to serialize and deserialize REST messages in a timely manner, justifying its use as client applications or

---

**TABLE IV: Average Client Energy Consumption (%)**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>700MHz</th>
<th>950MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Consumption(%)</td>
<td>3.414</td>
<td>2.972</td>
</tr>
<tr>
<td>Deserialization Consumption(%)</td>
<td>10.924</td>
<td>12.457</td>
</tr>
<tr>
<td>Service Consumption(%)</td>
<td>59.734</td>
<td>47.924</td>
</tr>
<tr>
<td>Others(%)</td>
<td>34.204</td>
<td>32.359</td>
</tr>
<tr>
<td>Total Consumption(J)</td>
<td>58.73</td>
<td>233.88</td>
</tr>
<tr>
<td>Frequency</td>
<td>950MHz</td>
<td></td>
</tr>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Consumption(%)</td>
<td>3.435</td>
<td>3.117</td>
</tr>
<tr>
<td>Deserialization Consumption(%)</td>
<td>3.435</td>
<td>3.117</td>
</tr>
<tr>
<td>Service Consumption(%)</td>
<td>58.345</td>
<td>41.024</td>
</tr>
<tr>
<td>Others(%)</td>
<td>35.253</td>
<td>36.480</td>
</tr>
<tr>
<td>Total Consumption(J)</td>
<td>56.26</td>
<td>203.78</td>
</tr>
</tbody>
</table>

However, the relation between energy consumption and total time or power, as showed in Table V, is proportional. Thus, the reduction in energy consumption related with higher frequencies is due to the shorter processing times caused by higher power levels.

**TABLE V: Average Client Times (%)**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>700MHz</th>
<th>950MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Time(%)</td>
<td>2.202</td>
<td>1.871</td>
</tr>
<tr>
<td>Deserialization Time(%)</td>
<td>3.423</td>
<td>3.875</td>
</tr>
<tr>
<td>Service Times(%)</td>
<td>94.646</td>
<td>95.143</td>
</tr>
<tr>
<td>Total Time (s)</td>
<td>15.16</td>
<td>93.13</td>
</tr>
<tr>
<td>Frequency</td>
<td>950MHz</td>
<td></td>
</tr>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Time(%)</td>
<td>1.968</td>
<td>1.777</td>
</tr>
<tr>
<td>Deserialization Time(%)</td>
<td>2.348</td>
<td>1.387</td>
</tr>
<tr>
<td>Service Times(%)</td>
<td>95.407</td>
<td>95.735</td>
</tr>
<tr>
<td>Total Time (s)</td>
<td>31.98</td>
<td>164.42</td>
</tr>
</tbody>
</table>

**TABLE VI: Average Server Times (%)**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>700MHz</th>
<th>950MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Consumption(%)</td>
<td>1,96</td>
<td>1,77</td>
</tr>
<tr>
<td>Deserialization Consumption(%)</td>
<td>3,140</td>
<td>4,007</td>
</tr>
<tr>
<td>Service Consumption(%)</td>
<td>3,403</td>
<td>3,641</td>
</tr>
<tr>
<td>Others(%)</td>
<td>10,909</td>
<td>1,897</td>
</tr>
<tr>
<td>Total Consumption(J)</td>
<td>31.98</td>
<td>164.42</td>
</tr>
<tr>
<td>Frequency</td>
<td>950MHz</td>
<td></td>
</tr>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Serialization Consumption(%)</td>
<td>3,403</td>
<td>3,641</td>
</tr>
<tr>
<td>Deserialization Consumption(%)</td>
<td>3,403</td>
<td>3,641</td>
</tr>
<tr>
<td>Service Consumption(%)</td>
<td>85,688</td>
<td>94,462</td>
</tr>
<tr>
<td>Total Consumption(J)</td>
<td>29.39</td>
<td>146.39</td>
</tr>
</tbody>
</table>

It is important to highlight that higher frequencies also reduces the total energy consumption needed in this phase which raise the energy consumption. Therefore, the shortest processing times caused by higher power levels.

**TABLE VIII: Server Power Consumption**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>700MHz</th>
<th>950MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Axis2</td>
<td>CXF</td>
</tr>
<tr>
<td>Message Size</td>
<td>100Kb</td>
<td>500Kb</td>
</tr>
<tr>
<td>Service Consumption(%)</td>
<td>1,96</td>
<td>1,77</td>
</tr>
<tr>
<td>Others(%)</td>
<td>1,81</td>
<td>1,82</td>
</tr>
<tr>
<td>Total Consumption(J)</td>
<td>1,81</td>
<td>1,82</td>
</tr>
</tbody>
</table>

Table VI shows time percentages for serialization, deserialization, and execution of service on server device. In this table, the greatest impact factor is service execution time. Because a CPU Bound service is executed and takes the most part (around 93%) of total time due to high process level. It is important to emphasize that for Axis2 framework the execution time includes the times to handle with a SOAP message which raise the execution time compared to CXF framework.

Serialization and deserialization times have little influence in this scenario, once the time to perform this steps is less than 7% in all cases. Likewise, in Table III, higher frequencies reduces the total time of service execution.

Like Table V, Table VIII shows that the relation between frequency and power is proportional. Therefore the shortest time achieved by higher frequencies also leads to lower total energy consumption rather than higher power levels.

Finally, when Raspberry Pi is working as a server, it increased the final time of the client application, although the serialization and deserialization times have few influence on the final result, once this device has low CPU resources. Also working at higher frequencies Raspberry Pi can achieve a better performance and energy consumption rather than higher power levels.

VI. CONCLUSION

A wide variety of projects explore the resources of mobile and embedded devices that, although their hardware are limited, they are able to perform personal and commercial tasks. The Raspberry Pi offers a computational architecture with a general purpose, low cost, and low power consumption. In this paper, the performance and energy results showed that Raspberry Pi resources are insufficient for the execution of tasks that depend mostly on the processor. However, these devices can be used to serialize and deserialize REST messages in a timely manner, justifying its use as client applications or
as service providers for applications that do not require large computational capacity.

Regarding the frameworks, both have several benefits to the developer. However, the CXF framework was more appropriate in this environment, once it provides support for the creation of both service providers and client applications. Furthermore, when using CXF in service providers applications it has obtained shorter overall times for larger messages (500Kb) while Axis2 was better for smaller messages (100Kb). The overlocking configuration also has influence in time and energy consumption results. These results are more efficient when the Raspberry Pi is overlocking, especially for large messages, but it is important to remember that the overlocking was done in a controlled environment and it has a higher power levels.

In terms of development experience, the small time and energy consumption difference between the frameworks does not justify Axis2 use, once it is more complicated to develop RESTful clients. In a few words, the Raspberry Pi has impressed because of its easy installation, configuration and use of resources to provide and access web services. It can be a good low-cost solution for applications that require a smaller processing capacity, such as a monitoring system. In future work we intend to monitor the energy consumption using batteries and also testing another kinds of applications, like IO-Bound services to verify the performance of reading and writing data into SD cards.

REFERENCES