Assessment of Different OPC UA Implementations for Industrial IoT-based Measurement Applications

Alberto Morato, Stefano Vitturi, Federico Tramarin and Angelo Cenedese

***Abstract*—The Industrial IoT (IIoT) paradigm represents an attractive opportunity for new generation measurement applications, which are increasingly based on efficient and reliable communication systems to allow the extensive availability of continuous data from instruments and/or sensors, thus enabling real-time measurement analysis. Nevertheless, different communication systems and heterogeneous sensors and acquisition systems may be found in an IIoT-enabled measurement application, so that solutions need to be defined to tackle the issue of seamless, effective, and low-latency interoperability. A significant and appropriate solution is the Open Platform Communications (OPC) Unified Architecture (UA) protocol, thanks to its object–oriented structure that allows a complete contextualization of the information. The intrinsic complexity of OPC UA, however, imposes a meaningful performance assessment to evaluate its suitability in the aforementioned context. To this aim, this paper presents the design of a general yet accurate and reproducible measurement setup that will be exploited to assess the performance of the main open source implementations of OPC UA. The final goal of this work is to provide a characterization of the impact of this protocol stack in an IIoT-enabled Measurement System, in particular in terms of both the latency introduced in the measurement process and the power consumption.**

***Index Terms*—Industrial internet of things (IIoT), distributed sensors, open platform communication unified architecture (OPC UA), latency assessment, performance evaluation, 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, to realize the Industrial Internet of Things (IIoT) [2], [3]. IIoT is a network of networks that connects industrial equipment, controllers, sensors and actuators, i.e. the "Things", to provide diverse and advanced types of services in manufacturing systems, aiming at improving quality, productivity, efficiency, reliability, safety, and security.

Traditionally, the foundation of manufacturing and process industries has been in the deployment of specific distributed sensor systems, to the purpose of monitoring (and then controlling) the production process, thus leveraging the concept of Distributed Measurement Systems (DMS) [4]. With the increased pervasiveness of the IIoT paradigm, DMSs come even more into focus, since the components of an IIoT system need an even further level of interaction to integrate instrumentation data, sensors, communication and processing. Moreover, the need of improving production capacity and optimizing the output process, and eventually exploiting predictive maintenance and machine learning approaches [5], requires an increased number of sensor devices and sensor swarms to be deployed.

This new IIoT–enabled DMS scenario is based on the support of efficient and reliable communication systems, which have to ensure widespread availability of data gathered from possibly heterogeneous measurement instruments and/or sensors [6], [7]. Overall, the IIoT paradigm may represent the enabler for several enhanced measurement features: continuous and thorough measurements through low-power wireless connections, measurement collection over considerably wide geographic areas, and real-time analysis of measurement data collected from the field [8].

Unfortunately, in the highlighted IIoT scenario it is common that components and sensors devices come from different producers and use different formats to represent measurement data, and also it is very likely that they intrinsically operate over heterogeneous networks. Hence, the provision of ways to enable communication and interoperability among such devices is of paramount importance. A key solution toward this goal is the Open Platform Communication (OPC) Unified Architecture (UA) [9]. OPC UA is a protocol defined by the IEC 62541 international standard [10] conceived to implement Machine–to–Machine (M2M) communication over possibly different physical media, while ensuring high level data protection against attacks and threats.

OPC UA represents hence an appealing and advantageous opportunity for the arising IIoT measurement paradigm. Particularly, its object–oriented structure allows a complete contextualization of the information. For instance, an OPC UA object could be used to store the value of a measurement, the features of the instrument/sensor, the measurement units, possible thresholds and so on. Such important characteristics allow for new generation of measurement instruments to deal with multiple and heterogeneous types of data.

At the same time, the complexity of the OPC UA protocol may also reveal an obstacle to its introduction within measurement systems. Indeed, sensors and actuators, field equipment and measurement instruments in the IIoT scenario are typically realized exploiting devices with limited hardware resources and low computational capabilities (and low costs). As a consequence, the implementation of OPC UA on such devices might be problematic and, furthermore, the performance might result compromised, in terms of increased latency and power consumption, hence impairing the quality and accuracy of

A. Morato is with the Dept. of Information Engineering, University of Padova, Italy and with CMZ Sistemi Elettronici srl, Carbonera, Italy.

A. Cenedese is with the Dept. of Information Engineering, University of Padova, Italy and with the CNR–IEIIT, National Research Council of Italy.

S. Vitturi is with the CNR–IEIIT, National Research Council of Italy.

F. Tramarin is with the Dept. of Engineering E. Ferrari, University of Modena and Reggio Emilia, Italy.

measurements.

II. Related Work and Contribution

The introduction of Industrial IoT technologies in the context of distributed measurement systems has started to be addressed some years ago. Paper [8] deals with the potential of IIoT for the instrumentation and measurement fields. Notably, the authors provide an accurate assessment that addresses benefits and challenges, including also some useful commercial aspects. In [11], [12] and [13] the authors address diverse Low Power Wide Area Networks (LPWAN) to enable IIoT-based measurement and monitoring applications over large distances. Both papers [14] and [15] describe the use of Wi-Fi, another important network for IIoT, to implement measurement systems that involve remote cloud data storage and analysis.

Moving to OPC UA, in [16] the authors describe a method to achieve synchronization among electrical drives connected via EtherCAT (a widespread real–time Ethernet network) using the OPC UA protocol. Particularly, the paper deals with the significant topic of obtaining a high accuracy synchronization over a geographically distributed system, that is of uttermost importance in distributed measurement applications. In [17], the authors refer to metrology assisted assembly systems, and introduce the optical large–scale metrology instruments, such as laser trackers and indoor GPS. Then they propose an object–oriented model to formally describe such instruments and investigate the suitability of OPC UA, as well as that of other protocols, to implement such a model. In [18] OPC UA is used to implement a smart sensor system to monitor the behavior of numerical control devices in the Industry 4.0 context. Specifically, OPC UA objects are used to store sensor information such as measurement, threshold, range, product data, etc. Another interesting OPC UA application is proposed in [19]. Here the authors present a system based on OPC UA to connect and integrate components typical of industrial automation as well as of distributed measurement systems. Examples of applications are provided that include smart microgrids, industrial laboratories and energy systems in general. Paper [20] considers an IIoT environment and focuses on the transfer of plant data to a cloud when OPC UA–based gateways are used to gather data directly at the production level. Notably, the authors implemented a measurement system that allowed to determine the impact of Quality of Service parameters on the communication delays.

