Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science with Specialization in Embedded Systems 30.0 credits

DIAGNOSTICS FRAMEWORK FOR TIME-CRITICAL CONTROL SYSTEMS IN CLOUD-FOG AUTOMATION

Johannes Deivard
jdd14001@student.mdu.se

Valentin Johansson
vjn20002@student.mdu.se

Examiner: Hossein Fotouhi
Mälardalen University, Västerås, Sweden

Supervisor(s): Ning Xiong
Mälardalen University, Västerås, Sweden

Company Supervisor(s): Pang Zhibo
ABB CRC, Västerås, Sweden

2022-06-09

Abstract

Evolving technology in wireless telecommunication, such as 5G, provides opportunities to utilize wireless communication more in an industrial setting where reliability and predictability are of great concern. More capable Industrial Internet of Things devices (IIoT) are, indeed, a catalyst for Industry 4.0. Still, before the IIoT devices can be deemed capable enough, a method to evaluate the IIoT systems unobtrusively—so that the evaluation does not affect the performance of the systems—must be established. This thesis aims to answer how the performance of a distributed control system can be unobtrusively evaluated, and also determine what the state-of-the-art is in latency measurements in distributed control systems. To answer the question, a novel diagnostics method for time-critical control systems in cloud-fog automation is proposed and extensively evaluated on real-life testbeds that use 5G, WiFi 6, and Ethernet in an edge-computing topology with real control systems. The feasibility of the proposed method was verified by experiments conducted with a diagnostics framework prototype developed in this thesis. In the proposed diagnostics framework, the controller application is monitored by a computing probe based on an extended Berkeley Packet Filter program. Network communication between the controller and control target is evaluated with a multi-channel Ethernet probe and custom-made software that computes several metrics related to the performance of the distributed system. The data from the unobtrusive probes are sent to a time-series database that is used for further analysis and real-time visualization in a graphical interface created with Grafana. The proposed diagnostics method together with the developed prototype can be used as a research infrastructure for future evaluations of distributed control systems.

Contents

List of Figures

List of Tables

Acronyms

ADC analog to digital converter

CFA Cloud-Fog Automation

CRC Corporate Research Center

DAC digital to analog converter

DPDK Data Plane Development Kit

E2E end-to-end

eBPF extended Berkeley Packet Filter

FaaS function as a service

FPGA Field-programmable gate array

IE Industrial Ethernet

IIoT Industrial Internet of Things

IoT Internet of Things

KPI Key Performance Indicators

MDA monitoring and data analytics

MIMO multiple-input multiple-output

NFVO Network Function Virtualisation Orchestrator

PaaS platform as a Service

PLC Programmable Logic Controller

PLP Packet Loss Problem

RTT round-trip time

SDN Software Defined Network

SoC system-on-a-chip

ULT unobtrusive latency tester

VM Virtual Machine

WIM wide area network infrastructure manager

1. Introduction

In the autonomous industry, intelligent agents need to perform their tasks with high reliability and do so in a predictable manner. One characteristic that well designed real-time systems have in common is predictability. Without predictability, the performance and safety of a system can be hard to guarantee. As a result, the end product can be unattractive and even dangerous to the end-user. Systems behaving predictably is not only beneficial for real-time systems. In fact, it is an attractive characteristic of any system, including automation systems, especially those that are critical for the revenue or safety of a company.

With the rise of 5G telecommunication, a reliable, high speed, ultra-low latency wireless communication technology, many new opportunities have appeared. A domain in which 5G technology may be especially fruitful is in the Industrial Internet of Things (IIoT) domain, more specifically in the control loop systems that are commonly present in IIoT. The higher speed, reliability and lower latency that comes with 5G has peaked the interest of adopting this new technology in the industry. The possible benefits of adopting 5G are many: Systems using a lot of cables could replace many of the cables with wireless technology, not only making them lighter and smaller but also cheaper and easier to produce. Computational capabilities are increased when the computations can be offloaded, with small or no negative effect on the performance, to a larger computer, either in the cloud, edge, or fog. Further, more complex systems can be built due to the increased availability, flexibility, reliability, and speed of communication in a distributed environment. To build said complex systems, however, the performance of the systems must be evaluated in some way. Stress testing and evaluating software during development is a well-established practice in professional software development. The same goes for the evaluation of networks. When it comes to evaluating software where the source code isn't available to instrument, i.e., in black-box software, the task gets harder. Similarly, incorporating detailed evaluations of networks—consisting of proprietary network devices and communication technology—with black-box software applications is an even more challenging task.

Early work in monitoring of physical hosts and virtual machines in a distributed environment [1], [2] used open-source tools that did not, at the time, support the resolution needed to monitor performance at container level. However, some leading commercial tools circumvent this shortcoming by instrumenting the application code [3], [4]. The approach using code instrumenting is also used by the open-source community [5]. When instrumenting code, the monitoring can no longer be considered black-box since access to the source code of the application is needed and that can be problematic in some cases, e.g., when the source code for the application is unavailable, or when the monitoring needs to be unobtrusive. However, there are solutions that opt for black-box approaches for monitoring [6]–[8], but they lack the level of detail needed to do low-level application performance monitoring, such as measuring the time it takes for a control loop to execute or identifying cycle times said control loop operates on.

In this thesis work, the task was to come up with some way to perform an unobtrusive online evaluation of distributed control loops to create a foundation for further development of distributed systems, especially systems using new wireless communication technologies, such as 5G. The two main components that need to be covered by the evaluation framework are the controller application and the network used to communicate with the control target. Further, it is beneficial if the evaluation framework provides some sort of metric related to the control target, but seeing as a control target can be almost any device that can receive input, it is almost impossible to create a generic enough method to perform a detailed evaluation on such a wide range of devices. Similarly, the number of combinations of controller applications, operating systems, network technologies and protocols, topologies, and control targets used in the industry is too many to investigate at a time. The scope of this thesis work is, therefore, limited to a subset of building blocks that should be supported by the proposed evaluation method. The building blocks consist of two different cloud-fog automation controllers, a virtualized controller engine called CODESYS, OS virtualization with KVM, and two different communication protocols, namely Modbus TCP and Profinet. Further, the virtualized controller engine will run on Linux, which consequently will be the virtualized OS, since the support for detailed process evaluation is generally better supported on Linux than it is on Windows.

There are two questions that are answered in this thesis work, both of which are formulated in

Section 1.1 below. The method used to answer these research questions is a system development research method proposed by Nunamaker and Chen [9]. First, in the pre-study phase, the previous knowledge from the team at ABB CRC was gathered, and related work in similar fields were found in a literature study. In the following exploration phase, an unobtrusive evaluation framework was iteratively developed with the systems development research method. Eventually, a working prototype of the framework took form, and experiments were conducted to evaluate the framework and its feasibility. The results of the experiments proved the feasibility of the framework, which was essential to answering the research questions. The selected method is appropriate for this kind of research since, in order to answer the research questions with confidence, a working prototype must be developed. The prototype is used to prove the feasibility of the approach and is a valuable contribution that can be used in further research and development of IIoT. Further, designing such a prototype is not straightforward since this type of evaluation tool is still very novel. Several approaches need to be considered before selecting one that seems feasible, and several iterations of that seemingly feasible approach might be needed before a viable implementation is done.

1.1 Problem formulation

This thesis work is a part of a bigger project, where the goal is to create a common testbed for Cloud-Fog Automation (CFA). The testbed should, among other things, support various application virtualization tools, e.g., orchestration frameworks and containers, a communication virtualization layer (e.g., messaging protocols and middleware), virtualized controller engines, such as CODESYS PLC and OpenPLC, different CFA controllers, and varying operating systems. To evaluate different technologies and algorithms used in the testbed, a diagnostics framework must be developed. The goal of the diagnostics framework is to perform unobtrusive live measurements of latencies in network communication and edge computing for time-critical applications. The live measurements should be visualized in a graphical interface so that the performance can be analyzed visually, e.g., by visually spotting bottlenecks and deviations. The measurements must also be persistent so that further analysis is possible. The task of unobtrusive live monitoring of the performance in a distributed system, e.g., a closed-loop control or robotics, is not something that is trivial and to the best of the authors' knowledge, is not something that has been done before. Solving this problem will provide a novel diagnostics method that can be used as a research infrastructure for further activities in the area of CFA.

In Table 1, the CFA building blocks that will be the initial focus for the development of the diagnostics framework are listed. In reality, the CFA testbed will host many more building blocks than what is listed in Table 1, but for an initial diagnostics framework prototype, the scope must be limited.

The problems and tasks described above can be redefined into two research questions that are central to this thesis work. The proposed research questions that are to be answered are as follows:

- **RQ1:** What is the state-of-the-art related to measurement of latencies in distributed control systems?
- **RQ2:** How can the performance of a distributed control system, consisting of the defined building blocks, be unobtrusively evaluated live?

Layer	**Building blocks**
CFA Controller	Open-Loop Motion Control in IEC61131
	PID-based Motion Control in IEC61131
Virtualized Controller Engine	CODESYS PLC
OS Virtualization	KVM
OS	Linux
Network protocols	Modbus TCP
	Profinet

Table 1: Testbed building blocks that are focused on in this thesis work.

1.2 Contributions

The result of this thesis work is a proposed diagnostic method that can be used to evaluate and identify deviations in both the network and the computing done by the controller application in a distributed control loop. In the proposed method, diagnostics of controller applications is achieved by a computing probe that is running in parallel to the controller application. The computing probe calculates the sleep and wake cycles of the controller application by utilizing system call tracepoints via an eBPF program. By keeping track of the sleep and wake cycles of the controller application, deviations can be identified, and by doing so, further deviations in the network and control target performance can, not only be anticipated but also explained and acted upon. The network evaluation method in the proposed diagnostics framework is based on previous thesis work that utilizes an industrial Ethernet probe, ET2000, and custom software that processes the data produced by the probe [10]. The network evaluation method used in the previous thesis work was proven by several experiments to be a feasible approach, thus making it a feasible method to use as a basis for further development in the diagnostics framework proposed in this thesis. Several metrics are captured by the network evaluation tool in the proposed framework, the most important being network latency in both directions, and controller application send intervals. Further, the prototype of the diagnostics framework uses InfluxDB as a database to store the data, and Grafana as a visualization tool so that evaluation sessions can be monitored in real-time. The prototype and the individual components in the prototype, such as the database, computing probe, and network sniffer, were evaluated in several experiments that each further proved the feasibility of the approach, both as individual components and in combinations as a diagnostics framework.

The contributions of this thesis work are useful for anyone that wishes to evaluate the performance of a distributed control system, e.g., by researchers at ABB CRC. In fact, part of the diagnostics framework is already used by other students in their thesis work at ABB CRC. The prototype is a strong foundation for further development that is needed before the framework can be used in a commercial tool. Currently, only a few network protocols and architectures have been proven to work, such as Profinet and Modbus TCP, and CODESYS on a Linux VM. Although the diagnostics framework has proved to work in the aforementioned settings it would benefit highly from further development and experiments, since there are many other protocols and architectures used in the industry.

