Chad Clusker

Investigating the use of a Python IEC61850 MMS server on the Vaisala Indigo500 transmitter

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

29 April 2025

Abstract

Author:	Chad Clusker
Title:	Investigating the use of a Python IEC61850 MMS server on the Vaisala Indigo500 transmitter
Number of Pages:	32 pages + 3 appendices
Date:	29 April 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT Systems: Embedded IoT Devices
Supervisors:	Roope Savolainen, Software Engineer Joseph Hotchkiss, Project Engineer

---

This study explores the process of adding support for the IEC61850 standard for the Vaisala Indigo500 transmitter. Furthermore, it details the development of a proof-of-concept Python IEC61850 MMS server on the Indigo500 transmitter that offers the measurements from a Vaisala MHT410 transmitter to connected IEC61850 clients. Further objectives include assessment of the performance of said server, the development process, future maintainability, and determining how much work is left before the project can be put into production.

Firstly, an assessment of available IEC61850 libraries was made after which the libiec61850 library written in C was chosen. Next, a comparison between a server written in the library's native language, C and a server written in Python using "experimental" Python bindings was carried out to determine the best language for the project. Further exploration was done after Python was chosen to determine how well the server would function on the Indigo500 transmitter.

After creating an initial server process design and developing a proof-of-concept server process based on the design, performance testing of the server was made to determine how well it performed. Furthermore, the groundwork was laid out to support future development by creating a feature branch on the Indigo500 Yocto layer consisting of required recipes for the feature as well as a new git repository for the server process source code.

Overall, the primary objectives of a proof-of-concept server functioning on the Indigo500 device was achieved. Further assessment of the work required before the new feature can be put into production was made, as well as suggestions on how to improve the performance of the IEC61850 MMS server. This study, as well as internal documentation support the future development and maintenance of this feature on the Indigo500.

Keywords: IEC61850, Yocto Project, Linux, Transformer oil, Python

---

Contents

List of Abbreviations and Key Concepts

AC:	Alternating current
API:	Application programming interface
BSP:	Board support package
CSV:	Comma-separated values
DGA:	Dissolved gas analysis
DNP3:	A set of communication protocols used in process automation
GOOSE:	Generic object-oriented substation event
ICD:	IED capability description
IED:	Intelligent electronic device
JDK:	Java development kit
JSON:	JavaScript object notation
MICS:	Model information conformance statement
MMS:	Manufacturing message specification
MVA:	Megavolt-amperes
PICS:	Product information conformance statement
ROC:	Rate of change
RTU:	Remote terminal unit

SV: Sampled value

SWIG: Simplified wrapper and interface generator

TCP/IP: Transport control protocol / Internet protocol

Key Concepts

awk: A text filter and manipulation tool

ctypes: A Python library that provides C compatible datatypes

Modbus: A client/server communication protocol

sed: A stream editing tool for filtering and editing text

setuptools: A Python library that is used for creating Python packages

top: A real time Linux process monitor

1 Introduction

Transformer oils are an important part of industrial power transformers that are in turn an integral part of the modern electrical grid. The regular monitoring of the state of transformer oils in transformers is critical to the safe and continual operation of the transformers and the electrical grid at large. Vaisala offer various products for the monitoring of transformer oil properties. These products include the OPT100, a multi-gas DGA monitor device, that can monitor temperature and air leaks as well as the gas build-up of up to nine fault gases in all types of ester liquids and mineral oil. The OPT100 also supports the IEC61850 standard which is widely used in power substations for communication. Vaisala's offering also includes the MHT410 transmitter which monitors hydrogen and moisture buildup and temperature in insulation liquids, for example, transformer oils. Although the MHT410 supports both DNP3 and Modbus, it does not support the IEC61850 standard. The MHT410 can be used with the Indigo500 transmitter that monitors measurements from connected measurement probes and can show these measurements and trends either locally on a display or remotely via a web interface. However, the Indigo500 does not support the IEC61850 standard. Currently, this means that if a customer uses a MHT410 in an environment using IEC61850 they must create their own solution for communicating the measurement data from the MHT410 to IEC61850 clients. Not all customers require all the features of the OPT100, and their needs could be met with a MHT410 transmitter. Therefore, if the Indigo500 had support for the IEC61850 standard, those customers would have an alternative that they could choose.

The objectives of this study include: choosing a suitable IEC61850 library that will enable building a prototype IEC61850 server on the Indigo500, building said server to read the measurements of a connected MHT410 and provide them to IEC61850 clients, assessing the performance impact of this feature on the Indigo500, observing how smooth the development process was and finally, making sure the server can easily be maintained in the future. If the objectives

are met, Vaisala can offer a new solution to customers who require transformer oil monitoring in an environment that uses IEC61850 communication.

2 Theoretical background

This chapter describes the application of transformer oils, why continuous monitoring of their state is critical for the electrical grid and how Vaisala's product offering fulfils the need of monitoring transformer oil. It explains what the IEC61850 standard is, what communication protocols it defines, how data is organised and what role it has in power substations. Lastly, this chapter describes what the Indigo500 transmitter is, what features it has, how software images are built for it, and provides a high-level overview of the software architecture.

2.1 Transformer oil measurements, OPT100 and MHT410

Transformers are pairs of wire coils used to increase or decrease voltage across two circuits. If the first coil has a changing current in it due to AC power, this results in a changing magnetic flux in the second coil which also induces an electromagnetic field in the second coil. This electromagnetic field drives current in a circuit that is connected to the second coil. Depending on the ratio of turns in the two coils, the resulting transformer is either a step-up transformer, increasing voltage in the second circuit or a step-down transformer which does the opposite. Figure 1 below shows the circuit diagram of a transformer. (Wolfson, 2014: 205.)

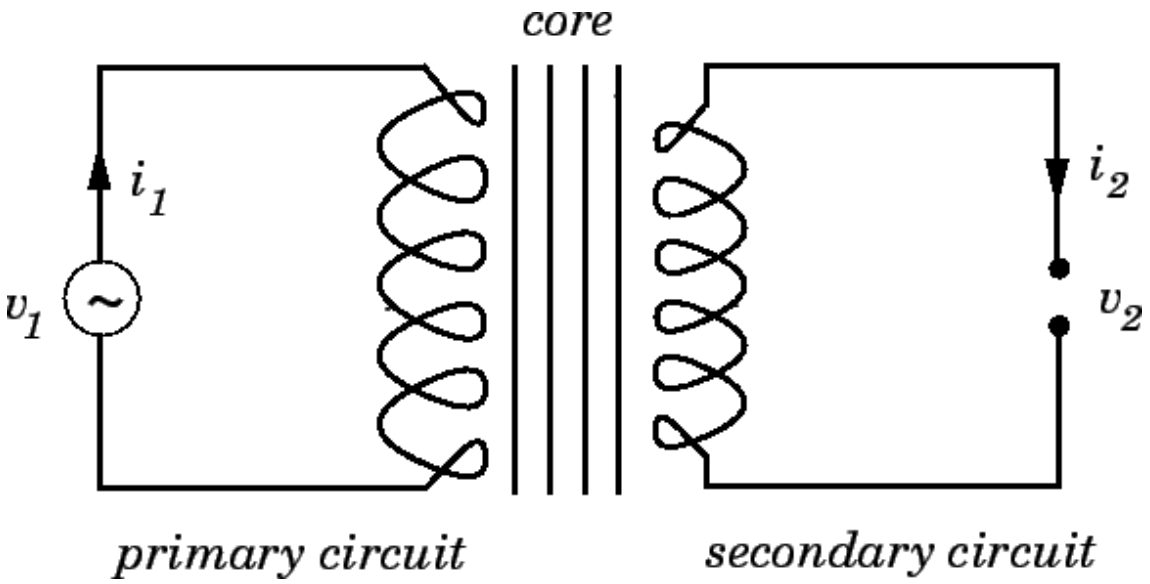

Figure 1: Circuit diagram of a transformer. (Fitzpatrick, 2007)

Power plants often generate power at a much higher voltage than what consumers connected on the grid require. The reason for this is that there is less power lost during transmission if the voltage is higher.

$$P = IV \tag{1}$$

Formula 1: P is power, I is current, V is voltage.