The papers cited above represent interesting contributions. However, as far as OPC UA is concerned, they mostly describe meaningful applications that make use of such protocol, without investigating its actual potential and capabilities. Nor, they consider protocol implementation aspects, which represent very important issues especially in the industrial scenario.

Moving from the above considerations, in this paper, which substantially extends [21], we propose a more structural assessment about the adoption of OPC UA in the context of IIoT–based measurement systems. In this respect, we report and discuss the results of an experimental work on some popular implementations of the OPC UA protocol stack, Particularly, we considered four OPC UA implementations. Three of them are open source, namely Open62541, FreeOPC UA C++ and FreeOPC UA Python, whereas the fourth one is a proprietary product, namely Prosys OPC Java. The work is aimed at investigating the behavior of OPC UA for the different implementations focusing on i) CPU usage, ii) communication times and iii) power consumption. In order to provide a meaningful and fair assessment, the protocol was implemented on a widespread commercially available Raspberry Pi Model 3B+ board that, thanks to its features, represents a manageable and effective test–bed. The measurement set–up has been designed to be of general usage and reproducible. Also, experiments have been mostly carried out using two widespread communication systems, namely Ethernet and Wi–Fi.

In detail, the paper is organised as follows. Section III gives a brief description of the OPC UA protocol and outlines the possible structure of distributed measurement systems that rely on OPC UA. Section IV introduces the experimental set–up implemented for the measurements. Section V describes the tests carried out and discusses the obtained results. Finally, Section VI concludes the paper and outlines some future directions of research.

III. Brief Introduction to OPC UA

OPC UA is based on a client–server relationship, in which the information is structured following an object–oriented model, where objects are formally referred to as "nodes". The OPC UA model defines the node objects in term of variables, methods and events. A Node is, hence, the fundamental entity of OPC UA and represents a basic object which has only the attributes necessary to define any kind of information item (e.g. ID, name, etc.). The set of nodes made available by an OPC UA server is referred to as the address space [22].

To access the information stored within the nodes, OPC UA provides for a number of service sets. The *attribute* service set, in particular, contains the `read/write` services that may be used by a client to access the attributes of a node. Thus, for example, the attribute "value" of a node of type "variable" may be read (respectively, written) by a client by suitably invoking the `read` (respectively, `write`) service.

With the *subscription* service set, a client may define a "subscription" and assign to it some "monitored items". Then, any attribute of any node in the address space may be associated to a monitored item. The change of one of such attributes triggers a "notification" to the subscription. Based on a "publishing interval" set by the client, the subscription periodically sends to the client the list of notifications that occurred during the publishing interval. In this way, for example, a client may be notified periodically with the list of changes of the attribute value, of a node of type variable, that occurred in the last publishing interval.

More recently, a further relationship to exchange information in the context of OPC UA has been added to the standard, namely, the *PubSub* (Publisher – Subscriber) relationship. It is complementary to the client–server one. With this model, a publisher is responsible for creating "DataSets" that may contain the attributes of variable or event nodes (for example the value of a variable). DataSets are grouped

within "DataSetMessages" that are sent continuously to a "Message Oriented Middleware" that, in turn, delivers them to subscribers. With such a model, it is possible to distribute data and events relevant to the publisher to a set of devices (subscribers) in a very efficient way.

In the context of IIoT-based measurement applications, the OPC UA protocol can be profitably exploited to allow seamless and secure interoperability among the heterogeneous sources of measurement data, as well as among the heterogeneous communication networks. Indeed, measurements can be stored by nodes that belong to one or more servers, so that they can be seamlessly accessed by distributed clients using, for example, the aforementioned service sets. 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, can be remotely accessed by an OPC UA client which implements techniques of real–time analysis and visualization.

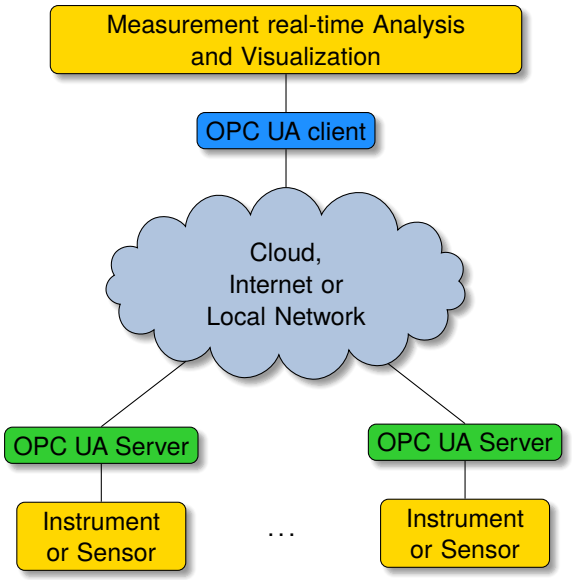


Fig. 1. Example of use of OPC UA in an IIoT-based Measurement System.

IV. Experimental Set–up

The experimental set–up has been redesigned, with respect to [21], to be as much general as possible, with reproducibility in mind in order to be easily reused in different scenarios. The set–up has been used to assess the behavior of some different OPC UA protocol stacks, implemented on lightweight embedded systems, that resemble those adopted by intelligent IoT sensors. Particularly, experiments have been carried out using Raspberry Pi Model 3B+ boards, over two diverse communication systems, namely Ethernet and Wi-Fi, as schematically shown in Fig. 2. Indeed, while both networks are meaningful in distributed measurement systems and IIoT scenarios, the former is much more targeted to high performance, ultra low-latency and local measurement applications, whereas the latter represents its wireless counterpart, allowing for increased mobility and larger installations. More importantly, although Wi-Fi networks are able to provide very high transmission rate and good reliability, the performance figures they provide are clearly different with respect to those of Ethernet networks,

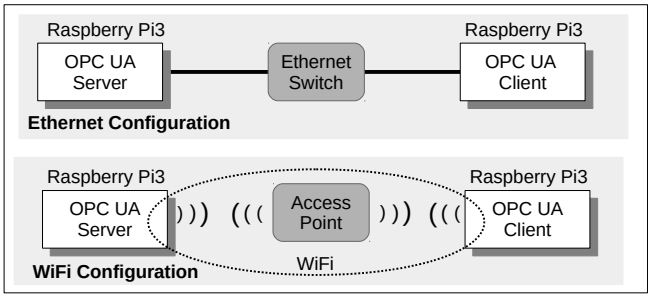


Fig. 2. Experimental set–up.

that may considered hence as a benchmark. An important aspect that will be analyzed in this work is hence the comparison between the two network supports in terms of latency to gather measurement data, which is a crucial parameter for accurate distributed measurements.

