Bottleneck Identification and Performance Modeling of OPC UA Communication Models

Andreas Burger
ABB Corporate Research Center
Ladenburg, Germany
andreas.burger@de.abb.com

Heiko Koziolek
ABB Corporate Research Center
Ladenburg, Germany
heiko.koziolek@de.abb.com

Julius Rückert
ABB Corporate Research Center
Ladenburg, Germany
julius.rueckert@de.abb.com

Marie Platenius-Mohr
ABB Corporate Research Center
Ladenburg, Germany
marie.platenius-mohr@de.abb.com

Gösta Stomberg
Technische Universität Darmstadt
Germany
sebastian_goesta.stomberg@stud.
tu-darmstadt.de

ABSTRACT

The OPC UA communication architecture is currently becoming an integral part of industrial automation systems, which control complex production processes, such as electric power generation or paper production. With a recently released extension for pub/sub communication, OPC UA can now also support fast cyclic control applications, but the bottlenecks of OPC UA implementations and their scalability on resource-constrained industrial devices are not yet well understood. Former OPC UA performance evaluations mainly concerned client/server round-trip times or focused on jitter, but did not explore resource bottlenecks or create predictive performance models. We have carried out extensive performance measurements with OPC UA client/server and pub/sub communication and created a CPU utilization prediction model based on linear regression that can be used to size hardware environments. We found that the server CPU is the main bottleneck for OPC UA pub/sub communication, but allows a throughput of up to 40,000 signals per second on a Raspberry Pi Zero. We also found that the client/server session management overhead can severely impact performance, if more than 20 clients access a single server.

KEYWORDS

Performance evaluation, performance modeling, OPC UA, client/server, pub/sub, bottleneck identification, dynamic multicast filtering, resource-constrained devices, M2M communication

ACM Reference Format:
Andreas Burger, Heiko Koziolek, Julius Rückert, Marie Platenius-Mohr, and Gösta Stomberg. 2019. Bottleneck Identification and Performance Modeling of OPC UA Communication Models. In *Proceedings of ACM International Conference on Performance Engineering (ICPE'19).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.475/123_4

ICPE'19, April 2019, Mumbai, India

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

Open Platform Communications Unified Architecture (OPC UA) (IEC 62541) is a service-oriented communication architecture for machine-to-machine (M2M) communication in industrial automation [8]. OPC UA servers typically run on x86-based CPUs and receive data from sensors and transfer to actuators used in electric power plants, chemical plants, or steel mills. OPC UA clients originally are workstations for human plant operators or other server-side systems. They use the information exposed by OPC UA servers to configure and operate a production process. OPC UA servers expose static data (e.g., configuration parameters, identification information), dynamic data (e.g., sensor readings, setpoints, alarms, events), as well as services. OPC UA communication runs on IP networks and supports a combination of different application-level protocols, such as UA Binary, HTTPS, SOAP/XML, MQTT[1], and AMQP[2]. Due to its powerful information modeling capabilities, OPC UA was hailed as a preferred communication standard for the Industrie 4.0 [13].

While OPC UA was initially designed for server to workstation communication, the OPC Foundation has recently enhanced it for field device communication [9], which requires short cycle times on resource-constrained devices. This so-called OPC UA publish/subscribe (pub/sub) communication relieves devices from session management overhead and is potentially suitable for sub-millisecond control loops using multicast UDP communication [2]. For example, if the pressure in a pipe of a chemical plant crosses a pre-defined threshold, a sensor must inform an embedded controller via OPC UA with a very short delay to open a valve to avoid damage to the equipment. Due to the novelty of the OPC UA pub/sub specification, experiences and guidelines on how to apply OPC pub/sub for different industrial automation application contexts is missing. Performance bottlenecks of pub/sub communication are unclear. A comparison between client/server and pub/sub communication is missing, and automation vendors have little guidance in sizing hardware environments appropriately.

Former evaluations of OPC UA focused on client/server communication and often investigated round-trip times for different numbers of communicated signals [4, 5, 8]. They also analyzed added latency for encrypted communication, but never constructed

[1] http://mqtt.org
[2] http://amqp.org

predictive performance models. Recent studies [2, 10] analyzed the jitter of communication delays for OPC UA pub/sub communication and find that a 50 ns jitter is achievable by exploiting features for time-sensitive networks (TSN). Our own former work specifically quantifies the encoding overhead for OPC UA pub/sub communication [7] and provides initial results on server-side CPU utilizations. None of these works systematically identified performance bottlenecks or created a predictive performance model.

The contributions of this paper are 1) an identification of OPC UA pub/sub bottlenecks via performance measurements on resource-constrained devices and 2) a performance model that allows predicting the CPU utilization of OPC UA servers given a specific application profile. We created a measurement testbed involving multiple computing devices and network switches, which we configured to carry out the required dynamic multicast filtering for pub/sub communication correctly. We used a commercial OPC UA SDK and created OPC UA servers and clients in C++ for our measurement experiments. We defined test scenarios and quantified the server-side session management overhead in client/server communication. Using the results, we created a performance model based on linear regression and predicted the CPU utilization for two typical application scenarios.

We found that the server CPU was the bottleneck in all analyzed scenarios. A Raspberry Pi Zero (as example for a cheap embedded device) was able to send up to 40,000 signals per second via OPC UA pub/sub and up to 20,000 signals per second for OPC UA client/server. The session management overhead for up to 30 connected clients incurred up to 45 percent CPU utilization. We also found that the memory and network utilization was low, as well as the overhead for encrypting the communication. The performance model showed a good accuracy and led to the conclusion that the Raspberry Pi Zero is powerful enough for the analyzed application scenarios.

This paper is organized as follows. Section 2 explains the basics of OPC UA and Section 3 reviews past work on OPC UA performance analysis. Section 4 introduces our measurement testbed and implementations. Section 5 reports the performance measurement results in terms of CPU, memory, and network utilization. Section 6 describes the constructed performance model and its application in the two application cases. Section 7 discusses limitations of our approaches, before Section 8 concludes the paper.

2 BACKGROUND

OPC UA is a platform-independent, service-oriented architecture that is typically used for machine-to-machine communication in industrial automation. OPC UA servers provide an object-oriented information model that allows a client to access live values and properties of industrial devices, such as sensors and actuators. OPC UA supports different transport protocols, such as HTTPS, SOAP, MQTT, and a self-defined OPC UA Binary format for fast communication. In the past, automation systems used OPC UA mainly between operator stations (PCs) and devices (embedded systems) and used industrial fieldbuses for latency critical communication between controllers and devices.

OPC UA's original communication model is client/server-based and was standardized in 2008. Fig. 1 depicts a high-level overview of this model. An OPC UA server exposes an object-oriented address space (i.e., instance of the information model), which for example may contain periodically updating temperature values of a temperature sensor in an industrial boiler. Each client may either read individual data items in the address space or register subscriptions for periodic updates with the server. The server then samples the address space for updated values according to the client-configured update interval (e.g., 500 ms) and sends the data to the client via a TCP connection. The server manages a dedicated subscription for each connected client, as clients may be interested in different data items and have different required updated intervals. That causes overhead in terms of CPU and memory usage, increasing with a higher number of clients.

