



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Time Constraints for OPC UA

Message Passing in the

Industrial Internet of Things

Nelson Igbokwe

Matr. No.: 903610

Master Thesis

for the Degree of

Master of Engineering (M. Eng.)

in the Degree Program

Information and Communications Engineering

at

Beuth University of Applied Sciences Berlin

Berlin, September 2021

1st Examiner: Prof. Dr.-Ing. Dietrich

2nd Examiner: Prof. Dr. Gober

Abstract

As Open Platform Communication Unified Architecture (OPC UA) is a common standard for distributed industrial applications, this thesis investigates an open-source OPC UA implementation written in Python. In combination with the commercial solution of MathWorks, Inc., a system is realized that allows the remote control of a motor. The system comprises an OPC UA Server in Python on a Raspberry Pi and a OPC UA Client in MATLAB. Further, measurements are presented that describe the delay and jitter of a Client-Server application in Python. The results point out limitations in terms of delay and jitter that must be considered as they do not allow the implementation of Hard Real-Time applications.

Code and data of this thesis can be accessed here:

<https://github.com/NelsonIg/OPC-UA>

Table of Contents

1	Motivation.....	1
2	Related Work	3
3	Background	4
3.1	General	4
3.2	Address Space	5
3.3	Services	7
3.4	Data Exchange	8
3.5	Systems Architecture	10
4	OPC UA in Python.....	13
4.1	Server Example	16
4.2	Client Example.....	23
4.3	Network Messages	27
5	Remote Motor Control	31
6	Measurements	39
6.1	Setup	39
6.1.1	Write & Method Invocation.....	40
6.1.2	Notification Messages.....	42
6.2	Results.....	45
7	Conclusions.....	51
I.	List of References	52
II.	List of Figures.....	53
III.	List of Tables	55
IV.	List of Listings	56
	Appendix A	i
	Appendix B.....	vi
	Appendix C	xi

1 Motivation

In industrial environments, the exchange of data between heterogenous devices is essential. Scenarios like detecting objects or controlling a motor already require devices that may be spatially distributed. In the case of a motor control system, the sensor detecting the rounds per minute and the respective controller leveraging the incoming pulses can be far away from the software that controls the system through human interaction. Terms like *Industrial Internet of Things* and *Industry 4.0* nourish the idea of interconnected devices. For this purpose, reliable network communication and standardized middleware are crucial.

Middleware that can be used reliably should be available for multiple programming languages and operating systems. Regarding the motor control, the embedded device sensing the rounds per minute and setting the motor voltage can have a Linux-based operating system. Whereas the software responsible for human interaction may run on a PC on top of Windows. In addition, embedded computers like the Raspberry Pi are usually programmed in C/C++ or Python while a motor control on a PC could be designed in MATLAB/Simulink.

Open Platform Communication Unified Architecture (OPC UA) can be found for many programming languages, such as those listed below:

- Free OPC-UA Library: Python [UAL]
- OPC Toolbox: MATLAB [MAT]
- Open62541: C/C++ [UAC]

Especially, the support for Python and MATLAB draws attention as Python is easy to use and useful in combination with a Raspberry Pi which accelerates and facilitates the prototyping of applications. MATLAB is commonly used in universities and companies for signal processing and simulations. This makes OPC UA an interesting standard to investigate.

When dealing with machine-to-machine communication, time constraints are important as every machine imposes certain constraints to be able to process data reliably. Requirements of processes can be divided into two groups [IOT]:

- **Hard Real-Time** imposes crucial deadlines that must be met for every event.
- **Soft Real-Time** accepts that deadlines are not always met and may jitter.

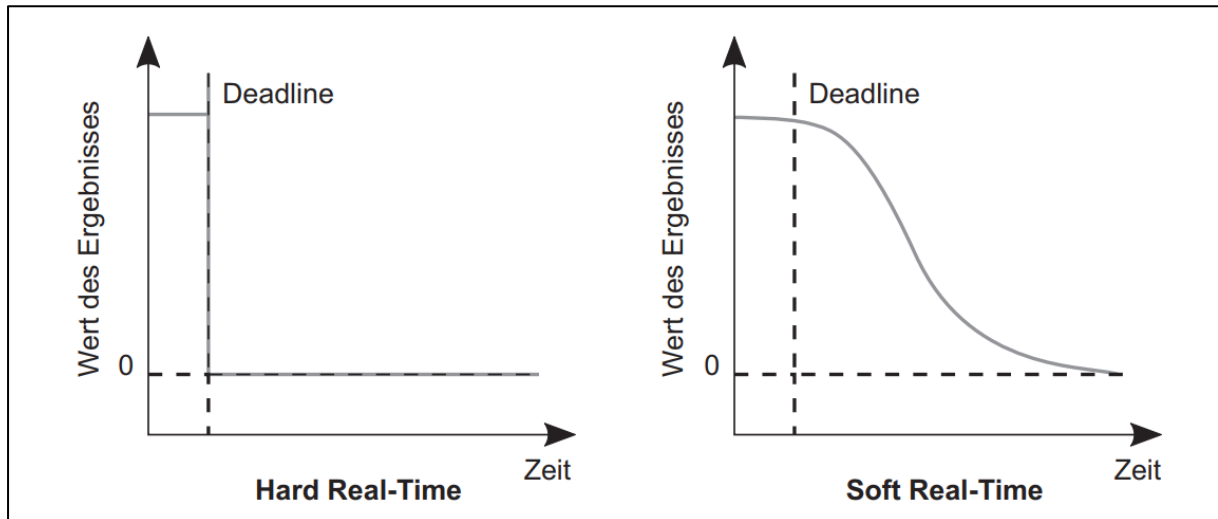


Figure 1: Hard and Soft Real-Time, Source: [IOT, Fig. 8.3]

Figure 1 supports the above explanation of Hard and Soft Real-Time. The y-axis shows the event value with respect to the time (x-axis).

As for OPC UA, this thesis shall provide some insight into what can be expected in terms of time constraints like delay and jitter when implementing an OPC UA Server in Python. For this purpose, a Raspberry Pi will be used to host the Server to allow the controlling of a DC motor. To reproduce an industrial application, the controlling of the motor is executed remotely through an OPC UA Client in MATLAB/Simulink.

2 Related Work

This chapter intends to put the thesis into context with current developments concerning OPC UA.

Feature-based Comparison of Open Source OPC-UA Implementations [FBC]

This paper investigates to what extent open-source implementations of OPC UA provide essential features and functionalities as well as their interoperability. The following implementations are investigated:

- open62541
- node-opcua
- UA-.NETStandard
- python-opcua

It is concluded that basic features are supported by all implementations and that they interconnect successfully. Also, they can be used to substitute or to complement commercial solutions.

Open-Source Implementation of OPC UA PubSub for Real-Time Communication with Time-Sensitive Networking [PUB]

(OT: Open-Source Implementierung von OPC UA PubSub für echtzeitfähige Kommunikation mit Time-Sensitive Networking)

This paper investigates how to use *Time-Sensitive Networking* to provide OPC UA PubSub in real time. For this purpose, the delay and jitter were minimized through a hardware interrupt that controlled the PubSub publisher. This was implemented for open62541 OPC UA SDK.

To contribute to the existing field, this thesis provides deeper insight into the performance (delay, jitter) of an open-source OPC UA implementation in Python. In addition, the interoperability with a commercial solution in MATLAB is demonstrated with a practical industrial application.

3 Background

The OPC UA specification is divided into multiple parts. This chapter presents a brief overview of key aspects of this standard.

3.1 General

OPC UA is a standard that is intended to be applicable to all industrial domains (sensors, actuators, ERP, etc.) and to facilitate the exchange of information as well as the controlling of industrial processes [OPC1]. In addition, OPC UA provides a meta model which allows an extensible description of data.

A key property of OPC UA is that it is a platform-independent standard that allows various entities to communicate by sending either request and response messages between Clients and Servers or network messages between Publishers and Subscribers over different types of networks [OPC1]. In each case, the Server defines what information is accessible and how it can be accessed.

The collection of information made visible to the Client is called *Address Space*, which can be accessed through services that are similar to method calls in programming languages [OPC1]. Clients that view items of the Address Space are provided with type definitions. This *Information Model* enables a concise description of the content of the Address Space. The standard also supports many formats for encoding data, such as binary structures, XML or JSON documents [OPC1]. The extent to which these formats can be used depends on the libraries implementing the OPC UA standard.

3.2 Address Space

As mentioned above, the Address Space represents the entire information visible to the Client. It comprises a set of nodes, which are represented according to the Information Model. In this model, object-oriented techniques are made use of as well as hierarchies and inheritances. For this purpose, node classes are defined which are referred to as metadata for the Address Space. Each existing node is an instance of one of these node classes from which fixed attributes are inherited. Some attributes may be optional like the description attribute and others mandatory like the node ID for identifying the node [OPC3]. The most important node classes are object, variable and method [HIOT].

- **Object:** Represents real-world systems and components since it can group other nodes.
- **Variable:** Contains a value that a Client can read/write or subscribe to.
- **Methods:** Callable functions that allow input and output parameters like starting a motor.

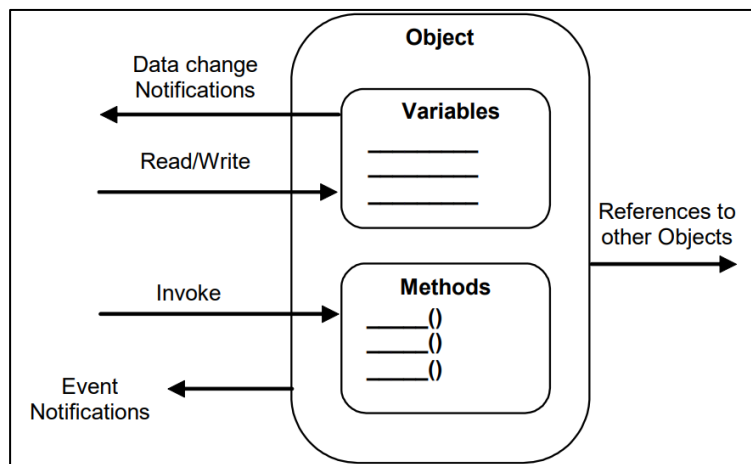


Figure 2: OPC UA Object Model, Source: [OPC3, Fig. 2]

Figure 2 shows the OPC UA *Object Model* from the OPC UA specifications containing the three most important node classes, variable, object and method, as well as related services to access them. Data change Notifications occur if Clients subscribe to variables by using the Monitoring and Subscription Services. Invoking methods leads to a process whose execution is bound to the Server and a result returned to the Client [OPC3] just like a remote procedure call (RPC). Below, the Object Model is applied to describe a simple DC motor with the respective start and stop methods. Its implementation will follow later.

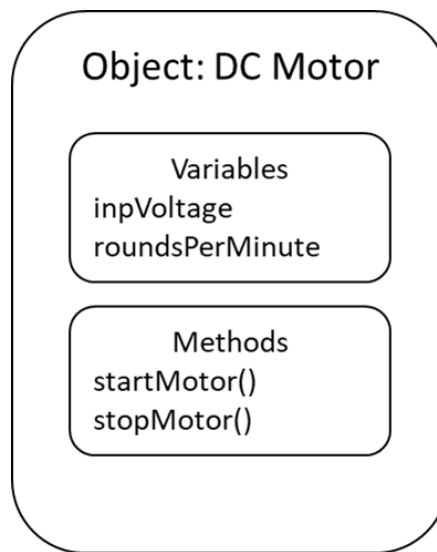


Figure 3: OPC UA DC Motor
Example

3.3 Services

Client and Server communicate through an interface provided by services, which are organized in service sets. OPC UA services are capable of two things. They allow Clients to create requests/subscriptions to Servers and to receive the respective responses. This section presents a selection of important services defined by the OPC UA specification [OPC4].

The **Discovery Service Set** groups services that permit Clients to discover endpoints of a Server and to read the security configurations required for the connection. This process does not require the establishment of a session.

After accessing the Discovery Endpoint, the **Secure Channel Service Set** creates a communication channel that ensures the confidentiality and integrity of messages. A Secure Channel describes a long-running logical connection between a single Server and a single Client, in which keys are used for authentication and encryption.

The **Session Service Set** establishes an application layer connection referred to as a session uniquely identified by its session ID and an authentication token. The latter is needed to associate an incoming request with a session.

Adding, modifying, and deleting nodes is handled by the **Node Management Service Set**. Nodes will not cease to exist even if the Clients that created them disconnect from the Server.

Read and write operations are services provided by the **Attribute Service Set**. For the write operation, it holds that the service does not return until the value has been written to the target attribute or the Server determines that the value cannot be written to. Similar to attributes, methods are invoked by means of the **Method Service Set**.

To subscribe to data, two service sets are used. A *Monitored Item* is created with the help of the **Monitored Item Service Set** to monitor value changes (attributes) or events (objects). The **Subscription Service Set** creates subscriptions, which send notifications generated by Monitored Items to the Client.

3.4 Data Exchange

There are two ways to exchange data between an OPC UA Client and Server. One option is to make use of the read and write services offered by the Attribute Service Set. Another way of transmitting data is to subscribe to notifications or events.

Read Service

This service is needed to read multiple attributes of various nodes. An important parameter of this service is the *max age* parameter, which informs the Server to access the value of a data source if the data is older than specified by *max age* [OPC4].

Write Service

As with the Read Service, multiple nodes can be written to. Some essential parameters are as follows:

- **Nodes to write:** List of nodes and their attributes to write to
- **Node ID:** Node ID of the node that contains the attributes
- **Attribute ID:** ID of the attribute
- **Value:** Value to write to

Subscription

Notifications are constantly created by Monitored Items until they are deleted, or the subscription is cancelled. Subscriptions gather these notifications and put them into notification messages to be published to the Subscriber. Several necessary parameters [OPC4] are listed below:

- **Publishing Interval:** Time interval at which the subscription sends notification messages.
- **Keep-alive Counter:** Counter set by the Client to count the consecutive publishing cycles without notifications sent to the Client. When its maximum is reached, a keep-alive message tells the Client that the subscription is still active.
- **Lifetime Counter:** Counts the number of consecutive publishing cycles, defined by the publishing interval, in which no publishing requests have been available to send a publishing response. When expired, the subscription is closed.

3.5 Systems Architecture

The OPC UA Client architecture [OPC1] defined by three important elements is shown below.

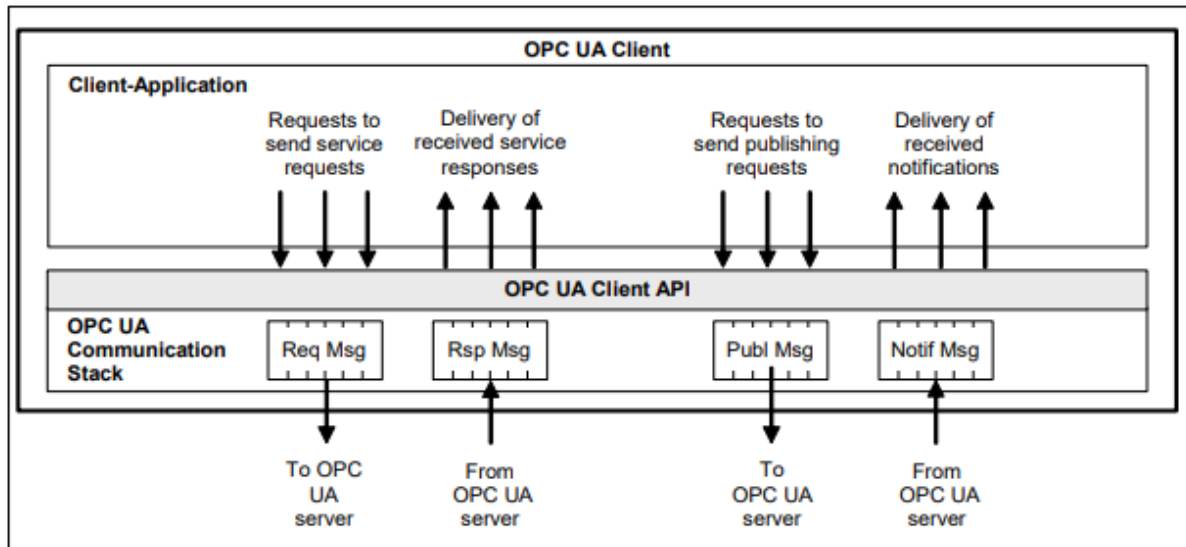


Figure 4: OPC UA Client Architecture, Source: [OPC1, Fig. 4]

Code that implements the function of the Client is found in the *Client-Application*. The *Client API* is an interface isolating application and *OPC UA Communication Stack*. API calls are converted by the underlying communication stack and then sent through the network. The architecture of Client and Server is the same with the exception that the Address Space, Monitored Items and Subscriptions are on the Server side in the application layer. Figure 4 also illustrates the various services used to request and send data as it shows the request and response messages for the read/write services and the publishing and notification messages of the subscription services.

