Assessment of Different OPC UA Industrial IoT solutions for Distributed Measurement Applications

Alberto Morato
CMZ Sistemi Elettronici S.r.l.
and *Dept. of Information Engineering*
University of Padova
alberto.morato@dei.unipd.it

Stefano Vitturi
National Research Council of Italy
CNR–IEIIT
stefano.vitturi@ieiit.cnr.it

Federico Tramarin
Dept. of Management and Engineering
University of Padova
federico.tramarin@unipd.it

Angelo Cenedese
Dept. of Information Engineering
University of Padova
angelo.cenedese@unipd.it

***Abstract*—The Industrial IoT scenario represents an interesting opportunity for distributed measurements systems, that are typically based on efficient and reliable communication systems, as well as the widespread availability of data from measurement instruments and/or sensors. The Open Platform Communications (OPC) Unified Architecture (UA) protocol is designed to ensure interoperability between heterogeneous sensors and acquisition systems, given its object–oriented structure allowing a complete contextualization of the information. Stemming from the intrinsic complexity of OPC UA, we designed an experimental measurement setup to carry out a meaningful performance assessment of its main open source implementations. The aim is to characterize the impact of the adoption of this protocol stack in a DMS in terms of both latency and power consumption, and to provide a general yet accurate and reproducible measurement setup.**

***Index Terms*—Industrial Internet of Things, Performance evaluation, OPC UA, Distributed Measurement Applications**

I. Introduction

In the last few years, the industrial world embraced the Industry 4.0 paradigm [1], which merges technologies with products, systems, and services, having its own intrinsic networked structures, in order to realize the Industrial Internet of Things (IIoT) [2], [3], to improve productivity, efficiency, reliability, safety, and security.

Basically, IIoT is a network of networks that connects industrial devices, equipment, sensors and actuators to provide high level services in several fields of applications. Indeed, with IIoT, heterogeneous data collected from different sources can be analyzed to better understand and drive the overall production in a fully automatic way, possibly adopting Machine Learning and Data Mining approaches [4]. Furthermore, thanks to the widespread connectivity, data can be effectively assessed from anywhere by several types of devices, such as Tablet PCs, Smartphones and Personal Computers. This implies the seamless connection of such devices, with the consequent adoption of several networks and protocols.

IIoT represents an interesting opportunity also for distributed measurements systems (DMS) [5]. Indeed, DMS are typically based on efficient and reliable communication systems [6], [7], as well as the widespread availability of data from measurement instruments and/or sensors.

Within the IIoT–enabled DMS scenario, a protocol of interest is represented by the Open Platform Communications (OPC) Unified Architecture (UA) [8]. OPC UA is an IEC standard protocol [9] designed to ensure interoperability between different equipments, effectively employed to implement Machine–to–Machine (M2M) communication. A typical application of OPC UA is found in factory automation systems where it enables to accomplish interoperable communication between PLCs and SCADA systems, allowing to ensure effective plant monitoring and control, also ensuring a high level of data protection against attacks and threats.

Analogously, OPC UA represents an appealing opportunity for distributed measurement systems as well. Indeed, its object–oriented structure allows a complete contextualization of the information. For example, an object could be used to store the value of a measurement, the features of the instrument/device that produced it, the measurement unit, possible thresholds and so on. These are important characteristics that allow to reduce ambiguities when the DMS has to deal with multiple and heterogeneous types of data.

Nonetheless, the complexity of the OPC UA protocol may have a negative impact on the performance and its evaluation, particularly in terms of response times. The issue that possibly introduced (and undesired) delays might compromise the behavior of distributed measurement systems has already been analyzed in the literature [10]–[13]. Nevertheless, this aspect may assume further relevance when devices with low computation capabilities and low costs are used, as it is often the case of field equipment like instruments and sensors.

Motivated by the above considerations, we designed an experimental measurement setup to carry out a meaningful performance assessment of the three main open source implementations of the OPC UA protocol stack, namely Open62541, FreeOPC UA C++ and FreeOPC UA Python. With the aim

of providing a meaningful and fair assessment, we exploited the widespread commercially available Raspberry Pi Model 3B boards, equipped with the Raspbian Operating System. A preliminary version of this measurement setup has already been discussed in [12]. In this work we introduce several improvements with respect to that analysis, and in particular: (i) we considered two different operating system configurations and the CPU isolation feature, (ii) we provide a more comprehensive analysis about the CPU usage statistics and (iii) take into account also power consumptions, and finally (iv) we corrected several implementation bugs and (v) improved the optimizations of the compiler, to provide more meaningful results. In the following, we first describe the measurement set–up, which has been designed to be independent from the specific protocol stack and hence of general usage, and then provide the results of an extensive measurement campaign aimed at evaluating the transmission times as well as the power consumption of the three different implementations.