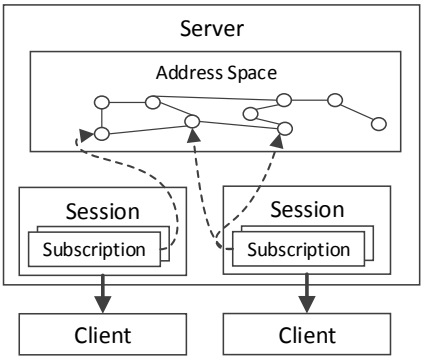

Figure 1: OPC UA slient/server subscription communication model (based on [9])

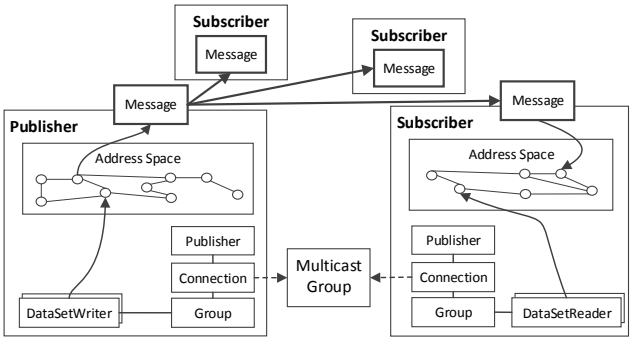

Figure 2: OPC UA Broker-less pub/sub communication model (based on [9])

In 2018, the OPC Foundation standardized an additional pub/sub communication model for OPC UA as Part 14 of the specification, which completely decouples the communication partners. Using OPC UA UDP, which is one of the new broker-less pub/sub modes, an OPC UA server now can group data items from the address space into DataSets and publish them with a configured update interval to UDP multicast groups (see Fig. 2). OPC UA clients interested in the data may subscribe to these data items by registering for updates of the respective UDP multicast groups. This is completely

independent of the publishing OPC UA server, which has no information how many subscribers receive the information. In this case, the OPC UA server does not need to manage a session for each client and is relieved of respective CPU and memory usage. However, due to the UDP transfer, there is no guaranteed delivery of the messages.

For OPC UA UDP, the OPC UA pub/sub specification recommends to use switches with Internet Group Management Protocol (IGMP)[3] support to limit the distribution of multicast traffic to the interested participants as depicted in Fig. 3. Hosts use IGMP to report their IP multicast group memberships in response to peridical IGMP queries received, e.g., from a router on the local network. Routers use IGMP messages to learn which multicast groups have members on their attached networks. Switches use IGMP snooping to listen to IGMP messages from hosts and then only send multicast network messages to ports that have joined the multicast group. This dynamic filtering of multicast messages is essential, as broadcast communication (without filtering) for the types of high-frequency communication characteristics in industrial automation systems would quickly overload the network. Using the network infrastructure for this purpose instead of a dedicated broker on the application level avoids an additional point of failure and larger attack profile.

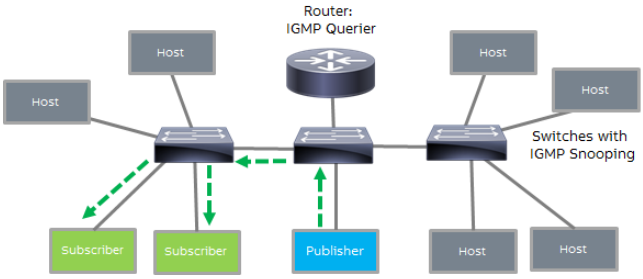

Figure 3: Dynamic Multicast Filtering using IGMP: switches only forward packets to hosts that registered for a multicast group of the publisher.

Due to the novelty of the pub/sub specification, no commercially released OPC UA SDKs support it as of now and experiences with its performance are missing. Engineers procuring network and computing hardware and setting up communication channels between hosts in industrial automation applications would benefit from knowing the performance limits of the technology and getting guidelines on when to use client/server or pub/sub communication.

3 RELATED WORK

Because the OPC Foundation released OPC UA pub/sub in early 2018 and there is still a lack of implementations, there are currently only a few performance studies available. This section first provides an overview of performance evaluations of OPC UA client/server communication and then describes three studies, which have dealt with OPC UA pub/sub communication up to now.

Mahnke et al. [8] provided the first comprehensive overview of the OPC UA technology in 2009. The book also contains a chapter with a rudimentary performance analysis, mainly targeting a comparison between classic OPC and the UA Binary and SOAP/XML communication protocols with a focus on round-trip latencies. They found that OPC UA is able to maintain the performance of classic OPC, while adding security and reliability. However, this analysis does not analyze CPU or memory utilization or a higher number of client sessions.

Cavalieri et al. [4] also analyzed OPC UA round-trip times in 2010, but took different message sizes into account. They also provided measurements about the induced delay for subscriptions in case of short update intervals and measure network bandwidth usage. Furthermore, they characterized the overhead on round-trip times induced by encryption. However, this work does not analyze different numbers of client sessions or propose a performance model.

In 2012, Fojcik et al. [5] discussed basic performance parameters for OPC and conducted performance measurements using the commercial Prosys OPC UA SDK on powerful PCs. The authors investigated the CPU utilization for up to 100 connected OPC UA clients as well as the update times for different sampling intervals. Since they used powerful hardware, the results may be hard to apply to resource-constrained devices. They also did not propose a performance model or compared client/server and pub/sub communication.

In 2016, Gruener et al. [6] created a RESTful extension for OPC UA communication and measured round-trip times for different number of read requests per session. They carried out these tests on x86 PCs, Raspberry Pi Model B devices, and an industrial fieldbus controller with 44 MHz and 16 MB RAM running μClinux. The authors also measured server throughput and found that the RESTful communication provides significantly higher throughput. However, they could not compare these results to pub/sub communication, yet.

Rocha et al. [11] did a performance comparison of quantity of used data and time spent to send and receive messages in the field of IoT and Industrial IoT (IIoT) with MQTT and OPC UA. The measurements were conducted over different servers using the Google Cloud Platform. The test implementation relied on Python3 for both protocols. The focus of the conducted measurements were on cloud-to-client communication, not in an industrial environment with OPC UA pub/sub or a comparison between client/server and pub/sub communication.

In 2018, after the release of the OPC UA pub/sub specification, the so-called "Shaper group" with industry representatives from B&R, ABB, Schneider Electric, and others released a whitepaper [2] on OPC UA communication for Time-sensitive Networks (TSN). They created a testbed with 50 devices and conducted measurements for the time synchronization between nodes. The time synchronization delay was around 100 μs with a 50 ns jitter in laboratory conditions. The whitepaper also presented a theoretical calculation on the minimum achievable cycle times with 100 MBit and 1 GBit switches. Up to 100 nodes and using a 1 GBit TSN switch with 100 bytes of payload, they predicted cycle times of around 100 μs. This was accompanied by a comparison to other Ethernet-based M2M fieldbuses (PROFINET IRT, EtherNet/IP, EtherCAT). The focus in the paper was on TSN communication, not on OPC UA

[3] https://tools.ietf.org/pdf/rfc3376.pdf

pub/sub specifics and did not compare client/server with pub/sub communication.

In a similar manner, Pfrommer et al. [10] measured jitter of OPC UA pub/sub using the open62541 SDK running on two PCs and a 1 GBit switch. They provided a demo involving Intel Atom processors and ARM Cortex M4 boards. These tests achieved a 100 μs cycle time and showed that with enabled time-based scheduling from the TSN specifications, the jitter can be reduced down to 40 ns.

Our own former work also involved OPC UA performance measurements. We introduced a reference architecture for plug-and-produce applications based on OPC UA client/server and pub/sub communication [7]. This study investigated specifically the encoding overhead at the server side for OPC UA pub/sub communication. It was found that a Raspberry Pi 3 could handle up to 800 signals per milliseconds under special conditions. However, this work did not analyze performance bottlenecks in detail, account for security overhead, multithreading implications, nor constructed a predictive performance model, which are the contributions of this paper.

4 EXPERIMENT METHOD

In this section, we describe how we carried out the experiments for our performance evaluation. This includes the derivation of research questions, the setup and configuration of a testbed, and corresponding implementations.

