Real-time capable OPC-UA programs over TSN for distributed industrial control

Christian Eymüller, Julian Hanke, Alwin Hoffmann, Markus Kugelmann, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Eymüller, Christian, Julian Hanke, Alwin Hoffmann, Markus Kugelmann, and Wolfgang Reif. 2020. "Real-time capable OPC-UA programs over TSN for distributed industrial control." In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 8-11 Sept. 2020, Vienna, Austria*, edited by Thilo Sauter, Francisco Vasques, Lucia Lo Bello, Valeriy Vyatkin, Andrés A. Nogueiras, and Stefan Wilker, 278–85. Piscataway, NJ: IEEE. https://doi.org/10.1109/ETFA46521.2020.9212171.

Real-time capable OPC-UA Programs over TSN for distributed industrial control

Christian Eymüller*, Julian Hanke*, Alwin Hoffmann*, Markus Kugelmann†, and Wolfgang Reif*
Institut for Software and Systems Engineering
University of Augsburg
Augsburg, Germany
*{eymueller, hanke, hoffmann, reif}@isse.de, †markus.kugelmann1@student.uni-augsburg.de

***Abstract*—A key aspect of Industry 4.0 is the continuous interconnectedness of components. The standardized industrial communication protocol OPC UA offers a solution to this problem by enabling the exchange of data between the shop floor level and the inter-enterprise level. Due to the integration of the Time Sensitive Network (TSN) into OPC UA, it is now even possible to exchange information in real-time. Especially on the shop floor, there are numerous heterogeneous distributed devices from sensors to robots which must communicate with each other in real-time to achieve a distributed industrial control. Therefore, we propose an approach to combine real-time communication over TSN with OPC UA Programs to synchronize multiple distributed OPC UA Programs and exchange process data between them without losing real-time guarantees. This can be seen as the enabler of Plug-and-Produce with real-time requirements.**

***Index Terms*—OPC UA Programs, OPC UA PubSub over TSN, Time-Sensitive Networking, Skills, Plug-and-Produce**

I. Introduction

Industry 4.0 induces remarkable social and economic challenges [1], in particular: shorter development and innovation cycles, individualization of products on demand, higher flexibility in product development, decentralized decision-making procedures, and resource efficiency. This results in large research efforts in intelligent manufacturing [2] in order to cope with these challenges. Cyber-physical production systems [3], digital twins [4] of the product and its manufacturing process as well as multi-robot cooperation [5] for flexible and adaptable assembly [6] are promising research directions.

A very promising concept for encapsulating procedural knowledge in a reusable way for intelligent manufacturing is that of skills [7]. In contrast to traditional PLC programs or robot scripts, skills can be characterized as general procedural descriptions which means that they can handle a variety of different parts and, thus, contain pre- and postconditions which must be met by the current situation [8]. The set of necessary skills for a given production process can be extracted by analyzing its industrial standard operating procedures [9]. Hence, skills offer the possibility to model a manufacturing process which can be adapted in a flexible way by changing the parameterization or the orchestration of its skills. Skills can be considered as an enabler for industrial Plug-and-Produce systems [10], [11].

For machine-to-machine communication in Industry 4.0, the OPC Unified Architecture (OPC UA, [12]) can be considered as the current de-facto standard [13]. It is based on an open, cross-platform service-oriented architecture (SOA) for accessing industrial equipment and systems. For modeling complex data, OPC UA has an integral information model which allows accessing the data with a client/server as well as a publish/subscribe communication pattern. Moreover, OPC UA offers so-called programs to model long-running control applications. Control applications in robotics and automation, however, need to meet real-time requirements in order to be executed in a deterministic and safe way [14]. That applies both to the software components and to the communication among them. OPC UA meets these requirements by using Time-Sensitive Networking (TSN) for its publish/subscribe communication pattern. TSN offers mechanisms for the time-sensitive transmission of data over shared ethernet networks. Hence, this allows OPC UA nodes to exchange data while meeting defined timing requirements.

When we consider control applications based on the orchestration of skills, we have a distributed real-time component system [15]. Skills can be implemented as long-running methods which must be started and stopped in a synchronized and time-sensitive manner while the control application is running. From the authors' point of view, there are currently no means to support the implementation of such real-time control applications in OPC UA. Hence, we propose the concept of real-time capable and long-running tasks in OPC UA as main contribution of this paper. Our approach uses OPC UA Programs [16] and combines it with publish/subscribe communication over TSN in order to realize distributed and synchronized OPC UA Programs over TSN. Based on our approach, skills and Plug-and-Produce systems with real-time requirements can be developed in the future.

This paper is structured as follows: Section II describes the state of the art in modeling industrial control applications, in particular with OPC UA. Our main contribution, i. e., distributed and synchronized OPC UA Programs over TSN, is explained in detail in Section III. Remarks on our implementation which is based on open62451 [17] can be found in Section IV. In order to show the appropriateness of

This work partly presents results of the project *WiR Augsburg* which is funded by the German Federal Ministry of Education and Research (BMBF) and the Bavarian Government.

our approach for industrial control, an evaluation is given in Section V. Finally, Section VI concludes the paper.