II. Brief Introduction to OPC UA

OPC UA is based on the Client–Server model, where the server is the source of information which is structured as objects, formally referred to as “Nodes”. The Server provides a Client with a set of Services, for example, read, write, browse, etc., which can be used to access the information stored on the server itself. The set of nodes made available by an OPC UA server is referred to as the address space [14].

The OPC UA model defines nodes in term of variables, methods and events. A Node is, hence, the fundamental entity of OPC UA and it represents a basic object which has only the attributes necessary to define any kind of information item (e.g. ID, name, etc.).

In a distributed measurement system that relies on the OPC UA protocol, measurements can be stored on nodes that belong to one or more servers, so that they can be accessed by the distributed clients. An illustrative sketch representing the described scenario is reported in Figure 1. As can be seen, measurements stored in different devices, and structured within diverse OPC UA servers, are remotely accessed by an OPC UA client which provides for their visualization.

III. Experimental Set-up

As already described in the Introduction, this manuscript proposes an experimental measurement setup, designed to allow the performance assessment of a given implementations of the OPC UA protocol stack. The main purpose of this experimental activity is to provide a figure about the performance of such implementation mainly in terms of latency and power consumption, when deployed within a lightweight embedded system similar to those adopted for intelligent IoT sensors.The design of the setup has been conceived to be as much general as possible, in order to avoid a strict dependence of particular software implementations as well as to enable the setup to be reused or reproduced in different scenarios.

All the experiments have been carried out on Raspberry Pi Model 3B+ boards. These have been equipped, alternatively

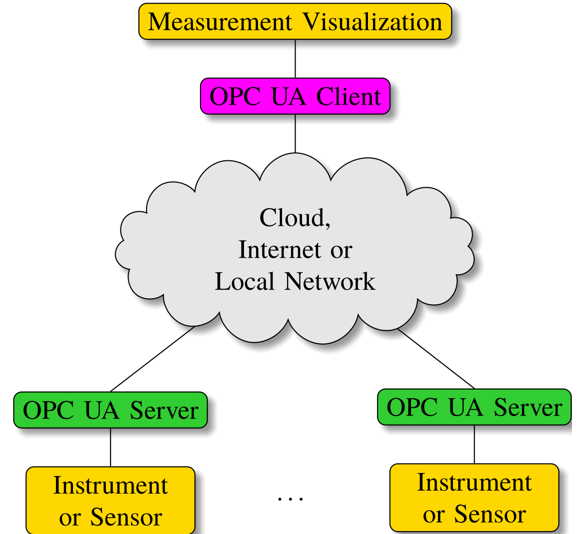


Fig. 1. Example of use of OPC UA in a Distributed Measurement System based on the IIoT Paradigm

on the basis of the measurement session, with two different operating system versions. The first one is represented by the default Raspbian OS (Kernel version 4.14.79). The second one is instead represented by a real-time version of the Rasbian OS (Kernel version 4.14.74–rt44). The latter one has been obtained starting from the default kernel version by the introduction of the `RT_PREEMPT` patch set, which enables a real–time behavior of the system allowing non critical part of the kernel to be preempted in favor of the execution of userspace applications.

Furthermore, as it will be better detailed in the next Section, we exploited a significant feature offered by the Linux operating system, that is, to isolate a group of CPU cores in which a process can be run. Indeed, the `isolcpus` boot parameter in combination with the `taskset` command, allows to isolate one or more cores from the kernel scheduling and to reserve them for the execution of userspace applications without interference from the OS.

In this way, we have been able to carry out experiments exploiting four different configurations, that is, experiments with or without real-time extensions, and experiments where the OPC UA processes were alternatively managed either by the kernel scheduler or allocated to the isolated core. As a further mean to minimize the external factors that may affect the accuracy of the measurements, the CPU governor (i.e. a kernel–level component responsible of scaling the CPU frequency based on the workload) has been disabled and the CPU frequency has been set to its maximum operable value of 1.4 GHz.