4.1 Research Questions

Fig. 4 depicts the potential bottlenecks for client/server and pub/sub communication. In the client/server communication model, the server has to handle the session management per-client. Thus, the server's CPU and memory utilization could become a bottleneck with an increasing number of clients, as explained in Section 2. Furthermore, the network can be a bottleneck because the data transmission increases with each client to send data to, as well as with the number of signals and the publishing rate.

In the pub/sub communication model based on OPC UA UDP, more potential bottlenecks exist. On the one hand, the network could be a bottleneck if a multicast group is overloaded or the filtering is ineffective or not working as expected. It could also be wrongly configured, such that the multicast communication becomes a physical layer broadcast. On the other hand, the CPU utilization of the subscribers could become a bottleneck in case of an overloaded multicast group as the DataSet filtering takes place on the application layer. Subscribers' CPU load is expected to depend on the DataSets received in a multicast group. The resource usage at the publisher is, however, expected to be dependent only on the number of signals and the publishing rate and not on the number of subscribers. Further factors that most likely influence the performance of the communication models are encryption and multithreading on the server side.

Motivated by the potential bottlenecks, the following research questions guided the work presented in this paper:

RQ1: How high is the CPU utilization of the publisher/ server in the pub/sub communication model compared to the client/ server model?

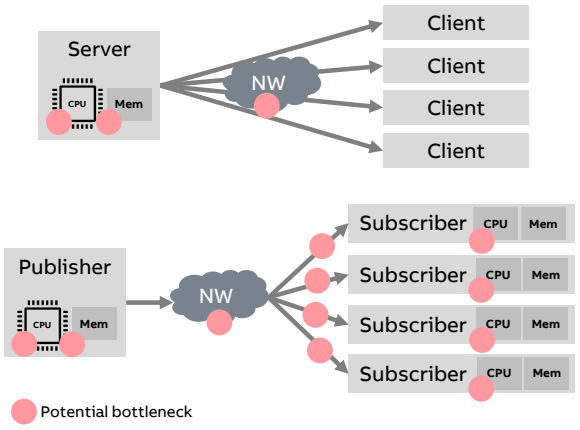

Figure 4: Potential bottlenecks in the client/server and pub/sub communication models

RQ2: How high is the memory usage of the publisher/ server in the pub/sub communication model compared to the client/ server model?

RQ3: How high is the network load in the pub/sub communication model compared to the client/server model?

RQ4: Which one is the bottleneck: CPU utilization, memory usage, or the network load?

RQ5: How much overhead in terms of CPU utilization is caused by security policies?

RQ6: What is the impact of multithreading on the server performance?

4.2 Testbed Setup and Configuration

For the execution of the performance measurements, we use a testbed consisting of three industrial-grade managed GBit network switches, as shown in Fig. 5. The switches (type Hirschmann RSP35) are connected to each other in a chain. In addition, we use Raspberry Pi devices with ARM processors running Linux (Raspbian with RT PREEMPT patch [4]), which are representative for modern industrial PCs. The server and publisher are deployed on a Raspberry Pi Zero, while the clients and subscribers are deployed on Raspberry Pi 3 devices. This setup is comparable to modern production plants. All Raspberry Pis are equipped with RTC DS3231 high precision real time clock modules.

For the pub/sub measurements, IGMPv3 snooping is enabled at the network switches together with the IGMP querier functionality with a query interval of 10 seconds [3]. The querier functionality is necessary as no router is connected to the testbed that could act as IGMP querier. In addition, we configured the ports interconnecting the switches as static query ports to ensure that IGMP membership reports of IP multicast clients arrive at all switches.

We used the Linux tool "iperf" to test the unicast and multicast switching performance and correct filtering. Without a correct IGMP configuration, the switches either broadcasted the IP multicast packets to all clients in the network (IGMP snooping not working) or did not forward the packets at all (no IGMP querier:

[4] https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/

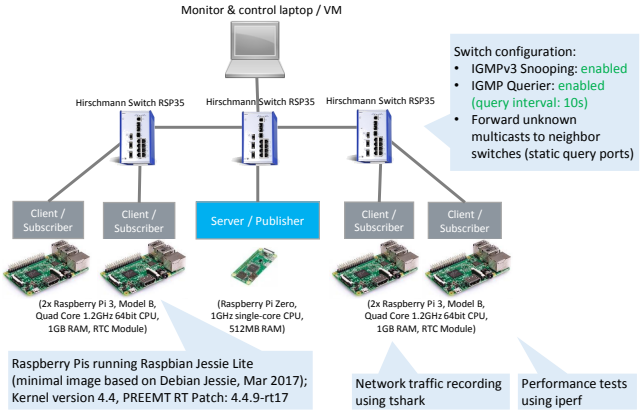

Figure 5: Testbed Setup and Configuration with Industrial Switches and different Raspberry Pis

forwarding paths could not be learned due to missing membership queries and reports). These first tests helped us to optimize the switch configuration for the following experiments. While the testbed supports executing more complex measurements, most of the results presented in this paper used just a subset of the configuration with clients being connected to only one of the switches as the network showed not to be the relevant bottleneck for our measurements.

4.3 Measurement Implementation

To answer the research questions, we developed test programs that use OPC UA functionality to simulate clients and servers, or publishers and subscribers, respectively. These programs monitor the performance metrics to be evaluated during experiments, i.e., CPU utilization, memory usage, and network load, and write them to log files. The programs support an easy variation of relevant parameters, e.g., the server's publishing rate and the number of published signals. For the client/server case, also the number of simulated clients could be configured. This way, multiple clients can be deployed on a single machine to enable larger experiments.

All components were implemented using the OPC UA C++ SDK by Unified Automation v1.5.2. As this version does not yet include any pub/sub functionality, we obtained an early prototypical pub/sub extension from Unified Automation, which was used in addition. Other SDKs can be evaluated as well using the concepts presented in this paper as soon as they support the OPC UA pub/sub specification.

4.3.1 Server Implementation. The OPC UA server representing the server component of the client/server model stores data in the server address space and communicates this data to one or more OPC UA clients running on the client component. In more detail, the address space is populated with a variable number of floating point values (floats) representing sensor readings, as well as with metrics used for the evaluation and validation of the experiments. In different scenarios, we varied the number of floats to study the influence on the performance. A loop in the server component updated the floats to new, random values to simulate the updating of signals in a typical control system scenario where signal values change due to new sensor data, for instance. The rate at which the loop is executed was made configurable to test the influence of different update rates, which corresponds to variable sampling rates of an industrial sensor. In addition to the signal updating, the program periodically writes CPU and memory measurements to the server address space.

4.3.2 Client Implementation. The client implementation consists of an OPC UA client and connects to the OPC UA server described above. It establishes an OPC UA session between client and server and client/server subscriptions (not to be confused with the pub/sub model, see Section 2) are used to communicate the data in the server address space to the client. For the experiments, the sampling and publishing rates are set to the update rate used for updating signals in the server address space. There is a callback function in the client program that is called upon the receipt of data notifications from the server. The program uses the callback to take CPU and memory measurements and to also create log entries that include the measurement values as well as additional metrics used for the validation of the experiments.

4.3.3 Pub/Sub Implementation. We developed similar programs for the pub/sub measurements. A configuration file includes the necessary publisher and subscriber settings and is read at the beginning of the program. The publisher stores data in an address space and publishes it to an IP multicast address. Again, the data in the address space consists of floats and metrics for evaluating and validating the communication model. Similar to the server program, a loop updates the float values at a specified update rate and takes CPU measurements at a defined interval. The memory measurements are only taken at the beginning of an experiment, because the memory usage was found to be stable over time for a given configuration and this way the measurement overhead could be reduced to a minimum. The subscriber program executes a callback function each time the subscriber receives messages from the publisher. As a result, CPU measurements are taken and written to a log file together with a number of other metrics. CPU and memory measurements consider all child processes of the program and additionally log the overall device CPU and memory usage. The CPU usage is calculated based on the time the CPU spends executing any of the program's processes and the total elapsed time.