1.3 Outline of the report

This thesis is written for researchers and students in the computer science field. Despite the fact that work done in this thesis is heavily focused on networking and low-level process diagnostics, not much prior knowledge in said fields are required to understand the content and the significance of the contributions. The report is structured as follows: Chapter 1 is the introduction to the thesis work, where the problem is formulated and the research questions to be answered are presented. Following the introduction is the background chapter that provides some background knowledge that is beneficial for the reader. Next, chapter 3 presents the work that is related to the work done in the thesis, e.g., cloud, fog, and edge computing, and performance evaluation of said computing paradigms. Chapter 4 presents the method used to answer the research questions and chapter 5 gives a short reflection on the ethical and societal considerations. Next is chapter 6, which presents the main contribution and work of this thesis: The Cloud-Fog Automation (CFA) Diagnostics framework. Following is chapter 7, which introduces the reader to the experiments, and their settings, that were performed to evaluate the proposed method. Naturally, the results of the experiments will be presented in the next chapter—chapter 8. The results of the experiments and the contribution of this thesis will be discussed in chapter 9, and finally, chapter 10 will conclude the thesis together with chapter 11, were future work will be proposed.

2. Background

In this chapter, core knowledge that is required to understand this thesis work will be presented. The reader will be introduced to the different concepts that are central to this work, such as extended Berkeley Packet Filter (eBPF), Software PLC, industrial communication protocols, network probing, and more.

2.1 Extended Berkeley Packet Filter

When implementing networking, security or diagnostics applications that need to be fast and have low overhead it is beneficial to have them operate in the kernel space so that the step where information propagate to the user space can be omitted and the application have access to the resources and events inside the kernel. Implementing kernel modules, however, can be very challenging due to the complex infrastructures that the programmer have to navigate and debug. By having a sandboxed Virtual Machine (VM) run inside the kernel with limited access to the kernel's resources, applications can instead be developed for this VM, which will result in lesser complexity and more robust applications. The use of a VM that runs inside the kernel also eliminates the need to change the kernel code when writing software the leverages kernel capabilities.

A powerful technology that uses the aforementioned approach—a VM running inside the kernel—is extended Berkeley Packet Filter (eBPF). eBPF programs executes BPF bytecode that is verified before it runs to ensure that the programs does not have any loops that compromise the kernel, by e.g., making the application's execution time too long. The eBPF programs are event-driven and only trigger when a specific trigger is reached. These triggers are called hooks and they exist in many places in the kernel, they can for example be found in network events, system calls, kernel tracepoints and function entries. Typical use cases for eBPF applications include tracing of user space applications, and network filtering, which was the primary use case when eBPF was first developed.

The learning curve to use eBPF directly to write BPF programs is rather steep. Even though using eBPF directly gives the developer the highest amount of control, it is far more common to use eBPF indirectly via various eBPF projects. A popular eBPF project that is focusing on networking, security or observability is Cilium [11] that is, among others, used by Google, GitLab and Amazon Web Services. Another popular project that aims to be a high level tracing language based on eBPF is bpftrace [12]. The bpftrace project compiles scripts to eBPF bytecode that can interact with the Linux eBPF subsystem and also access the tracing capabilities and attachments points found in Linux.

When it comes to kernels that support eBPF, the Linux kernel is, by far, the most supporting platform. The number of different hooks in the Linux kernel available via eBPF is large and the types of hook are varied. The case for Windows is another story. Even though Windows has realised the usefulness of eBPF and has an ongoing project [13] that aims to bring CFA capabilities to Windows, the amount of supported hooks on Windows is, at the time of writing, very limited. Further, the complexity of creating and compiling CFA programs in a Windows environment is also much higher than it is in Linux, where various open-source projects that simplify development can be used instead of using CFA directly.

2.2 Software Programmable Logic Controller

Programmable Logic Controllers (PLCs) originates in the automobile manufacturing industry where it provided a more flexible and easily programmable solution than the traditional hard-wired relay logic systems. Due to the harsh conditions in a manufacturing environments, PLCs are made as ruggedized computers, which essentially means they are designed and built to operate in harsh environments.

In the early years of PLCs, different vendors used different programming languages and structures for the programming of PLCs, causing confusion and unnecessary learning time when a switch between PLC vendors occur. Nowadays, however, standards for PLC programming languages have been put in place, which almost all PLC vendors comply with. The most well-adopted standard for PLC programming worldwide is IEC-61131, which was first published in 1993.

Software PLCs are a technology designed to turn an embedded computer into a fully functional PLC. Due to the advantages that software PLCs have over traditional PLCs, the industry is moving towards heavier usage of software PLCs. Software PLCs with their low maintenance and installation costs are more cost-effective than traditional PLCs. They are also more flexible and easier to use than traditional PLCs. Further, they also have support for more programming languages and are generally quicker and easier to install and update. When it comes to security, which is an important aspect of Industry 4.0, software PLCs also have an edge over traditional PLCs.

2.3 Industrial Communication Protocols

Communication has been essential for upholding an efficient and robust network link between distributed devices in industrial automation. This comes down to having a network infrastructure that can handle real-time control, data integrity, and withstanding harsh environments. As a result, the hierarchy of industrial communication networks usually consists of three different levels, as shown in figure 1. Having separated levels aims to efficiently structure the various distributed systems, controllers, drives, sensors, actuators, and processes in an industrial environment. Furthermore, this allows for using different network topologies based on each level's throughput, reliability, security, or availability requirements. One of the more widespread industrial environments for communication is the Industrial Ethernet (IE), which can provide deterministic communication and real-time control. Some of the protocols used within IE are PROFINET, Modbus TCP, EtherCAT and EtherNet/IP. For the scope of this thesis, the PROFINET and Modbus TCP protocols are covered in section 2.3.1 and 2.3.2.

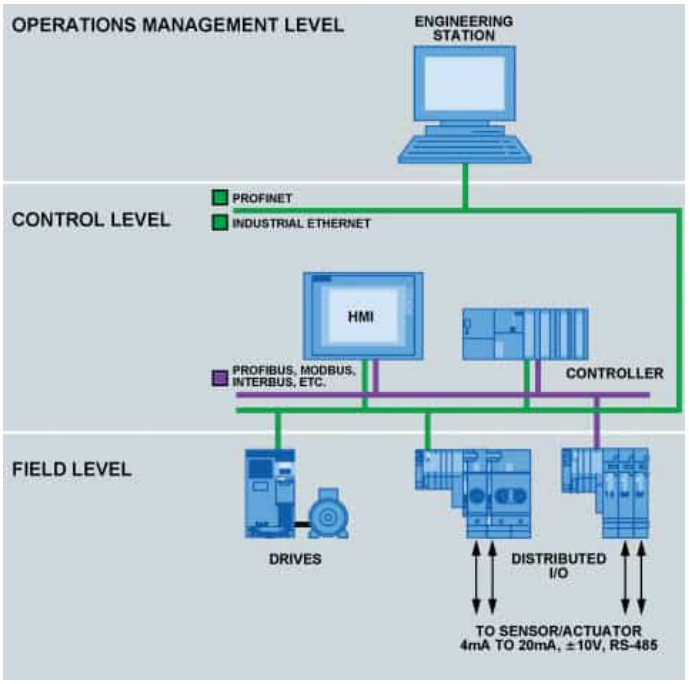

Figure 1: Overview of the hierarchical structure of industrial communication networks. The field level consists of various drives and distributed sensors, actuators, and machine processes that send information between devices, such as PLCs. The control level consists of controllers such as PLCs, distributed control units or computer systems that control the distributed I/O and drives. Finally, the operations management level consists of engineering stations that gather data from the lower levels to visualise various aspects of the plant. [14]

2.3.1 PROFINET

Profinet is an industrial automation protocol that is based on traditional Ethernet technology. It is used at the field level of industrial automation to support real-time communication. RTE protocols

can be divided into three types of classes to accommodate the requirements of various real-time communication layers. Class 1 handles real-time communication above the transport layer, with an achievable cycle time of around 100ms. This class is mainly used for low priority real-time data. Class 2 handles real-time communication above the MAC layer with an achievable cycle time of around 10ms. Soft real-time data is primarily used in this class. Lastly, class 3 handles real-time communication in the MAC layer, which has an achievable cycle time below 1ms. This class handles hard real-time data with strict time demands. Profinet supports all three classes, making it an optimal communication protocol for a broader range of industrial systems.

2.3.2 Modbus TCP

Modbus TCP is, similarly to Profinet, based on the traditional Ethernet technology. The protocol was initially published in 1979 by Modicon for use with PLCs, but has become de facto standard communication protocol in industrial automation. Today Modbus TCP is commonly available for connecting various industrial devices [15]. The benefits of the protocol are its resistance to noise and various disruptions due to the data transactions being stateless and its ability to hold a large number of concurrent connections active. The data transactions also require a small amount of information to remain intact on either end in case of disruptions in the network, making it a robust option [16].

2.4 Containerisation

One of the struggles with traditional coding has been the lack of portability in transferring code to various computing environments without causing bugs and errors. In addition, there are various setup costs involved when wanting to move a system between platforms in several cases. This can result in unnecessary time spent on resolving missing libraries and dependencies. A solution to this is containerisation [17], which is a way to package software code with the required libraries and dependencies that can execute on any infrastructure, also called a container. A runtime engine is used to facilitate the containers, which acts as a platform where the operating system can be shared among the containers. Today, one of the industry standards for containers is the open-source Docker Engine [18]. Containers can be viewed as similar to virtual machines, which share several similarities. However, one of the key differences is that containers run isolated from the host operating system. Comparing the architecture of the two, shown in figure 2, show that virtual machines run on an operating system each. Containers, however, share the kernel of one operating system. Due to these differences, containers are often more lightweight applications that consume fewer resources and can be deployed in more significant numbers on a single operating system.

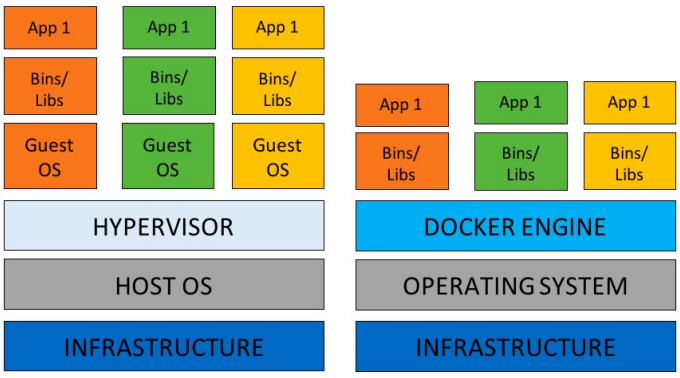

Figure 2: Comparison of the virtual machine and container architecture. [19]

2.5 Network probe

By probing a network, information about the network, and its traffic, that is not readily available to an observer can be extracted. A network probe is basically a messenger that monitors network traffic in some way and then reports the information to someplace else, e.g., a diagnostics framework

or monitoring application. Some probes can be obtrusive, depending on the implementation, but an ideal network probe is invisible to the devices in the observed network and does not affect the network's performance in any way. The unobtrusive probes often require dedicated hardware that can forward the network packets quickly and still compute additional information about the packets, such as metadata in the form of time of arrival timestamps. One such network probe that uses dedicated hardware is the ET2000, which is an industrial Ethernet multi-channel probe by Beckhoff that introduce next to no influence on the probed network. The location of the probe that is inserted into the network should be somewhere where the desired information can be collected from the traffic that passes through the probe junction.

Sometimes network probing is also defined as the act of probing a network for vulnerabilities. The act of probing a network for vulnerabilities usually includes having a probe scanning ports to identify open ports on devices in the network or assess well-known vulnerabilities. These types of probes are more active than the passive network probes mentioned in the previous paragraph, which only observe a network. Since they increase the network activity by sending messages and requests to devices on the network they should not be considered unobtrusive. In this thesis, the first-mentioned definition of network probing is used, meaning that when a network probe is mentioned, it does not refer to active probes that asses network vulnerabilities, but instead it refers to passive probes that collect information about the traffic as unobtrusively as possible.