The open-source implementations of OPC UA considered in the following experiments are reported below, along with the indication of the adopted programming language (PL). In order to ensure reproducibility of the experiments, we also provide the commit hash of the sources (all the implementations are available through the popular Github platform) at the time the experiments have been performed:

- **open62541** — PL: common subsets of the C99 and C++98. *Commit hash: 9f0c73d*
- **FreeOPC UA C++** — PL: C++11. *Commit hash: da2b76f*
- **FreeOPC UA Python** — PL: Python. *Commit hash: 83fb9ea*

All the aforementioned protocol stack work natively on the Raspberry PI boards, and consequently their setup procedure has not involved any further software adaptation. Nevertheless, they are conceptually different, so that their behaviors on the selected hardware device may provide useful insights for future developments. Particularly, two out of three implementations are implemented by means of compiled languages (C/C++), whereas the other one is implemented in Python, which is a high level interpreted language. Since the outcomes of the experiments also depends on the adopted development environment, for reproducibility purposes we also resume the most relevant technical details:

- Python version 3.5.3;
- glibc 2.23;
- gcc version 6.3.0;
- gcc optimization option: -O3 -s.

IV. Measurements Analysis

The objective of the measurements is to identify the achievable performance figures of different OPC UA implementations in terms of time and energy consumption, both parameters being meaningful for DMS deployment. To this regard, we developed a test communication task with which an integer variable stored in the OPC UA Server is read by the OPC UA Client. In this task, the server implements two separate threads as represented in Figure 2. Thread A simulates the acquisition of a new measurement (i.e. a physical quantity) every second, by increasing an integer variable. Thread B is instead devised to manage the whole OPC UA server. The measurement outcome, stored in an OPC UA object, is saved in a memory area common to both threads so that the server can access it. In order to acquire the variable, the client sends a read request to the server, which answers consequently in agreement with the OPC UA protocol rules. The time necessary to perform the whole procedure is defined as *service time*, T_s, which is obtained as the interval which elapses between the time in which the request is generated by the client (T_{req}) and that in which it actually receives the variable (T_{res}), i.e.

$$T_s = T_{res} - T_{req} \tag{1}$$

It is worth noticing that the time T_s is measured by acquiring the content of the internal CPU register "Cycle Counter Register" which is implemented within ARM processors. It performs as a counter of the processor clock cycles, hence allowing to directly obtain the elapsed time. Accessing the register requires only one CPU Cycle, so that its impact on the evaluation of the service time time T_s is negligible. Resuming, for each OPC UA implementation four experiments have been carried out, using both operating systems and with different CPU configurations (isolated/not isolated). For each

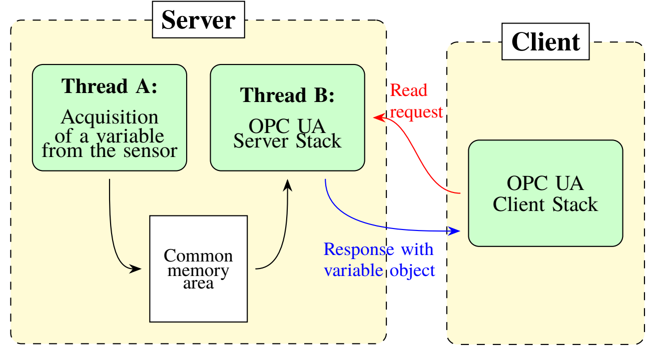


Fig. 2. Software architecture of the communication task

TABLE I
Statistics of the CPU Usage

	CPU Usage		
	Mean	In Kernel Space	In User Space
Open62541	17.2%	60.89%	39.11%
FreeOPC UA C++	26.1%	50.63%	49.37%
FreeOPC UA Python	51.2%	–	–

experiment, N = 100.000 measurements of the service time have been collected and analyzed.

A. CPU Usage

A first set of outcomes is resumed in Table I, which shows the statistics about the CPU usage for the three considered implementations. In particular, it can be seen that Open62541 is the most efficient from the average resources utilization point of view, followed by FreeOPC UA C++ and FreeOPC UA Python. Actually, the latter highlighted an almost doubled utilization value compared to the others, although this should be not much surprising given it is based on an interpreted language, being certainly less efficient. Nevertheless, it is interesting to note that in the case of the implementation based on Open62541 the subdivision of the used resources of the CPU is slightly unbalanced towards the Kernel Space, while in FreeOPC UA C++ we have a subdivision almost at 50%. Unfortunately, a comparison with the last implementation is not possible, because the tool with which the analysis was performed does not support measurements of the stack of interpreted languages.