4.3.4 Network measurements. We calculate inter-IP-packet time for the pub/sub measurements in the following manner: The Ethernet packets recorded by tshark [5] are first filtered for the packets that include complete IP and UDP headers. Second, the inter-arrival time of consecutive packets is calculated based on the recorded timestamps. These steps are necessary to account for the fact that in configurations with a lot of published signals, the sent UDP packets exceeded the IP packets' maximum transmission unit (MTU) size. As a result, IP fragmentation takes place, where a single UDP packet is split across multiple IP packets. Thus, only one of these packets includes a fully parsable IP and UDP header, which is used to determine the time between UDP packets, instead of IP packets, which would provide a wrong measure in this case. In the cases where the publisher is able to maintain a configured publishing

[5] https://www.wireshark.org/docs/man-pages/tshark.html

interval, the inter-arrival times of these UDP packets is expected to be equal (or smaller) than the configured publishing interval.

Furthermore, we calculate the network bandwidth based on the Ethernet frames recorded with tshark. They are filtered for the OPC UA traffic and the measurement period, which excludes the transient phases at the beginning and end of an experiment — similar to the other measures. We then sum up the size of the Ethernet frames within the remaining stable phase and divide by the length of the stable phase of the measurement to calculate the average bandwidth consumed during that time. On a smaller time scale, the actually used switching bandwidth shows spikes due to the bursty sending of packets according to the configured publishing interval. However, since the switching performance showed to be not a relevant bottleneck in any of our measurements, we decided to only report the average values. In resource-constraint scenarios with multiple communication partners on the same network, we propose to additionally study the bursts and derive worst-case aggregate bandwidth estimates in addition.

4.3.5 General Remarks. Recorded measurements from the first twenty seconds of each experiment are discarded to limit the analysis to the stable phase and ignore transient effects. All experiments were repeated at least five times to rule out random effects caused by the testbed setup and, e.g., operating system processes. We report results in the form of average values over the individual measurement repetitions.

5 PERFORMANCE MEASUREMENTS

This section presents our performance measurements results for CPU utilization, memory/network utilization, encryption overhead, and multithreading. The experiments had an update interval for each process of 100 ms.

5.1 CPU Utilization

Figure 6 shows the CPU utilization of the OPC UA server both for client/server and pub/sub communication to answer research question **RQ1**. The x-axis shows a growing number of signals (i.e., float values) delivered per second. The diagram shows 95% confidence intervals over the means of the individual measurements to provide a visual indication of the significance of differences between two observations. The individual client/server curves end with the most demanding setting for which the update intervals was still maintained.

Overall, the variation in CPU utilization for repeated measurement of any given data point (i.e. configured application profile) was small. The server CPU utilization using pub/sub communication is lower than for all cases using client/server communication. The publishing OPC UA server can transfer up to 40,000 signals per second via multicast UDP on a Raspberry Pi Zero, before the CPU is saturated. At that point, the server cannot achieve the configured update interval of 100 ms anymore (see further discussion of Fig. 7).

The server CPU utilization for client/server communication is shown for a growing signal rate (x-axis) and for a growing number of concurrently connected clients (individual curves) in Fig. 6. It was decided that, for an increasing number of clients, the signals published by the server are evenly distributed among connected clients, so that the total number of signals published by the server

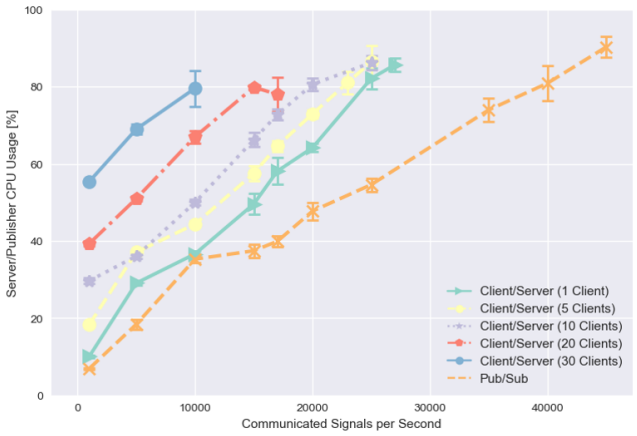

Figure 6: Server CPU utilization for OPC UA modes: pub/sub requires fewer CPU cycles than client/server and scales up to 40,000 signals per second before CPU saturation.

is kept constant. For example, for 5 clients and a total of 10,000 communicated signals, each client received 2,000 signals in each update interval. This approach helps us to characterize the session overhead induced by client/server connections.

In the client/server communication, the server CPU utilization was always higher than for pub/sub communication with the same signal rate. For a single client, the server could handle 25,000 signals per second before saturation. For 30 clients, the server could handle only a total of 10,000 signals due to the additional overhead to manage the individual sessions. More than 30 client sessions led to error messages issued by the server, therefore, these measurements could not be finalized. We assume that certain data structures of the used implementation cannot handle such a high number of sessions, at least on the Raspberry Pi Zero.

The overhead for client/server session management scales almost linearly up for higher numbers of concurrently connected clients. For 30 clients and 10,000 signals per second the CPU utilization is at 80 percent, compared with approx. 35 percent for one client. This means that the CPU spends 45 percent of its computing power only for managing the different sessions, despite communicating the same number of signals. The overhead can be attributed to context switches that the server needs when preparing the different communications with the clients.

To better understand the effect of a high server CPU utilization, Fig. 7 shows cycle times as observed by the client/subscriber when receiving signals. The time is measured as the duration between the receiving of two subsequent signal updates, which should adhere to the 100 ms update interval configured at the server/publisher. For client/server communication, the figure shows that client cycle time of 100 ms can be achieved in cases where the server CPU is not more than approximately 75 percent loaded. Upon higher utilizations, the cycle times are not met and are therefore considered invalid for the application case.

For example, for 10 clients, the client-side observed mean cycle time increases to 160 ms and the confidence interval indicates fluctuations across experiments when the server communicates

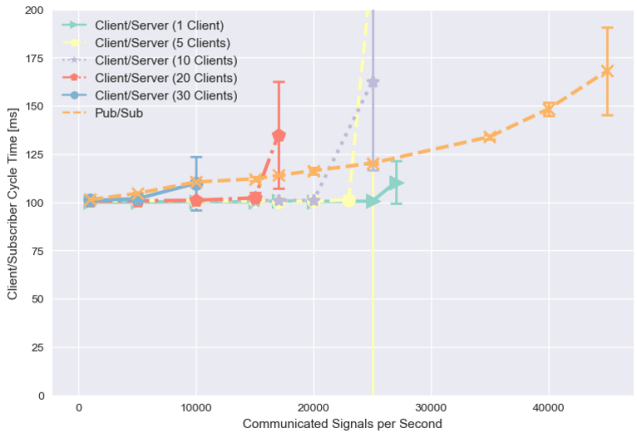

Figure 7: Cycle times observed at client/subscriber: 100 ms update interval not met for high signal rates with server CPU utilizations above 75 percent (cf. Fig. 6).

25,000 signals per second. In this case, the server is busy encoding and transferring the messages to the different clients and can no longer meet the configured updated interval. This indicates that the server CPU is the main bottleneck in the analyzed experiments and limits the number of concurrently sent signals per second.