2.6 Cloud-fog automation

In Industry 4.0, various systems must be efficient and operate with high precision. The most common requirements in the area are low latency, high reliability, and viable communication solutions [20]. However, with industrial equipment being more connected with the cloud, security is becoming increasingly important to avoid downtime, data breaches, or manipulation of systems [21]. Cloud-Fog Automation (CFA) aims to solve two issues: offload a large portion of the computation to the cloud and secure the connected industrial equipment. Depicted in figure 3 is the system design architecture of the CFA. Some of the main advantages are that it can accommodate more advanced algorithms and machine learning to be implemented, offering more cost-effective and robust solutions. In addition, having the heavy computational parts in the cloud and fog allows for more flexible load distribution. This enables the adaptation to various tasks and environments more dynamically. [20]

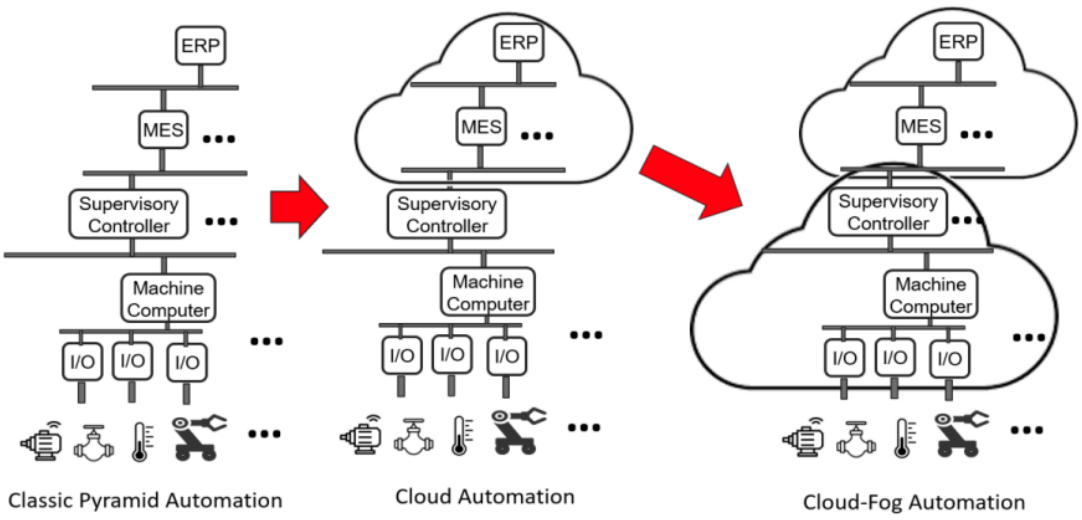

Figure 3: Overview of transition from classical automation to Cloud-Fog automation system design. [20]

3. Related work

Several essential aspects exist within 5G-based communication and edge computing for critical motion control systems. However, due to the 5G technology being relatively immature compared to its predecessors regarding researched capabilities, performance and industry evaluation, there is various ongoing work in this field. In the following sections the work related to this thesis will be presented.

3.1 5G in Cloud, Fog and Edge computing

The authors of [22] discusses the use of edge cloud computing that can accommodate the needs of systems with hard constraints regarding time and missions. The article proposes an edge cloud research test-bed that takes advantage of 5G, a platform as a Service (PaaS) framework, a distributed set of compute nodes, and a mission-critical and time-sensitive process under control. The results are a test-bed that can reliably distribute process resources with low latency over the edge cloud.

In [23], the author explores the potential and uncertainty of cloud based critical control systems. To this end, the author created two 5G-enabled testbeds, one with the controller adapted for a PaaS and another with a controller adapted for function as a service (FaaS). The testbed with the PaaS adapted controller, which could be dynamically relocated while controlling the plant, showed that this type of platform was only viable for the reference system when the controller were located in an edge-cloud with state-of-the-art wireless communication using LuMaMi, a massive multiple-input multiple-output (MIMO) research testbed [24], due to networking and computation delays in the other configurations. Results from the FaaS compatible testbed that focused on measuring response times in the cloud showed that this design also could be used to control the reference plant. The results from this testbed also showed that there is a significant overhead added by using cloud native services and the impact that distance have on the latency.

At ABB Corporate Research Center (CRC), there is numerous ongoing research in the 5G-based communication domain. Previous work has mainly investigated the communication Key Performance Indicatorss (KPIs) (such as round-trip time (RTT), availability, and reliability) of the entire control loop in motion control systems. A previous master student performed experiments to retrieve communication KPIs in an open-loop motion control system with wireless telecommunication technologies. The resulting KPIs were evaluated and compared with a baseline experiment where a wired connection was used. The results from the 5G-based experiments showed that the 5G-based communication is very fast when the reliability requirement is at 99.9%. However, the latency bound increases as the reliability requirement get more strict.

Another recent study conducted at ABB CRC investigated the end-to-end (E2E) latency in a 5G and edge-based control system. The study firstly investigated the E2E latency of three network architectures: local control, wired Ethernet control and wireless 5G control. In a latter part, a Ball and Beam control system was used as a time-sensitive and mission-critical process to propose conservative tuning approaches. The system measurement focused on analog to digital converter (ADC) and digital to analog converter (DAC) conversion, execution time, and communication latency. The KPIs were static error, rise time and settling time, calculated with respect to the average step response. An oscilloscope was used as a measurement tool in both of these parts. Results indicate that existing system bottlenecks most likely occur in the 5G network. To mitigate this, various tests were conducted where the control system took those latencies into account using the conservative tuning approach, which showed promising results. Finally, a proposal to shift from hardware to software-based solutions could be beneficial to improve the latency measurement. The reasoning is that software-based solutions could measure the latency at each sample and have more flexible usage.

3.2 Performance evaluation of Cloud, Fog and Edge computing

In [25], the authors discuss the importance of optimization for Internet of Things (IoT), Cloud, Fog and Edge computing and present novel solutions to achieve better evaluations. One of the crucial parts addressed is the shift from only evaluating latency to including factors such as reliability,

energy, cost, and security. Another aspect brought up is the importance of choosing the right metrics for assessing the performance of a system, which, if not considered carefully, could lead to misleading results and issues later on. The article presents various metrics in groups as more suitable when evaluating IoT, Cloud, Fog or Edge computing. Even though fog and edge computing is frequently used interchangeably, the authors of [25] decided to distinguish them from each other to give a more detailed review of the paradigms. The authors define fog as physical servers closer to the IoT devices than the servers in the cloud, but still have the same characteristics as a cloud environment, and edge as servers much closer to the IoT devices, mainly inheriting the characteristics of IoT devices. In the work presented in this thesis, however, fog and edge are indistinguishable from one another and can be used interchangeably.

When it comes to evaluating the performance of Cloud, Fog and Edge computing, there exist several solutions to accommodate this need. Today, numerous articles cover both hardware-based solutions [26]–[28] and software-based solutions [29]–[34].

3.2.1 Hardware-based performance evaluation

Hardware-based solutions have often been promoted as good alternatives when measuring high-speed networks, especially over 10 GB/s. The authors of [26] bring up the advantages of using hardware-based solutions, such as Field-programmable gate arrays (FPGAs) with system-on-a-chip (SoC) devices. The use cases of such devices are plenty. However, the article mainly focuses on testing the capabilities of network equipment and performing distributed measurements along a network path. A packet-train technique is proposed as a favourable option to measure the network, mainly due to its accuracy and robustness. The findings show that hardware-based solutions can achieve higher accuracy and throughput at higher speeds than software-based solutions that experience more jitter and worse results.

In another article [27], the authors demonstrate how an active measurement can be performed on a 100GB/s optical path in a Software Defined Network (SDN). The measurement technique can provide valuable data such as packet loss, capacity and delays. An FPGA is used as an active probe in the hardware to measure the network, which can achieve precision in nanoseconds. A multi-layer network in an office space was used to demonstrate the active measurement. The architecture consisted of a Network Function Virtualisation Orchestrator (NFVO), a wide area network infrastructure manager (WIM) and a monitoring and data analytics (MDA) controller. By injecting packet-trains on the network, the authors can measure the round-trip delay, jitter, packet loss, and throughput.

In a previous thesis work conducted at ABB CRC [10], the author investigates the integration of various network topologies, such as 5G and WiFi 6, on mission-critical motion control applications. It also proposes a systematic methodology to evaluate various network topologies in industrial applications using an unobtrusive layer 3 tester (UL3T) tool. The UL3T tool consists of an Ethernet probe, ET2000, that mirrors the network traffic unobtrusively, which is then fed to customized software that calculates the latency and stores it local text files. Another customized application is later used to calculate and visualize various KPIs based on the latency data stored in the text files. Several experiments were conducted during the experiment using 5G, WiFi 6, and a newly proposed hybrid topology using 5G and WiFi 6 together called FRER. The results showed that the hybrid topology performed better than the other tested network topologies. It was also concluded that the proposed UL3T tool was valuable for measuring and analysing network communication. The monitoring tool that was developed during the thesis [10] proved to be useful, but it had some limitations. It was not possible to do live evaluation of distributed control systems since the metrics are computed and presented after a monitoring session has ended. It was also limited in only monitoring some network KPIs, which is insufficient when an entire distributed control system is to be evaluated. Some metric more closely related to the computing part of the controller application is needed to determine cause and effect relationships for the performance of a distributed control system.

3.2.2 Software-based performance evaluation

More research has been conducted on various techniques to measure performance using software-based solutions. Today there exist numerous tools to measure performance and collect diagnostic

metrics on different layers. Some of the more common existing solutions are strace [35], DTrace [36], Sysdig [7], and extended Berkeley Packet Filter (eBPF) [37].

In [29] the authors present a performance-aware load shedding framework to lower the CPU overhead when measuring performance in container-based environments under heavier load. The proposed methodology is used on the monitoring tool Sysdig [7] together with a load manager that orchestrates the policies of the load shedding.

The authors in [30] propose a generalised black-box monitoring approach that can measure the performance of microservices in container environments in the cloud. One of the critical goals of the black-box approach is to avoid user intervention when setting up the monitor application onto microservices that are prone to evaluation. The article measures performance in three categories: low-level performance, application performance, and network performance. Measuring within these categories is to cover performance from high-level applications down to low-level applications. The proposed approach uses two data aggregation layers. The first layer uses an eBPF to generate performance metrics at the kernel level. The second layer collects data on the cluster-level by utilising a REST collector, which forwards the data to various endpoints to visualise the collected data. Results show that the proposed approach can achieve good results with respect to the initial goal of measuring performance with minimal overhead and not adding intrusive code.

4. Method

The methodology adopted throughout the thesis work was the system development research method derived from Nunamaker and Chen [9]. The method follows an iterative approach consisting of four research strategies depicted in figure 4.