TABLE II
CPU usage during the communication task

	context switches	CPU migrations	CPU cycles
Open62541	$100 \cdot 10^3$	1	$7.7 \cdot 10^9$
FreeOPC UA C++	$200 \cdot 10^3$	0	$13 \cdot 10^9$
FreeOPC UA Python	$283 \cdot 10^3$	29	$533 \cdot 10^9$

A second set of outcomes is relevant to general data concerning the use of the CPU during the communication task, as reported in Table II. In particular, the table reports the

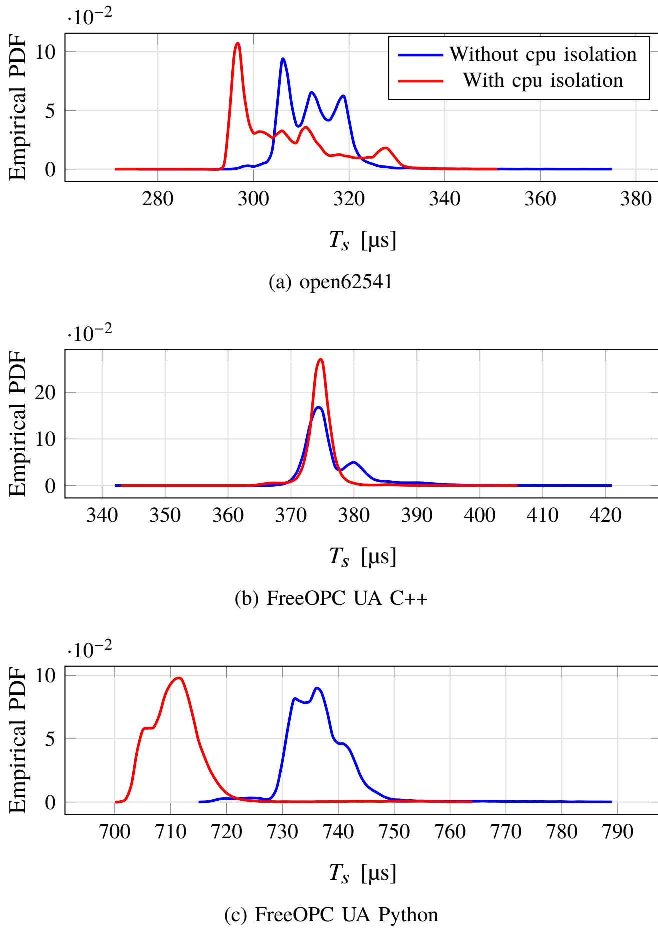


Fig. 3. EPDF of the service time for the configuration with the generic OS. Blue line: configuration without CPU isolation. Red line: configuration with CPU isolation enabled, where both server and client are forced to run on the isolated CPU.

number of context switches, CPU migrations and total number of CPU cycles to complete the exchange of 100.000 variables. These parameters are common indices exploited to determine the efficiency of a program, where high values indicate poor optimization and therefore long execution times. The outcomes actually confirm the previous observations, and also in this case it is worth noting how the compiled versions show similar values, while the Python based version has much higher values regarding in particular the number of CPU cycles and CPU migrations.

B. Generic OS

The first set of experiments was carried out using the setup with the generic operating system with both cpu isolation enabled and disabled. The empirical probability density function (EPDF) of the service time and the most relevant statistics of the experiments are reported in Figure 3 and Table III, respectively.

The obtained outcomes show that both the compiled implementations, Open62541 and FreeOPC UA C++, are characterized by rather similar average values of the service time. Conversely, the average T_s results much higher (about doubled) for the FreeOPC UA Python implementation, as it could be expected since Python is an interpreted language. Looking instead at the standard deviation, Open62541 is characterized by a non negligible values, which reflects on a considerable jitter of the service time, whereas both FreeOPC UA C++ and FreeOPC UA Python are definitely more stable.

The activation of the CPU isolation leads to appreciable benefits. All the three implementations exhibit a decreased average service time, which is more evident in relative terms for the FreeOPC UA Python implementation. Conversely, the impact on the standard deviation is rather limited, although in relative terms the FreeOPC UA C++ implementation is the one which mostly benefits from the CPU isolation.