II. State of the Art

For establishing a holistic communication infrastructure between devices on the shop floor and the cloud, the importance of OPC UA is increasing rapidly. OPC UA is an open and service-oriented machine-to-machine communication protocol. It is based on a client-server infrastructure which enables flexible and vendor-neutral communication in industrial applications. With its integral information model, every server is able to organize complex data or methods in a standardized way. In order to enable access to data or methods, a client must set up a dedicated TCP/IP communication channel to the server. With this one-to-many approach, the number of simultaneous connections is often limited by computing resources [12]. To overcome this limitation, Part 14 of the OPC UA specification [18] adds the possibility of many-to-many communication based on the publish/subscribe (*PubSub*) mechanism. With this mechanism, there is no need for a dedicated and direct communication channel between publisher and subscriber. This is similar to traditional field-buses in that many subscribers could receive the same message, but offers the advantage of being able to be reconfigured during runtime.

The publisher defines the published information with flexible and configurable *PublishedDataSets* which contain data variables or event sources from the information model. The subscriber can then consume a selected *PublishedDataSet* [18]. The routing of messages between a publisher and a subscriber is carried out by a message-oriented middleware. OPC UA PubSub supports two largely different variants: broker-based or broker-less middlewares. The broker-based variant integrates currently existing publish/subscriber protocols like AMQP [19] or MQTT [20]. In this case, the message-oriented middleware serves as a broker and is defined by one of the previously mentioned protocols. The broker-less option implements a custom UDP-based distribution protocol in the form of UDP multicasts by using the network infrastructure. Publisher can send packets to a UDP multicast group. Subscribers can register to this address and all messages get forwarded to them.

However, many sensors and actuators on the shop floor require real-time communication, because they have to react within a certain cycle time [14]. We can expect that plain UDP-based communication cannot achieve that due to the protocol specifications. To support time-deterministic applications such as real-time critical control systems, the OPC UA PubSub over TSN approach was introduced [21], which combines the OPC UA PubSub mechanism with real-time guarantees enabled by Time-Sensitive Networking (TSN). TSN is an extension to the Ethernet protocol developed within the IEEE 802.1 Working Group [22] to introduce a uniform and universal network convergence with timing guarantees for time-sensitive applications. The main concept of OPC UA PubSub over TSN is to utilize the TSN extension of the Ethernet protocol to ensure deterministic transport of OPC UA PubSub

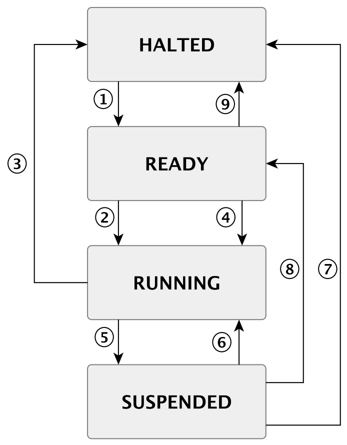


Figure 1: Standard OPC UA Program finite state machine with four default states and nine default transitions [16].

messages with real-time guarantees. Sending of messages for a publisher is handled by a hardware-triggered interrupt which ensures short delays and small jitter. This allows OPC UA subscribers to access a shared information model without the loss of real-time. Thus, it is now possible to synchronize several components and enable real-time communication between them [17]. With open62451 [17], there is an open source implementation in C available for OPC UA PubSub over TSN.

In addition to the previously addressed communication problem, real-time critical control systems need some kind of stateful and long-running service. OPC UA Programs [16] can be seen as a way to achieve this in a standardized way. Programs are stateful and model a long-running service by means of a predefined sequence of states they execute. Each program is represented by a finite state machine running on an OPC UA server. It allows clients to trigger or monitor the execution as well as to get the results back from a process in a generic manner. The default state machine is shown in Figure 1. It defines the four states *halted*, *ready*, *running* and *suspended* which must be implemented by every instance. But it is possible to extend the four states with substates to provide more resolution. The transitions between the four mandatory states are defined by nine basic transitions. These transitions can either be triggered by a client or by the program itself. So it may not require a client action to cause a transition but there are five optional methods (i. e., *start*, *suspend*, *resume*, *halt*, *reset*) to control the program execution. Each transition triggers an event, e. g., a value which can represent a final output or the actual state. The transitions and their implementation will be discussed in more detail in Section III. Overall, OPC UA Programs allow exposing, discovering and calling stateful and long-running services in a generic manner.

However, no current approach takes advantage of the combined benefits of the previously described technologies. Dorofeev et al. [23], for example, show how a manufacturing skill (service offered by an automation system) can be modeled by using an OPC UA Program. They implemented a generic interface to allow a higher control level to easily orchestrate their available skills. Furthermore, Profanter et al. [24] use OPC UA Programs to model skills of industrial robots. Kaspar et al. [25] model PLC function blocks similar to OPC UA Programs. Another approach can be found within the Robot Operating System (ROS). The ROS Action Protocol, which is provided by the ROS ActionLib [26], allows the execution of long-running services using topics and publish/subscribe communication. For this purpose, the ROS Actionlib offers a simple API for requesting goals on the client side and executing these goals on the server side via function calls and callbacks. But with regard to a holistic industrial communication, the aforementioned approaches lack the possibility of real-time communication and execution.