For pub/sub communication, the experiments showed an unexpected increasing trend already starting at 5,000 communicated signals per second. However, the CPU utilization at this point was only around 20 percent. The cycle times observed at the subscriber grow for an increasing signal rate, up to more than 150 ms, which violates the application requirements. Conceptually, there is no explanation why the server cannot meet the configured update intervals since the CPU and also other resources are not saturated. To rule out effects that could be caused by subscriber implementation or the network, the same measurements were repeated, this time analyzing the inter-arrival times of the UDP packets. For this, the network traffic was captured using tshark at both the publisher and subscriber side. Interestingly, the figures of the inter-arrival times for both sides showed almost similar curves as shown in Fig. 7, indicating that the publisher already sends out the packets too slowly. Thus, the increasing subscriber cycle times for pub/sub are caused by the publisher implementation and are not a result of a bottleneck for any of the monitored resources.

Based on these observations, we assume that the beta version of the pub/sub implementation of the OPC UA SDK is responsible for this glitch, especially as it was announced to not yet be optimized for performance. We expect that this aberration will not appear in the final release version of the SDK as, conceptually, we do not see any reason for such a steadily increasing overhead, especially as cycle times are met for the more demanding client/server mode.

Summarizing, to answer **RQ1**, the maximum CPU utilization to meet the desired update intervals for both client/server and pub/sub communication was at approx. 75 percent. Pub/sub communication managed to transfer approx. 40,000 signals per second, while client/server communication managed approx. 25,000 signals per second with a single client session and accumulated up to 45 percent of additional session management overhead for up to 30 concurrent clients. In addition to the server/publisher CPU utilization results, we also conducted first measurements on the client/subscriber-side CPU utilization, which was identified as another potential bottleneck in Section 4. For brevity reasons, however, we decided to leave a detailed study and discussion to future work.

5.2 Memory and Network Utilization

To answer **RQ2**, Fig. 8 shows the memory utilization of the OPC UA server in the different experiments. The Raspberry Pi Zero has 512 MB of RAM and the same scenarios as for the CPU utilization (i.e., 100 ms update interval, floats as signals) were executed. The server memory utilization is between 11 and 15 MB in all scenarios, even for a high number client sessions and communicated signals. This implies a memory usage of less that 4 percent (**RQ2**) and shows that the main memory is not the bottleneck resource in the analyzed cases. Transfer of larger data items could cause higher memory consumption, but this is rather rare for industrial automation applications as of today.

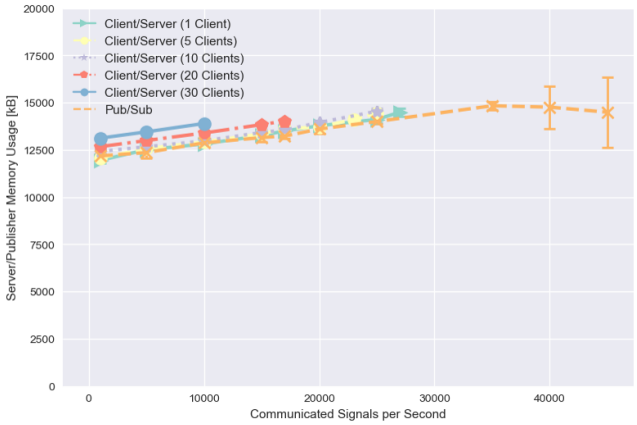

Figure 8: Memory utilization was below 4 percent of the available main memory of the device (512,000 KB).

For the network utilization (**RQ3**), Fig. 9 plots the trends for a growing number of communicated signals per second. The observed network utilization was lower than 6 Mbps in all experiments. For a 1 Gbit switch, this translates to a bandwidth usage of less than 1 percent. In case of pub/sub communication, the server outgoing network date rate is lowest in all cases, as the publisher does not deal with session overhead, but simply transfers the signals once via UDP to the configured IP multicast addresses.

In case of client/server communication, the network usage is higher but still smaller than 1 percent in all cases, considering a 1 Gbit network (**RQ3**). Thus, the network is not considered a bottleneck in the analyzed scenarios. Higher network utilizations may occur at the client side if multiple servers send as much data as they can to single clients. For pub/sub communication dynamic multicast filtering assures that packets are only sent to other nodes if they have subscribed to the respective multicast addresses. Otherwise, the packets are not forwarded by the network switch, therefore

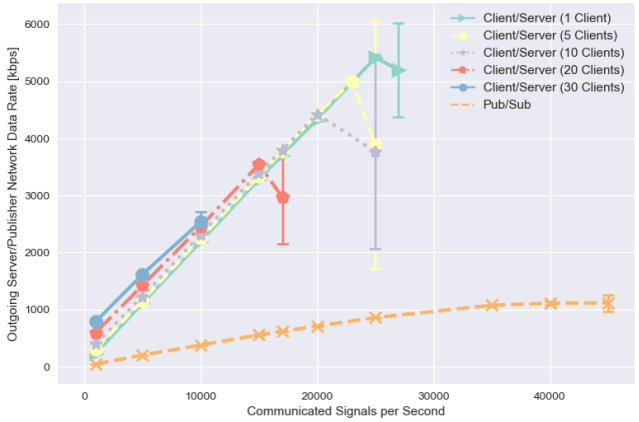

Figure 9: Network utilization at the server/publisher was always below 1 percent, for a 1 Gbit network.

reducing total network traffic. In summary, the CPU was the bottleneck in the analyzed scenarios while the experiments used the other resources only marginally (answering **RQ4**).

5.3 Security Overhead: CPU Utilization

Some scenarios in industrial automation may require encryption of the network traffic. OPC UA provides four different security policies, Basic256Sha256, Basic256, Basic128Rsa15, and None. Basic256 and Basic128Rsa15 have been declared obsolete as they are no longer considered secure. If the network is secured using other means (e.g., firewall) or even physically disconnected from any public network, the security policy "None" may be adequate. However, to get a better understanding on the overhead that encryption causes, we executed a number of test cases with the security policy Basic256Sha256, where all messages are signed and encrypted.

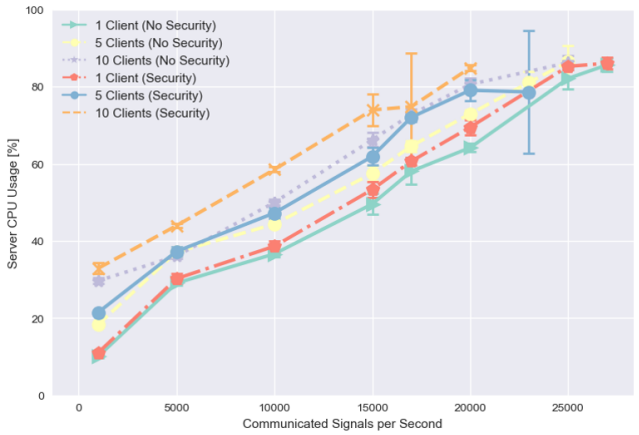

Figure 10: Server CPU utilization: only slight increase for security policy with message encryption.

Fig. 10 visualizes the server CPU utilization in encrypted and unencrypted client/server communication to answer **RQ5**. We were not able to activate the security policy for pub/sub communication, which was not yet implemented in the beta version of the SDK. A small increase in CPU utilization is visible in Fig. 10 for the different numbers of clients if encryption is enabled. The overhead increases for higher numbers of clients. For example, for 10 clients the CPU utilization at 10,000 signals per second is at approximately 50 percent for unencrypted communication, while the CPU utilization is at 60 percent for encrypted communication. For a low number of clients, however, the encryption overhead was low. Thus, we support that encryption should be always considered and evaluated as its impact on performance seemed almost negligible even for the used low-cost hardware platform here.

The figure for client-observed cycle times is not shown for brevity reasons. It can be summarized that for scenarios with enabled encryption, the cycle times increase slightly earlier than for unencrypted communication, although the difference is small. Memory and network overhead for encryption were negligible in the analyzed scenarios and are therefore not detailed.