In addition, some examples from the OPC UA specifications [OPC1] describing Server Client interactions are shown.

Server Aggregation

An OPC UA system is modelled to be able to contain multiple Clients and Servers, that concurrently communicate with one or more entities. Combining Client and Server in a component is also possible. This allows the creation of multi-tiered architectures such as the one in Figure 5, which enables the aggregation of data from lower-layer Servers [OPC1].

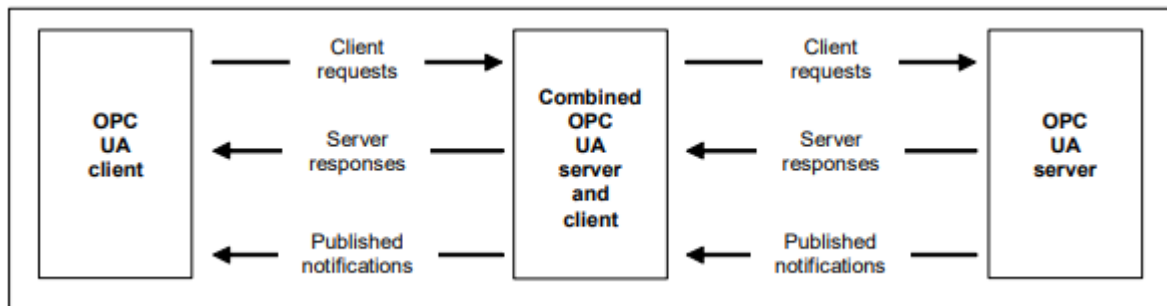


Figure 5: OPC UA System Architecture, Source: [OPC1, Fig. 3]

Publish & Subscribe

For Publish-Subscribe, two use cases are described in the specification, a broker-less and a broker-based form. For the broker-less form, the network infrastructure serves as the Message-Oriented Middleware (MOM), while in the case of a broker-based form, standard protocols like MQTT are used for the communication without establishing a session. Both cases are depicted below.

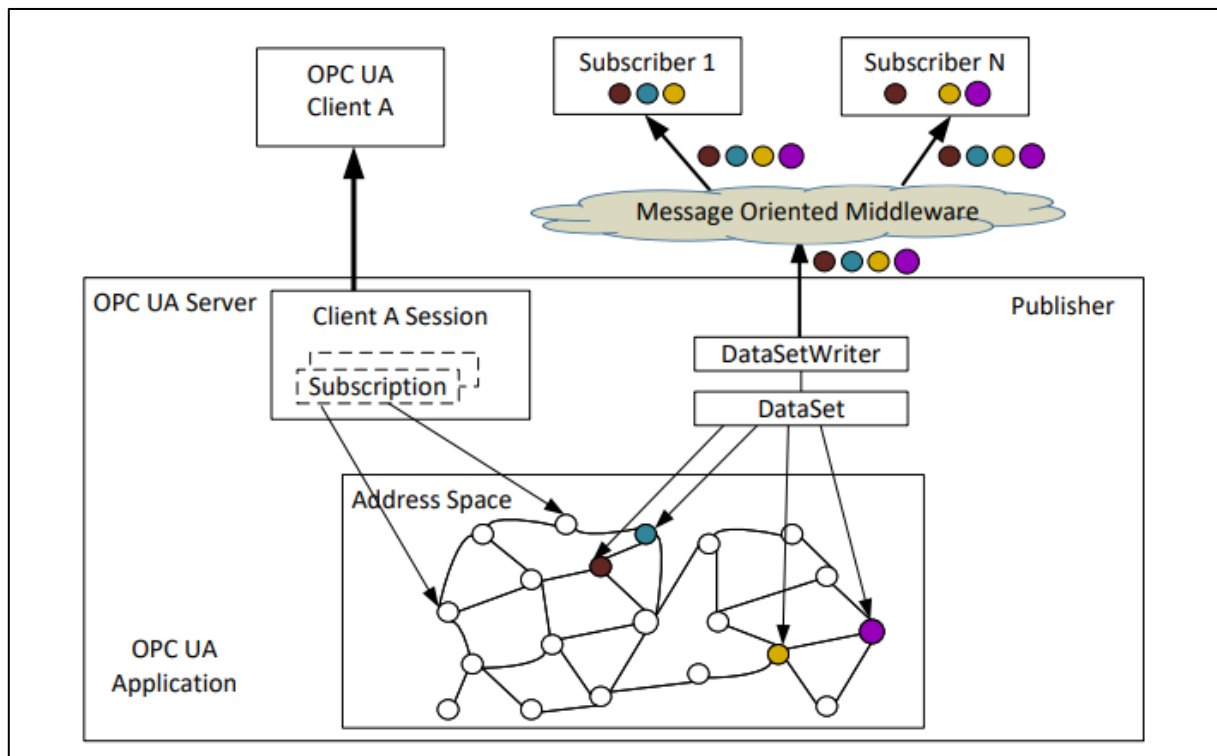


Figure 6: Publish-Subscribe Models, Source: [OPC1, Fig. 8]

4 OPC UA in Python

There are two open-source modules for Python that allow users to implement OPC UA Clients and Servers, *python-opcua* and *opcua-asyncio*. Both can be freely accessed on GitHub from the Free OPC-UA Library [UAL]. As *opcua-asyncio* supports asynchronous programming, it will be used here.

The first two sections of this chapter show the implementation of a simple Client and Server application. Here, fundamental services described in chapter 3 are applied. Then, with the help of Wireshark, the communication between Server and Client will be further investigated to display how application code translates to network messages. Figure 7 presents the application to be implemented.

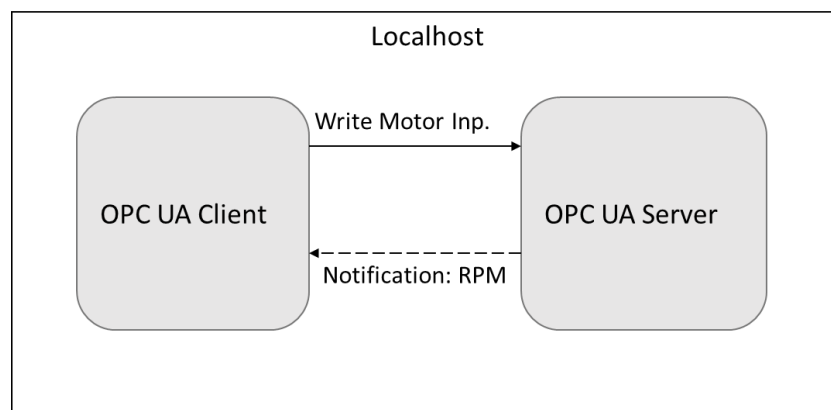


Figure 7: OPC UA – Client Server Example

To provide a better understanding of the following code examples, asynchronous programming in python is briefly explained. This way of programming is particularly useful for dealing with IO-bound network applications [AIO]. The key property of an asynchronous application is the *async-await* syntax. Here, *await* defines a process that may need some time to complete like a client-server request. In the meantime, other coroutines are allocated processing time. Every function that calls *await* must be declared as a coroutine by using the *async* key word. When simply calling a coroutine, a coroutine object is returned. In order to run it, *asyncio.run()* is used together with the coroutine as an input argument.

Listing 1 shows the output of an application that simulates the downloading of data from Tokyo and Munich. The data is downloaded twice, once sequentially and once concurrently.

```
Download data sequentially
Downloaded data from Tokyo
Downloaded data from Munich
Downloads took 12s
Download data concurrently
Downloaded data from Munich
Downloaded data from Tokyo
Downloads took 10s
```

Listing 1: Terminal Output for Concurrent & Sequential Application

The respective source code is shown below.

```
import asyncio, time

async def download_data(destination, delay):
    await asyncio.sleep(delay)
    print(f'Downloaded data from {destination}')

async def main():
    print('Download data sequentially')
    t1 = time.perf_counter()
    await download_data('Tokyo', 10)
    await download_data('Munich', 2)
    t2 = time.perf_counter()
    print(f'Downloads took {round(t2-t1)}s')

    print('Download data concurrently')
    task1 = asyncio.create_task(download_data('Tokyo', 10))
    task2 = asyncio.create_task(download_data('Munich', 2))
    t1 = time.perf_counter()
    await task1
    await task2
    t2 = time.perf_counter()
    print(f'Downloads took {round(t2-t1)}s')

asyncio.run(main())
```

Listing 2: Sequential and Concurrent Application

This simple example shows how to use coroutines sequentially and concurrently. For both cases, the main application is blocked until every function has returned. Processing time can be shared when calling the function as *tasks* that are created by means of *asyncio.create_task*. Note that the functions are not executed in parallel. Functions that permanently block must be allocated a new thread.

4.1 Server Example¹

The following properties shall be implemented into the Server:

- DC Motor
 - Input variable
 - Rounds per minute variable
 - Start and stop methods
- Simulation of DC motor process

Before the Address Space is created, a Server and the corresponding endpoint must be initialized as shown below.

```
# init server, set endpoint
server = Server()
await server.init()
server.set_endpoint(f"opc.tcp://{host}:4840/server_example/")
```

Listing 3: Example Server – Initialization

To correctly reference the nodes within the Address Space, a namespace is created. The variable *idx* indicates the node ID.

```
# setup of namespace
uri = "example-uri.edu"
idx = await server.register_namespace(uri)
```

Listing 4: Example Server – Namespace

Now that the Server is initialized, a new node class for the DC motor can be defined.

¹ For full code, see Appendix A-1

Three methods are used to create the new node class:

- *add_object_type*: Return base object
- *add_variable*: Add attribute to base object, define default value
- *set_modelling_rule*: Enable/disable instantiation of attributes together with object

```
# Create new object type
base_obj_motor = await server.nodes.base_object_type.add_object_type(
    nodeid=idx, bname="BaseMotor")

base_var_rpm = await base_obj_motor.add_variable(
    nodeid=idx, bname="RPM", val=0.0)

base_var_inp = await base_obj_motor.add_variable(
    nodeid=idx, bname="Input", val=0.0)

# ensure that variable will be instantiated together with object
await base_var_rpm.set_modelling_rule(True)
await base_var_inp.set_modelling_rule(True)
```

Listing 5: Example Server – Motor Base Object

The argument *bname* defines the browse name visible to the Client. To ensure that an instance of *base_obj_motor* is automatically provided with the parent attributes, *set_modelling_rule* is called.

After creation of the DC motor node class, the Address Space is populated.

```
# Address Space
dc_motor = await server.nodes.objects.add_object(nodeid=idx, bname="Motor",
                                                  objecttype=base_obj_motor)

global dc_motor_inp, dc_motor_rpm
dc_motor_inp = await dc_motor.get_child(f'{idx}:Input')
dc_motor_rpm = await dc_motor.get_child(f'{idx}:RPM')
await dc_motor_inp.set_writable()
await dc_motor_rpm.set_writable()
```

Listing 6: Example Server – Instantiation of Motor Object

The instantiation of *dc_motor* is similar to the creation of the node class. When calling *add_object*, an object type must be delivered as an argument together with the node ID and the browse name. The importance of the node ID is obvious when extracting a variable from the *dc_motor*. The method *get_child* expects a path which describes the hierarchy of the Address Space. Finally, permission to write to the variables is enabled with *set_writable*.

To conclude the population of the Address Space, two methods are added.

```
await dc_motor.add_method(idx, # nodeid
                          "start_motor", # browse name
                          start_motor, # method to be called
                          [], # list of input arguments
                          [], # list output arguments)

await dc_motor.add_method(idx, "stop_motor", stop_motor, [], [])
```

Listing 7: Example Server – Add Methods to Address Space

In addition to the node ID and the browse name, the target function and its input and output arguments must be provided. Since neither method requires any arguments, two empty lists are passed.

Once the Address Space is successfully created, the Server is started.

```
# create task for simulation of dc motor
task_motor_sim = asyncio.create_task(motor_simulation())
# start
async with server:
    while True:
        await task_motor_sim # runs concurrently with main()
        await asyncio.sleep(1)
```

Listing 8: Example Server – Start

Server and Client can be used as a context manager, which is convenient since it handles the connecting and disconnecting automatically.

The start and stop methods are defined through the decorator *uamethod*. This makes the creation of methods easier as the decorator automatically executes necessary conversions of arguments and outputs. The methods simply set the flag *MOTOR_STARTED*, which is used in *motor_simulation*.

```
@uamethod
async def start_motor(parent):
    global MOTOR_STARTED
    if not MOTOR_STARTED:
        MOTOR_STARTED = True
        print("Motor started.")

@uamethod
async def stop_motor(parent):
    global MOTOR_STARTED
    if MOTOR_STARTED:
        MOTOR_STARTED = False
        print("Motor stopped")
```

Listing 9: Example Server – Start Stop Methods

To simulate a DC motor, the function *motor_simulation* is defined to simply increase *dc_motor_rpm* depending on the input variable *dc_motor_inp* of *dc_motor*. This process is only started when *start_motor* is called. It is stopped with *stop_motor*.

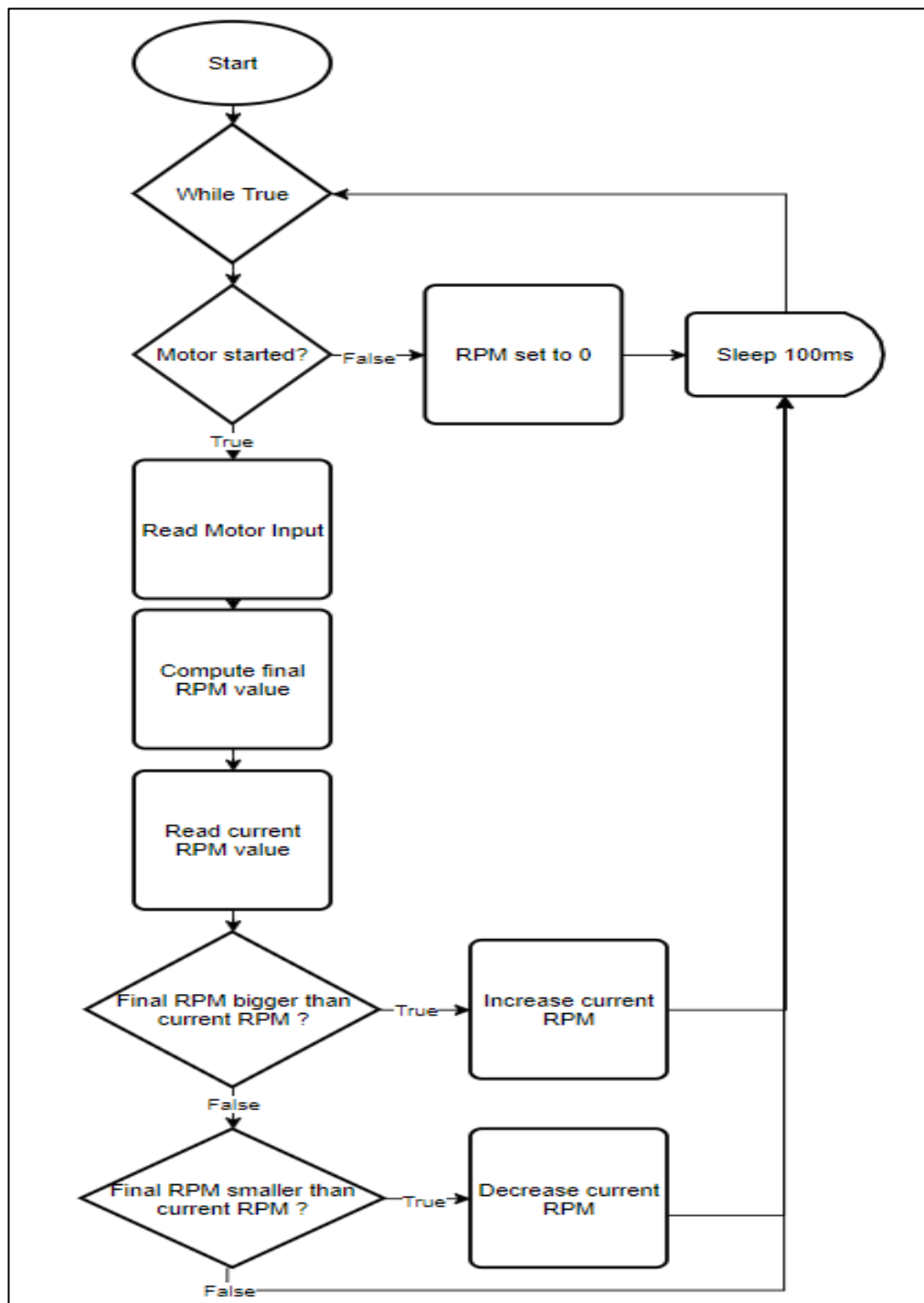


Figure 8: Example Server – Motor Simulation (Flowchart)

Browsing the Server

Before implementing the Client, the properties of the Server can already be discovered. For this purpose, an open-source OPC UA Client [GUI] with a graphical user interface is made use of. This is especially useful to understand the hierarchical structure of the Address Space.