III. Concept

In the context of distributed real-time critical control systems, four main requirements must be fulfilled:

1) synchronize clocks between several distributed components
2) enable real-time communication between these components
3) synchronize distributed control processes across multiple components without losing real-time capability
4) exchange information between the control processes in real-time

This paper proposes a novel approach to combine the concept of OPC UA Programs and the communication with OPC UA PubSub over TSN to accomplish these requirements. With OPC UA PubSub over TSN on the one hand, the possibility is given to synchronize clocks between several components and enable real-time communication between these nodes. On the other hand, OPC UA Programs add the ability to have continuously running processes. The problem, which must be solved, is how to get distributed continuously running processes (see Section III-B) that can be synchronized and exchange status as well as process information in real-time (see Section III-A). With this at hand, we are able to implement distributed, but synchronized industrial control software. The details of our concept are presented in the following two subsections.

A. Real-time communication

To meet the assumption that state changes can be communicated in real-time, we use OPC UA PubSub over TSN. The use of this technology enables sending and receiving messages between multiple devices with certain time limits. This ensures that changes in status on one device also lead to a state switch on other devices within a fixed timespan. The current state of the OPC UA Program state machine is provided by a topic that is published by an OPC UA Publisher over TSN. Other OPC UA Programs subscribe to this topic and thus have the current status of all other participating OPC UA Programs. The states of the other OPC UA Programs are monitored. All that remains to be done is to determine a protocol to synchronize the states of the distributed OPC UA Program state machines. The concept of this protocol is shown in Section III-B.

Of course, not only communication is needed to inform other OPC UA Programs about changes in status. Furthermore, real-time communication is needed to exchange process data across multiple running OPC UA Programs. This process data can be for example position data that has to be exchanged between multiple programs. It is therefore possible in our approach to add process data to the distributed OPC UA Programs that can be subscribed to and used in other OPC UA Programs. Adding OPC UA PubSub over TSN results in a deterministic real-time communication channel for exchanging process data across multiple running OPC UA Programs and the opportunity to transfer the current states of the state machines to other OPC UA Programs.

B. Distributed State Machines

We have revised the standard state machine of the OPC UA Program Specification (shown in Figure 1) by adding a handshake mechanism to synchronize multiple distributed OPC UA Programs with each other. Here, synchronisation means that the states of all distributed OPC UA Programs that form an overall behaviour must be exchanged and matched to a singular state. In our approach, we differentiate between two types of distributed OPC UA Programs. A distributed OPC UA Program always needs a single Master OPC UA Program that can be paired with multiple Slave OPC UA Programs. The main task of the Master OPC UA Program is to check whether all Slave OPC UA Programs are synchronized and ready to start the continuously running process. Therefore, the distributed OPC UA Programs must have the possibility to exchange their status with each other. The task of the Slave OPC UA Programs is to receive instructions from the master and react with state changes, which in turn are communicated back to the master.

Figure 2 shows a simple application example for distributed OPC UA Programs with one master and one slave. In the example, a trajectory generator is executed on one device. The task of the trajectory generator is to generate a trajectory and provide target axis positions at regular time intervals for the robot control. On the other device, a robot controller is

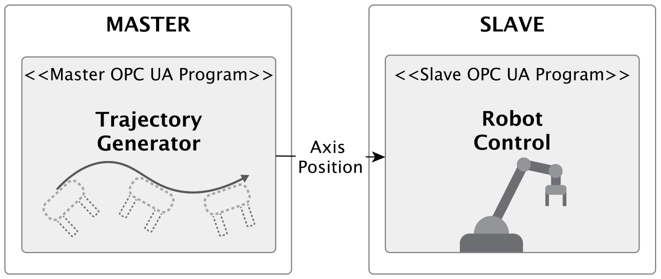


Figure 2: Application example for distributed OPC UA Programs with one master and one slave: Distributed execution of a trajectory on a robot arm

running. The robot control expects target positions for each axis in a predetermined time interval and translates these target axis positions into control commands for the robot. These control commands are then passed on to the robot. For this to work in a distributed manner, the two participants need to know when the other part is ready and start their execution of the program simultaneously. For this reason, it is necessary to have state machines synchronized across multiple devices.

Additional states have to be added to synchronize the distributed state machines. Due to the strict specification of OPC UA Programs, we did not add any additional states to the OPC UA Program to meet the mentioned challenges, but rather appended sub-states to the state machine. We added the sub-state *INIT* to the *HALTED* state of the OPC UA Program state machine. The *INIT* sub-state ensures that all involved OPC UA Programs are properly initialised and ready for upcoming state changes before they change into the state *READY*.

Figure 3 shows the sequence for synchronising and starting a distributed OPC UA Program. In the illustrations state transitions are represented by solid arrows (→) and notifications by dashed arrows (⇢). Initially, it must be determined how many OPC UA Slaves are participating. For the synchronisation of multiple OPC UA Programs, the Master OPC UA Program waits in the sub-state *INIT* till all Slave OPC UA Programs are initialised and switched into the *READY* state (①). The master gets a notification (②) from its slaves if the state *READY* is reached. As soon as the master has received all notifications of its slaves, it also switches into the state *READY* (③) and notifies (④) all slaves that the master is now ready for the execution of the program. After receiving the notification ④ the slaves switch into the state *RUNNING* (⑤), notify the master that the program has started (⑥) and start their continuously running process. Due to the received notifications (⑥) of all slaves, the master sets its state to *RUNNING* (⑦) and starts its program. Consequently, the handshake for starting a distributed OPC UA Program is completed and all participating OPC UA Programs are in the state *RUNNING* and executing the actual program. When all OPC UA Programs are running, the continuous exchange of process data starts (⑧).