As can be seen, the same topology was used for the two networks, with OPC UA client and server implemented on two diverse Raspberry Pi boards. A Netgear WGR 614 Wireless-G router was used to connect the two boards that was able to seamlessly implement the packet routing for both the Ethernet and Wi-Fi configurations.

Two different operating system versions have been used for the boards. The first one is represented by the default Raspbian OS (Kernel version 4.14.79), whereas the second one is its real-time version (Kernel version 4.14.74–rt44). The latter one has been obtained from the default kernel version with the introduction of the `RT_PREEMPT` patch set, which enables a real–time behavior of the system allowing non critical parts of the kernel to be preempted in favor of the execution of userspace applications. Furthermore, we exploited a useful feature offered by the Linux operating system, to isolate a group of CPU cores in which a process can be run. Specifically, 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. Furthermore, in order 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 during the experiments and the CPU frequency has been set to its maximum operable value of 1.4 GHz.

The implementations of OPC UA considered in the experiments are reported in Table I, along with the indication of the adopted programming language (PL). For the open source implementations (available through the popular Github

TABLE I
OPC UA Implementations.

OPC UA Implementation	Type	Programming Language	Commit Hash
Open62541	Open Src.	C99 / C++98	9f0c73d
FreeOPC UA C++	Open Src.	C++11	da2b76f
FreeOPC UA Python	Open Src.	Python	83fb9ea
Prosys Java v4.3.0-1075	Proprietary	Java	–

platform) we also provide the commit hash of the sources at the time the experiments were performed.

All the listed protocol stacks work natively on Raspberry Pi boards, and consequently their setup procedures have not involved any further software adaptation. However, they are conceptually different. In particular, as can be seen, two out of four stacks are implemented by means of compiled languages (C/C++), whereas the other ones are implemented in, respectively, Python and java, which are high level interpreted languages. As a consequence the analysis of their behaviors reveals necessary to provide useful insights for the applications that use them. In this direction, since the outcomes of the experiments also depend on the adopted development environment, the most relevant technical details are summarized below:

- Python version 3.5.3;
- openJDK 1.8;
- glibc 2.23;
- gcc version 6.3.0;
- gcc optimization option: -O3 -s.

Finally, it has to be pointed out that other valuable OPC UA implementations are available (either free of charge or commercial), that could have been considered in our work. For example Eclipse Milo, ASNeG OPC UA, OpenOpcUa, High Performance OPC UA, to mention some. To this regard, an interesting overview of such solutions is provided in [23]. However, the aim of our analysis is to investigate the behavior of OPC UA in the context of IIoT based measurement systems, not to provide a comparison of different OPC UA implementations. For this reason, we selected among the available solutions those listed in Table II paying attention to their diversity, so that the proposed assessment can result adequately comprehensive.

V. Measurement Results and Analysis

The objective of the measurements is to assess the behavior of the diverse different OPC UA implementations focusing on performance figures that are of interest for IIoT-based measurement instruments and applications. Specifically, we addressed the CPU usage, the power consumption and the task execution times. The latter indicator is of particular significance in the application context of this paper, since it has reflects the overall latency with which measurement data can be collected at the client, and is hence a meaningful indicator of the intrinsic capability of the system to sustain real-time measurement analysis over networks [8].

A. CPU Usage

A first set of outcomes is resumed in Table II, which shows the statistics about the CPU usage for the three considered implementations.

As can be seen, 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 one highlighted a rather higher utilization compared to the other ones. However, this is not surprising since FreeOPC UA Python and Prosys Java are based on interpreted languages,

TABLE II
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%	–	–
Prosys Java	50.9%	–	–

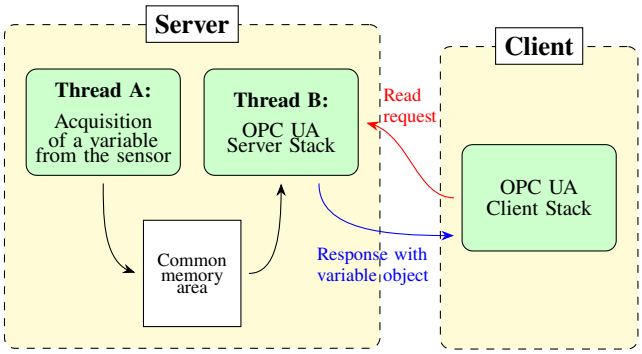


Fig. 3. Test Task for OPC UA.

that are certainly less efficient. It is interesting to note that, for the Open62541 implementation, the subdivision of the used resources of the CPU is slightly unbalanced towards the Kernel Space, while for FreeOPC UA C++ we have a subdivision almost at 50%. Unfortunately, values for FreeOPC UA Python are not available, because the tool `Perf`, with which the analysis was performed, does not support measurements of the stack of interpreted languages.

B. Read and Write Services

The experiments we carried out to test the OPC UA read and write services were based on a purposely developed test OPC UA task, with which an integer variable stored in the OPC UA server is read by the OPC UA client. In a first session, we focused on the open source implementations. In this task, the server implements two separate threads as shown in Figure 3. 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 the test task, the OPC UA client cyclically reads the value of the variable stored on the server. This is accomplished by a read request issued by the client, to which the server answers in agreement with the OPC UA protocol rules.

The outcomes relevant to the CPU usage for the read and write services are reported in Table III. The table refers to context switches, CPU migrations and total number of CPU cycles to complete the execution of 100.000 consecutive OPC UA test tasks. These outcomes are common indices exploited to determine the efficiency of a program, where high values indicate poor optimization and therefore long execution times.

The results are in good agreement with those shown in Subsection V.A. Also in this case, both the compiled implementations have comparable values, whereas both the Python and Java based ones show much higher values regarding in particular the number of CPU cycles.

TABLE III
CPU USAGE FOR THE OPC UA TEST 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	$533 \cdot 10^9$
Prosys Java	–	–	$174 \cdot 10^9$

To assess the performance of the read and write services, we measured the *task execution time*, T_s, defined as the time necessary to complete one instance of the OPC UA test task described in Figure 3. Specifically, T_s represents the time that elapses between the read request of the client, T_{req}, and the time at which it actually receives the OPC UA object containing the variable, T_{res}.