From Formula 1 it can be deduced that when voltage grows, current shrinks, which in turn means that the power dissipated will also shrink (Wolfson, 2014: 205). For example, a nuclear power plant has an output voltage of 20 kV and since the power outlets at a home in Finland provide 230V AC there must be transformers throughout the power distribution network that transform the 20kV to 230V. Without transformers there would be much more power lost during transmission since the voltage would have to be the same as the power outlet voltage throughout the distribution network.

For transformers to function properly, insulation must be used within and between the electrical circuits. Insulation is also used within the core (see Figure 1) and other structural parts of the transformer to eliminate circulating currents. Circulating current can produce excess or overheating and can result

in degradation of components and even partial discharges or short circuits (Moodley, 2024). A large portion of modern power transformers are oil immersed. This is for two reasons. Firstly, liquids that are used in transformers have higher dielectric strength and are good conductors of heat. Secondly, the oil can also double as a coolant for the transformer. An added benefit of liquid immersed transformers is that liquid tests such as DGA can provide useful information for assessing the condition of the transformer. (Ryder and Foata, 2024: 517) DGA is used to monitor the build-up of so-called fault gases, some of which are combustible. Usually when there are no abnormalities in the transformer oil, it is sampled for DGA every few months or sometimes annually. Sometimes on-line DGA is used which means the transformer can continue to operate until there is a problem detected by the analysis probe, saving money in the long run as routine checkups are avoided. (International Electrotechnical Commission, 2019) According to a study of transformer failures between 1997 and 2001, 28 out of 94 (29.8%) of all transformer failures of transformers with a power rating of 25MVA or above were caused by insulation failure or oil contamination. Transformers of such power ratings are commonly found in power transmission and power substations (Daelim Inc., no date). These failures resulted in almost 300 million USD in losses, adjusted for inflation. Using devices to monitor transformer oil contamination levels as well as oil heat, oxidation, pressure, and moisture can prevent transformer failures and even extend the operational life of a transformer by years. (Bartley, 2003: 3, 6; Vaisala, 2023.)

Vaisala offers several devices that have the capability to measure and monitor transformer oil. Although this study mainly focuses on the MHT410 transmitter, it is relevant to give a brief description of the OPT100 DGA monitor’s capabilities. The OPT100 is a maintenance-free multi-gas DGA monitor that is used to monitor the internal condition of an electrical transformer. It can be used with mineral oil and all ester liquid-based transformer oils. It draws oil from the transformer, performs DGA and releases the oil back into the transformer. It measures concentrations of up to 9 different fault gases, gas pressure as well as the moisture and temperature of the transformer oil. The MHT410

transmitter, on the other hand, measures moisture in oil, hydrogen concentration and temperature (Vaisala, 2022). The MHT410 does not provide all the detail of the OPT100 but it does provide all the relevant measurements for it to function as an early warning detector in transformer oil monitoring. (Vaisala, 2024; Vaisala, 2023.)

2.2 IEC61850 standard

IEC61850 is a standard for power or electrical substation automation created by the International Electrotechnical Commission. It is a collection of standards and protocols, it includes communication standards such as GOOSE, MMS, and SV and a data model describing how data is organised. Figure 2 below shows how the IEC61850 standard is organised.

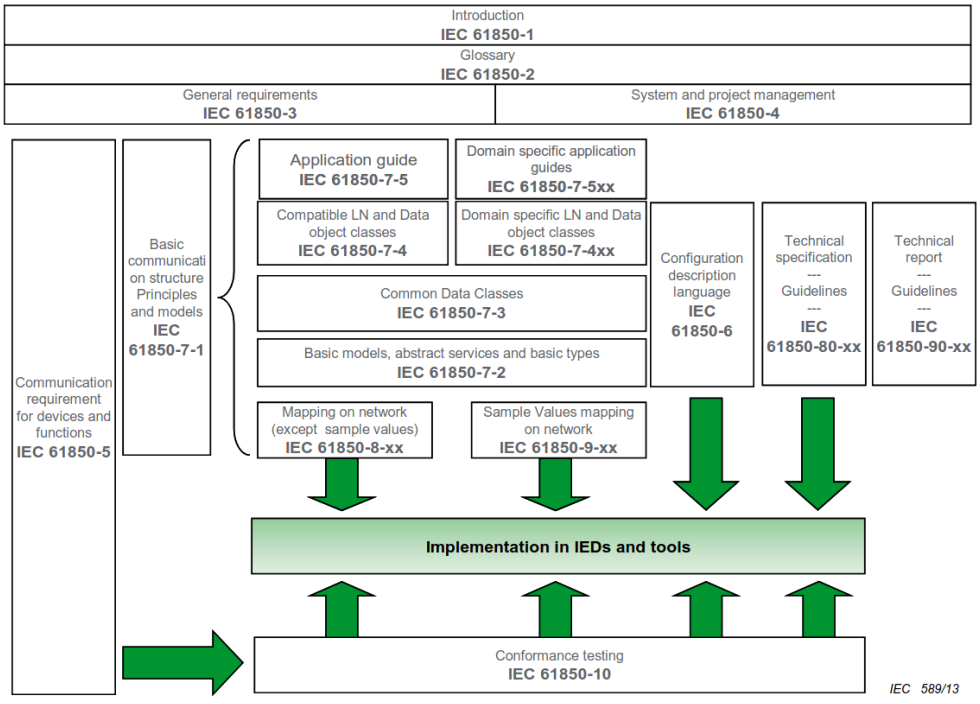

Figure 2: IEC61850 standard content. (International Electrotechnical Commission, 2013)

As shown in Figure 2, the IEC61850 standard contains various technical specifications and guides on how the standard should be applied in different scenarios. (International Electrotechnical Commission, 2013)

The standard defines three levels in the architecture of an electrical substation: station level, bay level and process level. The levels are displayed in Figure 3 below.

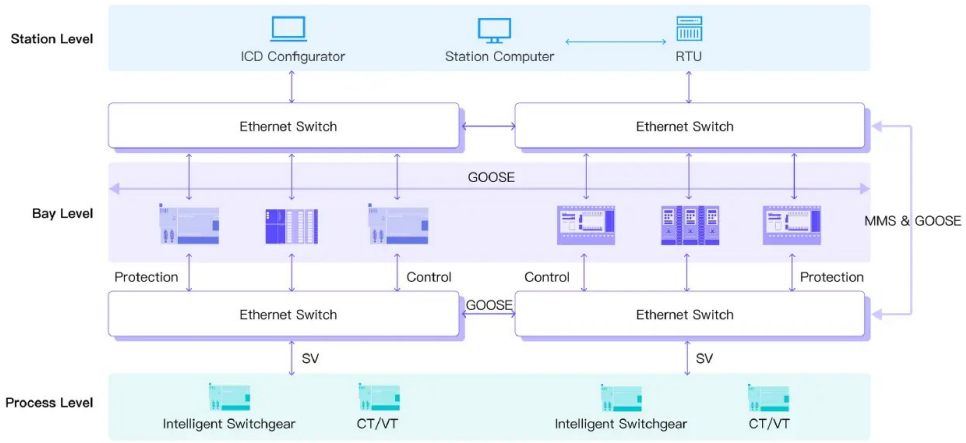

Figure 3: IEC61850 architecture. (Team Neuron, 2023)

The lowest level, as shown in Figure 3, is the process level. This level includes devices such as circuit breakers, voltage transformers and current transformers. Going a level up to the bay level, one finds IEDs or devices that control or monitor devices in the process level. An example of this could be an OPT100 device, which monitors the transformer oil of a voltage transformer. Lastly, the station level includes devices that act as clients and connect to the IEDs in the bay level requesting their data and controlling processes. (SGRwin, 2021; International Electrotechnical Commission, 2013.)

As mentioned earlier, the IEC61850 standard defines three different communication protocols. The first protocol, GOOSE, is mainly used for real-time information exchange between IEDs in the bay level and uses a publisher-subscriber model. In practice, this means an IED acts as a publisher, publishing event messages to all IEDs that have subscribed to the messages. These messages are called GOOSE messages. The second protocol defined by IEC61850 is the SV protocol. This protocol is mainly used to communicate analog and digital sampled values to and from devices on the bay level and

process level. SV communication could happen, for example, between a voltage transformer and an IED controlling and monitoring it. The third protocol is the MMS protocol that standardises communication between IEDs and RTUs at the station level. It is used, for example, for a power substation operator at the station level to be able to see the measurements from an IED on the bay level that is monitoring the temperature of transformer oil in a transformer on the process level. (SGRwin, 2023.)