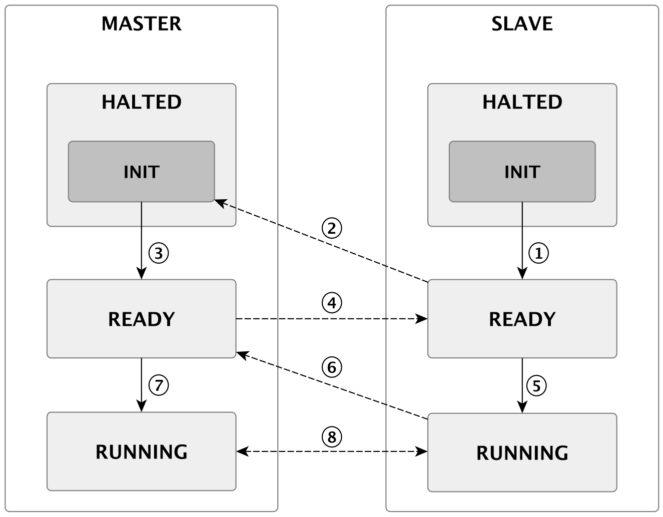


Figure 3: Handshake for starting distributed OPC UA Programs. Changes of state (→) are communicated via the OPC UA PubSub communication channel (⇢) between master and slaves. ① to ⑦ specify the order of the transitions and notifications. ⑧ represents the exchange of process data betwenn the master and slaves.

Now that the processes are running on all parts of the system, there must also be a possibility to halt all distributed OPC UA Programs simultaneously.

The state *HALTED* can be reached from all other states. In each of these cases the distributed OPC UA Programs behave identically. Figure 4 exemplifies the state transitions after an OPC UA Program stops and switches from any state into *HALTED* (1a or 1b). In case an OPC UA Program reaches the state *HALTED*, it notifies all other participating OPC UA Programs (2a or 2b). This ensures that every OPC UA Program is stopped and switched to state *HALTED*. Thus, all running OPC UA Programs are now halted (*HALTED*). In order to restart a program, the initialisation phase must repeated, i.e. the sub-state *INIT* must has to be called again. After words the sequence of starting distributed OPC UA Programs is used.

Finally, there is the option to suspend and resume OPC UA Programs. The procedure for switching from the state *RUNNING* to the state *SUSPENDED* is similar to the state switch from *RUNNING* to *HALTED*. In the event that an OPC UA Program suspends its running process, it switches into the state *SUSPENDED* and informs the running OPC UA Programs to change to this state as well and also suspend their work. For resuming the process, the transition changes shown in Figure 5 have to be executed. By the time an OPC UA Program goes back into the *RUNNING* state (1a or 1b), a notification is sent to all suspended OPC UA Programs (2a or 2b). This message prompts them to switch to the *RUNNING* state (3a or 3b) and resume their processes. Thus, it is possible to suspend and resume distributed OPC UA Programs simultaneously.

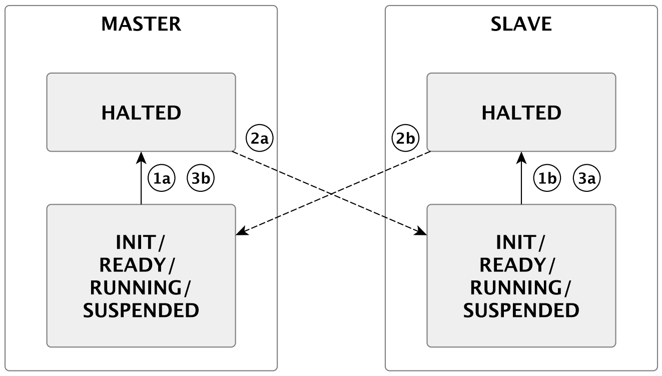

MASTER
SLAVE
HALTED
INIT/ READY/ RUNNING/ SUSPENDED
HALTED
INIT/ READY/ RUNNING/ SUSPENDED
1a
3b
2a
2b
1b
3a

Figure 4: Halting distributed OPC UA Programs. …a represents the sequence of halting initiated by the master and …b by the slave.

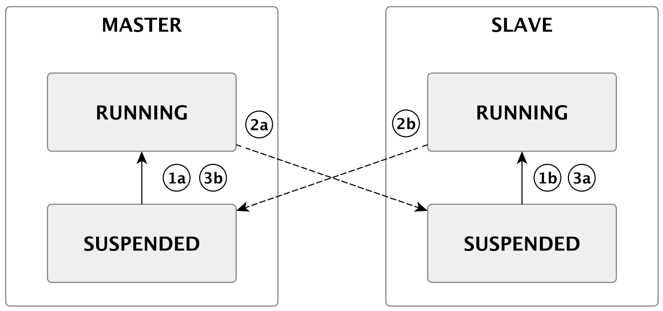


Figure 5: Restarting suspended distributed OPC UA Programs

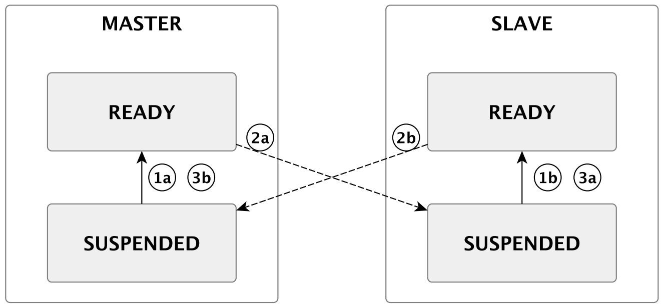


Figure 6: Reinitializing suspended distributed OPC UA Programs

Assuming a suspended OPC UA Program switches into state *HALTED*, the described routine shown in Figure 4 is applied. At last there is the possibility that transition (8) in Figure 1 is used. This transition represents the switch from state *SUSPENDED* to *READY*. Figure 6 shows the state sequence from state *SUSPENDED* to state *READY*. In case an OPC UA Program uses this transition ((1a) or (1b)), it notifies all other suspended OPC UA Programs that it switched to state *READY* ((2a) or (2b)). Subsequently the suspended OPC UA Program also switch into the *READY* state ((3a) or (3b)).