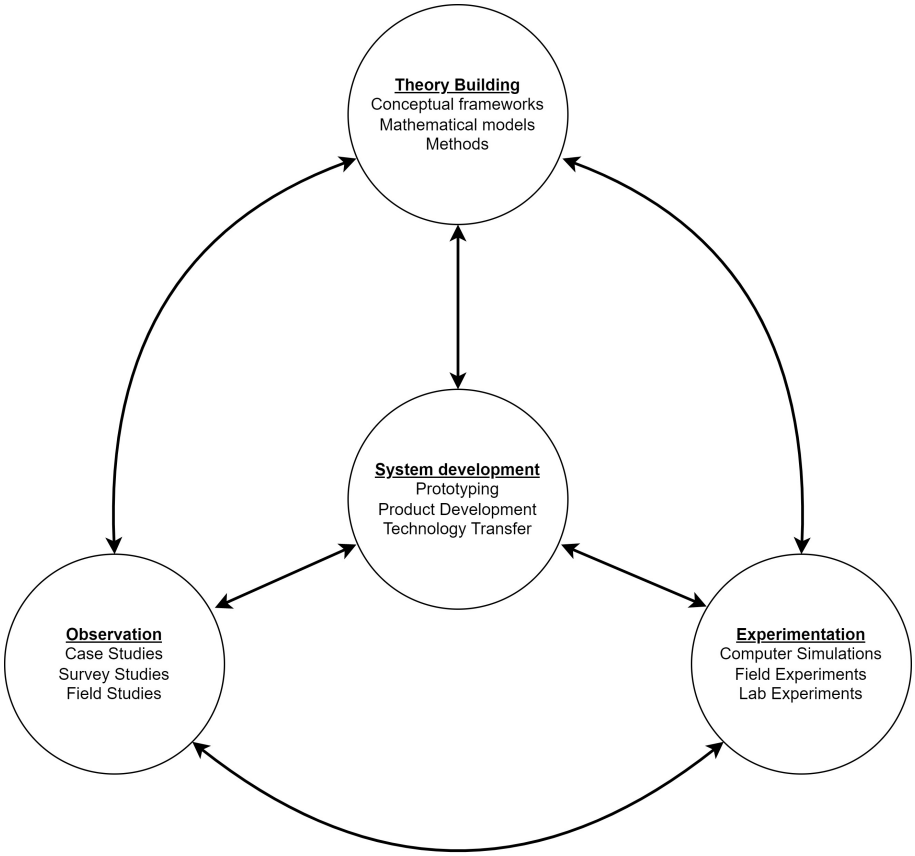

Figure 4: A multi methodological approach to information systems research [9].

The iterative process used consists of five stages, depicted in figure 5, which is the way the thesis work has been conducted in regards to system development. The process follows an iterative approach, making it possible to refine the work through iterations. Furthermore, being able to move back and forth between various stages allows for alternative ideas and concepts to be developed if previous choices do not meet specified requirements or needs.

In the first stage, a conceptual framework was constructed through a pre-study phase to identify suitable approaches for the thesis and limitations. This involved studying relevant research and investigating the system's functionalities and requirements. After a conceptual framework had been derived the system architecture that defines functionalities and interrelationships among system components was developed. The system was analysed and designed based on the requirements of the architecture. This stage also allowed alternative solutions to be designed, where one or several solutions could be chosen. A prototype was built based on the system design that could be tested and verified against the requirements. In addition, the prototype could provide a better understanding of the advantages and disadvantages of the chosen design, which can be of use if a redesign of the system was needed in the future. Lastly, an evaluation was conducted on the prototype to test the performance and other valuable metrics defined in the requirement from the earlier phases. The results from these evaluations were interpreted based on the conceptual framework, requirements, and prototypes used in previous work.

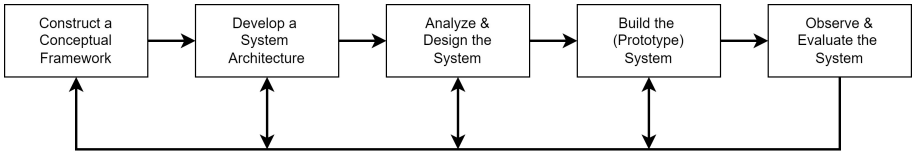

Figure 5: Process for the system development research [9].

5. Ethical and Societal Considerations

Since this thesis is purely technical work that does not contain any experiments where other people are directly involved, no ethical considerations are required. The results from this thesis are also purely technical and do not affect any individual directly, thus not requiring extra ethical considerations in this aspect either.

When it comes to societal considerations, however, there are some things to consider. The results from this work are directly contributing to the increased usage of wireless technologies in the industry. Increased wireless usage replaces the traditional heavy wiring that may be present normally in various equipment and systems, reducing the monetary and environmental impact of the products. Further, since wireless communication can enable more complex and capable products to be developed, the performance of the products can be increased and they can accomplish more complex tasks. That, in turn, will benefit society by, e.g., making products cheaper to produce, safer to use, advancing technology so that previously unsolvable tasks are now solvable, or making the environmental footprint smaller.

It is also worth mentioning that the provider of the 5G network that was used in some experiments is anonymous since, in its current state, is proprietary and under development. Further, no graphs created from tests on the 5G network are shown since characteristics of network performance can be reverse-engineered and reveal implementations that are secret. Leaking such graphs can not only hurt the 5G provider and give its competitors a technical edge over them but also hurt the reputation of the 5G provider if the graphs are misrepresented or misinterpreted.

6. Cloud-Fog Automation Diagnostics

The main contribution of this thesis work is the Cloud-Fog Automation (CFA) Diagnostics framework. The framework consist of two unobtrusive measuring tools that produce several metrics that can be visualized in a graphical interface. The measuring tools are focused on breaking down latencies in a distributed control loop and providing KPIs that aims to support the process of identifying and widening bottlenecks. One of the tools are based on network data, coming from dedicated sniffer hardware, and the other tool is monitoring the controller application, running as a separate process tracking system calls made by the controller application process. The framework was created with flexibility in mind, meaning that if a feasible solution was not generic enough, it would be discarded. The control target was treated as a black box, even though in some cases it isn't, in order to facilitate a more generic framework that could be used for many different systems and topologies, instead of singling out a single system and creating a tailored framework or tool for that specific system. The source code of the diagnostics framework prototype was developed at ABB CRC and since there are plans to evolve this prototype into a real product, the source code used in this thesis is proprietary and will not be shared to the public at this point in time.

In the following sections, the different parts of the CFA Diagnostics framework will be presented, and the steps taken that lead to the creation of the individual parts will be explained. First, the unobtrusive latency tester (ULT) for the network will be presented, along with possible alternative approaches. Following ULT Network, the ULT Computing tool will be presented, with the early prototype that led to the final implementation. Next, the time-series database that gives the readings from the tools persistence and collects all readings in a structured way so that they can be further used, e.g., by an online visualization tool that happens to be the final part of the CFA Diagnostic framework that will be presented and discussed in this chapter.

6.1 ULT Network

In a diagnostics framework that aims to break down latencies in a distributed control loop, it makes sense to treat the network as a single entity that produces latencies in two direction: downstream (to the control target), and upstream (from the control target). To capture the network latencies there are several possible approaches, some of which will be discussed in the following paragraphs.

Software on the control target and on the controller could be used to timestamp and track all outgoing and incoming network packets on both units. By combining the data produced by the two instrumented units, the network latency of the packets can be computed. This approach, however, requires the clocks of the two units to be accurately synced, which can be challenging in certain scenarios. It also requires the two units to be modifiable in such a way that the network packets can be caught and handled by the diagnostic code, e.g., an application created with Data Plane Development Kit (DPDK), such as PcapPlusPlus [38], or a BPF application that attach to network hooks in the system's kernel. If the control target is an industrial PLC, which is a likely scenario, the software approach will not be possible due to the customization constraints of PLCs. Also, when using a software based solution it has to be very fast and lightweight so no unwanted latencies are introduced in handling of the network packets on either of the devices.

Another possibility is to modify the software on the network devices that are closest to the endpoints, e.g., a router on either side of the network. The goal of the modified software is the same as in the previously discussed approach: Timestamp and track the traffic going in both directions. If the network devices nearest the endpoints—the controller and control target—are using a cabled connection, the extra delay introduced from the final hop to the endpoints can be omitted since it will typically be in the microseconds range. Although a feasible approach, it may prove challenging to modify the software of a proprietary network device, e.g., a router, since the software usually is a black box to the end-user. This, however, can be circumvented by using open-source software that is easier to modify, such as a software router using TheRouter [39], or by creating the required software and network devices from scratch.

Lastly, the final possible approach that will be discussed is the approach that is based on dedicated sniffer hardware. By utilizing a hybrid solution with dedicated hardware, much less overhead can be achieved than in a software-based or hybrid solution using proprietary network devices. The hardware in question would replace the supposed network devices near the endpoints in the

previous approach, but still have the same task: Timestamping and tracking the network traffic passing to and from the devices in the distributed control loop. There are three main advantages to using a dedicated sniffer hardware solution over a software-based or hybrid solution using proprietary network devices: (i) The overhead is expected to be lower in specialized equipment, which means that the measured latencies are closer to the actual latencies, (ii) the overhead is much more consistent and predictable since dedicated hardware is used, and (iii) specialized dedicated hardware is usually easier to integrate with custom software.

In the next section, the hard- and software architecture that was used for the network sniffer—ULT Network—will be presented.

6.1.1 Architecture

As mentioned in the previous paragraphs, there are several different possible solutions when approaching the creation of an unobtrusive network latency tester. The most attractive approach, however, is the hybrid approach that uses specialized hardware that was made for the task and then create software that accommodates the needs of the tool and the framework that it will be used in. The choice of specialized equipment is motivated by previous work that has been done in this area, namely the thesis work that was done before this thesis. In the previous student's thesis work [10], an industrial multi-channel network probe called ET2000 was used. The work done in that thesis used the device together with custom software and proved the feasibility of the approach. Therefore, in this thesis work, the same device and methodology were used in the creation of the ULT Network tool. The software that was used in previous work [10], however, was not used since more functionality was needed in the CFA Diagnostics framework and because the previous software would be hard to extend upon. The main things that ULT Network and UL3T have in common is the method used to monitor a network, i.e., using an ET2000 to sniff packets and special software running on a PC connected to the ET2000 uplink port, and the following four metrics that they both compute: Uplink and downlink network latency, dropped packets, and network reliability.

The ET2000 works by mirroring every packet that passes through it and attaching metadata, such as a timestamp and which port on the device received the message, and then sending that packet through a separate uplink port that a computer can be attached to. Overhead and timestamp granularity is not an issue when using the ET2000 device since it operates on 1 microsecond cycles and provides 1 nanosecond timestamp resolution. This means that the timing analysis of the network traffic can be precise and the effect of attaching the probe to a network will be minimal to the performance of the network.

The computer that is attached to the uplink port of the ET2000 device will receive all packets, with metadata, that pass through the device. By using a network monitor application, the attached computer can then analyze the packets that passed through the network probe. A common choice for network monitoring is Wireshark, but customized network monitor applications can easily be created with the help of various software libraries, such as Pcapy in Python, which was used in both the previous thesis work [10], and in the implementation of the ULT Network tool in this thesis work.

A typical setup using the ET2000 device, and the setup that the ULT Network tool was specifically developed to operate on is shown in Figure 6. This setup captures the timestamps as close to the end devices as possible by connecting the plant and the controller directly to the probing device via an Ethernet cable. Since ET2000 provides the capturing port ID in the meta-data that is attached to the mirrored packet, the direction and source of the network packets are always known. Packets that are sent from the controller to the plant will have port X30 in the metadata before it has been passed over the network, and port X11 after it has passed through the network. Similarly, packets that are sent from the plant to the controller will have port X10 in the metadata before it passes through the network and port X31 after the network. This means that a single packet will be sent twice via the uplink port, with the only difference between the packets being the metadata that is attached to them.