The IEC61850 protocol also describes a data model that devices in a substation network adhere to. This includes how the devices describe their data internally but also how they consume data from other devices. For example, a server will describe the data that they provide to clients in a standardised way so that clients who also adhere to the standard know how to request data and where from on the server. This means communication between devices from different manufacturers will work if they support the IEC61850 standard in some capacity. Figure 4 below shows a visual representation of the IEC61850 Data model.

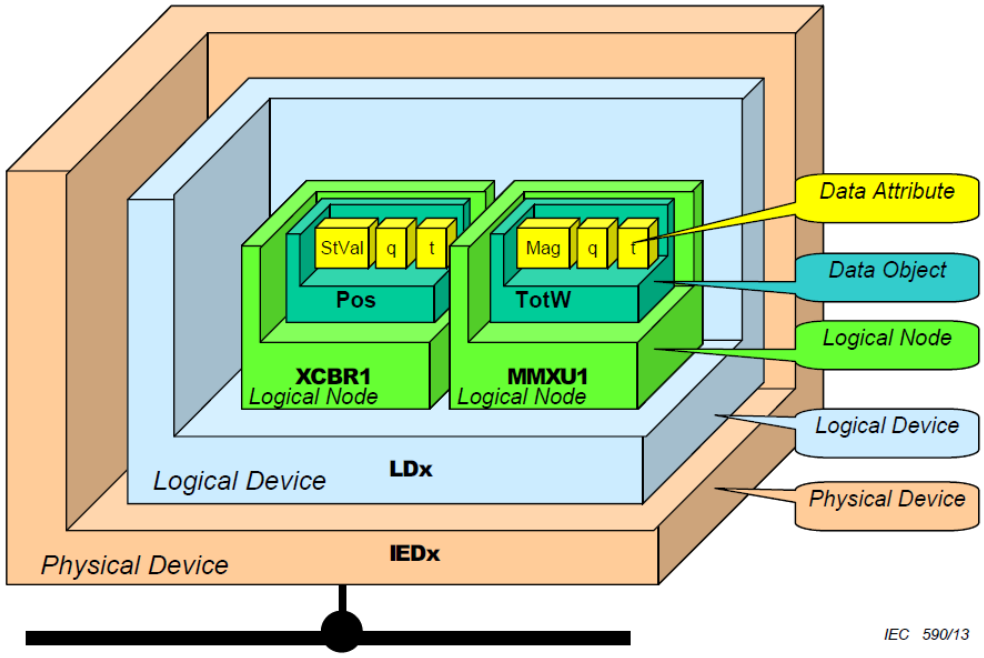

Figure 4: IEC61850 Data model (International Electrotechnical Commission, 2013)

The data model in IEC61850 is in the form of a hierarchical tree structure. The tree structure consists of five levels as shown in Figure 4: The highest level is the “Physical Device” level. This is a descriptor of a real (physical) device, in other words an IED which contains, for example, its network address, e.g. an IP address. These physical devices are further broken down into so-called “Logical Devices” which is the second level in the data model. A single physical device may have many “Logical Devices” within it. A “Logical device” often represents a group of functions of the IED. The next layer is the “Logical Node”. These represent smaller collections of functions or data nodes. Examples would be a measurement “Logical Node” that contains all the measurement data of the IED in question or a “Logical Node” that controls relays on the IED. The “Data Object” level would represent a specific function such as switching a relay or the temperature measurement data. “Data Objects” are further broken down into the smallest unit, “Data Attributes” which, in the case of, a temperature “Data Object” would consist of a timestamp, the quality of the reading and the reading itself. Data is referenced in the following format: “LogicalDevice.LogicalNode.DataObject.DataAttribute”. (Team Neuron, 2023; International Electrotechnical Commission, 2013.)

2.3 Vaisala Indigo500 transmitter

The Vaisala Indigo500 is an embedded Linux device that serves as a host for Indigo compatible, stand-alone “smart probes” or measurement probes. These probes differ from standard probes as they do more than just send measurements from a sensor forward. These measurement probes also have extra computational power meaning that they can calculate derivative measurands and host more complex communication. These probes measure a range of different quantities, from temperature and humidity to liquid concentration and moisture in oil. The device has a web interface and can be configured with a touch screen display from which one can monitor the current measured values of connected probes as well as the history of the measurements. The device also has 2 relays, analogue inputs, and outputs allowing for control over different processes. The Indigo500 is widely used in

factories, clean rooms and in industrial workflows where control over a workflow and measurements of the environment or a specific substance are crucial. Below is an image of the Indigo500 transmitter. (Vaisala, 2022.)

Figure 5: Vaisala Indigo500 transmitter (Vaisala, 2022)

For a complex device such as the Indigo500, as shown in Figure 5, there are many benefits to using Linux: it is ubiquitous, there is an endless number of resources regarding Linux and development for it on the internet and elsewhere. Linux ubiquity is reinforced by the sheer amount of software that can be run on the platform. If there is a need to add support for a library, there is a very high chance that there is a Linux implementation. To create Linux images for the Indigo500 device, Vaisala leverage an open-source tool called the Yocto Project. More specifically the device takes advantage of an internally maintained Yocto platform that is responsible for the maintenance of the kernel and other software tools that are not project-specific. This allows the Indigo500 team to focus on software development specifically for the Indigo500, without needing to spend time maintaining a custom Linux image. (Moseley, 2017.)

The different capabilities of the Indigo500 are run by different processes, as is typical in a Linux device. This means that the user interface is its own process

and program controlling the relays is another. Communication between these processes is achieved using D-Bus, a message bus system developed by freedesktop.org. To better illustrate how this works, when the UI process wants to let the relay process know a relay should be switched, it sends a signal over D-Bus that lets the relay process know to switch a relay. Below is an illustration of how different processes communicate with each other.

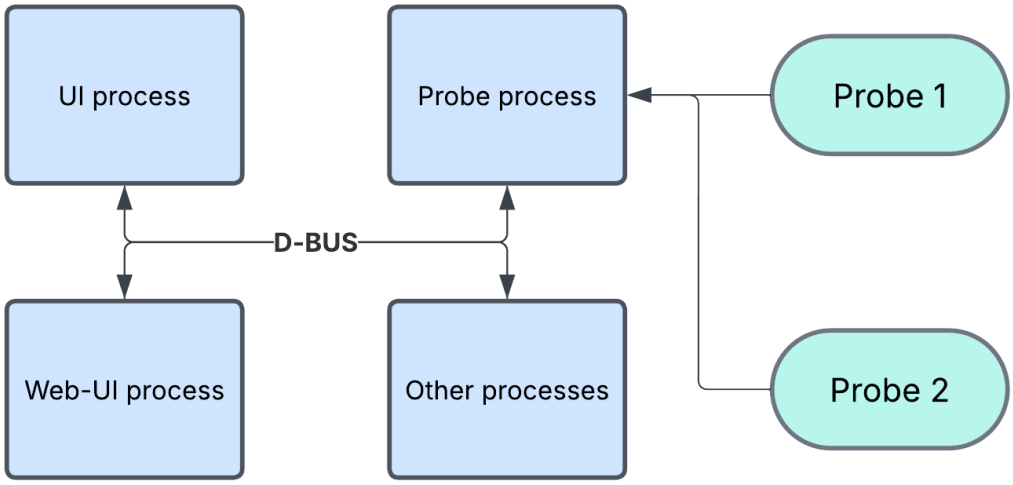

Figure 6: High level system architecture.

For the management of these processes, the Indigo500 uses the systemd software suite. Systemd is the very first process that starts when the system is booted and is responsible for starting and stopping all other services on the device. A systemd service file specifies which services to start and in what order and includes a logging daemon that is used to log the activity of a service. It is worth mentioning that processes use a “Mainloop” instance from the GLib library by the GNOME project instead of a manual infinite loop. This allows for more control over the timing of events when the process is running. Lastly, the Indigo500 project uses Git for version control of the software. This is in a “polyrepo”, meaning that the whole project consists of smaller repositories often for specific functions as opposed to all the source code being under one “monorepo”.