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

In the experiments, T_s has been measured by a direct access to the content of the *Cycle Counter Register* (CCR), an internal CPU register implemented within ARM processors, which is a counter of the processor clock cycles. This design choice is significant to improve the accuracy relevant to the measurement of the task execution time, because accessing the CCR register requires only one CPU cycle [24], hence introducing a negligible impact on the evaluation of the time T_s. The OPC UA test task was run continuously, meaning that a new instance of the task was started immediately after the conclusion of the former one. In the Ethernet configuration, the selected transmission rate was 100 Mbit/s whereas, for the Wi-Fi one, we chose the IEEE 802.11g mode, with transmission rate dynamically selected by the multi–rate support feature provided by such protocol[1]. For each experimental session, $N = 100.000$ measurements of the execution time have been collected and analyzed. Furthermore, all the components of the experimental set–up were located sufficiently close to each other to ensure, particularly for the Wi-Fi configuration, a high success probability in packet delivery. This has been subsequently confirmed by the traffic analysis we carried out, that showed a very low number of packet retransmissions and losses, with an average of about 3.6%.

The statistics of the execution time for the OPC UA test task are reported in Table IV for the case of non – isolated CPU and, respectively, in Table V for the isolated one.

At a first glance, the beneficial effect of introducing CPU isolation appears clear. Indeed, as shown in Table V, all mean values decrease when such a feature is used. The behavior of standard deviations is similar, with an exception relevant to the Open62541 implementation. In this case, a slight increase (from 12.56 to 12.76 µs) is observed when switching from the non–isolated case to the isolated one. We believe this aspect may be explained by the low average execution time of the Open62541 implementation. Indeed, we have checked that only the 30% of the CPU time was necessary to execute the Open62541 stack. Thus, both the task execution time and its variability, are mostly determined by the execution times of unbounded kernel threads (e.g. those concerned with the TCP/IP protocol suite, the network drivers, etc.), as well as by the network transmission times, that do not benefit from the CPU isolation.

[1]Please note that the multi–rate support feature could not be disabled on Raspberry Pi boards.

TABLE IV
STATISTICS OF THE EXECUTION TIME FOR THE OPC UA TEST TASK – NON–ISOLATED CPU.

		Execution Time T_s [µs]			
		Generic OS		RT OS	
		Mean	Std	Mean	Std
Ethernet					
	Open62541	312.83	12.56	382.48	24.44
	FreeOPC UA C++	377.30	4.54	467.59	15.01
	FreeOPC UA Python	736.79	7.44	778.43	20.63
Wi-Fi					
	Open62541	2036.12	501.60	3765.62	924.52
	FreeOPC UA C++	2063.38	488.10	3869.62	907.36
	FreeOPC UA Python	9274.24	750.59	12824.94	3901.25

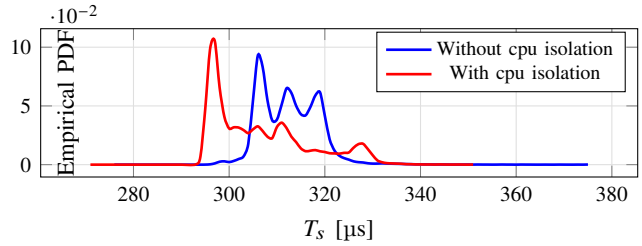
TABLE V
STATISTICS OF THE EXECUTION TIME FOR THE OPC UA TEST TASK – ISOLATED CPU.

		Execution Time T_s [µs]			
		Generic OS		RT OS	
		Mean	Std	Mean	Std
Ethernet					
	Open62541	306.67	12.76	368.78	10.13
	FreeOPC UA C++	374.74	3.32	457.60	11.32
	FreeOPC UA Python	711.27	6.52	735.84	9.69
Wi-Fi					
	Open62541	1950.29	460.55	3431.68	886.28
	FreeOPC UA C++	2017.71	457.51	3656.67	902.52
	FreeOPC UA Python	9031.62	713.10	12463.11	3807.16

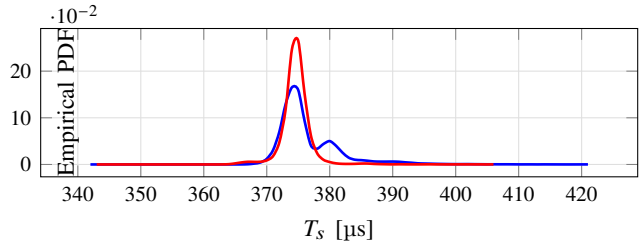
The results shown in Table V are confirmed by the probability density functions obtained experimentally (empirical probability density function, EPDF), reported for the Ethernet configuration, respectively, in Fig. 4 (generic operating system) and in Fig. 5 (RT operating system). As can be seen, the CPU isolation improves the EPDF shapes in most cases.

The tests carried out on the Wi-Fi configuration revealed similar trends, as shown in both Table IV and Table V and in both Fig. 6 and Fig. 7. In particular, the benefits brought by the CPU isolation are evident. As can be seen, both mean and standard deviation are greater with respect to the Ethernet case. This is due to the longer packet transmission times as well as to the possible retransmissions (interleaved by random backoff times) that could be necessary to successfully deliver a packet. Also, as discussed in [25], the task execution time is further negatively influenced by the Access Point which may introduce additional, non negligible, delays and randomness.

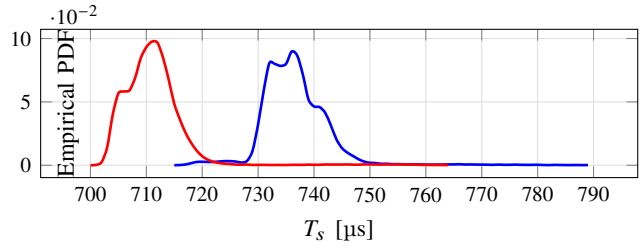
The introduction of the RT operating system, in general,



(a) Open62541

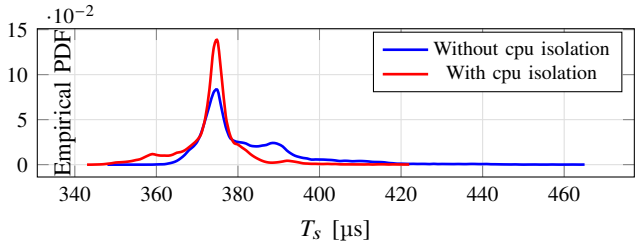


(b) FreeOPC UA C++

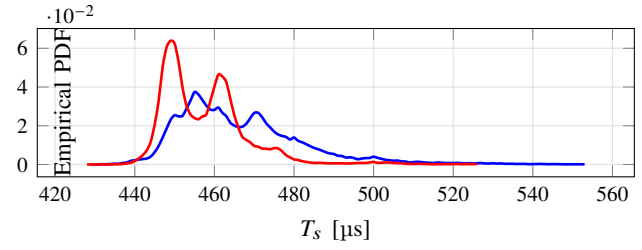


(c) FreeOPC UA Python

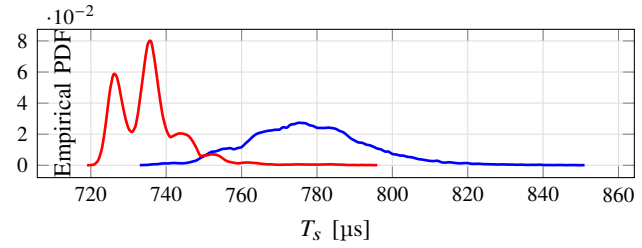
Fig. 4. Generic OS: EPDF of the execution time of the OPC UA test task for the Ethernet configuration. Blue line: configuration without CPU isolation. Red line: configuration with CPU isolation enabled, where both server and client are forced to run on the isolated CPU.