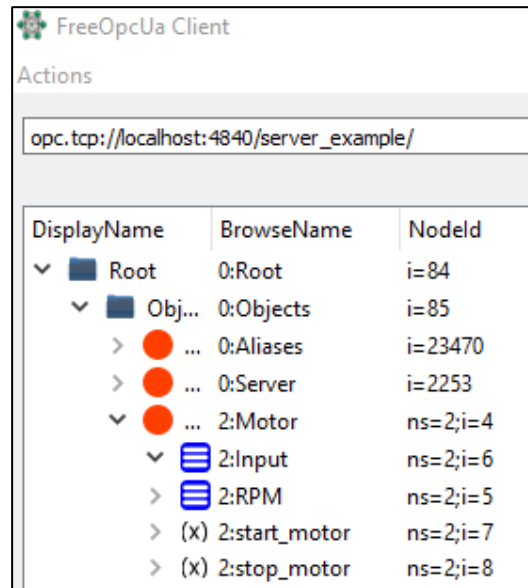


Figure 9: OPC UA Example Server –
Browsed Address Space

Figure 9 shows the created objects with *Motor* grouping the variables and methods. Services for invoking methods or for writing to variables are also accessible. This way, the functionalities of the Server can be tested as in Figure 10. Note that the input of the motor does not represent the voltage.

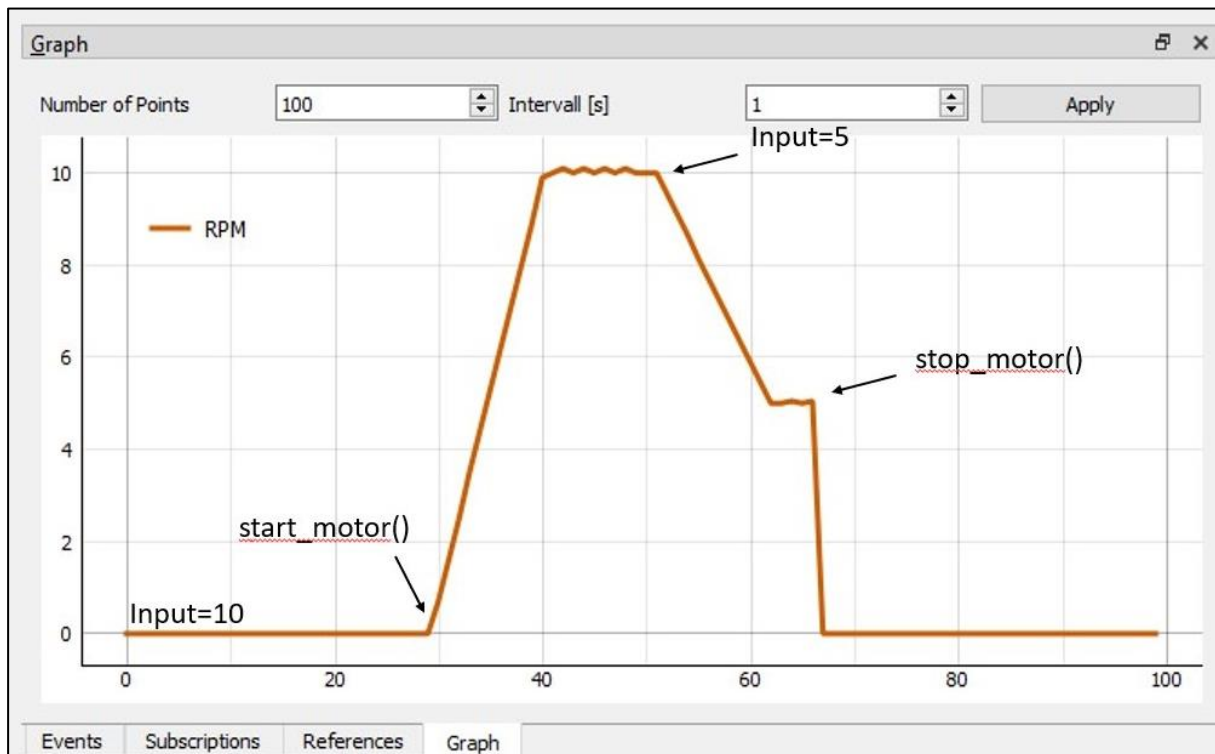


Figure 10: Value of RPM over Time

In Figure 10, the value of *RPM* is plotted over time. Through the GUI of the Client, *Input* is first set to 10, then *start_motor* is called and *RPM* increases. When setting *Input* to 5, *RPM* decreases and with *stop_motor* *RPM* is set to 0.

4.2 Client Example²

Instead of polling the current rounds per minute, the Client shall be notified every time the value changes. This is achieved through a subscription to the *RPM* variable. Every Subscription is an instance of a subscription handler. This handler is a class that follows a specific scheme in order to successfully handle notification messages. For convenience, one can inherit from a predefined handler class called *SubHandler*.

```
class SubscriptionHandler (SubHandler):

    def datachange_notification(self, node: Node, val, data):
        """
        This method will be called when the Client received
        a data change message from the Server.
        """
        global STOP_FLAG
        _logger.info(f'datachange_notification node: {node} value: {val}')
        if val > 1:
            STOP_FLAG = True
```

Listing 10: Example Client – Subscription Handler

For every data change notification, the method *datachange_notification* is called. If it is spelled differently, it will not be called. The stop condition for the motor is set to one, which sets *STOP_FLAG* to indicate that the motor should be stopped.

² For full code, see Appendix A-2

After starting the Client (context manager), objects are retrieved from the Address Space. Here, a path indicating the hierarchy of the Address Space must be passed to the method `get_child`.

```
server_endpoint = "opc.tcp://localhost:4840/server_example/"
client = Client(url=server_endpoint)
async with client:
    idx = await client.get_namespace_index(uri="example-uri.edu")

    # get motor object, here path from root folder
    motor_obj = await client.nodes.root.get_child(
        ["0:Objects", f"{idx}:Motor"])

    # get motor input variable, here path starting from Object folder
    motor_inp = await client.nodes.objects.get_child(
        path=[f"{idx}:Motor", f"{idx}:Input"])

    # get motor rpm variable
    motor_rpm = await client.nodes.objects.get_child(
        path=[f"{idx}:Motor", f"{idx}:RPM"])
```

Listing 11: Example Client – Get Address Space Objects

Once, the Variable *motor_rpm* is retrieved, it can be subscribed to as shown below.

```
# subscription
sub_handler = SubscriptionHandler()
subscription = await client.create_subscription(period=50,
                                                handler=sub_handler)

# subscribe to data change, only current data queuesize=1
await subscription.subscribe_data_change(nodes=motor_rpm, queuesize=1)
```

Listing 12: Example Client – Subscribe to Variable

In this example, the Client will first write to *motor_inp*, then invoke the method *start_motor* and finally call *stop_motor* if *STOP_FLAG* is set.

```
global STOP_FLAG # flag to stop motor
await motor_inp.write_value(1)
_logger.info('wrote 1 to input value')
await motor_obj.call_method(f'{idx}:start_motor')
_logger.info('motor started')
while True:
    await asyncio.sleep(0.005)
    if STOP_FLAG:
        await motor_obj.call_method(f'{idx}:stop_motor')
        _logger.info('motor stopped')
        STOP_FLAG = False
        subscription.delete()
        await asyncio.sleep(1)
        break
```

Listing 13: Example Client – Server Motor Control

When starting Server and Client, the manipulated variables are printed to the terminal on the Server side. The terminal output of the Server is shown in Listing 14.

```
Motor started.  
Input: 1  
RPM: 0  
Input: 1  
RPM: 0.1  
Input: 1  
RPM: 0.2  
Input: 1  
RPM: 0.30000000000000004  
Input: 1  
RPM: 0.4  
Input: 1  
RPM: 0.5  
Input: 1  
RPM: 0.6  
Input: 1  
RPM: 0.7  
Input: 1  
RPM: 0.7999999999999999  
Input: 1  
RPM: 0.8999999999999999  
Input: 1  
RPM: 0.9999999999999999  
Motor stopped  
RPM: 0
```

Listing 14: Example Server – Terminal Output

With the help of Wireshark, the communication between Server and Client can be analysed. The next section investigates what kind of messages are passed.

4.3 Network Messages

This section deals with the network messages sent between Server and Client in the preceding example. For this purpose, the exchange of messages was captured by means of Wireshark to display the messages of the various services. As OPC UA uses TCP in this example, there is a handshake at the start and at the end of a connection as well as acknowledgment messages between consecutive OPC UA messages. These will be omitted from the thesis for simplification. Further, the use of authentication mechanisms (username/password, certificates) also results in additional overhead.

Secure Channel

Establishing a secure channel and creating a session requires the following messages:

- *OpenSecureChannelRequest/ -Response*
- *CreateSessionRequest/ -Response*
- *ActivateSessionRequest/ -Response*

9	0.015616	::1	::1	OpcUa	136 Hello message
11	0.016038	::1	::1	OpcUa	92 Acknowledge message
13	0.016673	::1	::1	OpcUa	196 OpenSecureChannel message: OpenSecureChannelRequest
15	0.017401	::1	::1	OpcUa	199 OpenSecureChannel message: OpenSecureChannelResponse
17	0.018216	::1	::1	OpcUa	363 UA Secure Conversation Message: CreateSessionRequest
19	0.019037	::1	::1	OpcUa	632 UA Secure Conversation Message: CreateSessionResponse
21	0.020241	::1	::1	OpcUa	220 UA Secure Conversation Message: ActivateSessionRequest
23	0.020903	::1	::1	OpcUa	160 UA Secure Conversation Message: ActivateSessionResponse

Figure 11: Wireshark – Secure Channel & Session

Browsing Address Space

After establishing the session, the three objects (see Listing 11) are requested and received from the Address Space through

- *TranslateBrowsePathsToNodeIdsRequest*
- *TranslateBrowsePathsToNodeIdsResponse*.

Each response returns the Namespace Index and the Identifier for later use.

29	0.023068	::1	::1	OpcUa	173	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
31	0.023776	::1	::1	OpcUa	143	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
33	0.024461	::1	::1	OpcUa	171	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
35	0.025144	::1	::1	OpcUa	143	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
37	0.025717	::1	::1	OpcUa	169	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
39	0.026455	::1	::1	OpcUa	143	UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse

Figure 12: Wireshark – Request Objects of Address Space

Subscription

With the information about the objects, a subscription and a monitored item can be created on the side of the Server by means of the following messages:

- *CreateSubscriptionRequest/-Response*
- *CreateMonitoredItemsRequest/-Response*

41	0.027047	::1	::1	OpcUa	148	UA Secure Conversation Message: CreateSubscriptionRequest
43	0.027554	::1	::1	OpcUa	136	UA Secure Conversation Message: CreateSubscriptionResponse
45	0.028627	::1	::1	OpcUa	183	UA Secure Conversation Message: CreateMonitoredItemsRequest
47	0.028935	::1	::1	OpcUa	130	UA Secure Conversation Message: PublishRequest
49	0.029375	::1	::1	OpcUa	147	UA Secure Conversation Message: CreateMonitoredItemsResponse

Figure 13: Wireshark – Subscription & Monitored Item

Note that right after the *CreateMonitoredItemsRequest*, a *PublishRequest* is sent.

Write & Read

Writing to and reading from variables is done as follows:

- *WriteRequest/-Response*
- *ReadRequest/-Response*

Below, recorded packets of a write operation are shown. The request contains the browsed Namespace Index and the Identifier of the target variable.

51	0.030097	::1	::1	OpcUa	167 UA Secure Conversation Message: WriteRequest
53	0.030735	::1	::1	OpcUa	128 UA Secure Conversation Message: WriteResponse

Figure 14: Wireshark – Write Operation

Methods

The similarity of methods and write operation is obvious, when comparing the exchanged packets:

- *TranslateBrowsePathsToNodeIdsRequest/-Response*
- *CallRequest/-Response*

Calling a method results in a repeating overhead due to the browsing of the Address Space. This is not the case for write operations as two functions separate browsing from writing.

55	0.031447	::1	::1	OpcUa	162 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
57	0.032234	::1	::1	OpcUa	143 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
59	0.032882	::1	::1	OpcUa	148 UA Secure Conversation Message: CallRequest
61	0.033665	::1	::1	OpcUa	140 UA Secure Conversation Message: CallResponse

Figure 15: Wireshark – Method Invocation

Notification Messages

The Server only publishes notification messages if the Client sends a request. Once the data value changes, *PublishResponse* messages are sent. Note that the first *PublishRequest* was sent after creating the subscription (see Figure 13). Since the data does not change before calling *start_motor*, no *PublishResponse* was sent back.

63	0.068158	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse
65	0.069184	::1	::1	OpcUa	138	UA Secure Conversation Message: PublishRequest
67	0.192884	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse
69	0.193861	::1	::1	OpcUa	138	UA Secure Conversation Message: PublishRequest
71	0.241189	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse
73	0.242129	::1	::1	OpcUa	138	UA Secure Conversation Message: PublishRequest
75	0.351283	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse
77	0.352290	::1	::1	OpcUa	138	UA Secure Conversation Message: PublishRequest
79	0.461908	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse
81	0.462849	::1	::1	OpcUa	138	UA Secure Conversation Message: PublishRequest
83	0.574454	::1	::1	OpcUa	196	UA Secure Conversation Message: PublishResponse

Figure 16: Wireshark – Publishing Response & Publishing Request

By recording the communication between Server and Client, the difference between invoking a method and writing to a variable becomes visible. Even though both require just one line of code, calling a method requires four messages to be sent and received, whereas a write operation only needs two. This is because the variables are already retrieved and browsed in the Client setup (see Listing 11 and Figure 12).

5 Remote Motor Control

To model an industrial application, a Server-Client architecture is implemented to allow the controlling of a motor remotely. The table below lists additional components and software needed for this application.

Software				
MATLAB/Simulink R2020b			OPC Toolbox	
Hardware				
Description	Manufacturer	Manufacturer No.	Vendor	Vendor No.
DC Motor	Joy-it	com-Motor01	Conrad	1573543 - 62
Motor Driver	Joy-it	SBC-MotoDriver2	Conrad	1573541 - 62
Speed Sensor	Joy-it	SEN-Speed	Conrad	1646891 - 62

Table 1: Bill of Material for Remote Motor Control

MATLAB and the OPC Toolbox are freely available through the student licence of the Beuth University of Applied Sciences. Motor, sensor and driver can be purchased for a small amount of money. The whole architecture of this application is shown below.