Through the extensions of the OPC UA Program state machine, we are now able to fulfill the requirement of starting distributed processes across multiple components. What still has to be shown is the preservation of the real-time capability. The reaction time ($time(\dashrightarrow) + time(\rightarrow)$) for a change in status can be calculated by adding up the transmission time ($time(\dashrightarrow)$) to the OPC UA Program and the time for the state transition in the OPC UA Program state machine ($time(\rightarrow)$). Since the communication channel is deterministic and the status changes take place within fixed time barriers, the entire response time is deterministic. Hence the goal of starting distributed control processes across multiple components without losing real-time capability is fulfilled. This means that all initially defined goals for our approach have been achieved.

IV. Implementation

After the concept of distributed OPC UA Programs has been presented, we will introduce how this approach was implemented. As basis OPC UA implementation, we used the open source OPC UA stack open62541[1]. However, there are are no implementations of OPC UA Programs included in the stack yet. Moreover, at the time we started our implementation, there was no full real-time integration for OPC UA PubSub. Therefore, we used the open62541 fork of Kalycito[2] with an implementation of OPC UA PubSub over TSN [17], which meanwhile has been integrated into the master branch of open62541. As programming language we used C++.

Figure 7 shows a component diagram of the distributed Master OPC UA Program. The structure of the Slave OPC UA Program is identical. Each distributed OPC UA Program artifact consists of two main components: the `State Machine` component and the `Communication` component. The `State Machine` component represents the state machine of the OPC UA Program with the extensions described in Section III-B. For the current state of the state machine and process data to be exchanged, the `State Machine` component provides the interfaces `CurrentState` and `ProcessData`. The state and the process data of the other participating OPC UA Programs is accessed through the interfaces `ProcessData` and `ReceivedState` of the `Communication` component. The `Communication` component is responsible for the communication with other devices and is intended to provide an easy way to add data that have to be sent to other OPC UA Programs. To ensure the interchangeability of the OPC UA stack the interfaces `PublishData` and `SubscribedData` were added to the `Communication` component. In our case, the `PublishData` is composed of the `CurrentState` and the `ProcessData` and the `SubscribedData` is divided into the `ReceivedState` and the `ReceivedData`.

A. Encapsulation of the OPC UA PubSub configuration

At first, we encapsulated the communication of the open62541 stack to simplify the management of publishers and subscribers. The `Communication` component is responsible for updating the OPC UA information model with the values that have to be published and manages access to the data over the OPC UA PubSub mechanisms. Same applies to data that has to be subscribed.

Therefore, the `Communication` component provides the class `Publisher`, the class `Subscriber` and the templated class `<typedef DATA_TYPE> DataSet`. When objects of the type `Publisher` (or `Subscriber`) are created, a PubSub Connection and a default `WritingGroup` (or `ReadingGroup`) are generated automatically (cf. Figure 7). Created `DataSet` objects can then be added to `Publisher` and `Subscriber` objects by calling the method:

```
bool addDataSet(std::String* name,
    VariantDataSet dataset);
```

Unless it is defined otherwise, the added `DataSet` objects are appended to the default WritingGroup (or ReadingGroup).

[1]www.open62541.org
[2]https://github.com/Kalycito-opcua-tsn-LoI1/OPCUA-TSN-LoI1

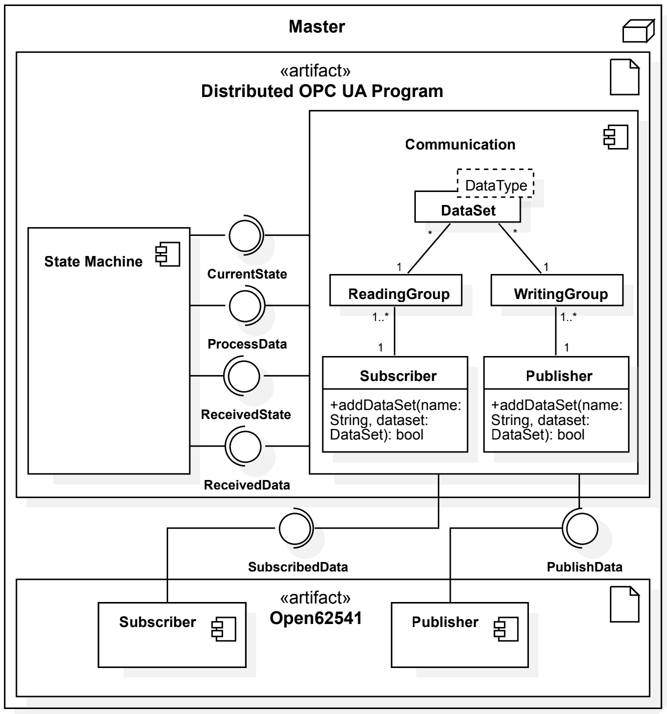


Figure 7: Component diagram of a distributed Master OPC UA Program. However, the structure of the slave is identical. Clear interfaces between the state machine component and the communication component were defined. In addition, interfaces were defined to make the OPC UA stack interchangeable.