3 Methods and materials

This chapter gives a short overview of software licenses and different library options and describes the IEC61850 library options for this project. The choice of library is also discussed here. Further, it covers the Yocto Project and how images are built for the Indigo500 transmitter. Lastly, the chapter discusses the reasons for a server language comparison regarding performance and ease of development.

3.1 Libraries and licensing

In some cases, the best course of action when there is a need to support a new protocol is to write one's own library for it. This can be worthwhile as one is not dependant on third parties to maintain the library. However, for this project building and then maintaining an internal library for IEC61850 would not be a good use of time or resources. IEC61850 is a large standard to write a library for and there are plenty of third-party libraries to choose from. When choosing a library for a commercial project, many things must be considered. One must consider whether the library is being actively maintained and supported, under what the license the software is published, in what language the library is written and what platform it is written for.

Broadly speaking, there are two types of libraries: proprietary or closed-source and open-source libraries. Both have their respective upsides and downsides. Proprietary libraries come with maintenance, support and licences that will not be a problem in a commercial setting, but the libraries come at a cost. It can be difficult to gauge the quality of and ease of integration of the code as most often one does not have access to the full source code prior to purchasing. Another downside is that proprietary libraries sometimes ship only as binaries not allowing the customer to make changes to the source code. Open-source libraries, by nature, give access to the full source code, meaning that it is easy to determine whether a library is sufficient or not. It is also a lot easier to test if a library would work well for ones needs as one is freely able to modify the source

code. What open-source libraries can lack is maintenance and support, sometimes completely if a project has been forgotten. Often it is difficult to determine how well or for how long a specific project will have support.

Another potential downside, in this case, is that some open-source licenses are so called copyleft licenses. This means that one is free to use and distribute the software under that license, but all modified or extended versions of that software must also be published under the same license. In the case that a piece of software published under a copyleft license would be included in the Indigo500 software the team would be required by the license to publish all the source code for the device as that would be considered an extension of the software published under a copyleft license. (GNU Project, no date.)

If there was an open-source library for IEC61850 available, that would mean development could be started immediately. This means open-source libraries have a slight priority. Ideally, the library would be written in Python as most of the software on the Indigo500 is written in the same language. There are a handful of proprietary libraries, but since there is no proprietary library that is written in Python the initial plan was to rule out all open-source libraries first. There were two clear open-source candidates, libiec61850 by MZ-Automation and iec61850bean from Beanit. If neither was suitable, there would have been further research into which proprietary library to choose for the project. The libiec61850 library is written in C, has experimental Python bindings and has the option for a commercial license if the GPLv3 license it is published under is too restrictive. This is because the GPLv3 license is a copyleft license (GNU Project, 2007). Since MZ-Automation offer commercial licenses, the library should not have support suddenly dropped since they have paying customers. This library is already in use for testing in another project at Vaisala and is liked by the team boosting confidence in it. The other library, iec61850bean, is published under the Apache-2.0 license which is not a copyleft license. This would be useful as there would be no license trouble and work could begin immediately. However, the downside of iec61850bean is that it is written in Java. This would require adding the JDK to the Indigo500 to run programs using

the library or writing Python bindings for the library. Both options mean unnecessary work. This is especially true when libiec61850 offers everything this project requires: it is open-source but has an option for a commercial license, it has Python bindings although they are experimental, and a recommendation from another team. Thus, libiec61850 was chosen as the library for the present project.

3.2 The Yocto Project and Indigo500

The Yocto Project is an open-source collaboration project enabling users to create customized Linux images regardless of hardware architecture. It provides tools and templates that enable image creation. These include recipes, configurations and dependencies. Software projects using The Yocto project are also using tools and software from OpenEmbedded. These include the OpenEmbedded build system and the meta-openembedded layer. (Moseley, 2017; The Yocto Project, 2025; Openembedded.org, no date.)

Central to the Yocto project is the Layer Model. Layers are repositories of instructions or “recipes” and other metadata that tell the OpenEmbedded build system what to do. Below is a diagram of the layer architecture.

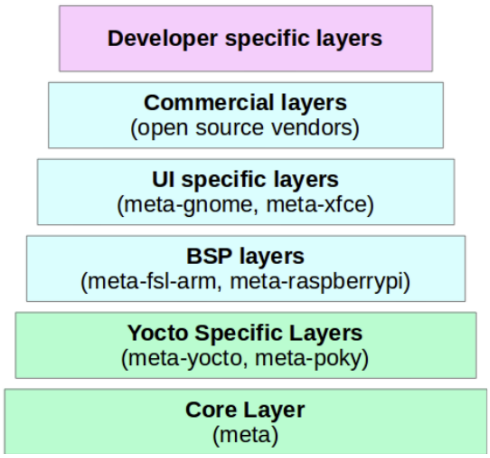

Figure 7: OpenEmbedded layer architecture. (Ozcelikors and Gumuskavak, 2020)

Figure 7 shows how different layers relate to each other and how they are named. Layer names are prepended with “meta-”. The bottom two layers are ones that all projects based on Yocto include. These are layers that include recipes for basic tools that most projects use. Further up are BSP layers that contain recipes and tools for specific hardware and thus will be included on a project-to-project basis. The same goes for all other layers that come from third party vendors. At the top of this architecture are the developer or project specific layers. These will include recipes for programs specific to a project. Isolating layers in this way helps future customisation and reuse. (The Yocto Project, 2025.)

As a user of The Yocto Project, the tool that will be the most familiar is the OpenEmbedded build system or Bitbake as it is more commonly known. It is maintained by OpenEmbedded and is a generic task execution engine written in Python. It is analogous to GNU Make but instead of “makefiles” Bitbake uses “recipes”. Recipes contain descriptive information of a package, such as where the source code for the package is and how to fetch it, how to compile said source code, how to make the generated artefacts into an installable package, and where on the target device the package should be installed. These recipes can also be amended and patched. Amending a recipe involves a new file called an amend file, and they are used to overwrite or extend a recipe. This is useful in situations where the original recipe should be kept the same or if there is a need to conditionally add amendments to a recipe. (Purdie, Larson and Blundell, 2025) Patching is like amending in that both require new files, and both modify the result of the package. However, patching a recipe modifies the source code that the recipe fetches instead of the recipe itself. This is used, for example, in situations where the source code requires a modification to run on the selected platform. When Bitbake is told to build a recipe, it will go through the steps outlined and then amend or patch the recipe if needed. The result will be a package that is ready to be a part of a full Linux image.

Customising Linux images using the Yocto project is done by first adding a new developer specific layer to a project (see Figure 7). The next step is including

required recipes in that layer by adding their names to the layer configuration file called “layers.conf”. After all required recipes are added to the layer, it must be added to the “bblayers.conf” file that will tell Bitbake to include recipes from the layer in the image it will create. The Indigo500 device has its own Yocto layer that contains recipes for all processes, programs and tools that the device uses. Vaisala also maintain another Yocto layer that is more generic that many different projects within the company use. This enables teams to focus on the specifics of a project and not have to spend as much time maintaining a complex Yocto environment.

3.3 Server language choice

The IEC61850 library that was chosen, libiec61850, is written in C, a language different to most software on the Indigo500. Introducing software written in a language other than the main language of a project can result in issues and challenges for the developers of the project. Some of these issues include added complexity, difficulties with data handling and memory management. Therefore, there must be a strong reason not to use the Python bindings provided by the library. Thus, a comparison must be done between a server written using the library’s native language, C, and one using the Python bindings. (Yang et al., 2024.)