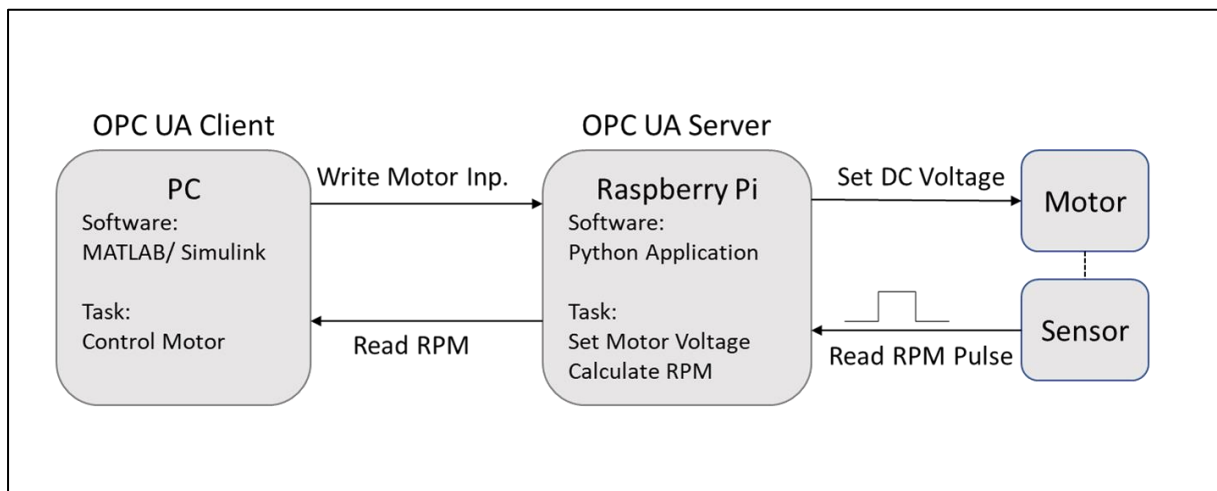


Figure 17: Remote Motor Control – Architecture

MATLAB Client

MathWorks provides a documentation that explains the OPC Toolbox in more detail [MAT]. In addition to OPC UA, other standards such as OPC Data Access and Historical Data Access are also supported. For the purpose of this application, the control of a remote motor through an OPC UA Server, it is necessary to obtain access to data with the help of an OPC UA Client. The respective library [MUA] supports the basic services that are needed for this application.

- Server Connection
- Namespace Browsing
- Read and Write
- Node Information

A support for Publish and Subscribe was not found.

To control the remote motor, an OPC UA Client needs to be implemented to write to the input node of the motor object as well as read from the rpm node. Also, the Client needs to be integrated as a function into a Simulink model. From the model's point of view, the function simply consists of one input and one output. Hence, the entire Server-Client application will be perceived as a monolithic system. To the Simulink model, it is not important where the motor is located. Only the input and output values are essential.

For demonstration purposes, the controller is kept simple. The main task is to compare the desired rounds per minute to the current rounds per minute. The error, which is the difference between the two, will be weighted with a step size and eventually added to the former input to yield the new input of the motor. To enable a convenient operation of this application, a dashboard will contain options like changing the rounds per minute and turning the motor on or off. The dashboard is shown in Figure 18.

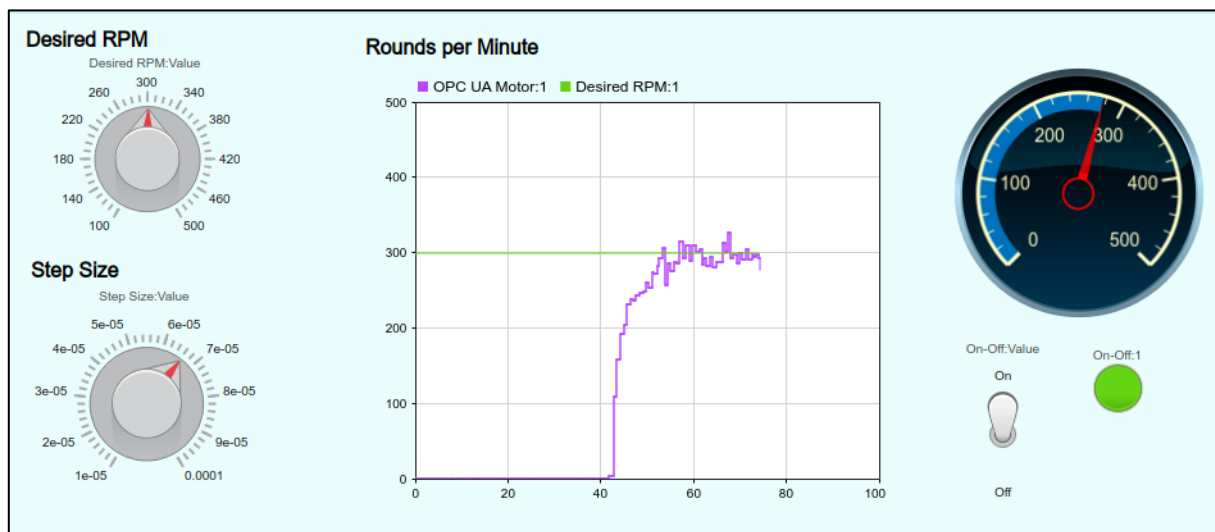


Figure 18: Remote Motor Control – Dashboard

The Simulink model is designed as follows. Some of the blocks are explained below the figure.

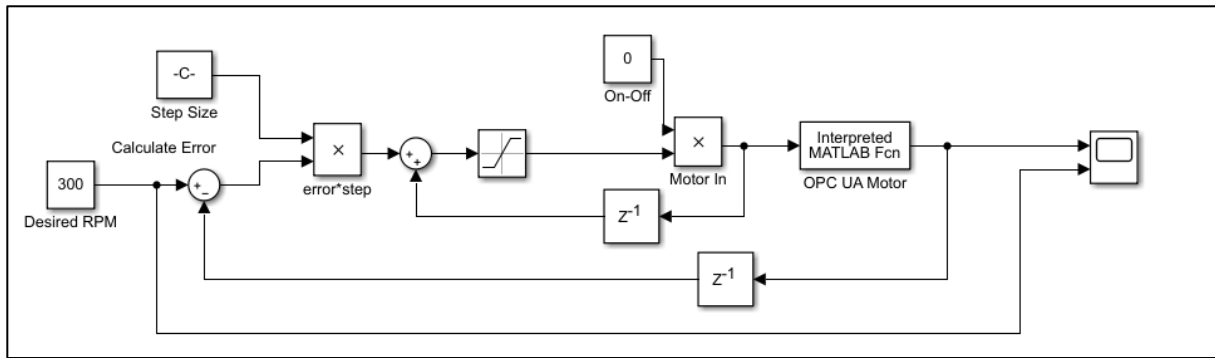




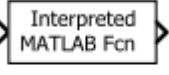
Figure 19: Remote Motor Control – Simulink Model

- 

Delay of 1 sample.

Delay
- 

Limit signal to upper and lower bounds.

Saturation
- 

Include MATLAB function in model (here: OPC UA Client).

Interpreted MATLAB Function

The start and stop functionality is implemented by multiplying the input value of the *Interpreted MATLAB Function* by a constant, which may have the value zero or one. To prevent input values that are not supported, upper and lower limits are defined in the upper branch of the model. Here, the upper limit is one and the lower limit is zero as the python application on the Server side does not support other values. Key tasks of the python application will be explained later.

The script for the OPC UA Client shall execute the following tasks:

1. Create an OPC UA Client and connect to a Server.
2. Browse the Namespace and retrieve nodes.
3. Write to and read from nodes.

The implementation is shown below.

```
function [outputVal] = read_write_opcua(inputVal)

persistent uaClient; % opcua client
persistent initClient; % variable to track initialised clients
persistent rpmNode; % OPC UA Node for rpm of motor
persistent inpNode; % OPC UA Node for motor input [0-1]
persistent initNodes; % variable to track initialised Nodes

host = '192.168.0.183';
port = 4840;

% Connect to server if no client was initialised
if (isempty(initClient))
    uaClient = opcua(host, port);
    connect(uaClient);
    initClient = 1;
% reconnect if connection lost
elseif uaClient.isConnected == 0
    connect(uaClient);
end

if isempty(uaClient) == 0
    % get nodes, if no nodes are already defined
    if uaClient.isConnected == 1 && (isempty(initNodes))
        rpmNode = opcuanode(2, 5);
        inpNode = opcuanode(2, 6);
        initNodes = 1;
    end

    % read from & write to server
    if uaClient.isConnected == 1 && (isempty(initNodes)) == 0
        writeValue(uaClient, inpNode, inputVal);
        % pause(0.100); % time for the motor to react --> not needed
        [rpmVal, ~, ~] = readValue(uaClient, rpmNode);
        outputVal = double(rpmVal);
    end
end
```

Listing 15: Remote Motor Control – MATLAB OPC UA Client

First, *persistent* variables are declared to store their values between function calls [PER]. This is needed to retain the information about the Client and the Server. Otherwise, the function would reconnect to the Server when called again. When first called, the Address Space is browsed to retrieve the motor input (here: *inpNode*) and the rounds per minute (here: *rpmNode*). Eventually, the input argument *inputVal* is written to *inpNode* and the respective rounds per minute are read from *rpmNode*. Between both function calls, an optional pause can be implemented to ensure that the motor can adjust to the new value. For this simple example, this aspect was not further investigated. Finally, the new rounds per minute are returned.

Python Server³

Apart from the populated Address Space, the python application continuously detects new pulses from the speed sensor and calculates the time difference between incoming edges. A call-back function is defined to be triggered when a rising edge is detected. Information about the detected edge and the time of its occurrence are forwarded to a subprocess. This subprocess computes the time difference between consecutive edges and the mean time difference. The main application uses the time difference to asynchronously set the rounds per minute. If a user changes the value of the motor input, the speed of the motor is adjusted accordingly. A summary is given with the following sequence diagram.

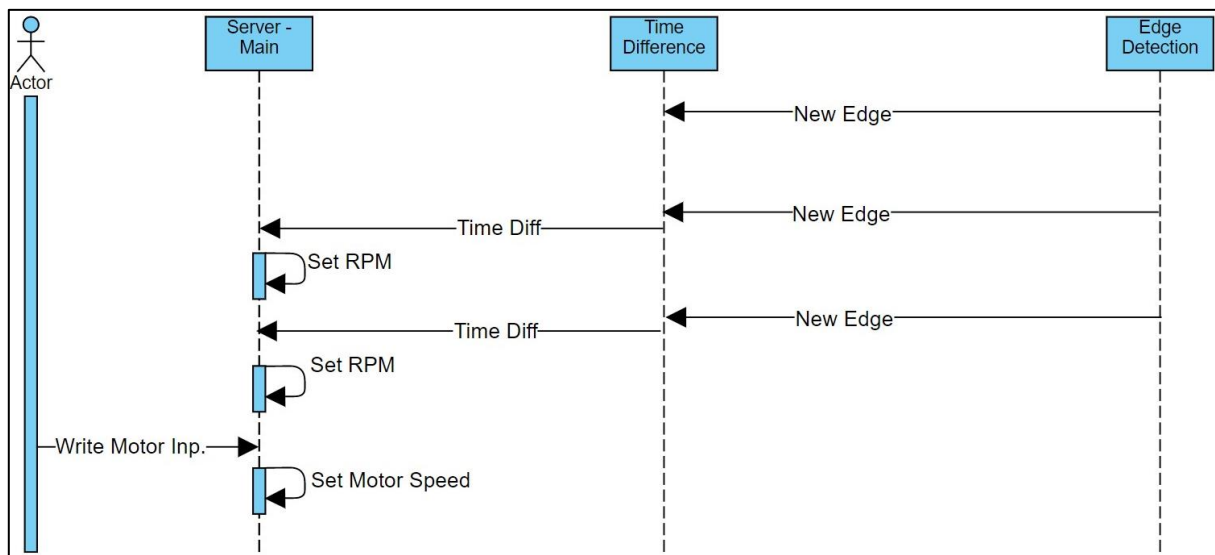


Figure 20: Remote Motor Control – Sequence Diagram of Python Server

³ For full code, see Appendix B

Below, some pictures of the setup are provided to enable better envisioning of the application.

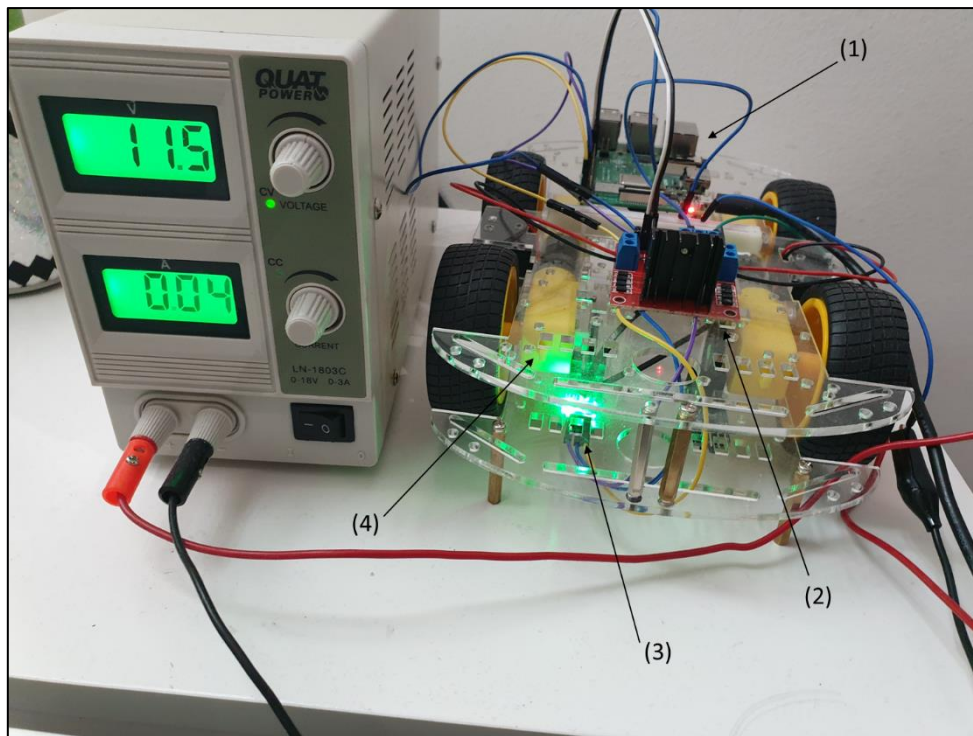


Figure 21: Remote Motor Control – Raspberry Pi and Peripherals

(1) Raspberry Pi, (2) Motor Driver, (3) Pulse Sensor, (4) DC Motor

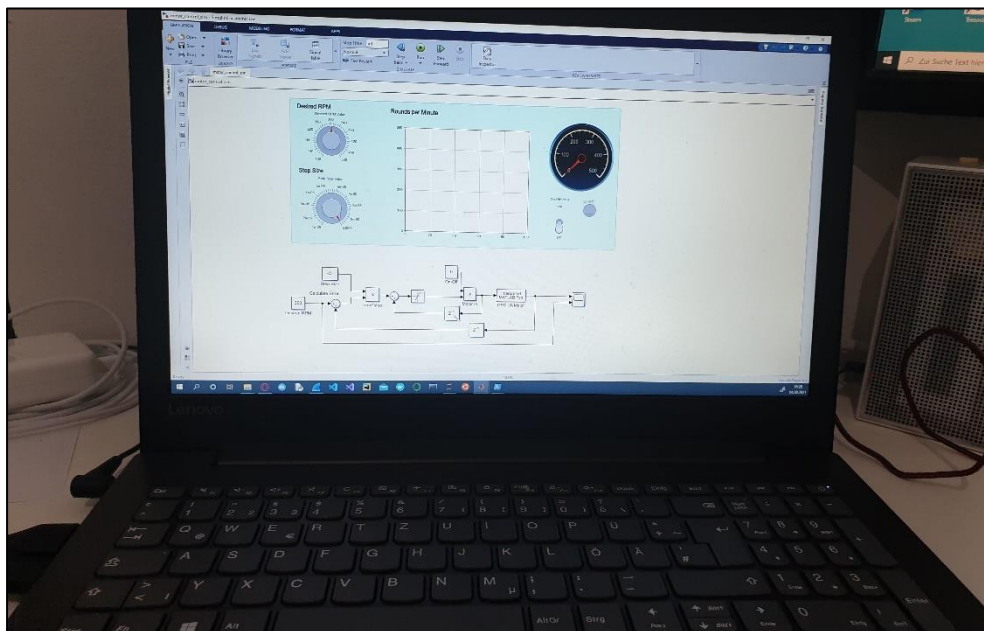


Figure 22: Remote Motor Control – Motor Controller on PC

6 Measurements

This chapter aims to describe an OPC UA Server Client application in terms of its timing properties. For this purpose, the findings regarding the execution time of methods and write operations as well as the reception of notification messages will be presented in the following.

6.1 Setup

Hardware List	
1x Raspberry Pi 3 Model B+	Raspberry Pi OS Lite (5/2021)
2x Ethernet Cable	CAT 5e
1x Laptop	8 GB RAM

Table 2: Measurement Setup – Hardware List

Since the data link can impact the measurements, two setups were chosen to test the application. One setup used Ethernet as data link and the other used WLAN. The time was always measured on the Client side. Another objective was to reduce the influence of the application code on the measurements, which was especially important for the invoked methods and the subscription handler. Both contained as few lines of code as possible. Code snippets will describe how the measurement of each functionality took place.