Additionally, the `Communication` component provides the methods

```
bool addPublisher(std::String* name,
    Publisher publisher)
```

and

```
bool addSubscriber(std::String* name,
    Subscriber subscriber)
```

in order to add the created `Publisher` and `Subscriber` objects to the distributed OPC UA Program. Thus, data of different types that is supposed to be published simply has to be generated as a `DataSet` object and can then be easily assigned to any publisher. The same applies to data that programs subscribe to. It is therefore not necessary to create PubSubConnections or DataSetGroups manually.

For example, if you would like to exchange a floating point value between a master and a slave, the following steps must be executed. On both sides a `<double>DataSet` object with a floating point datatype must be created. On the master this created object is added to the `Publisher` object by calling the `addDataSet` method. In order to subscribe to the floating point value on the slave, the created `<double>DataSet` object is added to the `Subscriber` object. Last but not least the `Publisher` object has to be added to the `Communication` component of the master and the `Subscriber` object to the `Communication` component of the slave. The floating point value in the `DataSet` can then be accessed via callback methods.

B. Exchange of states and process information in real-time

The encapsulation of communication also enables us to easily add publishers and subscribers to the distributed state machine. For the exchange of the current state between multiple distributed OPC UA Programs, each `State Machine` component appends one publisher which represents the current state as an integer value. Besides the publisher, a subscriber is also necessary to receive the state changes of connected OPC UA Programs. This is the minimum setup for synchronizing the state changes of several distributed OPC UA Programs with each other. In addition to this synchronization, it is possible to add process data in form of further publishers and subscribers to the OPC UA Program. Due to the determinism of the OPC UA PubSub communication over TSN, exchange of state and process information is also deterministic. In case process data is added to the OPC UA Program, the added `DataSets` are published with a defined cycle time after the OPC UA Program switches into the state *RUNNING*.

V. Evaluation

To evaluate the distributed OPC UA Programs we wanted to determine whether our approach fulfills the requirements of a distributed industrial control system. As an example of such a system, we chose the control of an industrial robot as a realistic use case for applying distributed OPC UA Programs. To this end we revisit and improve the example shown in Figure 2. For the control of an industrial six-axis robot, at least six floating-point values are required, indicating the target position of each axis. In a minimal setup, the robot subsequently returns the current axis position. Currently used robot controllers exchange the new axis values with a frequency between 0.1 and 1 kHz. In summary, the minimal goal is to send and receive at least six floating-point values with a frequency of 1 kHz (cycle time = 1 ms).

The performance measurements of the distributed OPC UA Programs are performed on two identical industrial PCs (IPCs) with an Intel Core i7-3615QE quad-core CPU running at 2.30 GHz. Both IPCs are equipped with an Intel i210 network card and are connected via a 1 Gbps link directly with each other. The computers operate with a real-time capable Preempt-RT Linux Kernel 5.0.14. For clock synchronisation between the devices, the standard IEEE 802.1AS-Rev is used. In order to enable TSN communication, an Intel patch for IEEE 802.1Qbv functionality was added.

For the measurement of performance, we instantiated one Master OPC UA Program on one of our IPCs and one Slave OPC UA Program on the other IPC. Both sides send and receive their current state and process data with a cycle time of 0.2 ms (frequency = 5 kHz). Each experiment has a recording period of one hour, which corresponds to approximately 180 million data messages. To check whether the process data size has an impact on the robustness of the system, measurements

Floating-point values (byte)	**Average round trip time [ms]**	**Max. round trip time [ms]**	**Standard deviation [ms]**
1 (8)	0.268	0.310	0.01030
5 (40)	0.285	0.344	0.01057
10 (80)	0.304	0.386	0.01011
15 (120)	0.338	0.559	0.00981
20 (160)	0.369	0.639	0.01190

Table I: Average and maximum round trip time for different payload sizes between 8 and 160 byte (cycle time = 200 μs, measurement period = 1 h, ≈180 million data messages)

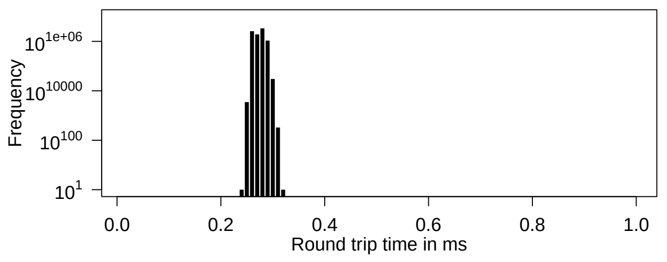
with 1, 5, 10, 15 and 20 floating-point values are performed. This leads to a data size of 8, 40, 80, 120 and 160 byte per transmission, respectively.

Figure 8 shows the results of the experiments with different payload sizes. In this experimental setup, the Master OPC UA Program publishes floating-point values that are subscribed to by the Slave OPC UA Program. After reception of an incoming message, the slave processes the data and immediately publishes the same amount of floating-point values. These values are subscribed to by the Master and processed. As a comparative value, the round trip time is measured from the point of publishing the floating-point values to the point of receiving the returned values from the slave. This includes sending the message on the Master, receiving and processing the data on the slave as well as the retransmission back to the Master.