(a) Open62541



(b) FreeOPC UA C++



(c) FreeOPC UA Python

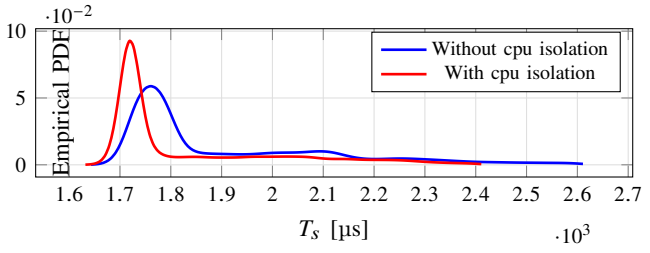
Fig. 5. RT OS: EPDF of the execution time of the OPC UA test task for the Ethernet configuration. Blue line: configuration without CPU isolation. Red line: configuration with CPU isolation enabled, and where both server and client are forced to run on the isolated CPU.

worsened the behavior of the OPC UA test task execution time, as can be evinced from the statistics and the EPDF reported above. Actually, the mean values are higher for all the OPC UA implementations, with respect to the correspondent cases in which the generic OS is used. The same happens for the standard deviation, with the unique exception of the Open62541 implementation (in this case, the value decreases from 12.76 to 10.13 µs when the RT OS is used). However, as already pointed out, these values, for the Open62541 implementation, are mostly influenced by the times necessary to execute unbounded kernel threads.

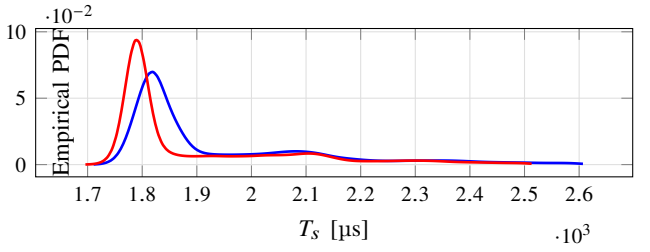
Although the worsening observed when the RT operating system is used may seem surprising, it may be explained making some considerations about the introduction of the Linux real–time extension. Actually, as can be seen from Table II, all the considered implementations of the OPC UA protocol stack make extensive use of the kernel functions, especially those concerning network connectivity. Nonetheless, the real-time patch makes some parts of the kernel preemptable, thus leaving up more space for executing instructions in the user space. Thus, the stack execution could be interrupted more frequently, resulting in longer execution times and greater jitter. Furthermore, as reported in [26], the application of the Linux real–time extension has negative effects on the throughput of the communication interface. This is confirmed by the traffic analysis that showed, in the worst case, an increment of 2.8 times of packet retransmissions.

Focusing on the stack implementations, the obtained results show that both the compiled versions, Open62541 and FreeOPC UA C++, are characterized by comparable average values of the OPC UA test task execution time. Conversely, the average T_s is much higher (about doubled) for the FreeOPC UA Python implementation, as it was expected since Python is an interpreted language. This aspect is exacerbated for the Wi-Fi configuration. As far as the standard deviation is concerned, with the Ethernet configuration, Open62541 presents higher values than both FreeOPC UA C++ and FreeOPC UA Python, especially as percentage of the mean, reflecting in a considerable jitter of the execution time. This feature is evident for the generic operating system, whereas it appears more vague for the RT operating system, likely due to the additional randomness introduced by this latter one. A similar consideration can be made for the Wi-Fi configuration. In this case, as can be seen, both the mean and standard deviation are increased with respect to Ethernet. However, the standard deviation values become comparable, especially for Open62541 and FreeOPC UA C++, likely as an effect of the randomness in accessing the physical medium introduced by Wi-Fi.

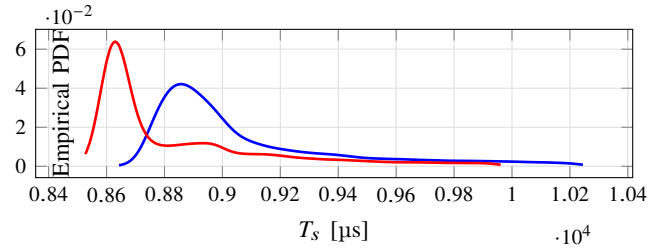
The final experiments of the `read` and `write` services



(a) Open62541

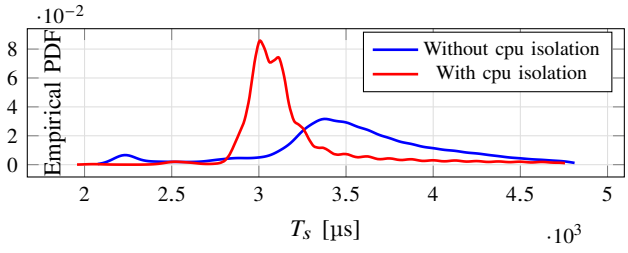


(b) FreeOPC UA C++

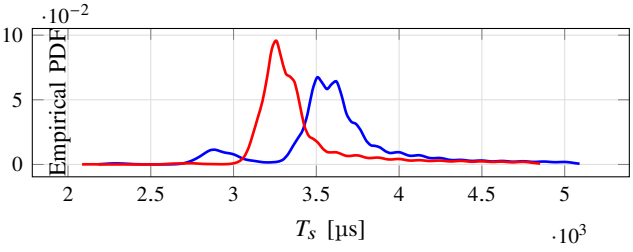


(c) FreeOPC UA Python

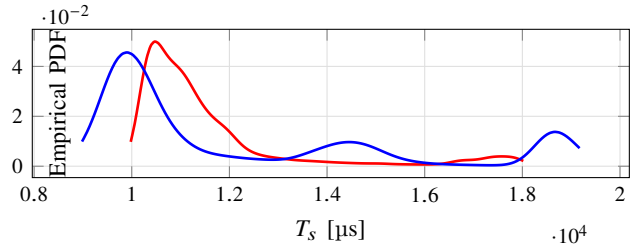
Fig. 6. Generic OS: EPDF of the execution time of the OPC UA test task for the Wi-Fi configuration. Blue line: configuration without CPU isolation. Red line: configuration with CPU isolation enabled, where both server and client are forced to run on the isolated CPU.