The Python bindings of the libiec61850 library are labelled as “experimental” and are not actively maintained. These bindings are implemented using SWIG, a software development tool that connects, for example programs written in C with various high-level programming languages, including Python. SWIG simply wraps the C functions in the libiec61850 library so that they can be called from, in this case, a Python program. In general, this setup should not be a problem for the present project despite the bindings not having active support. Nonetheless, an investigation must be made into how well the bindings work. A Python server must also be compared to one written in C to see how they differ in performance. An assessment must also be made to determine whether writing the server in Python adds any meaningful extra work during

development or for future maintenance of the server. (Simplified Wrapper and Interface Generator, 2024)

4 Exploration

The goal of this project was to add support for IEC61850 on the Indigo500 transmitter. More specifically it is to create an MMS server, henceforth called server, on the Indigo500 that provides the measurement data from a connected MHT410 transmitter. To do this, an IEC61850 library must be included in the Indigo500 firmware, and the measured values must be sent from the MHT410 to the server. This chapter covers acquiring and building the libiec61850 library and creating a simple server in Python to verify how well the Python bindings work. Further, a comparison between an identical server written in C is done to determine which language should be used for the actual proof of concept server. Lastly, an initial design of the proof-of-concept server is described.

4.1 Building libiec61850 and creating a simple Python server

The libiec61850 library source code can be acquired from MZ-Automation's GitHub repository. Once the software is cloned locally, the library can be built. The library supports both CMake and Make build systems and the build instructions can be found in the repository. The library includes examples written in C which can be used to test that the library functions as intended. The examples consist of many different servers and clients that demonstrate different parts of the IEC61850 protocol. After testing various combinations of server and client pairs, it could be determined that the library was functioning properly. The next step was to build the Python bindings and create a simple server using Python.

The libiec61850 library offers 3 different methods of creating a model to configure an IEC61850 server. The first two methods involve an ICD file. This is a simple XML file that describes the server model. Using the tools in the libiec61850 library, the ICD file can be converted into two different formats. The first format is a file with a ".cfg" suffix, in other words a configuration file. A FILE pointer pointing to this configuration file is passed to the "ConfigFileParser()" function from the library. This function returns a model object that can be used

by the server. The other format follows a similar logic but instead creates a C source file “static_model.c” and a header file “static_model.h” that include a model description. A struct from the “static_model.c” file is passed by reference to the function “IedServer_create()”. The third way of creating the model for a server involves manually making API calls to create the model dynamically. Below is an example of how an example written in C on the right is used as a template for the Python version on the left.

There were problems when trying to create a Python server using the first two methods of model generation. Passing a reference to a structure in a C source file from Python is possible by using the “ctypes” module or SWIG as a translation layer. However. because this proved to be non-trivial the other methods were tested before coming back to this. Using a “.cfg” file generated by the “model_generator” tool was simple at first since passing a FILE pointer to a function in Python is trivial. However, the function “IedServer_createWithConfig()” that was used to create the server takes a parameter called “tlsConfiguration”. If there is no need for this, as was the case for this simple server, a null pointer can be used as a function parameter instead. Since Python’s “None” is treated as a null pointer this should work. However, that was not the case. This was also attempted using a null pointer from “ctypes” but to no avail. After further debugging, this issue is most likely caused by SWIG expecting the function to be called with a pointer of a specific type, in this case a pointer to a “tslConfiguration” struct. As a result, when this function is called from Python using “None”, SWIG complains about an invalid function parameter data type. Another issue that was encountered at this stage of development was that error messages from the C functions would not be printed to stdout when using the Python bindings. The cause for this was not found and thus this makes debugging the C functions marginally more difficult if there were to be a problem.

Due to the difficulties described above, an attempt was made to build the server dynamically using API calls. This means that instead of having the model described in a separate file, each node in the model must be created using

explicit function calls. For example, to create a logical node the function “LogicalNode_create()” could be used. The examples include a dynamic server which could be easily used as a template. Listing 1 below is an example of how that was done.

```
ttmp1 = iec61850.LogicalNode_create("GGIO1", ldevice)
ttmp1_tmpsv = iec61850.CDC_SAV_create(
     "TmpSV", iec61850.toModelNode(ttmp1), 0, False)

temperatureValue = iec61850.toDataAttribute(
     iec61850.ModelNode_getChild(iec61850.toModelNode(ttmp1_tmpsv),
     "instMag.f"))
temperatureTimestamp = iec61850.toDataAttribute(
     iec61850.ModelNode_getChild(iec61850.toModelNode(ttmp1_tmpsv), "t"))
```

Listing 1: Creating a logical node with data attributes.

Listing 1 shows how data attributes are created. Some extra functions are provided by the Python bindings to make sure that the API functions are called with the correct data types.

Once the server model was ready and the server started, the next step was to update the server with some data points. Data attributes can be updated with real timestamps and random values from a function written to simulate measurements. Updating server data must be done after the data model is locked using the “IedServer_lockDataModel()”. After the required nodes have been updated, “IedServer_unlockDataModel()” is called and the server will begin processing client requests. After implementing this, the server (see Appendix 1) was verified by using “iec61850_client_example4” to ask for the data attributes from the server and their values.

4.2 C vs Python

After verifying that a Python server would work, the next step was to compare the performance of it to an identical one written in C. This involved logging the CPU usage and memory consumption of both programs for comparison. It is to be expected that the C server would have less memory usage as the Python interpreter always consumes some memory regardless of the program being

run. Another assumption is that the Python program would have higher CPU usage than the C server also due to the interpreter.

To test this a combination of command line tools were used. These tools were “top” for CPU and memory usage and then a combination of both the “sed” and “awk” tools to extract that information into CSV format. “Top” was configured to take a sample every 3 seconds for approximately 1 minute. Figure 8 below shows the samples taken.

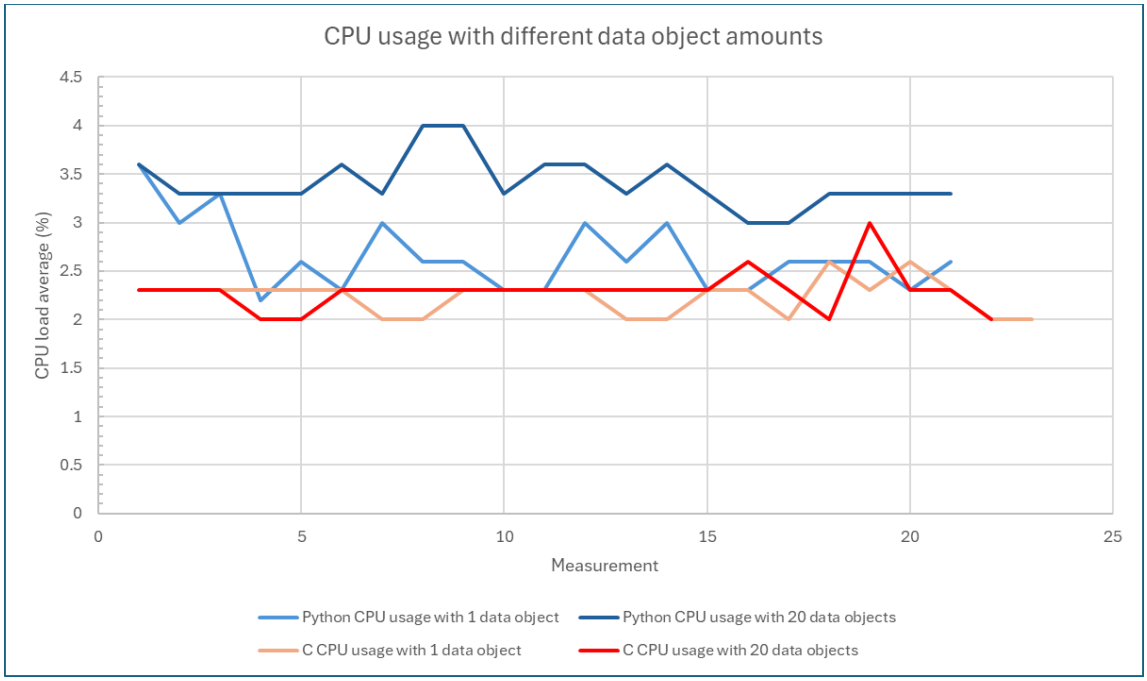

Figure 8: CPU usage with different data object amounts.

The tests were run first with each server updating one data object per second and then with 20 data objects being updated per second. This was done to better simulate the MHT410 that has 12 measurement points that the IEC61850 server will be updated with. Figure 8 shows that there is not a significant difference between the two C servers and there is a larger difference between the Python servers. However, the CPU usage across all types of servers is not high enough to warrant a move away from Python as the server choice. Table 1 below shows the memory usage of both servers.