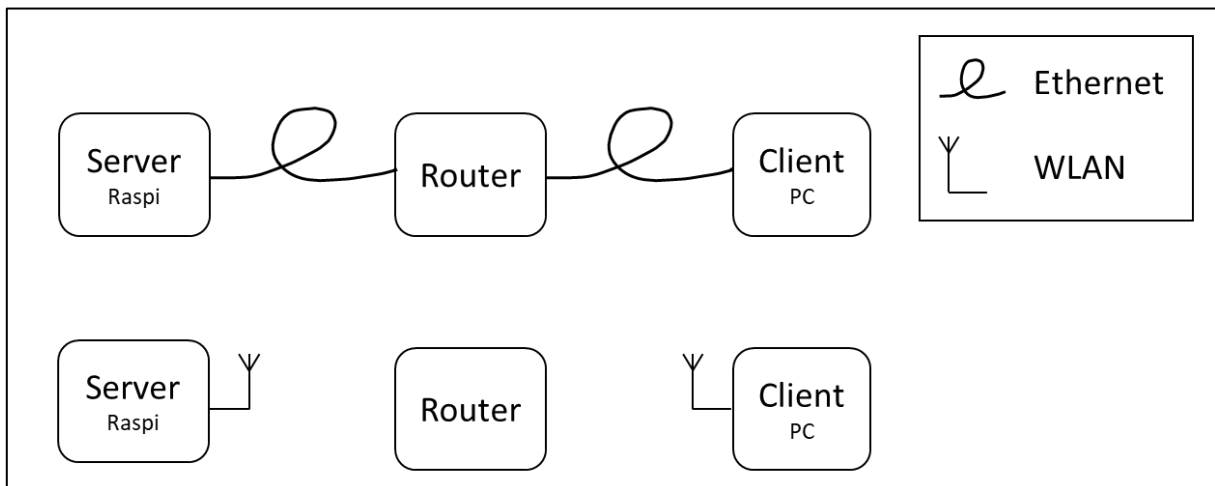


Figure 23: Measurement Setup – Block Diagram

6.1.1 Write & Method Invocation

Two sequence diagrams provide an overview of how the measurements took place as well as what kind of OPC UA messages were exchanged between Server and Client.

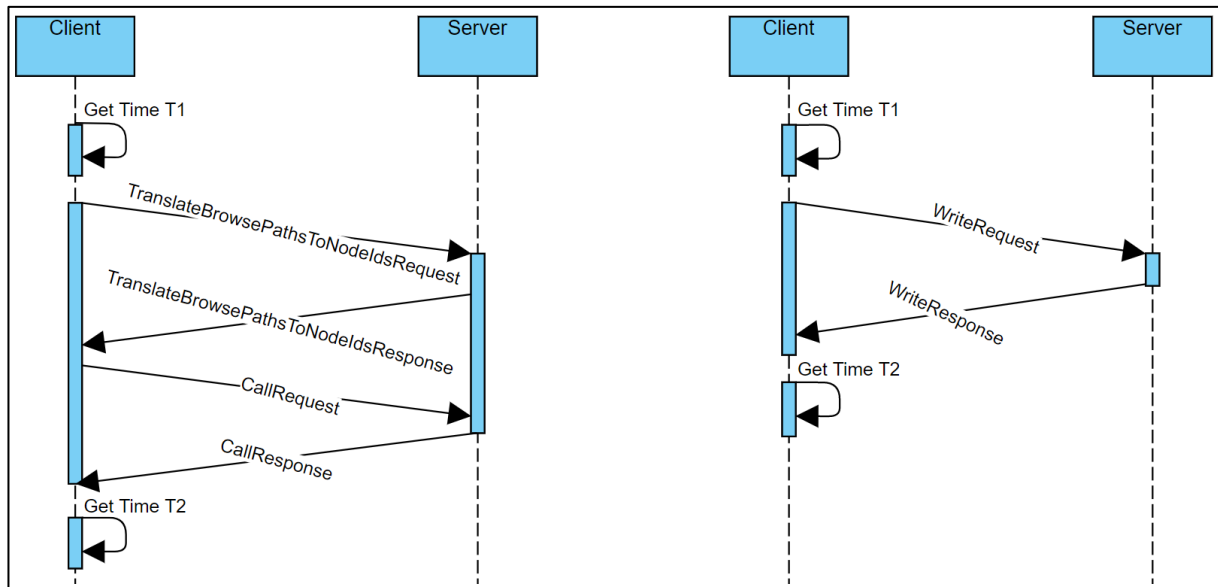


Figure 24: Measurement Setup – Sequence Diagram for Method Calls (left) & Write (right)

Server⁴

As invoked methods only return after they are processed, the functions under test will not contain any code apart from the *pass* statement.

```
@uamethod
async def start_motor(parent):
    pass

@uamethod
async def stop_motor(parent):
    pass
```

Listing 16: Measurement Setup – Methods for Invocations

⁴ For full code, see Appendix C-1

Client⁵

The Client Application stores the timestamp before and after the function call. With the help of `time.perf_counter_ns`, the measured time is system-wide and includes time elapsed during sleep. The function is called before and after the function under test; `cycles` sets the number of measurements.

```
async def time_method(var, idx, cycles, case):
    timing_vec = np.zeros(cycles)
    for i in range(cycles):
        t1 = time.perf_counter_ns()
        await var.call_method(f"{idx}:stop_motor") # DUT
        t2 = time.perf_counter_ns()
        timing_vec[i] = t2-t1
```

Listing 17: Measurement Setup – Timing of Method Calls

Write operations are measured similarly to method invocations. The only difference is that no application code apart from the Address Space is needed. The implementation of the Address Space does not deviate too much from the Client Server example described in chapter 4. Below, the function for measuring write operations is shown.

```
async def time_write(var, cycles, case):
    timing_vec = np.zeros(cycles)
    for i in range(cycles):
        random_value = round(rd.random(),2) # random float [0.00 - 1.00]
        t1 = time.perf_counter_ns()
        await var.write_value(random_value) # DUT
        t2 = time.perf_counter_ns()
        timing_vec[i] = t2-t1
```

Listing 18: Measurement Setup – Timing of Write Operations

The execution time is computed as the difference between the two timestamps in nano seconds.

⁵ Full code, see Appendix C-2

6.1.2 Notification Messages

Since notification messages are continuously published to the Client that subscribed to a monitored item, the time between consecutive messages is investigated. For this purpose, the Client subscribes to a data change event. When the measurement is started, the Server writes random values to the element that the Client subscribed to, which triggers a data change event. Depending on the defined period of the publishing interval set by the Client, notification messages will be sent from the Server to the Client. This triggers the *SubscriptionHandler* to execute the *datachange_notification* method. The time of arrival is stored inside this method. The Client starts and stops the measurement through the invocation of methods.

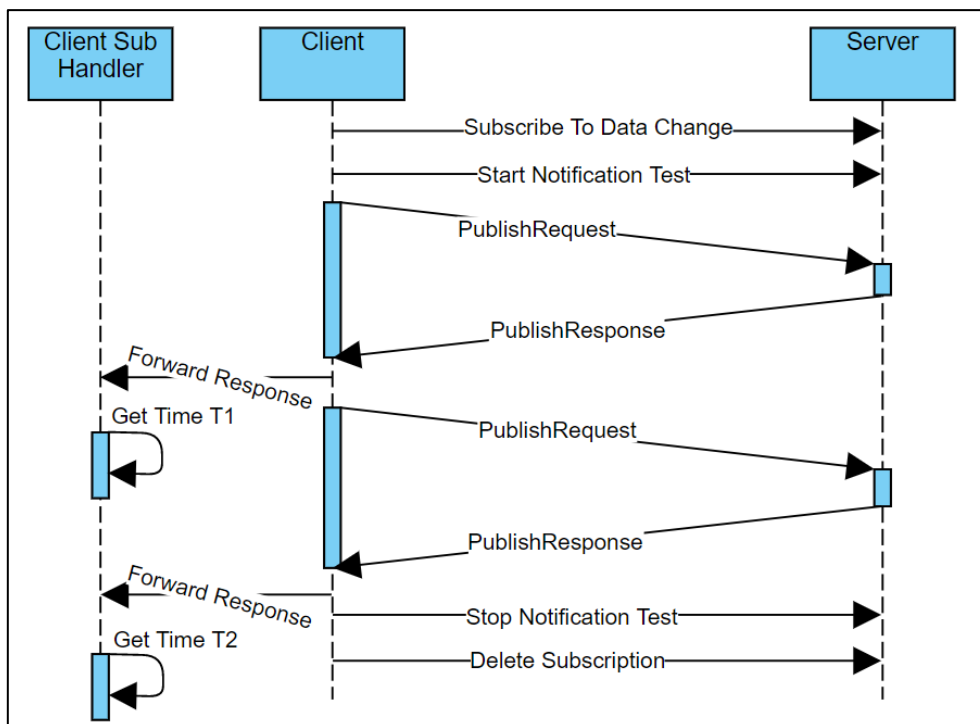


Figure 25: Measurement Setup – Sequence Diagram for Notification Messages

Server⁶

Source code to start and stop the measurement is shown below. One flag is used to stop and to start the test as well as variables that define the element to be changed and the period of the data change.

```
NOTIFICATION_TEST = False
notification_test_period = None
notification_test_variable = None
@uamethod
async def start_notification_test(parent, period, var_name='RPM'):
    global NOTIFICATION_TEST, notification_test_period, \
        notification_test_variable
    NOTIFICATION_TEST = True
    notification_test_period = period
    notification_test_variable = var_name

@uamethod
async def stop_notification_test(parent):
    global NOTIFICATION_TEST, notification_test_period
    NOTIFICATION_TEST = False
    notification_test_period = None
```

Listing 19: Measurement Setup – Methods for Notification Test

The test is then executed in a loop, which runs as long as the flag *NOTIFICATION_TEST* is set (see Listing 20).

```
var_dict = {'Input': dc_motor_inp, 'RPM': dc_motor_rpm}
async with server:
    while True:
        while NOTIFICATION_TEST:
            num = round(rd.random(), 3)
            await var_dict[notification_test_variable].write_value(num)
            await asyncio.sleep(notification_test_period)
        await asyncio.sleep(0.1)
```

Listing 20: Measurement Setup – Notification Test

⁶ For full code, see Appendix C-1

Client⁷

As previously mentioned, the *SubscriptionHandler* stores the time when a notification message is received (see Listing 21).

```
class SubscriptionHandler (SubHandler):
    def datachange_notification(self, node: Node, val, data):
        global timestamp_list
        timestamp_list.append(time.perf_counter_ns())
```

Listing 21: Measurement Setup – Timing of Notification Message

The function *time_subscription* (code snippet below) is put to sleep after starting the measurement. The duration of the sleep determines how many notification messages can be received before the measurement is stopped. Finally, the time difference of consecutive messages is computed.

```
async def time_subscription(client, obj, idx, var, var_name, case,
                           period=10, change_int=1, duration=100, queuesize=1):
    """
    period: desired period of notification messages [ms]
    change_int: interval of datachange of variable value [ms]
    duration: duration of test [sec]
    """
    global timestamp_list
    change_int_sec = change_int*10**-3
    # create subscription
    sub_handler = SubscriptionHandler()
    subscription = await client.create_subscription(period=period,
                                                    handler=sub_handler)

    # subscribe to data change; only current data --> queuesize=1
    await subscription.subscribe_data_change(nodes=var,
                                              queuesize=queuesize)

    # start notification test
    await obj.call_method(f"{idx}:start_notification_test", \
                          change_int_sec, var_name)
    await asyncio.sleep(duration)
    await obj.call_method(f"{idx}:stop_notification_test")
    subscription.delete()
    # compute timedifference between notification messages
    time_vec = []
    for i in range(len(timestamp_list)-1):
        time_vec.append(timestamp_list[i+1]-timestamp_list[i])
    time_vec = np.array(time_vec)[1::]
```

Listing 22: Measurement Setup – Execution of Notification Test

⁷ For full code, see Appendix C-2

6.2 Results

This section analyses the results of the measurements to find out how the systems behave in terms of their delay and jitter. For this purpose, the histogram and the cumulative distribution function (CDF) of every measurement are evaluated. An example of the configuration of the tests is given in the code below.

```
cycles=10**3
case='wlan'

await time_write(motor_rpm, cycles, case)

await time_method(motor_obj, idx, cycles, case)

await time_subscription(client, obj=motor_obj, idx=idx, var=motor_rpm, \
                        var_name='RPM', case=case, period=10,
                        change_int=10, duration=20)
```

Listing 23: Configuration of Measurements

Concerning the write operations and the method calls, both are executed a thousand times. This is enough to get a good estimate of the system's behaviour. For the notification messages, more arguments must be passed. The most important argument is the desired *period* which defines the publishing interval of the notification messages. Here, it is set to 10 milliseconds. The value of the monitored item is also changed every 10 milliseconds. Since *queuesize* is not passed as an argument, it is set to one, which means that only current values are sent to the subscriber. Finally, the *duration* of the test is set to 20 seconds.

For every histogram, occurrences of time values (delays) are depicted as well as the median, which describes the value that separates the lower 50% from the upper 50% of the data samples. Each CDF marks the 5% and the 95% quantile. The difference between both values provides an estimate of the jitter. The area within this interval is filled with colour.

Regarding the notification messages, the x-axis is modified to display the time interval of the notification messages relative to the desired period, which is set by the Client. This means that if it takes 12ms to receive a notification message, this value will be subtracted by 10ms to get the difference to the publishing interval. The shape of the histogram as well as the jitter of the CDF are not affected as this operation only results in a shift to the left.

Write Operations

Execution Time of 1000 Write Operations (Ethernet)

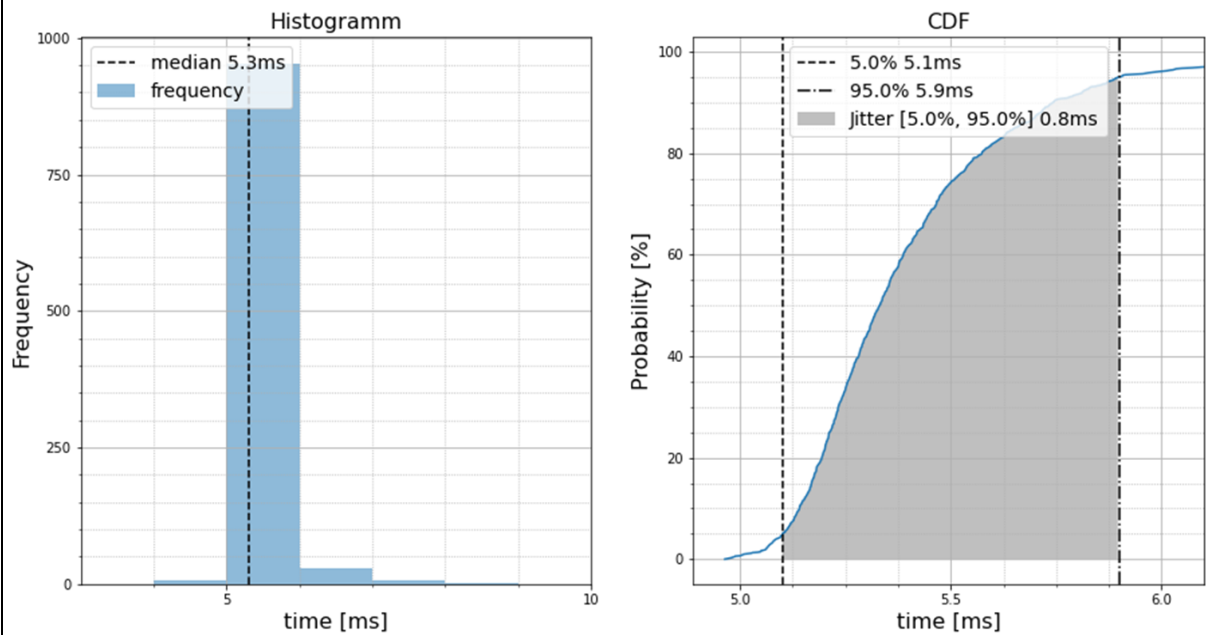


Figure 26: Write Operations with Ethernet

Execution Time of 1000 Write Operations (WLAN)