(a) Open62541



(b) FreeOPC UA C++



(c) FreeOPC UA Python

Fig. 7. RT OS: EPDF of the execution time of the OPC UA test task for the Wi-Fi configuration. 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.

were concerned with the Prosys Java implementation. In this case, due to space limitations we considered only the (most meaningful) case of generic OS over Ethernet. The EPDF of the test task execution time is shown in Fig. 8, whereas the statistics are reported in Table VI. As can be seen, both mean and standard deviation are considerably higher with respect to the open source implementations, confirming the trend already observed when an interpreted language (Java in this case) is used. Also, the EPDF shown in Fig. 8 puts in evidence the high variability of T_s, which is only partially mitigated by the CPU isolation.

TABLE VI
STATISTICS OF THE EXECUTION TIME FOR THE OPC UA TEST TASK FOR THE PROSYS JAVA IMPLEMENTATION ON ETHERNET WITH GENERIC OS.

Execution Time T_s [µs]			
Without CPU Isolation		CPU Isolation	
Mean	Std	Mean	Std
2793.19	348.50	2648.31	280.13

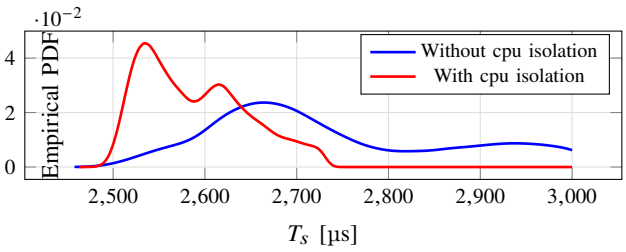


Fig. 8. EPDF of the execution time of the OPC UA test task for the Prosys Java implementation on Ethernet with Generic OS.

C. Subscription Services

In this session of experiments, we addressed the OPC UA subscription services. Similarly to the last test of the previous Subsection, we considered the case of generic OS over Ethernet. Moreover, we focused on only one open source implementation, namely Open62541, and on the Prosys Java proprietary solution. Notably, Open62541 revealed the most effective open source implementation among those addressed in the previous subsection. Also, as discussed in [27] and [28], Open62541 is well supported and suitable for applications in the IIoT scenario.

Subscription services are totally asynchronous and let the server to notify changes in its nodes. With refer to Fig 3, we associated a monitoring item to the sensor value simulated by Thread A. In this way, the server checks the data source with a period defined by the "Sampling Interval" and, when the publishing interval elapses, it sends a "Publish Response" message containing the notification of data change. Then the

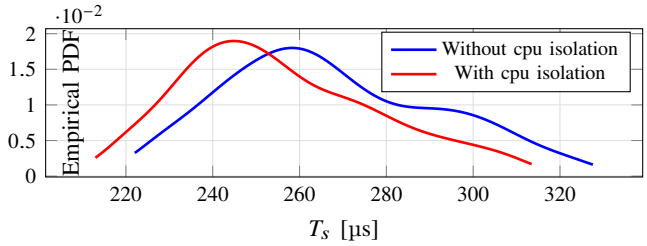


Fig. 9. EPDF of the delivery time for the subscription service – Open62541 implementation over Ethernet with Generic OS.

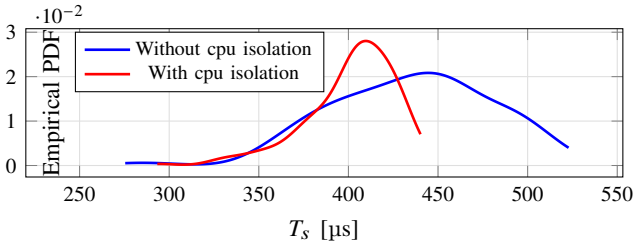


Fig. 10. EPDF of the delivery time for the subscription service – Prosys Java implementation over Ethernet with Generic OS..

client acquires the data and sends a "Publish Request" message that acknowledges the received data.

To assess the performance of the subscription service for the Open62541 implementation, we evaluated the *delivery time*, defined as the time employed by the client to acquire a data published by the server. The measurements have been carried out as follows. When the publish response message is ready to be transmitted, the server sets one of the GPIO pins of the Raspberry Pi. Similarly, when the client receives the message at the application level, it sets one of its GPIO pins. The interval elapsed between the generation of the two consecutive signal edges represents the delivery time. In this experiment, 6000 measurements have been collected and analyzed.

On the server, both the publishing interval and sampling interval have been set to 100 ms, which is the minimum selectable value on the default implementation of Open62541, whereas the update interval of Thread A has been set to 30 ms. The signal edges of the GPIO have been acquired by a logic analyzer with a sample rate of 24 MHz.

TABLE VII
STATISTICS OF THE DELIVERY TIME FOR THE SUBSCRIPTION SERVICE OPEN62541 IMPLEMENTATION OVER ETHERNET WITH GENERIC OS.

Delivery Time [µs]			
Without CPU Isolation		CPU Isolation	
Mean	Std	Mean	Std
267.77	24.55	255.68	22.59

The EPDF of the delivery time is shown in Fig. 9, whereas its statistics are provided in Table VII. Although an effective comparison with the results of the read and write services reported in Subsection V-B can not be done (the adopted measurement techniques had to be necessarily different), it may be observed that the delivery time has, on average, lower values than the task execution time of the read service. This can be explained considering that, with the subscription services, a single message transmission by the server is sufficient to make the measured data available to the client. Conversely, the jitter increases with respect to the read service for the Open62541 implementation. For the same considerations made in Subsection V-B, we believe this is most likely due to the execution of unbounded kernel threads that may heavily impact on the behavior of the delivery time. As a final observation, the CPU isolation is beneficial also in this case.

Moving to the Prosys Java implementation, an analogous procedure to evaluate the delivery time could not be used, since for such a proprietary implementation, it has not been possible to modify the stack core to set the GPIO pins. Thus, we resorted to analyze the time stamps of the messages exchanged over the network, acquired with Wireshark. The delivery time has been hence determined as the time interval between the generation of the publish response message and the arrival of the acknowledgment generated by the client.