Table 1: Virtual memory usage of a C server vs Python server.

Virtual memory usage (KiB)		
Data objects	1	20
C	78404	78420
Python	102968	103048

In terms of memory usage, a ~30% increase in memory usage of the Python server in comparison to the C server is indeed large. However, the amount of memory, ~102 MiB, will not be a problem on the Indigo500. Thus, after analysing the performance there was not a strong enough reason to move away from writing the server in Python.

4.3 Process design

The requirements for the server were the following: If a MHT410 transmitter is connected to the Indigo500, then the server should be on and updated with values from the transmitter, if there is no MHT410 connected, then the server should not run to save system resources, but the process should continue checking for new connected MHT410s. Figure 9 below shows the process in flowchart form.

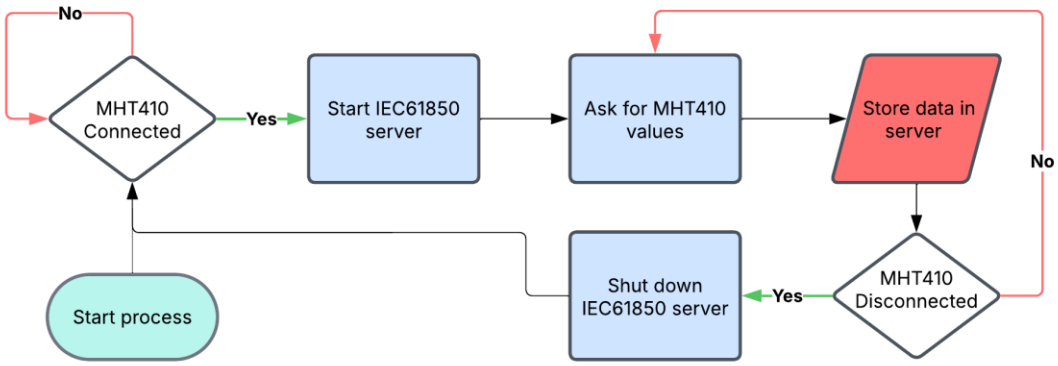

Figure 9: Flowchart of process design.

The flowchart in Figure 9 describes the initial plan for the design of the server process. First, it checks if there is a connected MHT410 transmitter. Once the probe is connected, the server starts. The first time this happens, the server model should be dynamically created. Then it begins reading measurements over D-Bus from the transmitter and updating the internal data structure of the server. This update loop should have an update cycle no less than 1 second as the measurements from the transmitter only update once a second. Lastly, if the probe disconnects for some reason, we must safely shut down the server and return to checking for new connections from a MHT410.

5 Proof-of-concept

This chapter outlines the process of obtaining, building and configuring the libiec61850 library on the Indigo500 and creating a server that takes measurements from a probe and sends them to an IEC61850 client over TCP/IP.

5.1 A simple server on the Indigo500

The libiec61850 library recipe is provided by the meta-networking layer from OpenEmbedded. Adding this recipe, along with its patches, on a new feature branch of the Indigo500 meta layer and adding libiec61850 into a package group enables subsequent image builds to use the library. Exploring a new build with libiec61850 included shows that the Python file “iec61850.py” is included in the site-packages folder where the rest of the Python modules are installed on the Indigo500. This means that the library can be used on the server by simply importing the “iec61850” module. Using Bitbake with the recipe is handy as there is no need to manually write a script to build and link the library on the device itself because Bitbake does that during the compilation of the Linux image. With this confirmed it is simple to test if the library works. In a Python shell, one must import the library and then call “LibIEC61850_getVersionString” to verify that API calls work. A slight inconvenience for development is the fact that the version of libiec61850 from the OpenEmbedded recipe is older than the one being used for local development. Functionally this only means that the Python module must be imported using a different name. It is “pyiec61850” on the version that is being used for local development and “iec61850” on the version that is being used on the Indigo500.

With this confirmed, the next step is to verify that TCP/IP traffic moves from the Indigo500 to a client. In addition, to support further work a new repository for the IEC61850 server was created. This repository includes the Python source code for the server, a configuration file that is used by the Python module “setuptools” to make the server into a module. After the setup, a new recipe, a new user that

is allowed to run the program, and a systemd service file that is responsible for starting the process can be added to the new Indigo500 Yocto layer feature branch. This recipe fetches everything from the repository based on the latest Git commit and packages it accordingly for use on the Indigo500. In order to test the TCP/IP traffic, the dynamic server from the previous chapter was used and so added to the repository. (See Appendix 1.) This would make it possible to verify that a client could connect to and read a value from the server. After a new image was built using the new server recipe together with the libiec61850 recipe and after the server program was run a client example could be used to test the connection. This did not work, as the port for TCP/IP traffic that was used was blocked by the firewall. The Indigo500 blocks all ports by default so a new port would have to be explicitly allowed for the IEC61850 client to connect. To enable traffic, new firewall rules were added to on the new Indigo500 Yocto layer feature branch. After adding the rules and using “netstat”, it could be confirmed that the server was listening to incoming connections on the selected port. Moreover, this was also confirmed by using a client example from the libiec61850 library.

5.2 Using a simulated environment for development

Because it is not feasible to do development on the Indigo500, the team has a simulated environment that simulates an Indigo500. The simulated environment uses “tmux”, a terminal multiplexer, that spawns a shell for each process and uses the session D-Bus bus for inter-process communication just like the Indigo500. This means that testing changes is faster as the developer is not required to remotely connect to the device every time they need to make a change, and they are not limited to the tools and programs the development images for the Indigo500 include. The development workflow approximately follows testing in the simulated environment until the feature works and only then verifies that it works on a manual build on the Indigo500.

The simulated environment would require modifications for the IEC61850 server to work as the virtual environment used by the simulator, which includes all the necessary Python libraries, does not include the libiec61850 library. This is required so that the server can access the API when being run inside the simulation. Once the virtual environment is activated, the libiec61850 Python bindings can be installed by simply copying the “_pyiec61850.so” file under “lib/python3.10/site-packages” in the virtual environment. This file is generated when the Python bindings are generated. The other modification was something that would simulate the MHT410 probe. The simulation environment uses a program that takes register maps in JSON form and “fakes” to the process handling probes that it would be a real probe with real measurements. After the modifications, the development of a prototype server could be started.

Using the design described in the previous chapter, the first thing to do was to recognise if a MHT410 was connected. This could be achieved by calling a D-Bus method that would return the information on the connected probes. Similarly, when the measurement data required, updating another D-Bus method was called with the exact registers of each measurand.