TABLE III
STATISTICS OF THE SERVICE TIME FOR THE GENERIC OS SET-UP

	Service time T_s [µs]			
	Whitout CPU isolation		With CPU isolation	
	Mean	Std	Mean	Std
Open62541	312.83	12.56	306.67	12.76
FreeOPC UA C++	377.30	4.54	374.74	3.32
FreeOPC UA Python	736.79	7.44	711.27	6.52

In general, regardless of the type of operating system used, the fact that the activation of the cpu isolation has had effects only on Open62541 and FreeOPC UA Python is an indirect proof of the measurements proposed in Table II. In particular, both have a non-zero number of CPU migrations, which implies that during the execution of the communication task the allocation of the process has been moved from a core to another, even multiple time. The involved operations to enable core migration cause a slight increase of the execution time. When the process is forced to run in the isolated CPU, migrations are no longer allowed, which explains the improvement in execution time. The effect is particularly visible with Python as on average it has a greater number of migrations.

C. Real-time OS

In the following set of outcomes we enabled the real–time kernel extensions. The EPDF of the service time and the most relevant statistics of the experiments are reported in Figure 4 and Table IV, respectively.

As a first observation it may be stated that, in general, the measured RTT values are slightly higher compared with the general purpose operating system for both the cases of the CPU isolated and non isolated.

From the results, it appears evident that only the Open62541 implementation obtained considerable benefits from the introduction of the real–time operating system. Indeed, the EPDF has a more compact shape resulting in a low jitter of the service time. Also, the isolation of the CPU brings a further improvement. Conversely, for both the FreeOPC UA C++ and FreeOPC UA Python implementations, the EPDFs behavior reveals a worsening with respect to the case of the generic operating system.

Although the outcomes of this set of measurements may seems counterintuitive, they allow to draw some useful con-

siderations about the general impact of the linux real–time extensions on latency. Indeed, as can be seen from Table I, all the considered implementations of the OPC UA protocol stack make extensive use of the kernel functions, especially those concerning network connectivity. Nonetheless, the real-time patch makes some parts of the kernel preemptible, thus leaving up more space for executing instructions in the user space. Also, as shown in [15], the application of the patch has negative effects on the throughput of the communication interface. For these reasons in the applications there is a worsening of the performance with the use of the real-time operating system.

It is worth highlighting that these observations may not be valid in devices without an operating system or in which there is no separation between Kernel and User Space (such as with FreeRTOS embedded systems). In these cases, it can be assumed that the system resources are allocated exclusively to the tasks with highest priority in that specific instant, leading to an increase in the performance of the communication task. However, tests on this type of systems are left for future work.

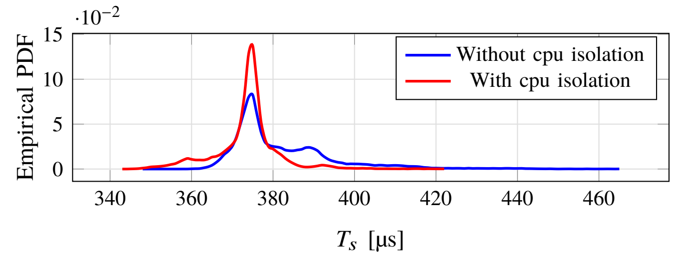
D. Power consumption

One of the main issues concerned with battery powered and possibly mobile devices is the autonomy. Indeed, such devices have to ensure a good level of performance for a given amount of time. To meet these requirements, modern processors are capable of Dynamic Voltage and Frequency Scaling (DVFS) to minimize energy consumption and, consequently, extend battery lifetime [16]. In the Raspberry PI boards used in the experimental set–up, the DVFS functionality is driven by a default kernel governor, called `ondemand`, that dynamically adjusts the CPU frequency in agreement with the workload variation. Specifically, if the workload exceeds a predefined threshold for a certain amount of time, then the governor increases the CPU operating frequency to its maximum value. Conversely, if the workload is below the threshold, the operating frequency is switched to the lowest feasible one [17].