5.4 Multithreading

Finally, we performed a series of experiments to determine the potential for performance increases using multithreading in the analyzed scenarios (referring to **RQ6**) and based on the studied hardware platforms. For this, we modified our server-side signal update functionality to be executed by multiple POSIX [6] threads to exploit possible parallelisms. In a real implementation, a similar approach could be used to parallelize other application tasks. In this case, the test scenarios communicated 100 signals with a 500 ms update interval and we repeated each experiment 15 times. The POSIX scheduling policy was configured to SCHED_FIFO, and the server POSIX thread priority was configured 98.

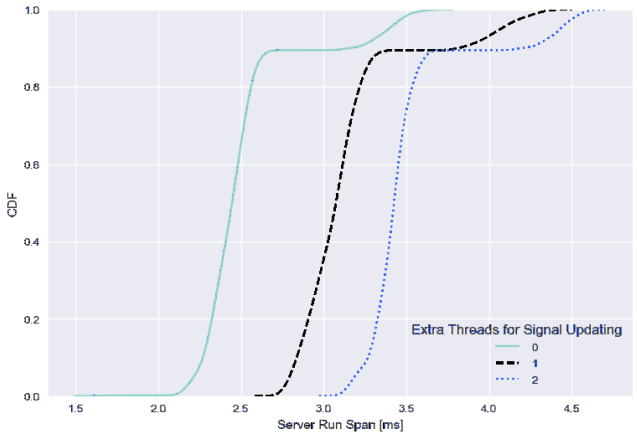

Figure 11: Impact of multithreaded server on Raspberry Pi Zero: additional threads reduce the performance.

Fig. 11 shows a cumulative distribution function for the server run span on the Raspberry Pi Zero. The server run span indicates how long it took the server to finish its work. The smaller the run span, the better as more CPU resources remain available to

[6] http://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

support scenarios with even higher signal rate requirements. As expected and because the Raspberry Pi Zero has only one CPU core, Fig. 11 shows that using additional threads yielded no performance improvement. On the contrary, more threads increased the server run span by up to 50 percent, possibly due to context switches. However, Fig. 12 shows that for a Raspberry Pi 3 with four CPU cores, a major performance improvement for multiple threads is possible. For two or three extra threads, the server run span was decreased by approx. 35 percent. Therefore considering a multi-threaded implementation has indeed a high potential to improve the maximum load a server/publisher device can handle in case multiple processor cores are available (**RQ6**).

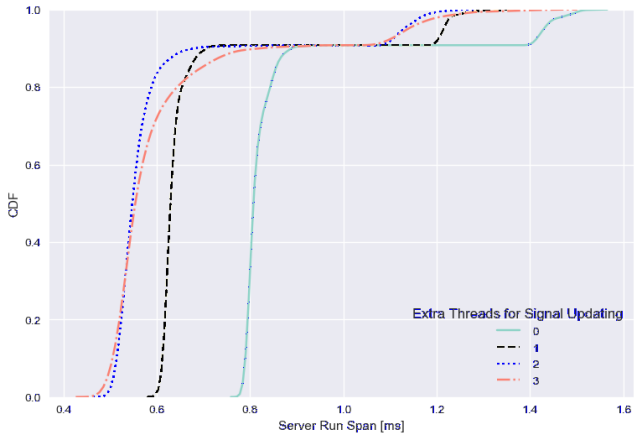

Figure 12: Impact of multithreaded server on Raspberry Pi 3: additional threads significantly increase the performance.

6 PERFORMANCE MODEL

In the following section, we describe a performance model which we created based on the measurements and performance evaluations in Section 5. Such a performance model can be integrated in development processes in order to evaluate whether client/server subscriptions or pub/sub are feasible for a given use case or a specific device.

6.1 Model Construction

The experiments described in Section 5 investigated the influence of several parameters on the performance. The performance model will now integrate these parameters. Parameters for the model are the number of signals hosted on the device, the update rate at which these signals are changed, the publishing rate at which those signals must be communicated to other parties or the number of parties to communicate.

The developed performance model enables predicting performance metrics such as CPU utilization from specified parameters for a given, untested, application context. For this model, the CPU utilization is considered a continuous output variable from one or more input parameters, resulting in a regression problem. Thus, well-studied regression models known from the field of machine learning are promising candidates for modeling the kind of system studied here.

In most cases, linear regression models [1] can be applied for this kind of prediction models. Due to the fact, that the performance measurements unveiled linear trends and that applying a linear regression with Maximum Likelihood estimation seemed straightforward, we decided to apply this model as a first approach for the performance prediction and summarize the taken steps for client/server communication in the following.

In order to formulate the performance model based on linear regression, the choice of the features are crucial. In our case, the feature vector $\vec{x}$ is calculated by using the number of signals in the server address space (ns), the publishing rate (spr), and, for client/server subscriptions, the number of subscribing clients (nc). Following the rules for linear regression, this feature vector is passed into a nonlinear feature vector $\vec{\phi}$ to include an offset and contributions of the respective parameters on the output y. In our case, y denotes the CPU utilization of the OPC UA server or the OPC UA publisher process. See (1) for the construction of $\vec{\phi}$ based on x_i, the feature vector of experiment i.

$$\vec{\phi}(x_i) = \begin{bmatrix} 1 \\ x_i \end{bmatrix} \tag{1}$$

The design of a suitable feature transformation $\vec{\phi}(x_i)$ is crucial and requires domain knowledge as well as the evaluations and comparisons of different possible transformations. In our case, the feature transformation finally selected and showing the most promising results for client/server measurements is given in (2). It was derived based on the observations of linear dependencies in the measurements and by iteratively tuning the contributions of the individual features, which is a typical approach in feature engineering.

$$x_i = ns[i] * spr[i] * nc[i]^{0.3} \tag{2}$$

The influence of the number of subscribing clients was stepwise subdued using an exponent which greatly improved the prediction accuracy. A model for pub/sub communication could be derived in a similar manner but is not presented here.

Once the feature transformation was in place, the weight vector $\vec{w}$ was inferred from the training data from our experiments in Section 5. This way, the generic model is calibrated to fit the measurement data. For this, the weight vector is calculated by using maximum likelihood linear regression which corresponds to the method of least squares applied to the feature transformations of the input and is given in (3). Here $[\Phi]^T$ denotes the Gram matrix of all transformed feature vectors as columns (describing the scenario parameters for individual experiments) and $\vec{t}$ a vector of the measured CPU load values for the individual experiments.

$$\vec{w} = ([\Phi]^T[\Phi])^{-1}[\Phi]^T\vec{t} \tag{3}$$

This method maximizes the likelihood of the measurement data given the weights and features and is basically used to calibrate the weights. It assumes the data set is sampled from a process and contains Gaussian noise.

Once $\vec{w}$ was learned based on the measurements, the CPU utilization y for a given setting of scenario parameters can be predicted

using (4), which summarizes the model.

$$y = \vec{w}^T \vec{\phi}(x) \tag{4}$$

In the following section, we present selected results of the application of the model and how these results support a development process for specific devices, like edge gateways or server aggregating several data sources into one.

6.2 Model Application

The model can be used to estimate the resource requirements of application scenarios which were not tested before using actual measurements. This greatly helps in resource planning and, e.g., evaluating implementation alternatives in scenarios relevant for process industries. In order to validate the accuracy of our model, we first apply it to scenarios discussed in Section 5 which were not used in the model training. In the following, we compare the actual measurement results for these scenarios to the model predictions.