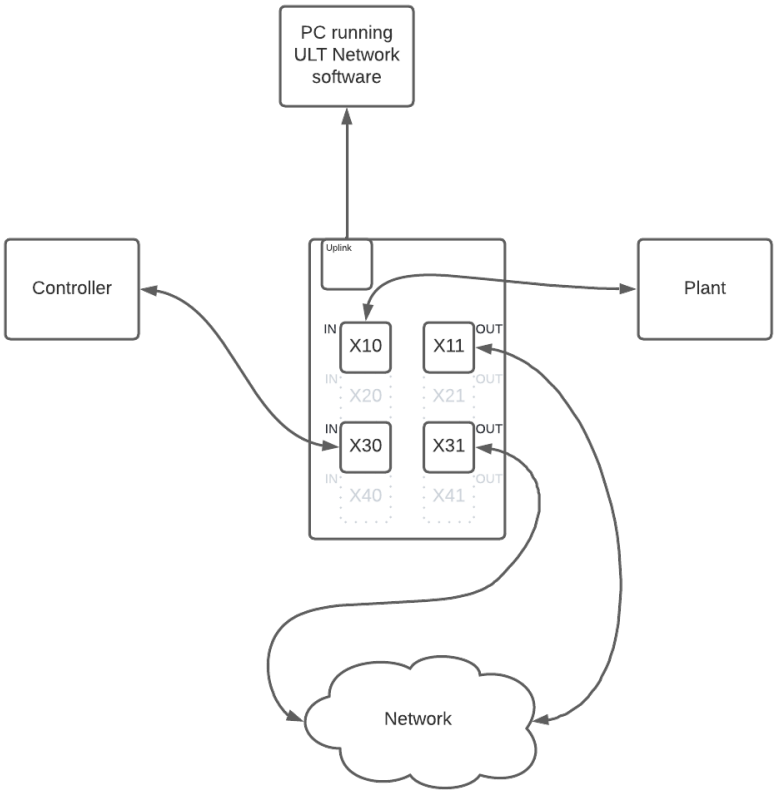

Figure 6: Illustration of a typical scenario when evaluating network communication with the ET2000. In the illustration, there are a total of 8 ports on the ET2000, as many as on the real device, only 4 of which are in use in a scenario with two end-devices, e.g., a single controller and a single plant as depicted here.

As mentioned previously, the programming language of choice when implementing the ULT Network tool was Python, and more specifically version 3.8.10. The main architectural choice when it comes to software, besides the programming language, was deciding to use Pcapy, which is a Python library that allows Python programs to access the functionality found in the PCAP packet capture library. Both Linux and Windows are supported by the library. However, WinPcap must be installed before the pcapy library can be used on Windows.

6.1.2 Implementation

The implementation of the ULT Network tool relies heavily on multiprocessing, assigning different tasks to different processes to ensure that the application can handle control loops with low cycle times that yield a high throughput of packets. By adopting a horizontal scaling philosophy in the form of more cores instead of faster cores, more metrics with higher computational cost can be added with minimal effect on the packet throughput. The application consists of four main components, which are processes that are needed regardless of the metrics that are computed, and a collection of sub-processes—one for each metric that is computed. The four main components of the application are the main process, the sniffing process, the packet processor process, and the results handler process. In the following paragraphs, the details of the components will be revealed as well as which metrics processes are currently implemented and how the metrics are computed.

The first process that will be presented is the first process that starts when running the application: The main process, which is responsible for spawning the other sub-processes and handling the setup of the analysis session, e.g., by parsing command-line arguments and prompting the user for which device to capture packets from. Many command-line arguments that change the behavior of the application are supported. For example, a user can choose between filtering on mac or IP addresses, specify the addresses for the controller and control target, select which Ethernet or IP

protocol to filter on, or give a path to a PCAP file that should be used for offline packet processing instead of live capturing packets.

Once the command-line arguments have been processed, the rest of the processes will be started. The processes are started in the opposite order of the data flow so that all processes are up and running before the sniffed packets start flowing into the application. It is more natural to present the processes in the order that data is flowing, however, so that is the order in which they will be presented below.

The packet sniffing process is responsible for sniffing the network packets and bringing them into the application. As mentioned earlier, this is done with the Pcapy framework, which in turn utilizes PCAP for capturing network packets. By applying specific filters, only the packets relevant to the diagnostics session are captured and brought into the application. The filtering is done by PCAP, which not only performs the filtering faster than a Python program would but also has a well-established and mature filter syntax [40].

Once a relevant packet has been captured by the sniffing process, it is placed in a multiprocessing queue that the packet processing process is reading from. The objective of the packet processing process is to parse the packet and identify which ET2000 port it was received on and then send it to the appropriate metrics processes via their own multiprocessing queues. The packet will be copied into each of the queues for each process that use the packet in its metric computation. This means that if a packet is used in the computation of several different metrics, the packet will be copied and placed in several different queues, yielding a larger amount of packets being processed than are actually captured by the sniffing process.

The packet processing process is also responsible for spawning the processes that compute the different metrics. Currently, there are five different metrics being computed, each metric process having its own queue and receiving its own copy of the packets needed for the metric calculation. In Table 2, the ET2000 port origin of the different packets that each metric use is shown. The table shows, e.g., that packets originating from the X10 port are copied to three different metric processes: *from_plant_network_latency*, *plant_send_interval*, and *plant_response_latency*.

	Packets used from ports			
Metric	**X10**	**X11**	**X30**	**X31**
from_plant_network_latency	X			X
to_plant_network_latency		X	X	
plant_send_interval	X			
controller_send_interval			X	
plant_response_latency	X	X		

Table 2: The five metrics that are currently implemented in the ULT Network tool and the ET2000 port that the packets used in the metric originate from. The number of X's in a column is the number of times a packet originating from that specific port will be copied, e.g., packets originating from the X10 port on the ET2000 will be copied into three different queues, read by three different metric processes.

As mentioned above, each metric process is responsible for computing a single metric. Considering a single packet flowing from the controller to the plant, i.e., the control target, the *to_plant_network_latency* metric represents the network latency of the considered packet. The single packet that is flowing from the controller to the plant will produce two sniffed packets since it passes through the ET2000 twice. For the sake of explaining the metric, the first capture, before the network, will be called packet X11, and the second packet after the network will be called packet X30. The latency is then computed by subtracting the timestamp of the X11 packet from the X30 packet. Similarly, the *from_plant_network_latency*, which represents the network latency of a packet flowing in the opposite direction—from the plant, to the controller—is computed by subtracting the timestamp of the X10 packet from the X31 packet.

The algorithm for computing the send interval metrics is also very straightforward. Considering two consecutive packets, P_1 and P_2, coming from the controller or the plant, the send interval is simply calculated by subtracting the first packet from the second packet, i.e., $P_2 - P_1$.

The *plant_response_latency* is calculated by subtracting the timestamp of the packet that is

received by the plant from the timestamp of the next packet that is sent by the plant. This metric, however, is not as reliable as the other four metrics since it is impossible to guarantee that the packet that is sent from the plant is a response to the packet it received earlier. It could very well be a status message, such as a heartbeat, to the controller, not related to the message it just received. For some scenarios and protocols, the *plant_ response_ latency* might be a reliable metric, but for others, it might be useless due to its unreliability.

For completeness, the metrics and their equations, as they were explained in the examples above, are shown in Table 3.

Metric	**Equation**
from_plant_network_latency	X31 - X10
to_plant_network_latency	X11 - X30
plant_send_interval	$X10_{n+1} - X10_n$
controller_send_interval	$X30_{n+1} - X30_n$
plant_response_latency	X10 - X11

Table 3: The five metrics that are currently implemented in the ULT Network tool and the corresponding equation that is used to compute them. The variables in the Equation column represent packets that have been captured by that specific port on the ET2000. In the equations, a simple scenario with a single packet flowing back and forth through the distributed control loop is considered. The n variable represent an arbitrary packet, and $n + 1$ represents the consecutive packet following the arbitrary packet.

Once a metric process has computed a metric, i.e., a result, it is placed in the results queue, which is read by the results handler process—the first process that is started by the main process and the last process that is left to explain. The results handler process is responsible for gathering the results from the computed metrics in the metrics processes and sending the results to the database. The results are communicated to the results handler process via a single multiprocessing queue, which all metrics processes use to send their results through. The database writes to InfluxDB are done in batches of 1000 to impede network congestion and minimize the network packet overhead.

An illustration of the relationship between the processes in the ULT Network application is shown in Figure 7. The illustration is showing which processes that are spawning other processes and what communication channels are being used between the processes. Further, the illustration is also showing the data flow between the processes.

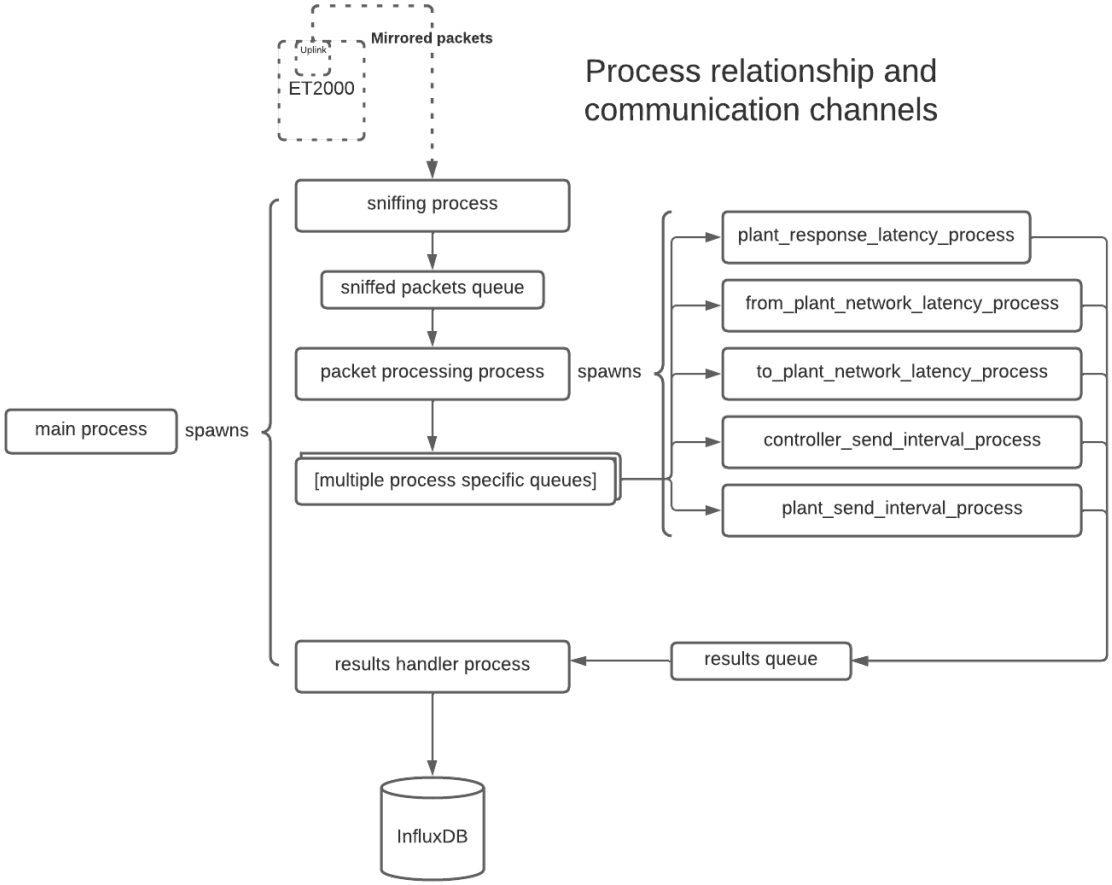

Figure 7: An illustration of the process relationships and communication channels in the ULT Network tool. The curly braces indicate the processes on the right side that are spawned by the process on the left side that is connected to the curly brace via the word "spawns". The arrows are showing the data flow, which starts by receiving sniffed packets from the connected ET2000 device and ends with metric results being sent to the InfluxDB database.