The EPDF of the delivery time is shown in Fig. 10, whereas the statistics are reported in Table VIII

As can be seen, also in this case, the subscription service appears more efficient than the read one (although, as already pointed out for the Open62541 implementation, an effective comparison can not be made). Indeed, the mean time necessary to acquire a measurement data by the client is definitely lower than that shown in Table VI for the read service. Also, the beneficial effect of the CPU isolation is evident.

TABLE VIII
STATISTICS OF THE DELIVERY TIME FOR THE SUBSCRIPTION SERVICE PROSYS JAVA IMPLEMENTATION OVER ETHERNET WITH GENERIC OS.

Execution Time T_s [µs]			
Without CPU Isolation		CPU Isolation	
Mean	Std	Mean	Std
434.33	47.57	401.53	24.84

D. PubSub Communication Profile

In a final experimental session we addressed the OPC UA PubSub communication model. Currently, PubSub is only supported by Open62541 and, although it provides all the main methods, it is an experimental version still under heavy development. Similarly to the subscription services carried out for Open62541, the server has been configured to publish a message every 100 ms. These messages were sent to a message oriented middleware via UDP multicast transmissions using the UADP encoding. As for Open62541, we measured the delivery time as the difference between the generation of the two consecutive signal edges on the GPIO pins. The EPDF of the delivery time and its statistics are reported respectively in Fig. 11 and Table IX. Contrary to expectations, with the use of PubSub, there is a remarkable worsening of performance with respect to the subscription services. Indeed both mean and jitter values increase considerably. This is an unexpected result, since the PubSub communication model has been conceived to minimize protocol overhead and hence to reduce the transmission times between publishers and subscribers. Thus, there are not logical explanations of these outcomes.

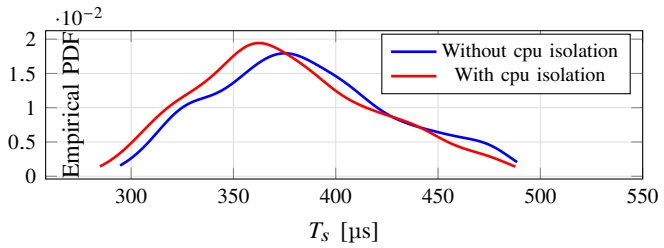


Fig. 11. EPDF of the pubsub transmission time for the Open62541 implementation on Ethernet with Generic OS.

We suppose the problem is due to the fact that the PubSub implementation of Open62541 is still under development.

TABLE IX
STATISTICS OF THE DELIVERY TIME FOR THE PUBSUB COMMUNICATION PROFILE OVER ETHERNET WITH GENERIC OS.

Delivery Time T_s [µs]			
Without CPU Isolation		CPU Isolation	
Mean	Std	Mean	Std
384.77	43.93	376.28	43.69

E. Power Consumption

One of the main issues concerned with, possibly mobile, battery powered 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 their power consumption and, consequently, extend the battery lifetime [29]. In the Raspberry PI boards used in the experimental setup, 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 [30]. Such an approach, clearly, represents an optimal trade-off between performance and power consumption in a generic processing system.

We carried out an analysis of the DVFS impact on both power consumption and performance for the experimental setup considered so far. The first tests have been performed using the Open62541 stack, on the Wi-Fi configuration, with the generic operating system and without CPU isolation using read/write services. In detail, we measured the current consumption on the client side, as well as the time necessary to complete the experimental session described in Subsection V.B, that comprised the execution of $N = 100.000$ OPC UA test tasks.

The circuit implemented for current measurement is described in Figure 12. The Raspberry Pi was powered by a stabilized power supply, providing a 5 V continuous voltage. The adsorbed current was measured using a Hall effect sensor whose sensitivity is 185 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 kHz. Each time the OPC UA test task is started, the Raspberry Pi rises a signal triggering a new acquisition of the current level, which is also timestamped for further analysis.

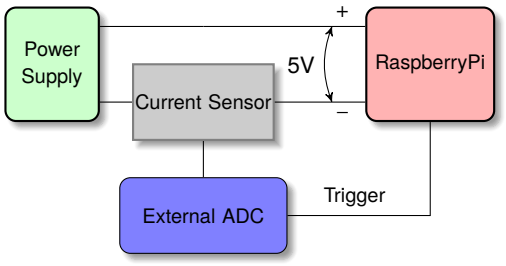


Fig. 12. Setup adopted for current measurements.

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

	Current [A]			**Session completion time [ms]**
	Mean	Std	Idle	
Governor Enabled	0.4175	0.0587	0.3928	235924
Governor Disabled	0.4767	0.0760	0.4462	215259

The results of this new set of experiments are summarized in Table X, which reports the statistics of the current consumption for the two cases in which the governor was, respectively, enabled and disabled. The table also shows the current consumption in idle state (i.e. while the experimental session was not carried out), for comparison.

As can be seen, enabling the governor leads to a slight decrease of the current consumption, for both average and standard deviation values. However, the time necessary to complete the experimental session increases, by almost 10%, as a result of the continuous CPU frequency adjustments caused by the workload variations. Thus, at a first glance, it might not be worth to maintain the governor enabled, since the benefits achieved in term of power savings may result nullified by the performance degradation. However, a decision in this direction has to take into consideration other aspects, such as the specific devices adopted and the performance requirements.

A further observation can be made with respect to the current consumption in idle state. Table X clearly shows only a limited increase of the current consumption, when moving from this state to that in which experiments were executed, regardless of the governor status (enabled or disabled). This may be explained considering that Open62541 uses very low CPU resources that, evidently, are not sufficient to imply a remarkable variation in current consumption, as confirmed by the results presented in Table II.

Finally, we carried out additional tests for the subscription service set and PubSub communication profile. In both cases, we used we measured the power consumption for the Open62541 implementation, over Ethernet, with the governor disabled. The analysis, actually, has not revealed any significant change with respect to the former measurements.

VI. Conclusion and Future works

In this paper we considered the case of IIoT measurement applications and proposed the adoption of OPC UA protocol to enable a seamless interoperability when heterogeneous sources of measurement data coexist sharing information over the network. We focused on four widespread implementations of the protocol, to analyze their impact on a networked measurement system, mostly in terms of latency and power consumption. A reproducible and effective measurement setup has been designed that allowed to carry out a thorough assessment, thus providing a rather complete characterization of OPC UA for network-based measurement instrumentation systems.

Meaningful results have been obtained, allowing also to provide some interesting implementation guidelines. For instance, it has been verified that the use of a real-time operating system does not bring specific advantages whereas, in general, the best performances are achieved with a generic operating system exploiting the CPU isolation for the measurement application.