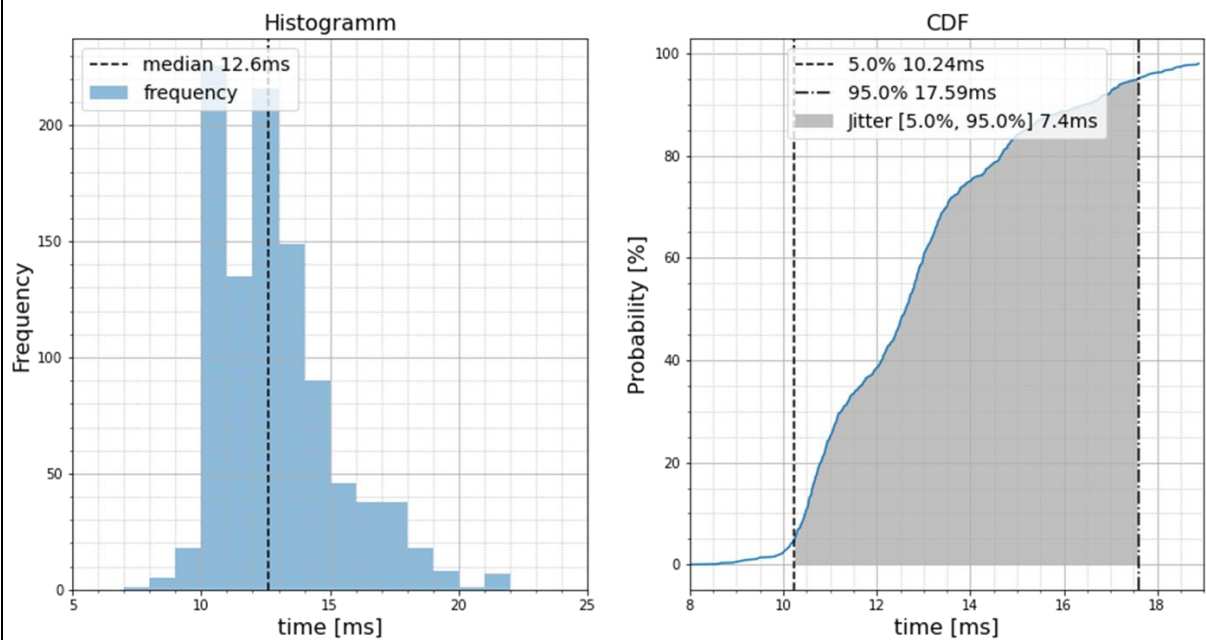


Figure 27: Write Operations with WLAN

Method Calls

Execution Time of 1000 Method Calls (Ethernet)

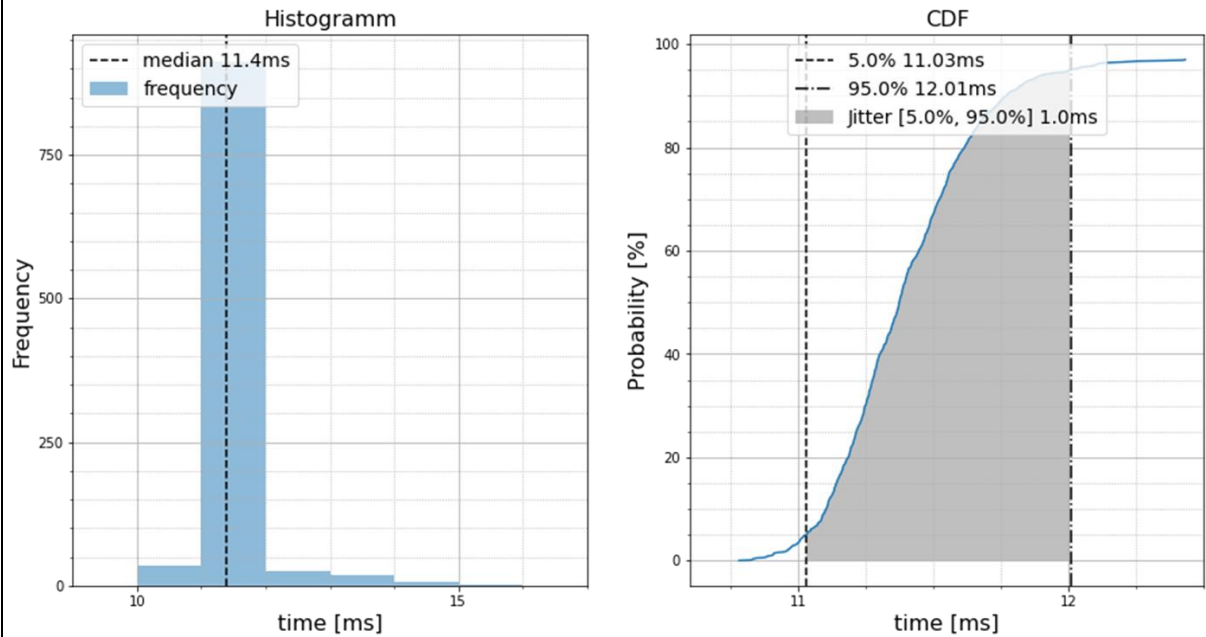


Figure 28: Method Calls with Ethernet

Execution Time of 1000 Method Calls (WLAN)

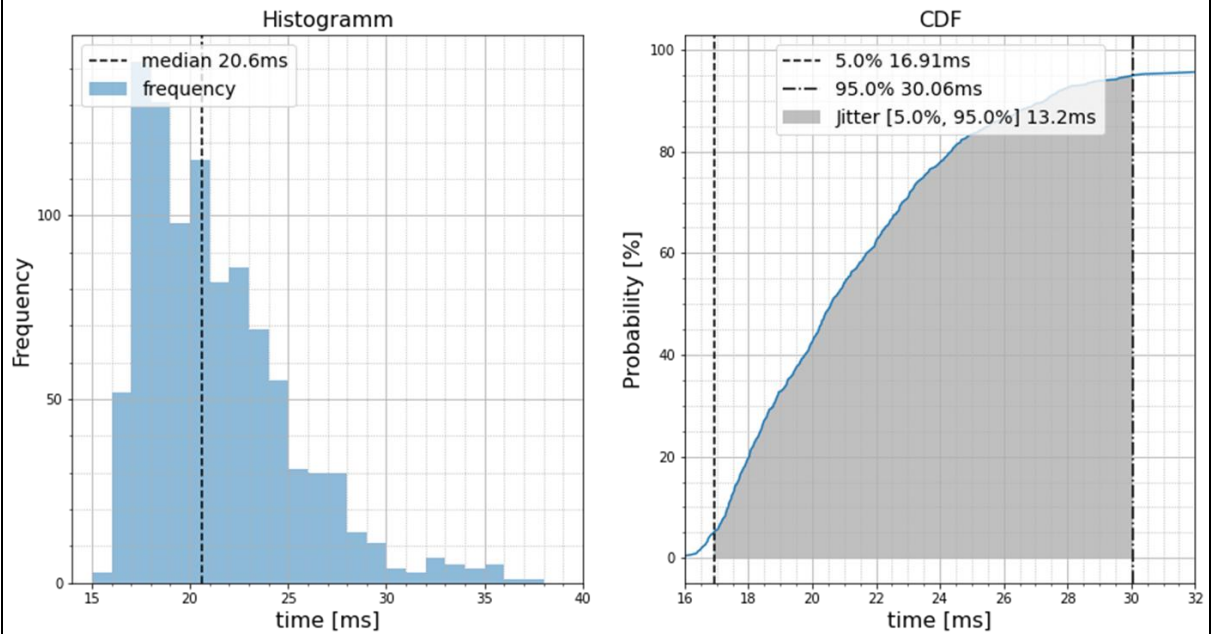


Figure 29: Method Calls with WLAN

Notification Messages

Time Between 1560 Notification Messages, Period=10.0ms (Ethernet)

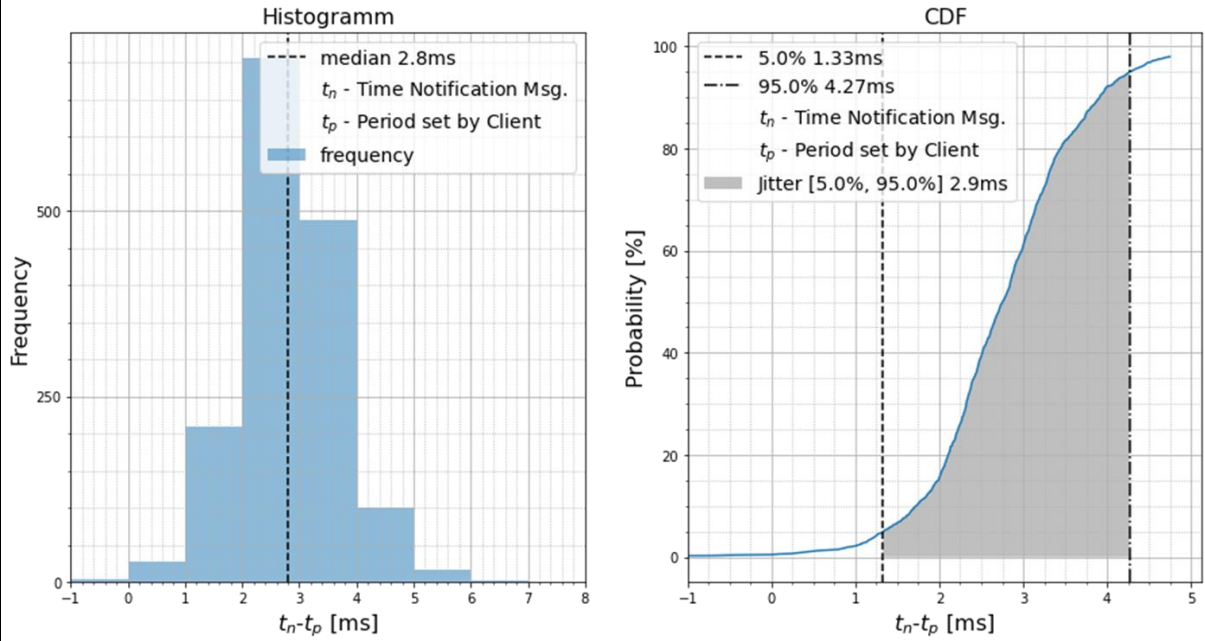


Figure 30: Notification Messages with Ethernet

Time Between 1474 Notification Messages - Period=10.0ms (WLAN)

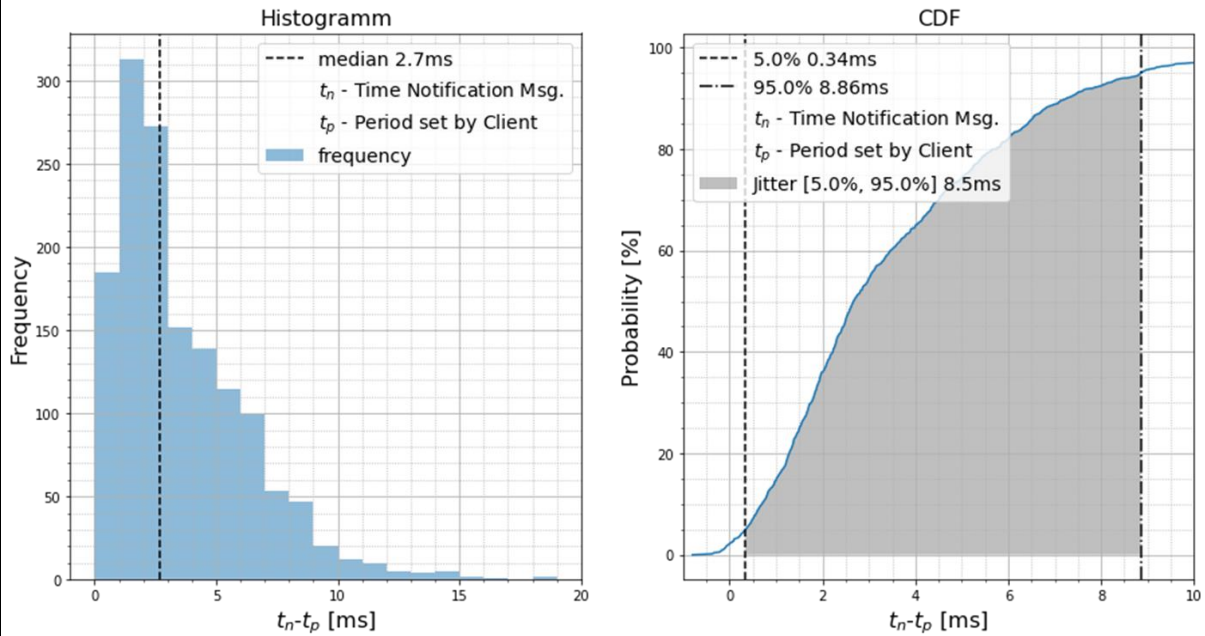


Figure 31: Notification Messages with WLAN

Summary of Results						
Service	Write Operation			Method Call		
	Delay ⁸	Jitter	Ratio	Delay ⁸	Jitter	Ratio
Ethernet	5.3ms	0.8ms	0.15	11.4ms	1ms	0.09
WLAN	12.6ms	7.4ms	0.59	20.6ms	13.2ms	0.64
Service	Notification Message			Ping		
	Delay ^{8,9}	Jitter	Ratio	Delay ⁸	Jitter	Ratio
Ethernet	12.8ms	2.9ms	0.23	1ms	0ms	0
WLAN	12.7ms	8.5ms	0.67	8ms	14ms	1.75

Table 3: Influence of Data Link on Delay and Jitter

The table above shows notable performance differences between the two data link variants. For write operations and method calls, the Ethernet application's delay and jitter perform better than their WLAN counterparts. This is not the case for Notification Messages. Here, only the jitter deviates. Eventually, the services can be compared to the performance of the *ping* command. The resolution for this command is 1ms so that values below 1ms are mapped to 1ms.

In addition, the jitter-delay-ratio is computed as follows:

$$ratio = \frac{jitter}{median\ delay} \quad \text{Equation 1: Jitter-Delay-Ratio}$$

This ratio provides information independent of the tested service. Regarding Ethernet, the jitter-delay-ratio ranges from 0.09 to 0.23 and for WLAN from 0.59 to 0.67 (see Figure 32).

⁸ median

⁹ Values represent the actual delay t_n of the notification message.

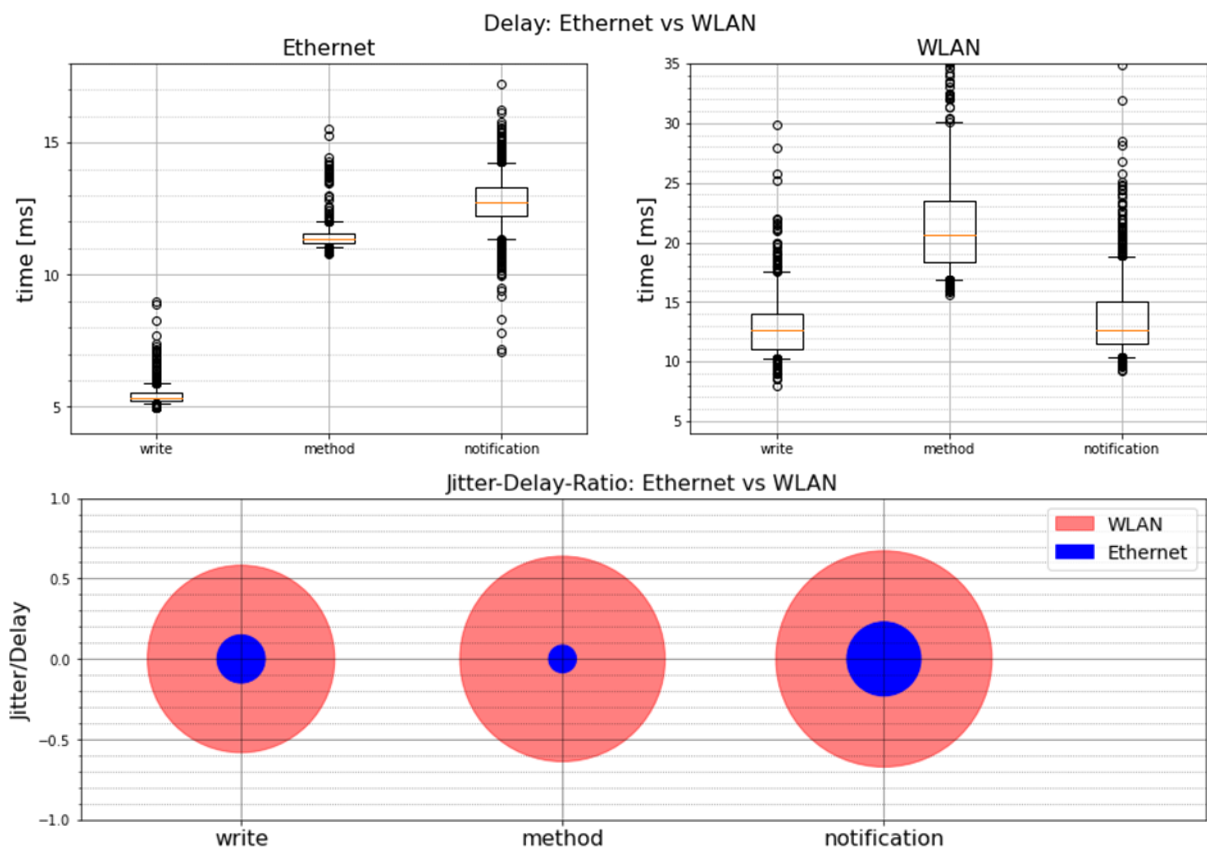


Figure 32: Performance: Ethernet vs WLAN

7 Conclusions

Summary

This thesis investigates the Python open-source OPC UA implementation *opcua-asyncio* from the Free OPC-UA Library. It gives an overview of essential functionalities and shows how services like writing to variables, invoking methods and publish/subscribe are translated into network messages. Further, it is demonstrated that open-source and commercial solutions can be successfully combined to implement a prototype of a distributed industrial application. For this purpose, a controller was designed in MATLAB/Simulink to remotely control a motor through an OPC UA Server implemented with *opcua-asyncio* on a Raspberry Pi.