The capabilities of the ULT Network tool was explored in several experiments that are detailed in Sections 7.2 and 7.3.

6.2 ULT Computing

To give a higher level of detail in the diagnostics data, and in turn give a more favorable basis for finding bottlenecks and eventual points of failure in a control loop, a computing probe was developed to run in parallel to the controller. The goal of the computing probe is to give an insight of how the control loop in the controller application behaves without being obtrusive. The computing probe must also treat the source code of the controller application as a black box, meaning that no code instrumentation to collect and extract KPI:s is allowed. By creating a probe that treats the controller applications as black boxes, the resulting probe will be much more generic and be a viable tool for more products than it would if it relied on instrumentation of the controller source code, which is not even possible in many cases.

The following sections will present the prototypes and experiments that lead to the final prototype, i.e., the computing probe used in the diagnostics framework. Further, the architecture of the final prototype, the implementation, and the result of the computing probe will be presented.

6.2.1 Early prototype

When it comes to process diagnostics there is usually a lot of data available in various places in the operating system. The means to retrieve that data, and what data is actually accessible outside of the kernel varies between operating systems. Generally speaking, Linux is, not surprisingly, more generous when it comes to ways to access the data generated by the kernel than Windows is.

Before determining how to collect the data that will be used in the diagnostics, a decision of what data to collect must be made. The behavior of the controller and the control target ultimately depends on whether the control signals are sent and received in a predictable and expected manner—in addition to the actual algorithms governing the controller and control target. This means that the metrics of interest should be in the time domain.

If a control message is sent too late, the performance of the control target may suffer. The reasons for a control message may being sent too late can be many, but two of the arguably most common reasons are (i) the host system running out of resources, and (ii) the controller application runs into an edge case that makes the control loop execution take significantly longer time than expected. If the host system runs out of resources it may lead to the controller application not having enough resources to do its job properly. For example, if the controller application is not being awoken from its sleeping state in time, it may lead to a delayed control message since the computations to produce and send the control message is delayed. The same outcome can be expected in the aforementioned control loop edge case scenario if the computations leading to a sent control message is taking significantly longer time than it usually does.

The first prototype was based on the idea that by continuously polling a Process object that contains information such as CPU time for a process in C# or Python, the time that the process sleeps and the time it is awake can be inferred. By recording the time intervals that the controller application process does not spend on the CPU, i.e., polls to the Process object that have the same CPU time value as the preceding poll, the time spent sleeping can be estimated to the time interval that the polls had the same CPU time value. Similarly, the time spent awake can be inferred from the time intervals that the CPU time is continuously incremented. This prototype worked under the assumption that the granularity and update frequency of the process' CPU time is higher than the in the clocks used for time-stamping in C and Python.

During the theoretical investigation of the first prototype two different scenarios were considered: (i) A non-preemptive scenario where the controller application would have the highest priority and thus not be preempted during the execution of the control loop, and (ii) a preemptive scenario where there could be other processes with equal or higher priority competing for the resources that could lead to preemption during the execution of the control loop.

An illustration of the non-preemptive scenario is shown in Fig. 8. The figure is illustrating a plausible scenario of a controller application with 4 ms cycle time that has the highest process priority on the system. The figure shows that the first time interval that the controller spends on the CPU is the longest since, in this scenario, it is assumed that there are some initialization happening before the application enters the main control loop. On the following time intervals, however, the time spent on the CPU is lower and it is from these intervals that useful information could be retrieved, e.g., if there is a large enough deviation from the mean. The goal of the prototype would be to capture the behavior shown in Fig. 8, where the different sleep and wake cycles, as well as the actual cycle time could vary over time. If the prototype would be able to capture the context switches and do accurate measurements of the time that the process spends on and off the CPU, not only a visual graph similar to the illustration could be created from the data, but also deviations that would affect the rest of the distributed control loop could be identified and highlighted.

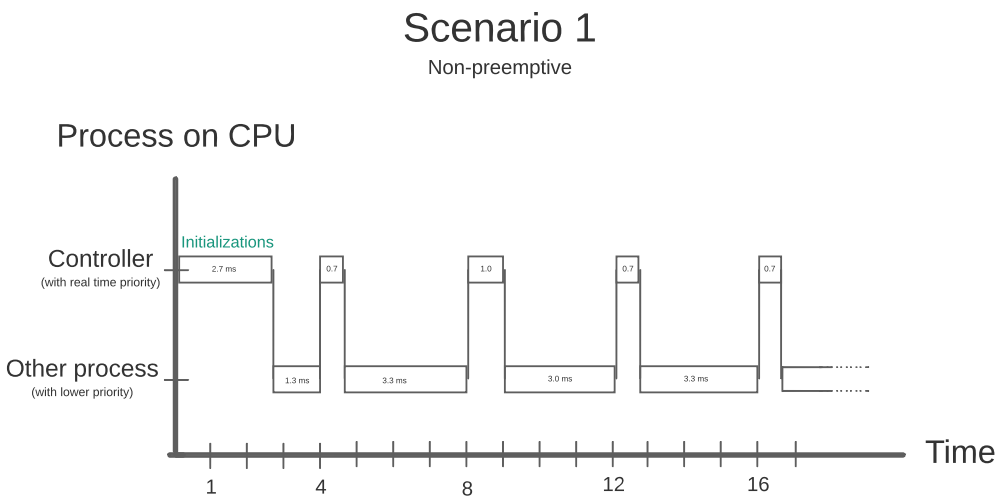

Figure 8: An illustration of the CPU time for a controller application with the highest priority and other processes with lower priorities running on the same system. The controller application have a cycle time of 4 ms, which can be seen in the illustration by looking a the size of the gap between the starts of the controller CPU time intervals. The first interval have the longest time spent on the CPU since, in this scenario, the controller application is assumed to execute some initialization code before entering the main control loop.

The second scenario that the prototype would have to handle is illustrated in Fig. 9. In this scenario it is assumed that there are other processes with the same priority as the controller application, which is normal priority in this case. The effect of this setup is that the controller application may be interrupted during the execution of the control loop. The prototype must be able to capture intervals that are very small and somehow determine if the deviating time interval is caused by an preemption or something else. A possible solution is to examine the length of the time interval following the deviating time interval, i.e., the time interval that the controller process does not spend on the CPU directly after the deviation. If the following time interval is significantly shorter than the mean, it could mean that the controller was preempted and that the following interval for time spent on CPU should be considered as the continuation of the preempted control loop iteration.

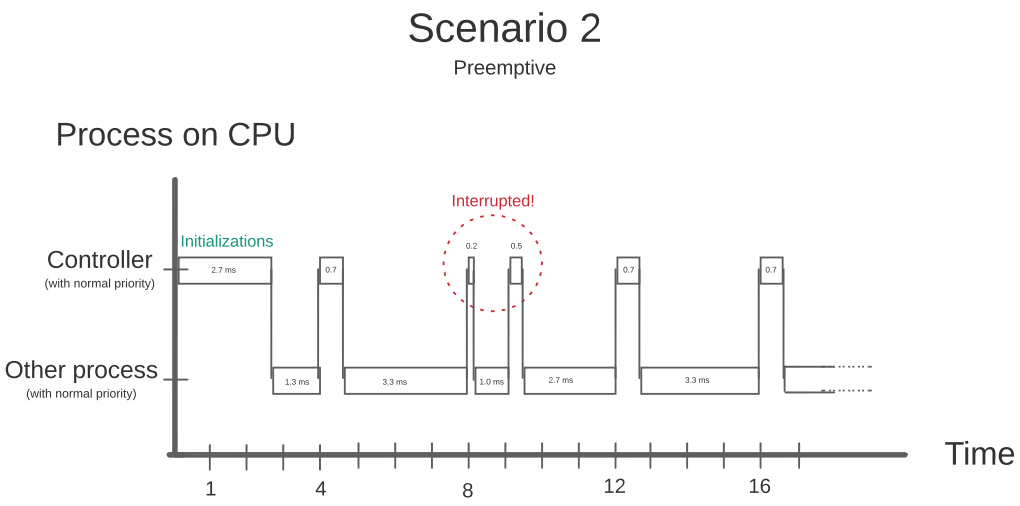

Figure 9: An illustration of the CPU time for a controller application with the same priority as other processes running on the same system. The controller application have a cycle time of 4 ms, which can be seen in the illustration by looking a the size of the gap between the starts of the controller CPU time intervals. With the exception of the two middle intervals where the controller was preempted. The first interval have the longest time spent on the CPU since, in this scenario, the controller application is assumed to execute some initialization code before entering the main control loop.

To test the feasibility of the method described in this section, the prototype was implemented in Python, using the psutil library, however, alternative implementations were also considered, such as the Process object in C# that can obtain similar information. The main loop of the prototype basically consists of a timestamp retrieval, simple polling of the corresponding object containing information about the process, and some logic to determine if a context switch happened or not. The information retrieved from the process object is the *cpu_ times* in Python, and if implemented in C#, the *TotalProcessorTime* would be used instead. If the CPU time did not increment since the last iteration of the loop, it means that a context switch must have happened. The execution time of the main loop determines how accurate the prototype is since the granularity of the timestamps depend on how often they can be read, up to the point where the granularity matches the actual resolution of clock that the timestamps are read from. With this in mind, efforts were in place to make the execution time of the main loop as low as possible during the development of the prototype.

6.2.2 Architecture

The findings from the tests of the early prototype led to the creation of the second and final prototype that uses an event based approach instead of continuous polling. With an event based approach, no context switches will be missed, and a timestamp can be taken immediately when the event happens. An event based approach leads to very accurate reading and a very good utilization of processing power.

There exists several technologies that are capable of realising an event based unobtrusive computation diagnostics tool. One technology that was explored was DTrace, which operates on Windows, which is the most common platform for software PLC. Investigation of DTrace on Windows showed that it is very limited in its capabilities and supported platforms: It can only run on Windows 10 x64 Build 18342 or higher, which means it needs a 64 bit platform. Further, it can only trace 64 bit processes. These limitations resulted in DTrace not being able to trace the wake cycles of the simulated controller. Since the goal of the diagnostics framework is to be versatile, it is not wise to select a technology with as much limitations as DTrace.

Another technology that is widely recognized, especially in the networking and security domain, is eBPF [37]. During initial tests of the feasibility of eBPF, hooks that were able to trace the sleep and wake cycles of the simulated controller application were found. The identified hooks, however, were not available on Windows systems so it was decided that the final implementation of the ULT Computing probe was to be developed for Linux environments. This decision was further motivated by the increasing support for software PLCs running on Linux. To make the implementation quicker than it would be if eBPF were used directly, a scripting language called bpftrace [12], which wraps eBPF, was used instead. Due to the inherent limitations of eBPF, a wrapper was needed to forward the information extracted in the eBPF program to a database so it could be used further. The wrapper for the bpftrace program was created in Python and is responsible for spawning the bpftrace process. The bpftrace program and the Python wrapper have a one-way communication via the stdout of the bpftrace program. The stdout of the bpftrace program is piped to the Python wrapper that processes the data and then finally sends the processed data to the database. The architecture and a simple use case of the ULT Computing probe is illustrated in Fig. 10. The illustration is showing a use case where the probe is running in parallel to the controller application directly on the operating system, i.e., not in a container, however, the probe is capable of tracing controller applications running in separate containers as well.