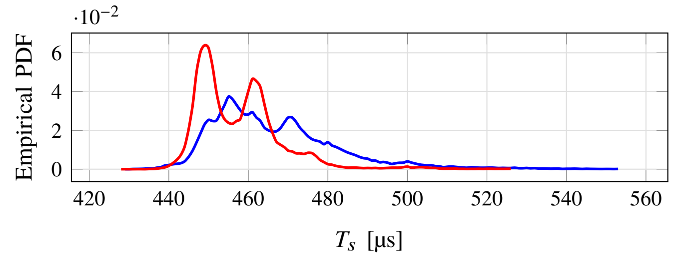
It is clear that such an approach may represent an optimal trade-off between performance and power consumption in a generic processing system. However, it may also introduce a certain latency that impacts on the system responsiveness. Moreover, frequent, sudden and unforeseen frequency switches may also impair the accuracy of time readings from the processor timestamp counter. The aforementioned reasons are at the basis of the choice to disable the CPU frequency

TABLE IV
STATISTICS OF THE SERVICE TIME FOR THE REAL–TIME OS SETUP

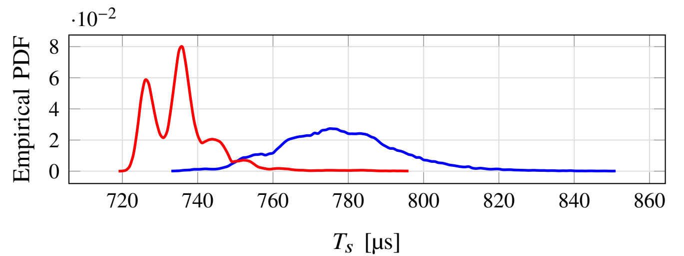
	Service time T_s [µs]			
	Whitout CPU isolation		With CPU isolation	
	Mean	Std	Mean	Std
Open62541	382.48	24.44	368.78	10.13
FreeOPC UA C++	467.59	15.01	457.60	11.32
FreeOPC UA Python	778.43	20.63	735.84	9.69



(a) open62541



(b) FreeOPC UA C++



(c) FreeOPC UA Python

Fig. 4. EPDF of the service time for the configuration with the Real–time OS. Blue line: configuration without CPU isolation. Red line: configuration with CPU isolation enabled, and where both server and client are forced to run on the isolated CPU.

governor during all the measurement sessions discussed so far, with the aim of minimizing the external factors that may affect the accuracy of the measurements.

Nevertheless, the power consumption of the adopted devices is a metric of definite interest in an IIoT scenario. Consequently, we carried out a preliminary analysis of the impact of the governor on both the power consumption and the performance of the developed OPC UA based distributed measurement system developed for the experiments. The tests have been performed using only the Open62541 stack, with the generic operating system and without CPU isolation. We measured the current consumption on the client side.

The experimental set-up for the current measurement is reported in Figure 5. The Raspberry Pi has been powered with a stabilized power supply, providing a 5 V continuous voltage. The current has been measured using a Hall effect sensor whose sensitivity is 66 mV/A. Current measurements have been acquired using an external digital acquisition system equipped with a 12 bit ADC with an input range of [0, 3.3]V at a sampling rate of 1 Hz. Each time the communication task is started, the Raspberry Pi rises a signal triggering a new acquisition of the current level, which is also timestamped for

further analysis.

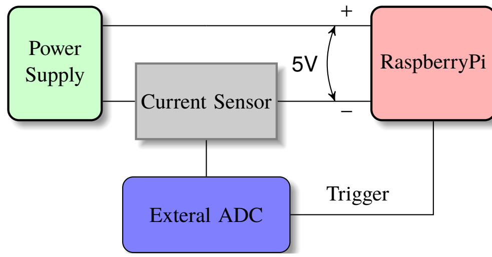


Fig. 5. Block diagram of the current measurement setup.

TABLE V
STATISTICS OF THE CURRENT CONSUMPTION WITH CPU GOVERNOR DISABLED AND ENABLED

	Current [A]	
	Mean	Std
Governor Disabled	0.4862	0.0379
Governor Enabled	0.4838	0.0358
Percentage Change	0.49%	5.86%

The results of this new set of experiments are summarized in Table V, which reports the statistics of the power consumption for the two cases in which the governor was, respectively, enabled and disabled. Looking at Table V, it can be seen that the difference in current current consumption with both the configuration of the governor is negligible. This is mostly due to the low CPU resources used by Open62541 (see Table I for reference) which, although the CPU frequency is at maximum value, are not sufficient to imply an appreciable variation in current consumption.