Additionally, the delay and jitter of a Client-Server application was investigated to provide insight about limitations as *opcua-asyncio* is written in Python. The measurements display the delay and jitter for write operations, method invocations and notifications (publish and subscribe) for Ethernet and WLAN as the data link. From the results, it can be concluded that an OPC UA implementation in Python is suitable for applications that do not impose hard deadlines. However, the data link impacts the delay and jitter significantly. Here, Ethernet instead of WLAN should be used for better timing performances.

Outlook

For future investigations, it is essential to evaluate to what extent the Server on the Raspberry Pi can handle incoming requests until the service is denied. This is especially important, as this thesis only covers applications with one Client and one Server. In a real industrial environment, this will rarely be the case.

I. List of References

- [AIO] **Python Software Foundation:** Asyncio – Asynchronous I/O (Accessed: 4/2021), <https://docs.python.org/3/library/asyncio.html>
- [FBC] **Mühlbauer, N.; Kirdan, E.; Pahl, M; Waedt, K.** (2020): Feature-based Comparison of Open Source OPC-UA Implementations
- [GUI] **Free OPC-UA Library:** opcua-client-gui (Accessed: 4/2021), <https://github.com/FreeOpcUa/opcua-client-gui>
- [HIOT] **Veneri, G. & Capasso, A.** (2018): Hands-On Industrial Internet of Things Packt Publishing Ltd., Birmingham, UK
- [IOT] **Hüning, F.** (2019): Embedded Systems für IoT, Springer-Verlag GmbH Deutschland
- [MAT] **MathWorks, Inc.:** OPC Toolbox, User's Guide (Accessed: 6/2021), https://de.mathworks.com/help/pdf_doc/opc/index.html?s_tid=mwa_osa_a
- [MUA] **MathWorks, Inc.:** Unified Architecture (Accessed: 6/2021), <https://de.mathworks.com/help/opc/unified-architecture.html>
- [OPC1] **OPC Foundation** (2017, Release 1.04): OPC Unified Architecture Part 1: Overview and Concepts
- [OPC3] **OPC Foundation** (2017, Release 1.04): OPC Unified Architecture Part 3: Address Space Model
- [OPC4] **OPC Foundation** (2017, Release 1.04): OPC Unified Architecture Part 4: Services
- [PER] **MathWorks, Inc.:** Persistent (Accessed: 07/2021), https://de.mathworks.com/help/matlab/ref/persistent.html?s_tid=srchtitle
- [PUB] **Pfrommer, J.; Usländer T.** (2020): Open-Source Implementierung von OPC UA PubSub für echtzeitfähige Kommunikation mit Time-Sensitive Networking
- [UAL] **Free OPC-UA Library** (Accessed: 07/2021): <https://github.com/FreeOpcUa>
- [UAC] **Open62541** (Accessed: 07/2021): <https://open62541.org/doc/current/>

II. List of Figures

Figure 1: Hard and Soft Real-Time, Source: [IOT, Fig. 8.3]	2
Figure 2: OPC UA Object Model, Source: [OPC3, Fig. 2].....	5
Figure 3: OPC UA DC Motor Example	6
Figure 4: OPC UA Client Architecture, Source: [OPC1, Fig. 4]	10
Figure 5: OPC UA System Architecture, Source: [OPC1, Fig. 3]	11
Figure 6: Publish-Subscribe Models, Source: [OPC1, Fig. 8]	12
Figure 7: OPC UA – Client Server Example	13
Figure 8: Example Server – Motor Simulation (Flowchart)	20
Figure 9: OPC UA Example Server – Browsed Address Space	21
Figure 10: Value of RPM over Time	22
Figure 11: Wireshark – Secure Channel & Session	27
Figure 12: Wireshark – Request Objects of Address Space.....	28
Figure 13: Wireshark – Subscription & Monitored Item	28
Figure 14: Wireshark – Write Operation.....	29
Figure 15: Wireshark – Method Invocation	29
Figure 16: Wireshark – Publishing Response & Publishing Request	30
Figure 17: Remote Motor Control – Architecture.....	31
Figure 18: Remote Motor Control – Dashboard	33
Figure 19: Remote Motor Control – Simulink Model.....	34
Figure 20: Remote Motor Control – Sequence Diagram of Python Server	37
Figure 21: Remote Motor Control – Raspberry Pi and Peripherals	38
Figure 22: Remote Motor Control – Motor Controller on PC	38
Figure 23: Measurement Setup – Block Diagram	39
Figure 24: Measurement Setup – Sequence Diagram for Method Calls (left) & Write (right).....	40
Figure 25: Measurement Setup – Sequence Diagram for Notification Messages.....	42
Figure 26: Write Operations with Ethernet	46
Figure 27: Write Operations with WLAN.....	46
Figure 28: Method Calls with Ethernet	47
Figure 29: Method Calls with WLAN.....	47

Figure 30: Notification Messages with Ethernet.....	48
Figure 31: Notification Messages with WLAN	48
Figure 32: Performance: Ethernet vs WLAN.....	50

III. List of Tables

Table 1: Bill of Material for Remote Motor Control	31
Table 2: Measurement Setup – Hardware List.....	39
Table 3: Influence of Data Link on Delay and Jitter.....	49

IV. List of Listings

Listing 1: Terminal Output for Concurrent & Sequential Application	14
Listing 2: Sequential and Concurrent Application	15
Listing 3: Example Server – Initialization	16
Listing 4: Example Server – Namespace	16
Listing 5: Example Server – Motor Base Object	17
Listing 6: Example Server – Instantiation of Motor Object.....	18
Listing 7: Example Server – Add Methods to Address Space	18
Listing 8: Example Server – Start	19
Listing 9: Example Server – Start Stop Methods	19
Listing 10: Example Client – Subscription Handler	23
Listing 11: Example Client – Get Address Space Objects	24
Listing 12: Example Client – Subscribe to Variable.....	24
Listing 13: Example Client – Server Motor Control.....	25
Listing 14: Example Server – Terminal Output	26
Listing 15: Remote Motor Control – MATLAB OPC UA Client.....	35
Listing 16: Measurement Setup – Methods for Invocations	40
Listing 17: Measurement Setup – Timing of Method Calls.....	41
Listing 18: Measurement Setup – Timing of Write Operations	41
Listing 19: Measurement Setup – Methods for Notification Test.....	43
Listing 20: Measurement Setup – Notification Test	43
Listing 21: Measurement Setup – Timing of Notification Message	44
Listing 22: Measurement Setup – Execution of Notification Test.....	44
Listing 23: Configuration of Measurements	45

Appendix A

1. Server Example – server_example.py

```
#!/usr/bin/env python3
"""
Example of OPC UA Server populated with custom Object DC Motor

Object: DC Motor
    Variables: inp_value, rounds_per_min
    Methods: start_motor(), stop_motor()
"""
import sys
sys.path.insert(0, '..') # import parent folder
import asyncio # documentation -->
https://docs.python.org/3/library/asyncio-task.html
from asyncua import ua, Server, uamethod
import argparse

parser = argparse.ArgumentParser(description='Start an OPC UA Server, that \
                                         controls a DC Motor')
parser.add_argument('--host', default='0.0.0.0', type=str,
                    help='Define the host IP of the Server.', dest='host')

args = parser.parse_args()

# add start, stop methods for motor
global MOTOR_STARTED, dc_motor_inp, dc_motor_rpm
MOTOR_STARTED = False
# @uamethod: Method decorator to automatically
# convert arguments and output to and from variant
@uamethod
async def start_motor(parent):
    global MOTOR_STARTED
    if not MOTOR_STARTED:
        MOTOR_STARTED = True
        print("Motor started.")

@uamethod
async def stop_motor(parent):
    global MOTOR_STARTED
    if MOTOR_STARTED:
        MOTOR_STARTED = False
        print("Motor stopped")

# simulate motor
async def motor_simulation():
    """
    simulate a dc motor
    """
    global MOTOR_STARTED, dc_motor_inp, dc_motor_rpm
    while True:
        while MOTOR_STARTED:
```

```

        inp = await dc_motor_inp.get_value() # motor input
        print("Input:", inp)
        rpm_fin = inp*10 # final rpm value
        rpm_now = await dc_motor_rpm.get_value() # current rpm value
        print("RPM:", rpm_now)
        if rpm_fin>rpm_now: # increase rpm
            await dc_motor_rpm.write_value(rpm_now+inp/10)
        elif rpm_fin<rpm_now: # decrease rpm
            await dc_motor_rpm.write_value(rpm_now - inp / 10)
        else: # do nothing
            pass
        await asyncio.sleep(0.1) # sleep 100ms
    # motor stopped
    await dc_motor_rpm.write_value(0)
    print("RPM:", await dc_motor_rpm.get_value())
    await asyncio.sleep(0.1) # sleep 100ms

async def main(host='localhost'):
    # init server, set endpoint
    server = Server()
    await server.init()
    server.set_endpoint(f"opc.tcp://{host}:4840/server_example/")

    # setup of namespace, not needed
    uri = "example-uri.edu"
    idx = await server.register_namespace(uri)

    # Create new object type
    base_obj_motor = await server.nodes.base_object_type.add_object_type(
        nodeid=idx, bname="BaseMotor")
    base_var_rpm = await base_obj_motor.add_variable(nodeid=idx,
        bname="RPM", val=0.0)
    base_var_inp = await base_obj_motor.add_variable(nodeid=idx,
        bname="Input", val=0.0)

    # ensure that variable will be instantiated together with object
    await base_var_rpm.set_modelling_rule(True)
    await base_var_inp.set_modelling_rule(True)

    # Address Space
    dc_motor = await server.nodes.objects.add_object(
        nodeid=idx, bname="Motor",
        objecttype=base_obj_motor)

    global dc_motor_inp, dc_motor_rpm
    dc_motor_inp = await dc_motor.get_child(f'{idx}:Input')
    dc_motor_rpm = await dc_motor.get_child(f'{idx}:RPM')
    await dc_motor_inp.set_writable()
    await dc_motor_rpm.set_writable()
    # add methods
    await dc_motor.add_method(idx, # nodeid
        "start_motor", # browse name
        start_motor, # method to be called
        [], # list of input arguments
        [], # list output arguments
    )
    await dc_motor.add_method(idx, "stop_motor", stop_motor, [], [])

    # create task for simulation of dc motor
    task_motor_sim = asyncio.create_task(motor_simulation())

```

```
# start
async with server:
    while True:
        await task_motor_sim # runs concurrently with main()
        await asyncio.sleep(1)

if __name__ == "__main__":
    host = args.host
    asyncio.run(main(host))
```

2. Client Example – client_example.py

```
#!/usr/bin/env python3
"""
Example OPC UA Client
that Subscribes to DC Motor Object Attributes (RPM, Input)
"""

import sys
sys.path.insert(0, "..")
import asyncio
import logging
from asyncua import Client, Node, ua
from asyncua.common.subscription import SubHandler

logging.basicConfig(level=logging.INFO) # logging.INFO as default
_logger = logging.getLogger() #'asyncua')

import argparse

parser = argparse.ArgumentParser(description='Start an OPC UA Server, that \
                                         controls a DC Motor')
parser.add_argument('--host', default='0.0.0.0', type=str,
                    help='Define the host IP of the Server.', dest='host')

args = parser.parse_args()

global STOP_FLAG
STOP_FLAG = False

class SubscriptionHandler (SubHandler):
    """
    Handle the data that is received for the subscription.
    """

    def datachange_notification(self, node: Node, val, data):
        """
        Callback for asyncua Subscription.
        This method will be called when the Client receives a data change
        message from the Server.
        """
        global STOP_FLAG
        _logger.info(f'datachange_notification node: {node} value: {val}')
        if val > 1:
            STOP_FLAG = True

async def main(host):
    """
    Open communication to Server
    """
    server_endpoint = f"opc.tcp://{host}:4840/server_example/"
    client = Client(url=server_endpoint)
    async with client:
        idx = await client.get_namespace_index(uri="example-uri.edu")
        # get motor object, here path from root folder
```



```

motor_obj = await client.nodes.root.get_child(["0:Objects",
                                                f"{idx}:Motor"])

# get motor input variable, here path starting from Object folder
motor_inp = await client.nodes.objects.get_child(
    path=[f"{idx}:Motor", f"{idx}:Input"])

# get motor rpm variable
motor_rpm = await client.nodes.objects.get_child(
    path=[f"{idx}:Motor", f"{idx}:RPM"])

# subscription
sub_handler = SubscriptionHandler()
subscription = await client.create_subscription(period=50,
                                                handler=sub_handler)

# subscribe to data change, only current data queuesize=1
await subscription.subscribe_data_change(nodes=motor_rpm,
                                         queuesize=1)

global STOP_FLAG # flag to stop motor
# write to input value of motor
# then, start motor
# finally, stop motor if stop flag is set by sub_handler
# and delete subscription and exit context manager
await motor_inp.write_value(1)
_logger.info('wrote 1 to input value')
await motor_obj.call_method(f'{idx}:start_motor')
_logger.info('motor started')
while True:
    await asyncio.sleep(0.005)
    if STOP_FLAG:
        await motor_obj.call_method(f"{idx}:stop_motor")
        _logger.info('motor stopped')
        STOP_FLAG = False
        subscription.delete()
        await asyncio.sleep(1)
        break

if __name__ == "__main__":
    host = args.host
    asyncio.run(main(host))