The first validation scenario is a prediction for a server node handling 5 client/server connections and up to 2500 signals with a publishing rate of 100 ms. Fig. 13 shows the performance of the model compared to measured data for this particular setup. The

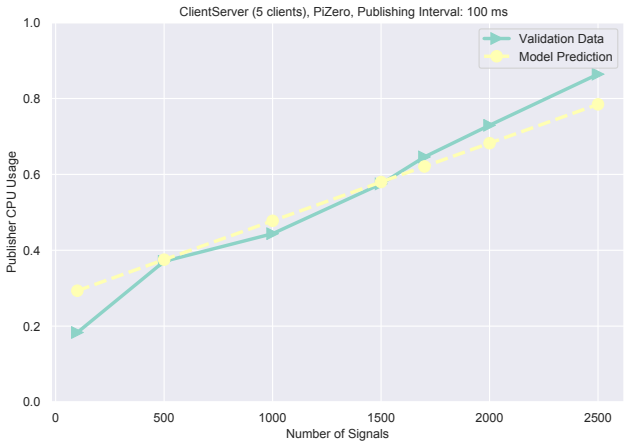

Figure 13: Model evaluation for client/server scenario with 5 clients.

prediction results show a good match to the measured data. For this prediction, we trained the model with a training data setup of 10 client/server connections and a publishing rate of 100 ms.

We used the same trained model to predict a scenario with 20 client/server connections and a publishing rate of 100 ms. Fig. 14 shows the results of this validation. For this specific scenario, the validation data includes a step at about 1,500–1,700 signals. This particular step was not predicted by the model. However, the predicted curve is close to the measured values. Thus, both validation scenarios show that the model is already fairly accurate in the prediction of the CPU usage for specific scenarios. For the sake of completeness, we jointly show the training data and the prediction in Fig. 15 to visually verify the correct model calibration.

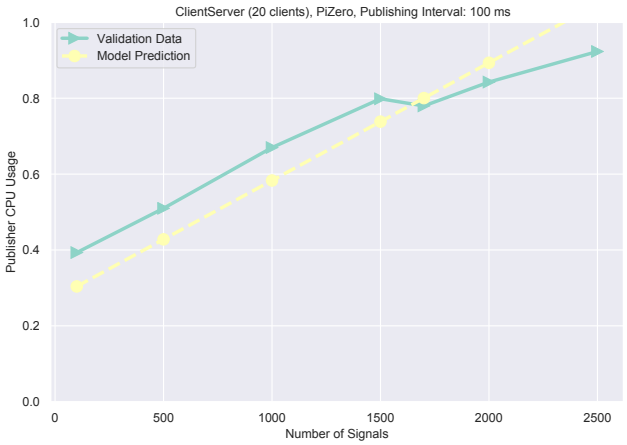

Figure 14: Model evaluation for client/server scenario with 20 clients.

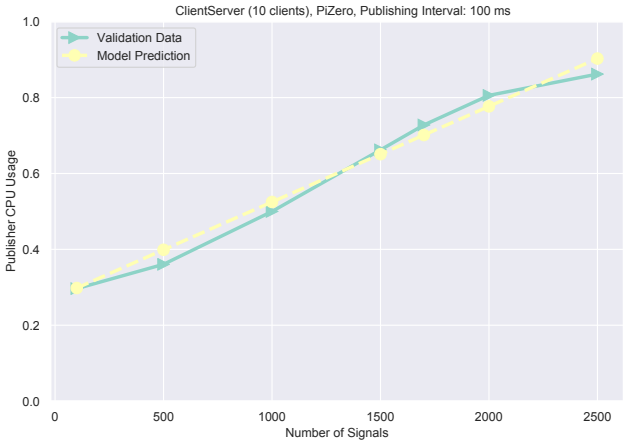

Figure 15: Training data vs. prediction (10 clients).

To show the power of such a model, we used it to predict two major use cases which are relevant for industrial automation systems. The first use case serves a large number of IO points per node and has the following properties:

- 6,000 IO points, 2 large process controllers
- 10 applications in each controller
- 2 controllers share 8 field comm. interfaces (FCIs)
- A controller handles up to 3,000 IO points and 4 FCIs

In particular, that means for the controller node, the CPU utilization for up to 3,000 IO points and 4 client/server connections needs to be predicted. Using the model, for each FCI a configuration of up to 750 IO points and 2 client/server connections is predicted. Fig. 16 shows the results for the prediction of the CPU utilization of the controller node.

The prediction results show that handling such a high number of IO points is challenging for the CPU. A controller with such a hardware setup would be very busy and likely unreliable. However, the model was trained on a Raspberry Pi Zero with a 1 GHz

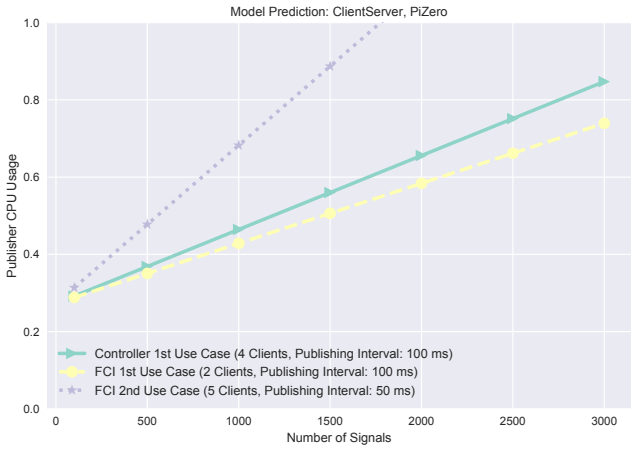

Figure 16: Prediction of controller and FCI loads for two industrial automation system use cases.

single-core CPU and 512 MB memory. We assume that a future controller system architecture has at least dual core CPUs and the same frequency per core or even higher, so that the setup is considered feasible. The prediction model presented here provides a first guidance on how powerful such a hardware architecture should be for the given setup. A model trained on the target hardware platform would help to further improve the predictions.

The prediction for the FCI node shows a different picture. In Fig. 16 it can be seen that a hardware architecture similar to a Raspberry Pi Zero would be sufficient for an FCI. Especially, if we assume that one FCI needs to handle up to 750 IO points with a publishing interval of 100 ms. In that case, the CPU utilization would be at about 40 percent which gives enough space to scale up the number of IO points if necessary as well as to do different tasks, e.g., analog-to-digital signal converting or evaluation of signals.

The second use case focuses on a low number of IO points per node and has the following properties:

- 6,000 IO points, 10 small controllers
- 2 applications in each controller
- 10 controller share 40 FCIs
- Each FCI is connected to 5 controllers

In particular, one controller handles 600 IO points and 4 Client/Server connections with a publishing interval of 100 ms. An FCI is connected to 5 controllers, that means 5 client/server connections, and handles 150 IO points with a publishing interval of 50 ms. The prediction for the controller is similar to the previous use case. Here, the controller needs to handle fewer IO points at the same publishing rate of about 100 ms. In Fig. 16, the CPU usage for 600 IO points is about 0.4 GHz. Hence, a hardware platform similar to the Raspberry Pi Zero is appropriate in that use case.

Fig. 16 also shows prediction results for the FCI in the second use case. Due to the shorter publishing interval of about 50 ms, the CPU utilization is higher than in the previous predictions. Nevertheless, in this case the FCI has to handle 150 IO points which results in a CPU utilization of about 35 percent even for the higher publishing rate. Hence, the hardware setup for an FCI can be realized based on a CPU with 1 GHz or less.

Predicting these two use cases demonstrates the benefit of the proposed performance model. It provides a first estimation of hardware requirements for a specific node or to determine that the number of IO points is too high to be handled. After selecting the actual hardware and application use cases, clearly a more thorough evaluation of the overall system architecture is needed. Nevertheless, possible bottlenecks can be easily and early identified in the development process using the proposed modeling approach.