Furthermore, in compliance with modern IIoT-based applications, we considered the case of battery powered wireless measurement systems, thus providing some valuable insights about the expected power consumption in some selected and relevant cases. Indeed, the measurement campaign highlighted that the DVFS feature should be enabled, allowing for lower power consumption without compromising the performance.

The current work opens up to future analysis. For instance, power consumption depends also on intrinsic parameters of the accumulator and on environmental conditions, hence requiring a more extensive experimental campaign focused on mobile battery powered measurement instruments. In addition, the proposed experimental setup for latency analysis seems to be overkill for small integrated smart sensors. Hence, we plan to test the framework on low power embedded devices, like widespread microcontrollers with no operating systems.

References

[1] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *J. Ind. Inf. Integr.*, vol. 6, pp. 1–10, Jun. 2017.

[2] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial internet of things: Challenges, opportunities, and directions," *IEEE Trans. Industr. Inform.*, 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," *Proc. IEEE*, vol. 107, no. 6, pp. 944–961, Jun. 2019.

[4] D. Grimaldi and M. Marinov, "Distributed measurement systems," *Measurement*, vol. 30, no. 4, pp. 279–287, Dec. 2001.

[5] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Boston: Morgan Kaufmann, 2011.

[6] G. Y. Tian, "Design and implementation of distributed measurement systems using fieldbus-based intelligent sensors," *IEEE Trans. Instrum. Meas.*, 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 *Proc. IDAACS*, Rende, Italy, 2009, pp. 673–678.

[8] B. Ooi and S. Shirmohammadi, "The potential of IoT for instrumentation and measurement," *IEEE Instrum. Meas. Mag.*, vol. 23, no. 3, pp. 21–26, May 2020.

[9] D. Bruckner, M.-P. Stanica, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, and T. Sauter, "An Introduction to OPC UA TSN for Industrial Communication Systems," *Proc. IEEE*, vol. 107, no. 6, pp. 1121–1131, Jun. 2019.

[10] International Electrotechnical Commission, *IEC 62541: OPC unified architecture - Part 1: Overview and concepts*. IEC, 2016.

[11] M. Rizzi, P. Ferrari, A. Flammini, and E. Sisinni, "Evaluation of the IoT LoRaWAN Solution for Distributed Measurement Applications," *IEEE Trans. Instrum. Meas.*, vol. 66, no. 12, pp. 3340–3349, Dec. 2017.

[12] H. Lee and K. Ke, "monitoring of large-area iot sensors using a lora wireless mesh network system: Design and evaluation," *IEEE Trans. Instrum. Meas.*, no. 9, pp. 2177–2187.

[13] R. Palisetty and K. C. Ray, "FPGA Prototype and Real Time Analysis of Multiuser Variable Rate CI-GO-OFDMA," *IEEE Trans. Instrum. Meas.*, vol. 67, no. 3, pp. 538–546, Mar. 2018.

[14] V. Bianchi, A. Boni, S. Fortunati, M. Giannetto, M. Careri, and I. De Munari, "A Wi-Fi Cloud-Based Portable Potentiostat for Electrochemical Biosensors," *IEEE Trans. Instrum. Meas.*, vol. 69, no. 6, pp. 3232–3240, Jun. 2020.

[15] Y. Liao and H. Lai, "Investigation of a Wireless Real-Time pH Monitoring System Based on Ruthenium Dioxide Membrane pH Sensor," *IEEE Trans. Instrum. Meas.*, vol. 69, no. 2, pp. 479–487, Feb. 2020.

[16] J. Cavalaglio Camargo Molano, A. Lahrache, R. Rubini, and M. Cocconcelli, "A new method for motion synchronization among multivendor's programmable controllers," *Measurement*, vol. 126, pp. 202 – 214, Oct. 2018.

[17] B. Montavon, M. Peterek, and R. Schmitt, "Model–based interfacing of large–scale metrology instruments," *Proc. SPIE 11059, Multimodal Sensing: Technologies and Applications, 110590C*, Jun. 2019.

[18] S. Lee, C. Kim, and J. Lee, "Development of a Smart Sensor System Using OPC UA," in *Proc. MoMM*, 2017, pp. 220–225.

[19] I. González, A. J. Calderón, A. J. Barragán, and J. M. Andújar, "Integration of Sensors, Controllers and Instruments Using a Novel OPC Architecture," *Sensors*, vol. 17, no. 7, Jul. 2017.

[20] P. Ferrari, A. Flammini, S. Rinaldi, E. Sisinni, D. Maffei, and M. Malara, "Impact of Quality of Service on Cloud Based Industrial IoT Applications with OPC UA," *Electronics*, vol. 7, no. 7, p. 109, Jul. 2018.

[21] A. Morato, S. Vitturi, F. Tramarin, and A. Cenedese, "Assessment of Different OPC UA Industrial IoT solutions for Distributed Measurement Applications," in *Proc. I2MTC*, Dubrovnik, Croatia, 2020, pp. 1–6.

[22] M. Damm, S.-H. Leitner, and W. Mahnke, *OPC Unified Architecture*. Springer-Verlag Berlin Heidelberg, 2009.

[23] H. Haskamp, M. Meyer, R. Möllmann, F. Orth, and A. W. Colombo, "Benchmarking of existing OPC UA implementations for Industrie 4.0-compliant digitalization solutions," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 589–594.

[24] "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile". Accessed: 2020-10-05. [Online]. Available: https://developer.arm.com/documentation/ddi0487/fb/

[25] L. Seno, F. Tramarin, and S. Vitturi, "Performance of Industrial Communication Systems - Real Application Contexts," *IEEE Ind. Electron. Mag*, vol. 6, no. 2, pp. 27–37, Jun. 2012.

[26] "Raspberry Pi: The N-queens Problem (benchmark) Preempt-RT vs. Standard Kernel". Accessed: 2020-10-05. [Online]. Available: https://lemariva.com/blog/2018/04/raspberry-pi-the-n-queens-problem-performance-test

[27] F. Palm, S. Gruener, J. Pfrommer, M. Graube, and L. Urbas, "Open source as enabler for OPC UA in industrial automation," in *Proc. ETFA*, Luxembourg, Luxembourg, 2015, pp. 1–6.

[28] J. Pfrommer, "Semantic interoperability at big-data scale with the open62541 OPC UA implementation," in *Proc. Workshop Interoperability and Open-Source Solutions for the Internet of Things*, Stuttgart, Germany, 2016, pp. 173–185.

[29] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van Der Wijngaart, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE J. of Solid-St. Circ.*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[30] M. P. Karpowicz, "Energy-efficient CPU frequency control for the Linux system," *Concurr. Comput.*, vol. 28, no. 2, pp. 420–437, Feb. 2016.