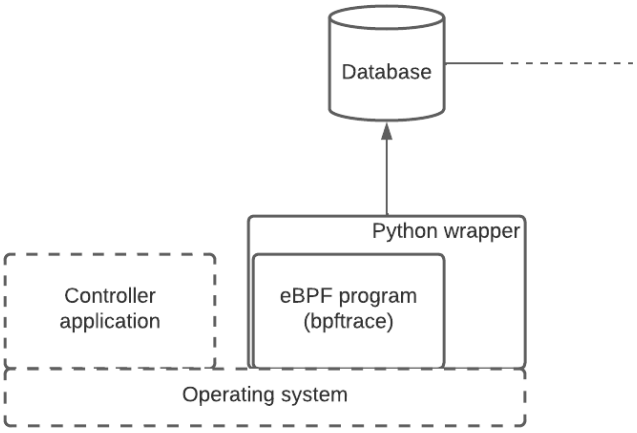

Figure 10: An illustration showing the architecture of the ULT Computing probe. A wrapper written in Python starts a bpftrace script and pipes the stdout of the process in order to further process the data and send it to a database. The data in the database can be used by other applications, which is indicated by the dashed line going from the database. In this illustration, the simulated controller that is supposed to be probed is running in parallel to the probe directly on the operating system, i.e., not in a container or on a virtual machine.

To conclude what has been said about the first prototype and the final architecture: The first prototype was based on a procedure-driven approach that utilized the information accessible in Process objects, both Python and C# implementations were tested. The accuracy of the first procedure-driver prototype was low, and instead, an event-driven approach was adopted. The final prototype, and the architecture that was ultimately settled on, uses eBPF via the bpftrace project on a Linux platform and a Python wrapper that sends the extracted data to a database.

6.2.3 Implementation

In this section, the implementation of the ULT Computing probe will be presented. First, the eBPF program and bpftrace script will be discussed, and following, the wrapper program and how it interacts with the eBPF program will be explained.

At the core of the ULT Computing probe is the eBPF program that is created by the bpftrace script. The program utilizes two hooks, one that gets triggered when a process calls a system call that puts it to sleep and the other one when it exits that system call. By keeping track of the time of entry and exit, the time spent awake and the time spent sleeping can be computed,

which is exactly what is done in the two functions that are attached to the hooks. The tracepoint that gets triggered, however, may differ between different controller applications. For example, when monitoring the simulated controller application, which is written in Python, the voluntary sleep system call named *sys_enter_select* is used, and when monitoring a SoftPLC application, the *sys_enter_clock_nanosleep* found in the scheduler must be used. Regardless of the system call that is traced, the corresponding exit point of the tracepoint is also traced. The names of the exit tracepoints are *sys_exit_select* and *sys_exit_clock_nanosleep* respectively. The tracepoints to use must manually be modified in the bpftrace script between different use cases since it is too complex for bpftrace to be handled by command-line arguments.

Further, the name of the executable and process ID of the process to be monitored must be specified to filter out any unwanted processes that also trigger the aforementioned hooks. The executable name is usually the same between sessions, e.g., "python3" for the simulated controller and "Maintask" for a SoftPLC application. The process ID, however, may differ between sessions. To accommodate for this variation, the executable name and process ID can be passed as command-line arguments when starting the script. The executable name is mandatory, but the process ID is optional. If a process ID is not provided, all processes with the provided executable name will trigger the hooks. It is, therefore, although optional, advised to always provide a process ID as well.

Once a time interval is computed by utilizing the *enter* and *exit* timestamps, it is printed to the stdout. The format of each sample is a white space separated line consisting of the following entries: name of the process, process ID, action (awake or sleeping), and duration in nanoseconds. As an example, an entry for a sleep interval lasting roughly 8 ms may look like this:

python3 31415 SLEEPING 8015320

Since communication with a database over a network connection is far too complex of a task for a eBPF program, another program is needed to take care of the task of sending the data to a database. Having the data produced by the bpftrace script be formatted a specific way, and keeping the stdout reserved for printing the samples, provides a means for easily extracting and forwarding the data. A wrapper program was written in Python with the purpose of doing just that: Extracting the data in the stdout of the process and forwarding it to a database.

The wrapper starts the eBPF program as a sub-process via the asyncio framework and pipes the stdout of the process to a handle that is accessible to the Python script. Before the main loop of the wrapper program is executed, it identifies how the data is formatted by waiting for a line that starts with "headers:". The line that initializes the headers specifies the names and order of the features present in a sample. This approach allows the wrapper program to function even if the bpftrace script modifies the structure of the samples. The wrapper also processes any command-line arguments that may have been passed to the application, e.g., process ID and executable name of the process to monitor, which will be passed down to the bpftrace process.

Once the headers have been initialized, the wrapper program enters a loop that continuously reads a line from the stdout of the bpftrace process, processes it, and sends it to a database. In this case, the wrapper is configured to send the data to an InfluxDB database. The data is sent in large batches to avoid network congestion. Further, the wrapper program also prints out the moving mean, and can also easily be modified to print out every sample it collects, so that it can run without a database if the user just wants to quickly observe the behavior of a process.

Several experiments were performed to test and verify the feasibility of the implementation. However, the details of the experiments will not be presented here. They will be presented in Section 7.1.

6.3 Time-series database

In the proposed system architecture for the thesis, a database was planned to be used to store all measurements from the controller, network, and control target monitoring probes. Because the current critical motion control systems could be running with cycle times as low as 1 ms, each monitoring probe could handle several thousand packets per second. Thus, the database had to be able to handle large quantities of writing operations towards the database without causing any

bottlenecks. In addition, the database should also be capable of handling larger periodic reads from the database to the visualisation tool. The decision was to use a time-series database, which is optimized for writing large quantities of data and reading less frequently.

There were several commercial available solutions available at the time, such as Prometheus [5], InfluxDB [41], TimescaleDB [42], GridDB [43], and OpenTSDB [44]. Two of the more popular options were Prometheus and InfluxDB. Both are well established open-source time-series databases that provide several features for querying, storing, and visualising data. They have a wide range of libraries to extend functionality in the framework and an active community. When it comes to integration, both can integrate with various other systems, such as Docker [18], OpenStack [45], GitHub [46] and Grafana [47]. One of the key differences is that Prometheus supports data writes with a millisecond resolutions timestamp, while InfluxDB supports nanoseconds resolutions timestamps. Prometheus is also designed to pull data from the target system periodically, while InfluxDB expects the target system to push data to the InfluxDB server.

After evaluating these time-series database alternatives, the project chose to use InfluxDB. The reasoning was threefold: It can handle a large number of consecutive writes per second [48], which is critical to storing all captured data successfully. It supports nanoseconds resolution timestamps, which is beneficial for more precise measurements. Finally, it can scale horizontally with increased data flow while requiring minimal additional configuration changes.

6.3.1 Architecture

The InfluxDB database architecture consists of two sections for the ULT Network and Computing, depicted in figure 11. These sections reside in a bucket, which corresponds to a measurement session. Each table in the ULT Network and Computing is structured to accommodate the raw monitoring data. The reason is foremost to store unmodified data to minimise storage space overhead. Storing the raw data also allows for further processing in other nodes, such as the visualisation tools. The measurement tables for the ULT network consist of one table for each specific measurement: plant response latency, controller send interval, plant send interval, latency from plant to controller, and latency from the controller to plant. For ULT Computing, there exists one table to keep track of process cycle times.

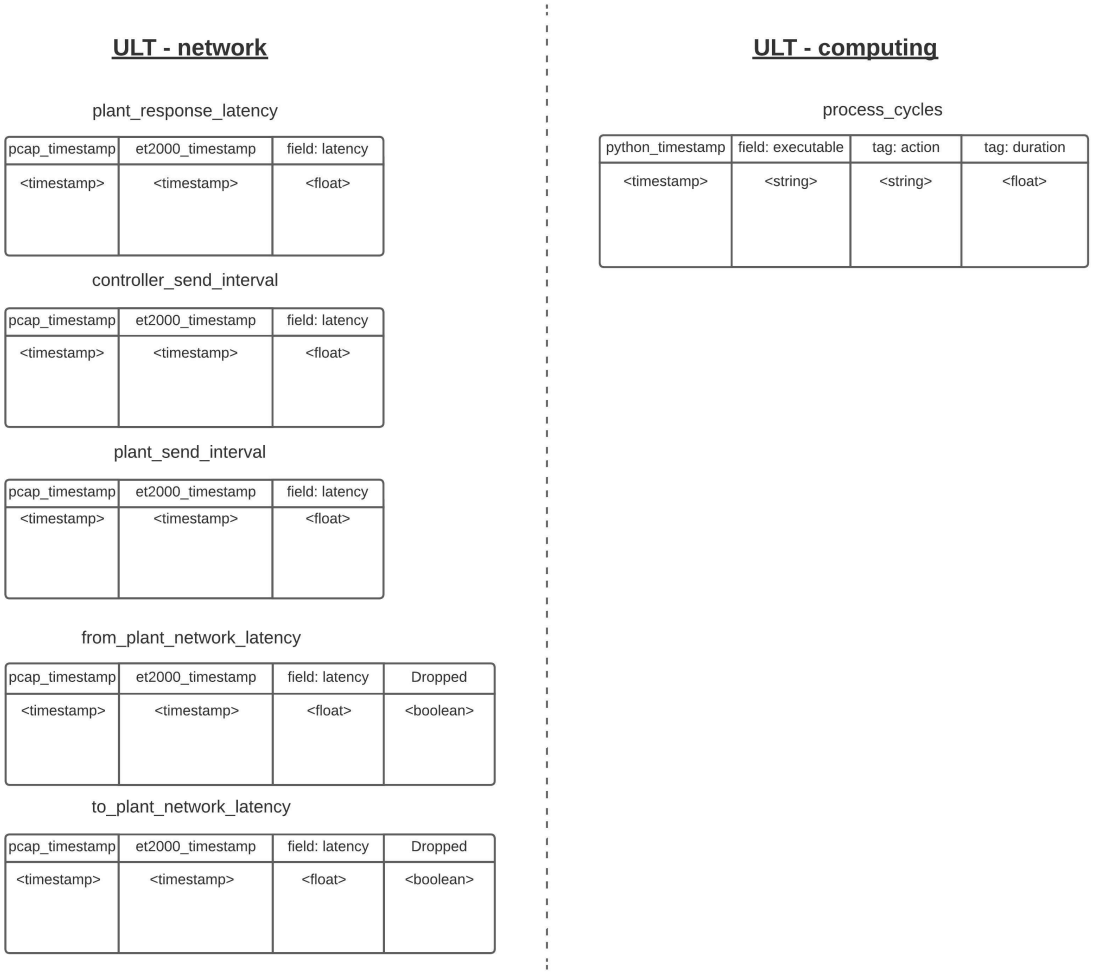

Figure 11: Overview of the InfluxDB database architecture. The ULT network consists of five separate tables for latency measurements calculated over the network between two nodes. Each table contain timestamps from the ET2000 capturing tool, the PCAP software, and the measured latency. Finally, the ULT Computing consists of one table for the controller and control target process separately. The table contain timestamps from Python and the executable name, action, and duration of the awake and sleeping cycle.