V. CONCLUSION AND FUTURE WORKS

In this paper we proposed the usage of OPC UA as communication interface for distributed measurement systems. In particular we focused on analysis of a communication task in term of service time of three open source implementation of the stack with different configuration of operating systems and isolated CPUs. Generally speaking, we observed that using a Real-Time operating system lead to an increase of the service time. This is most likely due to the preemption of part of the kernel and consequently implies the reduction of the network communication port throughput. On the other hand, the allocation of the stack instance on an isolated CPU allow to obtain a significant reduction of the jitter. Based on these observation, is clear that the use of an Real-Time operating system does not bring any advantage and in general the best performances are achieved with a generic operating system with an isolated CPU. In particular, it has been shown that Open62541 is the most efficient among the examined implementations.

The proposed application of OPC UA can also be extended to wireless battery powered measurement systems. For this reason we gave an insights on the power consumption of one of the experimental setups for different configuration of the CPU governor. It has been show that, due to the relatively low demand of CPU resources required by the Open62541 implementation, the configuration of the CPU governor has almost no impact of the power consumption. Anyway, a more extensive experimental campaign focused on mobile battery powered measurement system as well as on the performances of the communication task over wireless link, will be object of future works.

In addition, since the proposed experimental setup seems to be overkill for small integrated sensors, we plan to test the framework on low power embedded devices as, for example, microcontroller with no operating system or with real time OS such as FreeRTOS.

REFERENCES

[1] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, Jun. 2017.
[2] E. Sisinni, A. Saifullah *et al.*, "Industrial internet of things: Challenges, opportunities, and directions," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 11, pp. 4724–4734, Nov 2018.
[3] S. Vitturi, C. Zunino, and T. Sauter, "Industrial Communication Systems and Their Future Challenges: Next-Generation Ethernet, IIoT, and 5G," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 944–961, Jun. 2019.
[4] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*, 3rd ed. Boston: Morgan Kaufmann, 2011.
[5] D. Grimaldi and M. Marinov, "Distributed measurement systems," *Measurement*, vol. 30, no. 4, pp. 279–287, Dec. 2001.
[6] G. Y. Tian, "Design and implementation of distributed measurement systems using fieldbus-based intelligent sensors," *IEEE Trans. on Instr. and Measurement*, vol. 50, no. 5, pp. 1197–1202, Oct 2001.
[7] L. Skrzypczak, D. Grimaldi, and R. Rak, "Analysis of the different wireless transmission technologies in distributed measurement systems," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2009. IDAACS 2009. IEEE International Workshop on*, Sept 2009, pp. 673–678.
[8] D. Bruckner, M.-P. Stanica *et al.*, "An Introduction to OPC UA TSN for Industrial Communication Systems," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1121–1131, Jun. 2019.
[9] International Electrotechnical Commission, *IEC 62541: OPC unified architecture - Part 1: Overview and concepts*. IEC, 2016.
[10] J. W. Overstreet and A. Tzes, "Internet-based client/server virtual instrument designs for real-time remote-access control engineering laboratory," in *Proceedings of the 1999 American Control Conference*, 1999, pp. 1472–1476.
[11] M. Bertocco, G. Gamba *et al.*, "Investigating wireless networks coexistence issues through an interference aware simulator," in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2008, pp. 1153–1156.
[12] A. Cenedese, M. Frodella *et al.*, "Comparative assessment of different OPC UA open–source stacks for embedded systems," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Zaragoza, Spain: IEEE, Sep. 2019, pp. 1127–1134.
[13] L. Cristaldi, A. Ferrero *et al.*, "The impact of internet transmission on the uncertainty in the electric power quality estimation by means of a distributed measurement system," *IEEE Transactions on Instrumentation and Measurement*, vol. 52, no. 4, pp. 1073–1078, Aug 2003.
[14] M. Damm, S.-H. Leitner, and W. Mahnke, *OPC Unified Architecture*. Springer-Verlag Berlin Heidelberg, 2009.
[15] "Raspberry Pi: The N-queens Problem (benchmark) Preempt-RT vs. Standard Kernel," https://lemariva.com/blog/2018/04/raspberry-pi-the-n-queens-problem-performance-test, (accessed 2019-11-19).
[16] J. Howard, S. Dighe *et al.*, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.
[17] M. P. Karpowicz, "Energy-efficient CPU frequency control for the Linux system," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 420–437, 2016.