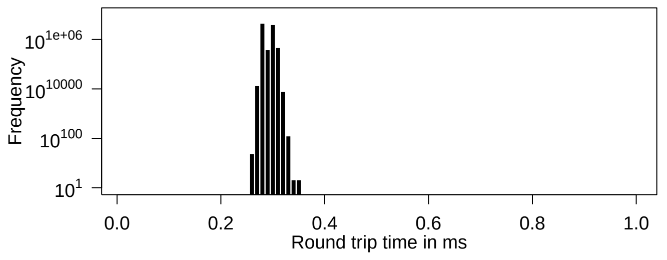
The measured round trip times are plotted in a histogram, with the horizontal axis indicating the measured time in milliseconds and the vertical axis the number of occurrences of that specific time. Figure 8a to 8e show the histogram for measurements with a payload size of 1 (8 bytes), 5 (40 bytes), 10 (80 bytes), 15 (120 bytes) and 20 (160 bytes) floating-point values. It appears that the round trip times are uniformly distributed with a low payload size. With increasing size, the minimal round trip times remain. However, the maximum duration of the message transmissions increases. This is mainly due to longer processing times on the slave. The measured transmission times have not changed significantly.

In addition to the histograms, the average and maximum round trip time was recorded during the measurements and the standard deviation was calculated. Table I shows these records for the different payloads in milliseconds. The experiments show that, as expected, the round trip time increases with growing payload size. But still, there are no major outliers for the round trip time. This is also evident from the almost constant low standard deviations (<0.01190 ms). This indicates that only a few measurements have slightly increased round trip times. In this regard, the logarithmic scale of the histograms in Figure 8a to 8e can be misleading.

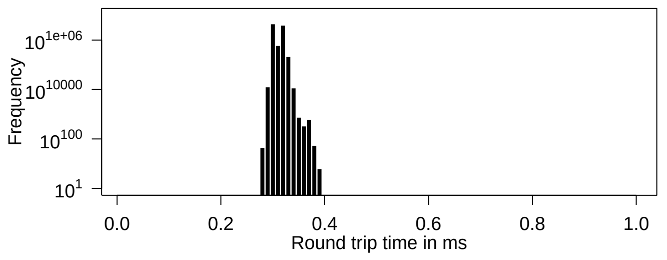
With a maximum round trip time of 0.639 milliseconds and a payload size of 20 floating-point values (160 bytes), we are still far below the initially defined real-time barrier goals specified with 1 millisecond at a payload of six floating-point values (48 bytes). Especially since robot control systems only expect the values every millisecond (1 kHz) and we measure



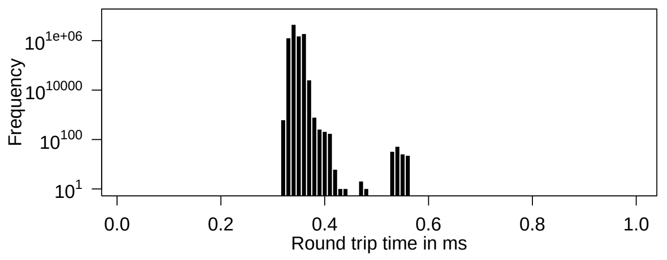
(a) Histogram of round trip times for exchanging one floating-point value (8 bytes)



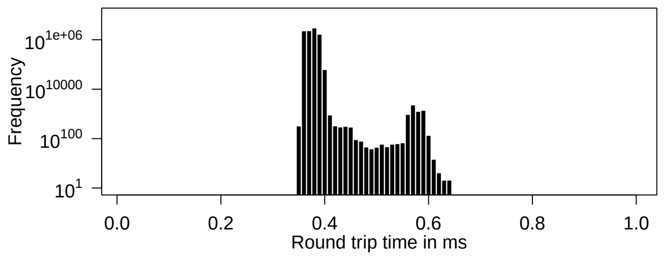
(b) Histogram of round trip times for exchanging 5 floating-point values (40 bytes)



(c) Histogram of round trip times for exchanging 10 floating-point values (80 bytes)



(d) Histogram of round trip times for exchanging 15 floating-point values (120 bytes)



(e) Histogram of round trip times for exchanging 20 floating-point values (160 bytes)

Figure 8: Histograms of round trip times (log scale) with different payload sizes between 8 and 160 byte (cycle time = 200 μs, measurement period = 1 h, ≈180 million data messages)

the time including the transmission of the response. Thus, you could halve the measured round trip times.

VI. Conclusion

In the context of real-time critical control systems, we modeled the problem of running continuous and distributed processes across multiple components which need the possibility of exchanging information in real-time. In order to solve this problem, we proposed an approach to combine the concept of OPC UA Programs and OPC UA PubSub communication over TSN. With OPC UA PubSub over TSN, we can time-synchronize several components and transmit state changes as well as process data in real-time. OPC UA Programs expand these concepts by adding the ability to have continuously running processes. Since the combination of these two techniques is not sufficient to meet all the requirements, we introduced a handshake protocol to start and end the synchronization between various distributed state machines of OPC UA Programs.