```

Appendix B

Server for Motor Control – server_motor_rpm.py

```
#!/usr/bin/env python3
"""
Server which
    - controls DC Motor
    - computes RPM

Object: DC Motor
    Variables: inp_value, rounds_per_min
    Methods: start_motor(), stop_motor()
"""
import sys
sys.path.insert(0, '..') # import parent folder

import asyncio # documentation -->
https://docs.python.org/3/library/asyncio-task.html
from asyncua import ua, Server, uamethod
from asyncua.common.subscription import SubHandler
from asyncua import Node, ua

from gpiozero import Motor, Button

import logging, argparse

from multiprocessing import Process, Value

import time
import numpy as np

##### ARGUMENTS PARSER #####
parser = argparse.ArgumentParser(description='Start an OPC UA Server, that \
                                         controls a DC Motor')
parser.add_argument('--host', default='0.0.0.0', type=str,
                    help='Define the host IP of the Server.', dest='host')
parser.add_argument('--port', '-p', default='4840', type=str,
                    help='Define the port of the Server.', dest='port')
parser.add_argument('--debug', '-d', action='store_true',
                    help='Enable debugging mode.', dest='debug')
args = parser.parse_args()
#####

puls = Button(14)

logging.basicConfig(level=logging.DEBUG) # logging.INFO as default
_logger = logging.getLogger(__name__)

##### Callback functions#####
global MOTOR_STARTED, NEW_MOTOR_INP
global STOP_FLAG, START_FLAG
```

```

class SubscriptionHandler (SubHandler):
    """
    Handle the data that is received for the subscription.
    """

    def datachange_notification(self, node: Node, val, data):
        global STOP_FLAG, START_FLAG, NEW_MOTOR_INP, motor_speed_is
        if DEBUG: _logger.debug(f'datachange_notification node: {node} \
                                value: {val}')

        motor_speed_is = val
        if val > 0:
            START_FLAG=True
        else:
            STOP_FLAG=True
            NEW_MOTOR_INP=True

# Define Flags as multiprocessing.Value so memory is shared
# Values are thread safe
global EDGE_DETECTED, OLD_EDGE, NEW_EDGE
EDGE_DETECTED = Value('i', False)
OLD_EDGE, NEW_EDGE = Value('i', False), Value('i', False)

def callback_high_edge():
    global EDGE_DETECTED, OLD_EDGE, NEW_EDGE
    OLD_EDGE.value = NEW_EDGE.value
    NEW_EDGE.value = time.perf_counter_ns()
    EDGE_DETECTED.value = True

#***** Server Methods *****#
START_FLAG, STOP_FLAG, NEW_MOTOR_INP = False, False, False # inititate
flags
# add start, stop methods for motor
# @uamethod: Method decorator to automatically
# convert arguments and output to and from variant
@uamethod
async def start_motor(parent):
    global START_FLAG
    if not START_FLAG:
        START_FLAG = True
        if DEBUG: _logger.debug("start_motor Method called")

@uamethod
async def stop_motor(parent):
    global STOP_FLAG
    if not STOP_FLAG:
        STOP_FLAG = True
        if DEBUG: _logger.debug("stop_motor Method called")

#***** Other functions *****#
global dc_motor_inp, dc_motor_rpm, motor_speed_is

global motor
motor = Motor(26, 20)

```

```

async def set_speed():
    global START_FLAG, STOP_FLAG, NEW_MOTOR_INP, motor_speed_is
    if START_FLAG:
        if NEW_MOTOR_INP:
            if motor_speed_is >=0 and motor_speed_is <=1:
                motor.forward(motor_speed_is)
            NEW_MOTOR_INP = False
            START_FLAG = False
            if DEBUG: _logger.debug(f'motor set to {motor_speed_is}')
        if STOP_FLAG:
            motor.forward(0)
            NEW_MOTOR_INP = True
            STOP_FLAG = False
            if DEBUG: _logger.debug(f'motor set to 0')

global mean_diff
mean_diff = Value('i', 0)
def calc_time_diff():
    """
        Calculate Time Difference between pulses
    """
    counter = 0 # count cycles without new edge detected
    n_pulses = 3 # define how many pulses are stored
    diff = 0 # time difference between two edges
    diff_vec = np.zeros(n_pulses) # stores last time differences of pulses
    while True:
        global EDGE_DETECTED, OLD_EDGE, \
            NEW_EDGE, mean_diff
        # set counter to 0 if edge is detected
        # and start computing
        if EDGE_DETECTED.value:
            EDGE_DETECTED.value = False
            counter = 0
            # check old and new edge, no computation at first edge
            if NEW_EDGE.value and OLD_EDGE.value:
                # update mean difference between pulses
                diff = NEW_EDGE.value-OLD_EDGE.value
                if diff >0: # ignore random wrong values
                    # shift to right to update new values
                    diff_vec[1:] = diff_vec[:-1:1]
                    diff_vec[0] = diff # latest value
            # Count cycles without edge
        else:
            counter +=1
            if counter>50:
                # detect when motor has stopped
                diff_vec = np.zeros(n_pulses)
                mean_diff.value = int(np.mean(diff_vec))
                time.sleep(0.001)

async def set_rpm():
    global mean_diff, dc_motor_rpm
    # print(f'async\tmean_diff {mean_diff.value}')
    if mean_diff.value==0:
        rpm=0
    else:
        rpm = 60/((mean_diff.value)*20*(10**(-9)))
    await dc_motor_rpm.write_value(rpm)
    if DEBUG: _logger.debug(f'rpm\t{rpm}')

```

```

async def main(host, port):
    # init server, set endpoint
    server = Server()
    await server.init()
    server.set_endpoint(f"opc.tcp://{host}:{port}")

    # setup of namespace, not needed
    uri = "example-uri.edu"
    idx = await server.register_namespace(uri)

    # Create new object type
    base_obj_motor = await server.nodes.base_object_type.add_object_type(
        nodeid=idx, bname="BaseMotor")
    base_var_rpm = await base_obj_motor.add_variable(nodeid=idx,
        bname="RPM", val=0.0)
    base_var_inp = await base_obj_motor.add_variable(nodeid=idx,
        bname="Input", val=0.0)

    # ensure that variable will be instantiated together with object
    await base_var_rpm.set_modelling_rule(True)
    await base_var_inp.set_modelling_rule(True)

    # Address Space
    dc_motor = await server.nodes.objects.add_object(nodeid=idx,
        bname="Motor", objecttype=base_obj_motor)

    global dc_motor_inp, dc_motor_rpm
    dc_motor_inp = await dc_motor.get_child(f'{idx}:Input')
    dc_motor_rpm = await dc_motor.get_child(f'{idx}:RPM')
    await dc_motor_inp.set_writable()
    await dc_motor_rpm.set_writable()

    # add methods
    await dc_motor.add_method(idx, "start_motor", start_motor, [], [])
    await dc_motor.add_method(idx, "stop_motor", stop_motor, [], [])

    # subscription
    sub_handler = SubscriptionHandler()
    subscription = await server.create_subscription(period=10,
        handler=sub_handler)

    # subscribe to data change, only current data queuesize=1
    await subscription.subscribe_data_change(nodes=dc_motor_inp,
        queuesize=1)

    # start
    async with server:
        while True:
            await asyncio.gather(set_speed(), set_rpm())
            await asyncio.sleep(.10)

if __name__ == "__main__":
    global DEBUG
    # callback for detecting rising edges
    puls.when_pressed = callback_high_edge # rising edge
    # thread for computing time difference of rising edges
    p = Process(target=calc_time_diff)
    p.start()

    # command line arguments
    DEBUG = args.debug
    host = args.host

```

```
port = args.port
# start server
try:
    asyncio.run(main(host, port))
except Exception as e:
    print(e)
    p.kill()
    p.join()
```

Appendix C

1. Server for Measurements – server_timeit.py

```
import asyncio
from asyncua import ua, Server, uamethod
from asyncua.common.subscription import SubHandler
from asyncua import Node, ua

import logging, sys
import random as rd

logging.basicConfig(level=logging.DEBUG) # logging.INFO as default
_logger = logging.getLogger(__name__)

import argparse
##### ARGUMENTS PARSER #####
parser = argparse.ArgumentParser(description='Start an OPC UA Server, that \
                                     controls a DC Motor')
parser.add_argument('--host', default='0.0.0.0', type=str,
                    help='Define the host IP of the Server.', dest='host')
args = parser.parse_args()
#####

##### Server Methods #####
# add start, stop methods for motor
# @uamethod: Method decorator to automatically
# convert arguments and output to and from variant
@uamethod
async def start_motor(parent):
    pass

@uamethod
async def stop_motor(parent):
    pass

NOTIFICATION_TEST = False
notification_test_period = None
notification_test_variable = None

@uamethod
async def start_notification_test(parent, period, var_name='RPM'):
    """
        var_name: 'RPM' or 'Input'
    """
    global NOTIFICATION_TEST, notification_test_period, \
        notification_test_variable
    NOTIFICATION_TEST = True
    notification_test_period = period
    notification_test_variable = var_name
```

```

@uamethod
async def stop_notification_test(parent):
    global NOTIFICATION_TEST, notification_test_period
    NOTIFICATION_TEST = False
    notification_test_period = None

async def main(host='0.0.0.0'):
    # init server, set endpoint
    server = Server()
    await server.init()
    server.set_endpoint(f"opc.tcp://{host}:4840")

    # setup of namespace, not needed
    uri = "example-uri.edu"
    idx = await server.register_namespace(uri)

    # Create new object type
    base_obj_motor = await server.nodes.base_object_type.add_object_type(
        nodeid=idx, bname="BaseMotor")
    base_var_rpm = await base_obj_motor.add_variable(nodeid=idx,
        bname="RPM", val=0.0)
    base_var_inp = await base_obj_motor.add_variable(nodeid=idx,
        bname="Input", val=0.0)

    # ensure that variable will be instantiated together with object
    await base_var_rpm.set_modelling_rule(True)
    await base_var_inp.set_modelling_rule(True)

    # Address Space
    dc_motor = await server.nodes.objects.add_object(nodeid=idx,
        bname="Motor", objecttype=base_obj_motor)

    global dc_motor_inp, dc_motor_rpm
    dc_motor_inp = await dc_motor.get_child(f'{idx}:Input')
    dc_motor_rpm = await dc_motor.get_child(f'{idx}:RPM')
    await dc_motor_inp.set_writable()
    await dc_motor_rpm.set_writable()

    # add methods
    await dc_motor.add_method(idx, "start_motor", start_motor, [], [])
    await dc_motor.add_method(idx, "stop_motor", stop_motor, [], [])

    # Methods for notification test
    inarg_period = ua.Argument()
    inarg_period.Name = 'period'
    inarg_period.ValueRank = -1
    inarg_period.ArrayDimensions = []
    inarg_period.DataType = ua.NodeId(ua.ObjectIds.Float)
    inarg_period.Description = ua.LocalizedText("period of datachange in \
        sec.")

    inarg_var_name = ua.Argument()
    inarg_var_name.Name = 'variable_name'
    inarg_var_name.ValueRank = -1
    inarg_var_name.ArrayDimensions = []
    inarg_var_name.DataType = ua.NodeId(ua.ObjectIds.String)
    inarg_var_name.Description = ua.LocalizedText("name of variable to be \
        changed")

    await dc_motor.add_method(idx, "start_notification_test",

```



```

        start_notification_test,
        inarg_period, inarg_var_name], [])
await dc_motor.add_method(idx, "stop_notification_test",
        stop_notification_test, [], [])
# dictionary for notification test
var_dict = {'Input': dc_motor_inp, 'RPM': dc_motor_rpm}

# start
async with server:
    while True:
        while NOTIFICATION_TEST:
            num = round(rd.random(), 3)
            await var_dict[notification_test_variable].write_value(num)
            await asyncio.sleep(notification_test_period)
            await asyncio.sleep(0.1)

if __name__ == '__main__':
    host = args.host
    asyncio.run(main(host))

```

2. Client for Measurements – client_timeit.py

```
'''
Client to time interactions with server_timeit.py
'''

import sys, os
sys.path.insert(0, "..")

import asyncio
from asyncua import Client, Node, ua
from asyncua.common import subscription
from asyncua.common.subscription import SubHandler

import time
import numpy as np
import pandas as pd
import logging
import random as rd

# create directory for test data
data_path = "./timing/"
if not os.path.exists(data_path):
    os.mkdir(data_path)

logging.basicConfig(level=logging.INFO) # logging.INFO as default
_logger = logging.getLogger(__name__) #'asyncua')

global timestamp_list
timestamp_list = []
class SubscriptionHandler (SubHandler):
    """
    Handle the data that received for the subscription.
    """

    def datachange_notification(self, node: Node, val, data):
        """
        Callback for asyncua Subscription.
        This method will be called when the Client received a data change
        message from the Server.
        """
        global timestamp_list
        timestamp_list.append(time.perf_counter_ns())

async def time_subscription(client, obj, idx, var, var_name, case,
period=10, change_int=1, duration=100, queuesize=1):
    """
    period: desired period of notification messages [ms]
    change_int: interval of datachange of variable value [ms]
    duration: duration of test [sec]
    """
    global timestamp_list
    change_int_sec = change_int*10**-3
    # create subscription
```

```

sub_handler = SubscriptionHandler()
subscription = await client.create_subscription(period=period,
                                                handler=sub_handler)

# subscribe to data change; only current data --> queuesize=1
await subscription.subscribe_data_change(nodes=var,
                                         queuesize=queuesize)

# start notification test
await obj.call_method(f"{idx}:start_notification_test", \
                     change_int_sec, var_name)
await asyncio.sleep(duration)
await obj.call_method(f"{idx}:stop_notification_test")
subscription.delete()
# compute timedifference between notification messages
time_vec = []
for i in range(len(timestamp_list)-1):
    time_vec.append(timestamp_list[i+1]-timestamp_list[i])
time_vec = np.array(time_vec)[1::]
# save dataframe to csv
df = pd.DataFrame({'datachange_notifications': time_vec, \
                  'period': np.ones(time_vec.shape)*period, \
                  'queuesize': np.ones(time_vec.shape)*queuesize, \
                  'change_int': np.ones(time_vec.shape)*change_int})

df.to_csv(data_path+f'{case}_subscription_duration_{duration}_period_{period}_changeInt_{change_int}_queuesize_{queuesize}.csv')


async def time_method(var, idx, cycles, case, delay=0):
    """
        time the daly of method calls and return timing_vec of len(cycles)
    """
    timing_vec = np.zeros(cycles)
    for i in range(cycles):
        t1 = time.perf_counter_ns()
        await var.call_method(f"{idx}:stop_motor")
        t2 = time.perf_counter_ns()
        timing_vec[i] = t2-t1
        if delay >0:
            await asyncio.sleep(delay)
    df = pd.DataFrame({'method_call': timing_vec,
                      'delay': np.ones(timing_vec.shape)*delay})
    df.to_csv(data_path+f'{case}_method_cycles_{cycles}_delay_{delay}.csv')


async def time_write(var, cycles, case, delay=0):
    """
        time the delay of write operations and save as csv-file
    """
    timing_vec = np.zeros(cycles)
    for i in range(cycles):
        random_value = round(rd.random(),2) # random float [0.00 - 1.00]
        t1 = time.perf_counter_ns()
        await var.write_value(random_value)
        t2 = time.perf_counter_ns()
        timing_vec[i] = t2-t1
        if delay >0:
            await asyncio.sleep(delay)

```

```

df = pd.DataFrame({'write_value': timing_vec,
                  'delay': np.ones(timing_vec.shape)*delay})
df.to_csv(
    data_path+f'{case}_write_value_cycles_{cycles}_delay_{delay}.csv')

async def main(host='0.0.0.0'):
    server_endpoint = f"opc.tcp://{host}:4840"
    client = Client(url=server_endpoint)
    not_connected = True
    while not_connected:
        try:
            async with client:
                not_connected = False
                idx = await client.get_namespace_index(
                    uri="example-uri.edu")

                # get motor object, here path from root folder
                motor_obj = await client.nodes.root.get_child(["0:Objects",
                    f"{idx}:Motor"])

                # get motor variables
                motor_rpm = await client.nodes.objects.get_child(
                    path=[f"{idx}:Motor", f"{idx}:RPM"])
                motor_inp = await client.nodes.objects.get_child(
                    path=[f"{idx}:Motor", f"{idx}:Input"])

                cycles=10**3
                delay = 0
                case='wlan'

                _logger.info('start timing of write operations')
                t1 = time.perf_counter()
                await time_write(motor_rpm, cycles, case, delay)
                t2 = time.perf_counter()
                _logger.info(
                    f'finished timing of write operations: {t2-t1}s')

                _logger.info('start timing of method calls')
                t1 = time.perf_counter()
                await time_method(motor_obj, idx, cycles, case, delay)
                t2 = time.perf_counter()
                _logger.info(f'finished timing of method calls: {t2-t1}s')

                _logger.info('start timing datachange_notifications')
                t1 = time.perf_counter()
                await time_subscription(client, obj=motor_obj, idx=idx,
                    var=motor_rpm, var_name='RPM',
                    case=case, period=10, change_int=10,
                    duration=20)
                t2 = time.perf_counter()
                _logger.info(
                    f'finished timing of datachange_notifications: {t2-t1}s')

            except asyncio.exceptions.TimeoutError:
                _logger.warning(f'Connection failed. Connecting again to
{server_endpoint}')
                continue

```

```

if __name__ == "__main__":
    if len(sys.argv)>1:
        # arguments passed to script
        # accepted options: -h/--host
        # syntax <option> <host>
        if '-h' in sys.argv:
            idx_option = sys.argv.index('-h')
        elif '--host' in sys.argv:
            idx_option = sys.argv.index('--host')
        else:
            raise ValueError('only -h and --host accepted as options\n \
                               <option> <host>')

        # set host
        host = sys.argv[idx_option+1]
        asyncio.run(main(host))
    else:
        asyncio.run(main())

```