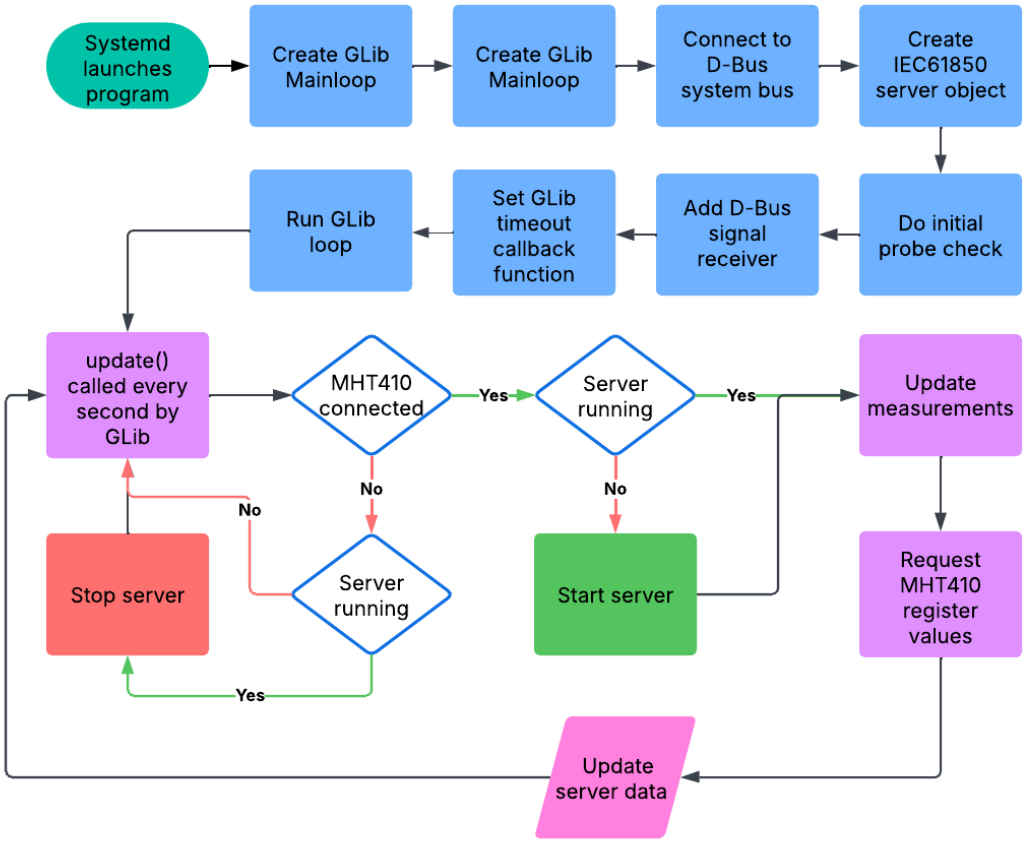

Figure 10: Flowchart of prototype server.

Once the program finishes setup, a GLib control loop is entered with a timeout of one second, meaning that every second the program does a check if the server must be shut down or started, and updates measurements if the server is running.

5.3 Moving the server to Indigo500

Moving the prototype server to the Indigo500 requires changing the source code on the Git repository created earlier. Another optimisation is to include libiec61850 as a dependency for the IEC61850 server Yocto recipe. This is possible as the recipe comes from OpenEmbedded meta layers. This allows for fewer files in the Indigo500 layer and means that manual updates of the library will not be necessary since when the Indigo500 moves to a new version of Yocto the potential recipe updates will automatically be included. In other

words, the team is not responsible for maintaining an extra recipe since it is already being maintained.

After these updates were ready and a new build was tested on the Indigo500, it was discovered that the server process would start but encounter a D-Bus exception and promptly hang instead of proceeding to start the actual IEC61850 server. For security reasons, there are rules on the Indigo500 that dictate which users can call and listen to D-Bus methods and signals respectively. Simply including the user for the IEC61850 server in the D-Bus configuration files of the processes that the new server communicates with and adding the users of said processes in the D-Bus configuration file of the new server resolved this issue. Again, using one of the example clients, the IEC61850 server could be verified to work and further verified to show the correct measurements from the MHT410 transmitter.

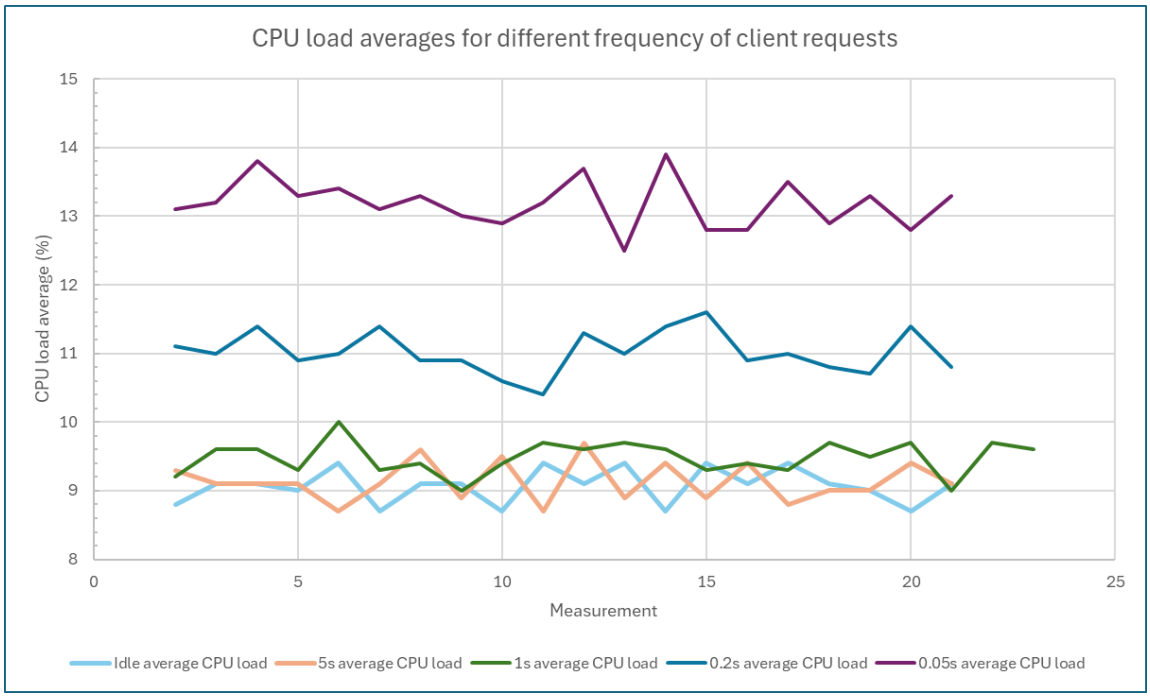

Figure 11: CPU load averages vs client requests

Some light performance testing was done to find out if the CPU load on the Indigo500 or the memory usage would be problematic. Using a simple bash

script an example server could be configured to send both connection and read requests at different intervals. Figure 11 shows the average CPU load of the server process at idle and at different intervals of client requests. Realistically, clients should not be polling the measurements more often than once a second, but it is useful to know that the CPU load average is not unmanageable. The memory usage was a stable 3.8% and did not change depending on client request frequency.

6 Conclusions & future

The primary goal of adding support for the IEC61850 standard on the Indigo500 was achieved as well as providing a prototype IEC61850 MMS server on the device. Overall, the library choice of libiec61850 is a good one. It suits the development environment of the Indigo500 with its Python bindings despite some issues with the SWIG wrapping process. It also has good licensing options that suit a commercial environment.

The performance of the prototype server on the Indigo500 is adequate; however, there is room for optimisation. The current average CPU load is slightly higher than ideal. This is partly due to the current prototype server calling D-Bus every second using the “pydbus” D-Bus library. There is a possibility that changing the from “pydbus” to “sdbus” could bring the average CPU load down from where it currently is.

Setting up a repository for the server source code and a new feature branch with a recipe for the server to the Indigo500 Yocto layer lay the groundwork for smooth future development. Together with this study, Vaisala internal documentation on IEC61850 and the libiec61850 documentation, the tools developers require to expand on the prototype, continue development of this feature are provided.

For the feature to be shipped, the server should support sending reports. Further unit and system tests are required to ensure good code quality and to minimise bugs. Adding a factory flag that would enable the server only on devices that the IEC61850 protocol should be on. Also, PICS and MICS documents must be created so that customers know how much of the IEC61850 protocol has been implemented and how the model is structured. This will make it clear how to bring the IEC61850 MMS server into use in existing IEC61850 networks.

References

Bartley, W. (2003) ‘Analysis of Transformer Failures’. Hartford Steam Boiler Inspection & Insurance Co.

Daelim Inc. (no date) Comprehensive Guide to 18MVA to 90MVA Transformers | Daelim Transformer. Available at: https://www.daelimtransformer.com/18mva-to-90mva-transformers.html (Accessed: 26 April 2025).

Fitzpatrick, R. (2007) Transformers. Available at: https://farside.ph.utexas.edu/teaching/316/lectures/node106.html (Accessed: 18 April 2025).

GNU Project (2007). The GNU General Public License v3.0. Available at: https://www.gnu.org/licenses/gpl-3.0.en.html (Accessed: 20 April 2025).

GNU Project (no date). What Is Copyleft? Available at: https://www.gnu.org/licenses/copyleft.en.html (Accessed: 26 April 2025).

International Electrotechnical Commission (2013) ‘IEC Technical Report 61850-1’. IEC. Available at: http://www.iec.ch

International Electrotechnical Commission (2019). ‘IEEE Guide for the Interpretation of Gases Generated in Mineral Oil-Immersed Transformers - Redline’. IEEE Std C57.104-2019 (Revision of IEEE Std C57.104-2008) - Redline, pp. 1–179.