We used the open source OPC UA open62541 stack as basic API and extended it with an additional interface to encapsulate communication between multiple state machines. The results of our evaluation shows that it is possible to employ the combination of OPC UA Programs and OPC UA PubSub over TSN for distributed industrial control. The maximum round trip time of 0.639 milliseconds with a symmetrical transmission of 160 bytes payload length shows that a delay of approximately 0.4 milliseconds can be expected for an asynchronous transmission. Thus, a cycle time of 1 to 10 millisecond (0.1 kHz - 1 kHz) with small amounts of data ($<$ 1 kilobyte) is possible without difficulty.

In future experiments, we plan to evaluate and optimize our handshake mechanism to handle the synchronization of multiple Slave OPC UA Programs. In particular, it must be evaluated whether our approach also works with more complex network topologies other than direct connections. Moreover, we are working on the opportunity to coordinate nested OPC UA Programs which requires dealing with data transmitted at different cycle times. In the future, we plan to investigate how our approach can serve as a basis to enable an industrial Plug-and-Produce architecture. Hence, we investigate how real-time capable and reusable skills can be implemented with our approach.

References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & information systems engineering*, vol. 6, no. 4, pp. 239–242, 2014.

[2] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent manufacturing in the context of industry 4.0: a review," *Engineering*, vol. 3, no. 5, pp. 616–630, 2017.

[3] L. Monostori, "Cyber-physical production systems: Roots, expectations and R&D challenges," *Procedia Cirp*, vol. 17, pp. 9–13, 2014.

[4] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia Cirp*, vol. 61, pp. 335–340, 2017.

[5] A. Angerer, M. Vistein, A. Hoffmann, W. Reif, F. Krebs, and M. Schönheits, "Towards multi-functional robot-based automation systems," in *Proc. 12th Intl. Conf. on Informatics in Control, Autom. and Robotics, Rome, Italy*, 2015.

[6] L. Nägele, A. Schierl, A. Hoffmann, and W. Reif, "Multi-robot cooperation for assembly: Automated planning and optimization," in *Informatics in Control, Automation and Robotics, 16th International Conference, ICINCO 2019, Prague, Czech Republic, July 29-31, 2019 Revised Selected Papers*, ser. LNEE. Springer, 2020, to be published.

[7] A. Björkelund, J. Malec, K. Nilsson, and P. Nugues, "Knowledge and skill representations for robotized production," *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 8999 – 9004, 2011, 18th IFAC World Congress. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667016450561

[8] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif, "Service-oriented robotics manufacturing by reasoning about the scene graph of a robotics cell," in *Proc. 41st Intl. Symp. on Robotics, Munich, Germany*. VDE, June 2014, pp. 1–8.

[9] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen, "Robot skills for manufacturing: From concept to industrial deployment," *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.

[10] M. Onori, N. Lohse, J. Barata, and C. Hanisch, "The IDEAS project: plug & produce at shop-floor level," *Assembly automation*, 2012.

[11] J. Pfrommer, D. Stogl, K. Aleksandrov, S. E. Navarro, B. Hein, and J. Beyerer, "Plug & produce by modelling skills and service-oriented orchestration of reconfigurable manufacturing systems," *at-Automatisierungstechnik*, vol. 63, no. 10, pp. 790–800, 2015.

[12] OPC Foundation, *OPC Unified Architecture Part 1: Overview and Concepts*, 2017.

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

[14] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, St. Louis, Missouri, USA*. IEEE, Oct. 2009, pp. 2108–2113.

[15] M. Vistein, A. Hoffmann, A. Angerer, A. Schierl, and W. Reif, "Towards re-orchestration of real-time component systems in robotics," in *IEEE Intl. Conf. on Robotic Computing (IRC)*. IEEE, 2017, pp. 60–68.

[16] OPC Foundation, *OPC Unified Architecture Part 10: Programs*, 2017.

[17] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, "Open source OPC UA PubSub over TSN for realtime industrial communication," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2018, pp. 1087–1090.

[18] OPC Foundation, *OPC Unified Architecture Part 14: PubSub*, 2018.

[19] J. O'Hara, "Toward a commodity enterprise middleware," *Queue*, vol. 5, no. 4, pp. 48–55, May 2007. [Online]. Available: https://doi.org/10.1145/1255421.1255424

[20] International Organization for Standardization, *Information technology - Message Queuing Telemetry Transport (MQTT) v3.1.1 (ISO/IEC 20922:2016)*, 2016.

[21] D. Bruckner, M. Stanica, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, and T. Sauter, "An introduction to OPC UA TSN for industrial communication systems," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 121–1131, 2019.

[22] IEEE. (1999) IEEE 802.1 Working Group. [Online]. Available: https://1.ieee802.org/

[23] K. Dorofeev and A. Zoitl, "Skill-based engineering approach using opc ua programs," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, July 2018, pp. 1098–1103.

[24] S. Profanter, A. Breitkreuz, M. Rickert, and A. Knoll, "A hardware-agnostic OPC UA skill model for robot manipulators and tools," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 1061–1068.

[25] M. Kaspar, J. Bock, Y. Kogan, P. Venet, M. Weser, and U. E. Zimmermann, "Tool and technology independent function interfaces by using a generic OPC UA representation," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2018, pp. 1183–1186.

[26] A. Koubaa, *Robot Operating System (ROS): The Complete Reference (Volume 2)*, 1st ed. Springer Publishing Company, Incorporated, 2017.