7 THREATS TO VALIDITY

This section discusses the threats to validity both to our measurement and modeling results. We review construct validity, internal validity, and external validity [12].

7.1 Construct Validity

Construct validity reflects whether the measures used in the experimentation represent what the research had in mind with the research questions. Our experiments employed Raspberry Pi devices as substitutes for industrial embedded systems. We argue that Raspberry Pis have a similar computing performance as modern industrial devices. We also equipped the Raspberry Pis with high precision real-time clock modules. However, some cheaper or power-constrained industrial field devices may use even less powerful hardware. For this kind of devices, our experiments and models provide limited guidance.

With Debian Linux with the RT PREEMPT patch we used a soft real-time operating system (RTOS) that may not offer the same deterministic behavior with guaranteed deadlines, such as commercial RTOS (e.g., VxWorks or QNX). However, we followed the guidelines specified by the Open Source Automation Development Lab [7], which have tested Linux RT PREEMPT on multiple platforms and deemed it usable for many typical industrial automation applications. More and more commercial industrial devices are based on Linux, therefore we deem this choice representative for the target application domain.

The chosen OPC UA SDK from Unified Automation can be considered representative for the whole class of other commercial (e.g., Matrikon, Softing, Prosys) and OSS (e.g., FreeOpcUa, open62541) OPC UA SDKs. Unified Automation provides a comprehensive list of reference customers [8], which includes Siemens, Robert Bosch GmbH, Trumpf and other companies, showing that it is used throughout the industry.

We used update intervals down to 50 ms in our experiments and performance models. This does not cover scenarios with shorter cycle times, which are however only used in specific situations in industrial automation (e.g., power electronics, machine control). We argue that the chosen application profiles cover a range of scenarios and similar profiles have been used in other OPC UA measurements [4, 5]. We focused on server-side publishing, but did not investigate subscriber-side filtering in more detail, which could become a problem if the published DataSets are not well chosen.

[7] https://www.osadl.org/
[8] https://www.unified-automation.com/references.html

We only used IGMP for dynamic multicast filtering, whereas other protocols (e.g., MMRP [9], TSN-based) or also application-level brokers could fulfill similar purposes. Here we followed guidelines from the OPC UA pub/sub specification. Finally, we mainly analyzed resource utilizations, but not latencies. However, we claim that OPC UA latency has been extensively covered in other studies (e.g., [2, 10]) and was therefore out of scope in this paper, which was more interested in scalability.

7.2 Internal Validity

Internal validity reflects whether the effects shown in the experiments can correctly be attributed to the causes (e.g., the chosen application profiles and testbed), and whether there are interfering variables that may distort the cause-and-effect relationship.

One interfering variable in our experiments was the beta-status of the chosen OPC UA SDK. The software measured had not yet been performance-optimized, therefore the results for CPU and memory utilization need to be considered with care. Fig. 7 showed an unexpected increase in cycle times for higher signal rates, which we attributed to the unstable OPC UA SDK. Future measurements need to validate our findings with commercially released SDKs, but we expect rather an improvement in performance, therefore our results can be considered lower bounds.

Additional bottlenecks could hide in the network, if larger sets of individual publishers would be analyzed, and multiple clients would have overlapping information needs. This could lead to a highly utilized network in specific situations.

For the performance model, we used a simple maximum likelihood linear regression model. More sophisticated models, such as neuronal networks, Gaussian processes, or ridge regression could be used alternatively, as they would provide a probability distribution and not only a single scalar prediction value. If more features would be added to the model (e.g., the hardware platform), an alternative model choice would be required, as linear regression tends to overfitting for a complex feature vector.

7.3 External Validity

External validity is concerned with question to what extent the findings can be generalize. As our testbed hardware and application profiles are considered representative for many control applications in process automation, the results should be valid in many similar scenarios. For applications with 50-500 ms cycle times and 1-20,000 signals per second, our results should be applicable. This may, for example, cover many chemical processes, electric power generation, or paper production. Scenarios involving cloud communication may add network latency and, thus, need an additional analysis.

Users with more powerful hardware could achieve even higher server throughputs, while users with much less powerful hardware could run into different bottlenecks. We did not conduct a comparison of different OPC UA SDKs for pub/sub performance, so our results may not be easily generalized to applications using different SDKs. We also did not investigate other M2M middlewares, e.g., based on OMG DDS [10], AMQP, or JMS [11], which are used in other domains.

[9] http://www.ieee802.org/1/pages/802.1ak.html

8 CONCLUSION

This paper showed that CPU utilization is the performance bottleneck for OPC UA pub/sub communication using typical application profiles from industrial automation. We found that the memory and network overhead was negligible, that encryption added a low overhead, and that OPC UA communication could benefit from multithreading. The constructed performance model helped answering hardware sizing questions in two different scenarios.

The measurements and models may be used by practitioners and researchers designing industrial automation architectures and systems. Given the number of signals to communicate and their required update intervals, developers can size hardware environments to minimize costs while providing an appropriate performance. This is considered especially useful in designing industrial Internet-of-Things applications with many OPC UA communication partners.

To improve our understanding of OPC UA performance further, we plan to conduct additional experiments on different hardware platforms and even more application profiles. We want to investigate the impact of sizing DataSets of OPC UA publishers on the subscriber performance. We plan to improve our performance model to make it applicable in even more situations.

[10] https://www.omgwiki.org/dds/what-is-dds-3/

[11] https://jcp.org/en/jsr/detail?id=343

REFERENCES

[1] C. M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York.

[2] D. Bruckner, R. Blair, M-P. Stanica, A. Ademaj, W. Skeffington, and D. Kutscher. 2018. *OPC UA TSN A new Solution for Industrial Communication.* Technical Report. Shapers Group White Paper. 1–10 pages. https://www.br-automation.com/smc/953ce46647cb909f0cce603249fb229e29f0a30a.pdf

[3] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. 2002. Internet Group Management Protocol, Version 3. RFC 3376. https://tools.ietf.org/html/rfc3376

[4] S. Cavalieri and F. Chiacchio. 2013. Analysis of OPC UA performances. *Computer Standards & Interfaces* 36, 1 (2013), 165–177.

[5] M. Fojcik and K. Folkert. 2012. Introduction to opc ua performance. In *International Conference on Computer Networks*. Springer, 261–270.

[6] S. Grüner, J. Pfrommer, and F. Palm. 2016. RESTful industrial communication with OPC UA. *IEEE Transactions on Industrial Informatics* 12, 5 (2016), 1832–1841.

[7] H. Koziolek, A. Burger, and J. Doppelhamer. 2018. Self-Commissioning Industrial IoT-Systems in Process Automation: A Reference Architecture. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 196–19609.

[8] W. Mahnke, S.-H. Leitner, and M. Damm. 2009. *OPC unified architecture*. Springer Science & Business Media.

[9] OPC Foundation. 2018. OPC Unified Architecture Specification Part 14: PubSub.

[10] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran. 2018. Open Source OPC UA PubSub over TSN for Realtime Industrial Communication. In *IEEE Emerging Technologies in Factory Automation (ETFA)*. 1–4.

[11] M. Silveira Rocha, G. Serpa Sestito, A. Luis Dias, A. Celso Turcato, and D. Brandão. 2018. Performance Comparison Between OPC UA and MQTT for Data Exchange. In *2018 Workshop on Metrology for Industry 4.0 and IoT*. 175–179. https://doi.org/10.1109/METROI4.2018.8428342

[12] P. Runeson and M. Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[13] M. Schleipen, S.-S. Gilani, T. Bischoff, and J. Pfrommer. 2016. OPC UA & Industrie 4.0 – Enabling technology with high diversity and variability. *Procedia Cirp* 57, 1 (2016), 315–320.