Moodley, N. (2024) Mitigation of Thermal Faults in Power Transformers, Power Transformer Health. Available at: https://powertransformerhealth.com/2024/06/02/mitigation-of-thermal-faults-in-power-transformers/ (Accessed: 26 April 2025).

Moseley, D. (2017) ‘Why the Yocto Project for my IoT Project?’, Embedded. Available at: https://www.embedded.com /why-the-yocto-project-for-my-iot-project (Accessed: 20 April 2025).

Openembedded.org (no date). OpenEmbedded and The Yocto Project. Available at: https://www.openembedded.org/wiki/OpenEmbedded_and_The_Yocto_Project (Accessed: 27 April 2025).

Ozcelikors, M. and Gumuskavak, A. (2020) ‘Platform-independent Infotainment and Digital Cluster Development using Yocto Project’, in 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE). 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), pp. 1–5. Available at: https://doi.org/10.1109/ICECCE49384.2020.9179288.

Purdie, R., Larson, C. and Blundell, P. (2025) 1 Overview — Bitbake dev documentation. Available at: https://docs.yoctoproject.org/bitbake/2.10/bitbake-user-manual/bitbake-user-manual-intro.html (Accessed: 20 April 2025).

Ryder, S. and Foata, M. (2024) 'Liquid Insulation Ageing in Transformers and Reactors', in Transformer and Reactor Life Management. Springer, Cham, pp. 515–549. Available at: https://doi.org/10.1007/978-3-031-77219-1_18 (Accessed: 26 April 2025).

SGRwin (2021) 'Basic understanding of IEC 61850 - What are the key aspects?', SGRwin, 30 April. Available at: https://www.sgrwin.com/basic-understanding-iec-61850/ (Accessed: 18 April 2025).

SGRwin (2023) 'GOOSE, MMS, and SV Protocols in Digital Substations', SGRwin, 29 June. Available at: https://www.sgrwin.com/goose-mms-and-sv-protocols/ (Accessed: 18 April 2025).

Simplified Wrapper and Interface Generator (2024). Available at: https://www.swig.org/ (Accessed: 20 April 2025).

Team Neuron (2023) IEC 61850 Protocol: Features, Information Model, and Combination with MQTT, www.emqx.com. Available at: https://www.emqx.com/en/blog/iec-61850-protocol (Accessed: 18 April 2025).

Vaisala. (2023). Dissolved Gas Analysis (DGA) for Power Transformers. Available at: https://www.vaisala.com/en/measurement/dissolved-gas-oil-dga-measurement (Accessed: 21 April 2025).

Vaisala (2022). Moisture, Hydrogen and Temperature Transmitter MHT410 Available at: https://www.vaisala.com/en/products/instruments-sensors-and-other-measurement-devices/instruments-industrial-measurements/mht410 (Accessed: 21 April 2025).

Vaisala (2024) 'OPT100 User Guide M211858EN-AA'.

Vaisala (2022). Indigo500 Series Transmitters. Available at: https://www.vaisala.com/en/products/systems/indoor-monitoring-systems/indigo500 (Accessed: 26 April 2025).

Wolfson, R. (2014) '6 Transformers and Power Supplies', in *Essential University Physics*. 2nd edn. Pearson, p. 205.

Yang, H. et al. (2024) 'Multi-Language Software Development: Issues, Challenges, and Solutions', IEEE Transactions on Software Engineering, 50(3), pp. 512–533. Available at: https://doi.org/10.1109/TSE.2024.3358258.

The Yocto Project (2025). Introducing the Yocto Project Available at: https://docs.yoctoproject.org/overview-manual/yp-intro.html (Accessed: 20 April 2025).

Appendix 1: Python test server

Below is the test server:

```
 # Copyright (c) Vaisala Oyj. All rights reserved.


import os, random, signal, time
from datetime import datetime, timezone

import pyiec61850 as pli #iec61850 on device

running = 1
CDC_CTL_MODEL_HAS_CANCEL = 1 << 4
CDC_CTL_MODEL_SBO_ENHANCED = 4

RPT_OPT_SEQ_NUM = 1
RPT_OPT_TIME_STAMP = 2
RPT_OPT_REASON_FOR_INCLUSION = 4

TRG_OPT_DATA_CHANGED = 1

def sigintHandler(signalID, frame) -> None:
    """Signal handler."""
    print(signalID)
    global running
    running = 0

def random_measurement() -> int:
    """Mock measurements."""
    # perhaps redundant to seed it every function call
    random.seed(None)
    return random.randint(0, 25)

def main() -> None:
    """Main."""
    global running
    tcpPort = 10288
    print("Using version:", pli.LibIEC61850_getVersionString())

    model = pli.IedModel_create("testmodel")
    ldevice1 = pli.LogicalDevice_create("SENSORS", model)
    lln0 = pli.LogicalNode_create("LLN0", ldevice1)
    lln0_mod = pli.CDC_ENS_create("Mod", pli.toModelNode(lln0), 0)
    lln0_health = pli.CDC_ENS_create("Health", pli.toModelNode(lln0), 0)

    pli.SettingGroupControlBlock_create(lln0, 1, 1)

    # add tmp sensor logical node
    ttmp1 = pli.LogicalNode_create("GGI01", ldevice1)
    ttmp1_tmpsv = pli.CDC_SAV_create("TmpSv", pli.toModelNode(ttmp1), 0,
False)

    temperatureValue = pli.toDataAttribute(
         pli.ModelNode_getChild(pli.toModelNode(ttmp1_tmpsv), "instMag.f"))
    temperatureTimestamp = pli.toDataAttribute(
         pli.ModelNode_getChild(pli.toModelNode(ttmp1_tmpsv), "t"))

    ggio1 = pli.LogicalNode_create("GGIO1", ldevice1)
    ggio1_anIn1 = pli.CDC_APC_create("AnOut1", pli.toModelNode(ggio1), 0,
                                     CDC_CTL_MODEL_HAS_CANCEL |
CDC_CTL_MODEL_SBO_ENHANCED,
                                     False)
    dataSet = pli.DataSet_create("events", lln0)
    pli.DataSetEntry_create(dataSet, "TTMP1$MX$TmpSv$instMag$f", -1, None)

    rptOptions = RPT_OPT_SEQ_NUM | RPT_OPT_TIME_STAMP |
RPT_OPT_REASON_FOR_INCLUSION
    pli.ReportControlBlock_create("events01", lln0, "events01", False, None,
1, TRG_OPT_DATA_CHANGED, rptOptions, 50, 0)
```

```
    pli.ReportControlBlock_create("events02", lln0, "events02", False, None,
1, TRG_OPT_DATA_CHANGED, rptOptions, 50, 0)
    pli.GSEControlBlock_create("gse01", lln0, "events01", "events", 1, False,
200, 3000)


    iedServer = pli.IedServer_create(model)

    pli.IedServer_start(iedServer, tcpPort)

    if(pli.IedServer_isRunning(iedServer) == False):
        print("Are you sudo ? for some ports you need sudo")
        pli.IedServer_destroy(iedServer)


    # set signal handler for sigints
    signal.signal(signal.SIGINT, sigintHandler)

    v = float(0)
    while(running):
        """Main loop of server."""
        ret = None
        print(v)

        pli.IedServer_lockDataModel(iedServer)
        pli.IedServer_updateUTCTimeAttributeValue(
             iedServer, temperatureTimestamp, int(time.time() * 1000))
        pli.IedServer_updateFloatAttributeValue(iedServer, temperatureValue,
v)

        ret = pli.IedServer_getAttributeValue(iedServer, temperatureValue)
        print("updated value: ", pli.MmsValue_toFloat(ret))

        ret = pli.IedServer_getUTCTimeAttributeValue(iedServer,
temperatureTimestamp)
        print("updated time: ", datetime.fromtimestamp(ret/1000))

        pli.IedServer_unlockDataModel(iedServer)

        v += 0.1

        time.sleep(0.1)

    pli.IedServer_stop(iedServer)
    pli.IedServer_destroy(iedServer)


if __name__ == "__main__":
    main()
```