6.3.2 Implementation

Since the tables in the InfluxDB architecture are created automatically when uploading data, it allowed setting up the database quickly on the desired platform. During the implementation phase, the database was set up in a docker container on a centralised server for the CFA testbed, shown in figure 12. This allowed for having several ULT network and computing nodes connected simultaneously, and allowing connecting other measurement probes into the database in the future. In addition, having the database in a docker container makes it possible to move it to other infrastructures if needed. It also allows to set up the container and store data on local computers, which makes it a portable solution that can be used in remote areas if required. Several experiments were conducted to verify the feasibility of the implementation, which are covered in Section 7.3.

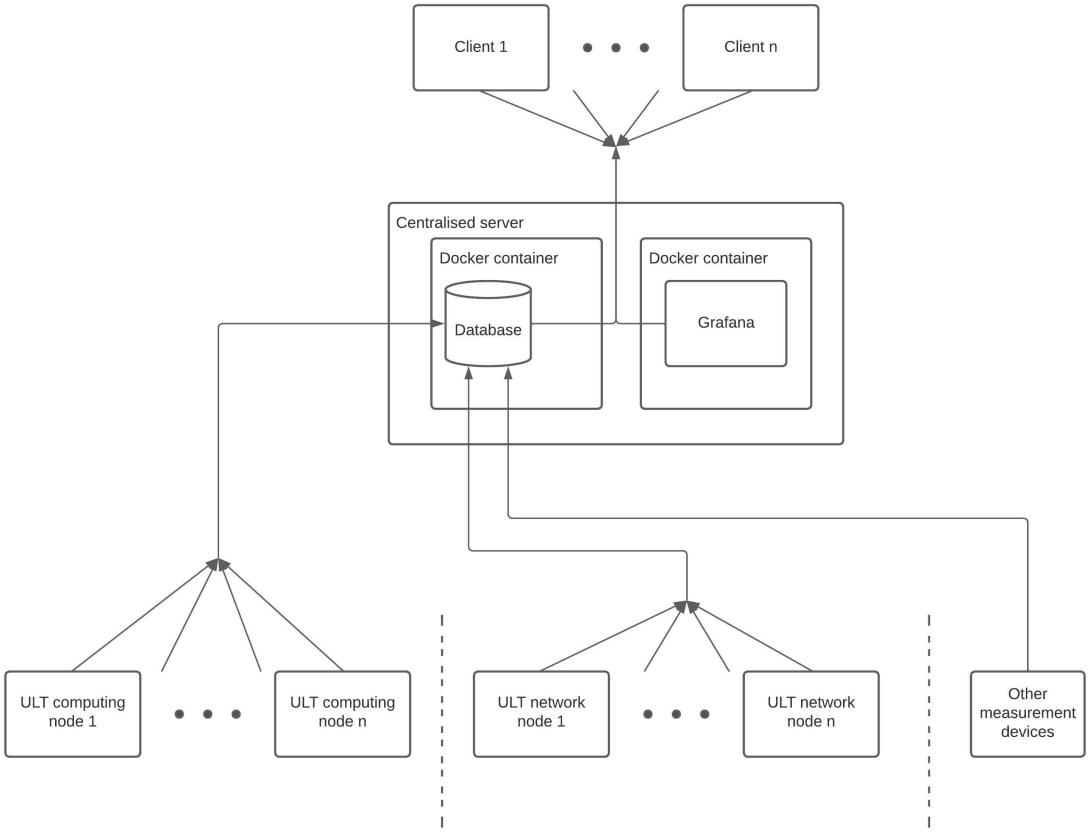

Figure 12: Overview of the centralised database that runs inside a docker container. The centralised database can accommodate several ULT Computing nodes, ULT Network nodes, and other measurement devices, simultaneously. In the centralised server lies also the graphical interface that can be utilized to visualize the captured data. Both the centralised database and graphical interface can be accessed by various clients simultaneously.

6.4 Online visualisation

Visualising monitoring data in a meaningful way can have several benefits for the end-user when examining performance and key limitations in various systems. A valuable aspect is that the visualisation tool should be flexible enough to simplify what data to visualise and how to visualise it. The tool should also be built with extensibility in mind, allowing for expanding the visualisation with more functionality down the line. Various online visualisation alternatives were considered during the pre-study phase, from building a custom visualisation tool to using existing commercial solutions. At ABB CRC, a visualisation tool from a previous thesis work [10] already existed that could visualise various KPI graphs based on previously captured network data. Among these KPIs were availability, reliability, and latency. However, since it was an offline visualisation tool, it was not possible to view the KPIs in real-time.

Two key factors played a significant role in deciding if a custom visualisation tool was to be built or using an existing commercial alternative, setup cost and extensibility. First, setting up the tool and implementing basic functionality should not take too long since the project time plan was relatively short. Secondly, it should not be too complicated for other groups of people wanting to add their functionality to the visualisation tool in the future. Therefore, a decision was made to primarily look for existing commercial visualisation tools that could fulfil these factors. Some of the more popular available visualisation tool solutions to date were Datadog [3], Grafana [47], and Kibana [49].

After evaluating the various alternatives, the project chose Grafana as the online visualisation tool. The reason was due to Grafana being a tool that focuses on providing a wide range of

tools to visualise time series metrics through various graphs. It is also easy to set up and use and requires minimal coding to get data from a database and visualise it. In addition, the tool has an active community with numerous custom built plugins that can be integrated into the framework. Furthermore, since Grafana supports various time-series databases out of the box, such as InfluxDB, it felt more natural to use it since InfluxDB was the choice of database for the project.

6.4.1 Architecture

The architecture consists of two separate sections, depicted in Figure 13, to accommodate the ULT Network and Computing measurements. In the computing section, the cycle time metrics are visualised. For example, the process cycle time consists of when the process is awake and asleep. The network section visualises four types of metrics: latency, reliability, dropped packets, and send interval. The latency, reliability, and dropped packets metrics are structured in two sets to showcase both network directions, i.e. controller to plant and vice versa. Send interval metrics are also structured in two sets to show the send interval of the controller and plant. The latency and send interval metrics consists of three graphs that show the maximum, mean, and minimum latency values over a set period. These three graphs make it possible to observe any latency spikes during a test and compare the difference in latency over time. The reliability metrics consist of calculating the probability of packets missing the deadline. Finally, the dropped packets metrics showcase each time a packet is dropped due to missing the deadline or not reaching the destination target over a set period.

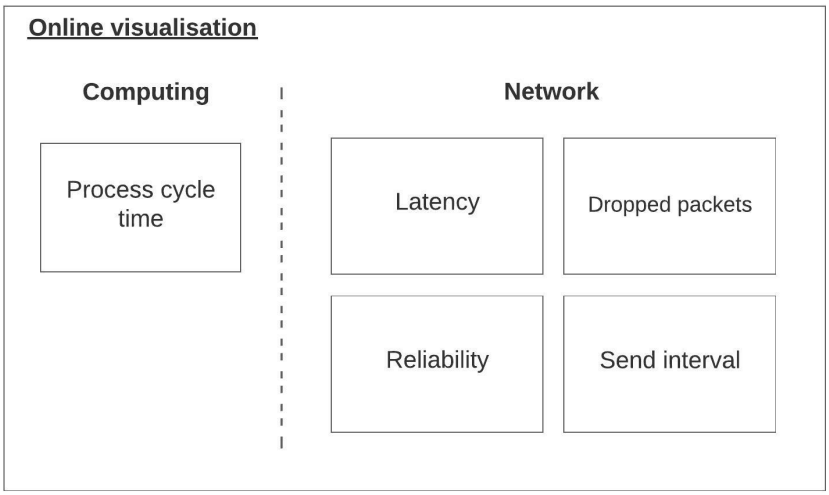

Figure 13: Overview illustration of the online visualisation architecture. The graphical interface is divided into two separate dashboards that visualize metrics for the ULT Computing and Network tool. In the ULT Computing dashboard the process awake and sleep time metrics are shown separately, which together adds up to the process cycle time. The ULT Network dashboard consists of four panels each for the network downstream and upstream.

The measurement data from InfluxDB is queried using Flux query language [50]. Shown in Figure 14 is the process for querying data and visualising it in Grafana. First, a range is inputted in Grafana to define an interval in which data is fetched. Next, the queried measurement data is filtered to only include the necessary data for each panel in Grafana. Because of the large amount of data captured every second, having an extensive range can cause significant bottlenecks when visualising the whole data set. Therefore, all queried data is fed through an aggregate window that aggregates a value based on the inputted data set. It is also possible to extract the min-, mean-, and max value for each aggregated value, which is useful when presenting the latency between the controller and control target.

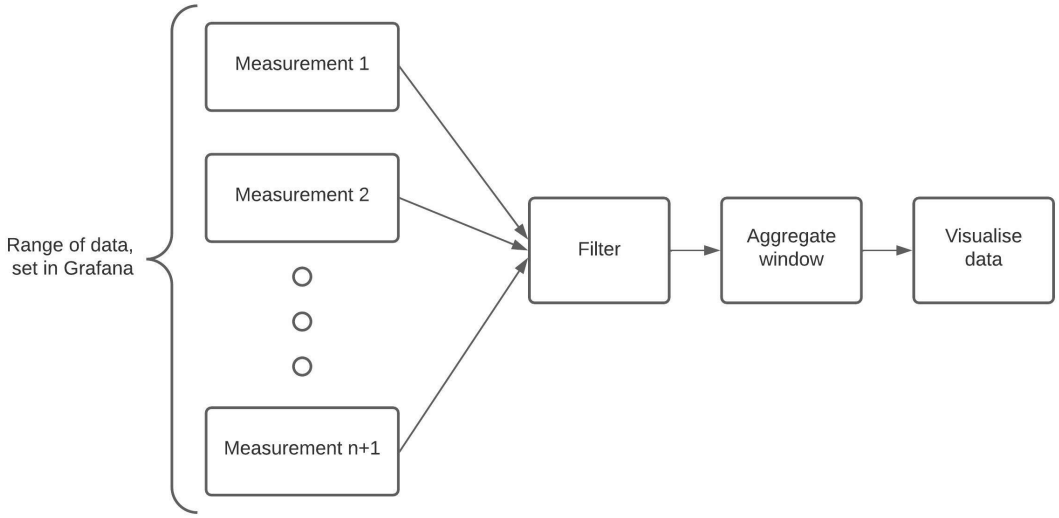

Figure 14: Overview of the visualisation process of measured data in Grafana. First a range is set in Grafana to fetch measurement data within a specified time frame. The fetched data is filtered in each panel to only contain the necessary data useful to that specific panel. Next, the filtered data is fed through an aggregate window to decrease overload of the graphical interface with too many data points. Lastly, the data is visualised in each respective panel.

6.4.2 Implementation

The Grafana dashboard was implemented, similar to InfluxDB, in a docker container on a centralised server, shown in Figure 12. The goal was to have the same capabilities as the database, serve multiple clients simultaneously and be portable to other infrastructures. Figure 19 and 20 showcase the dashboard for the ULT Computing and network. In the ULT Computing dashboard there exists two panels that show the sleeping and awake time of the monitored process, respectively. The ULT Network dashboard consists of three rows of panels. In the first two rows the latency, reliability, and dropped packets, from the controller to the drive are displayed, respectively. The third row shows the send interval of the controller and drive. Several experiments were conducted to verify the feasibility of the implementation, which are covered in Section 7.3.

7. Experiments

Several experiments have been performed to test the individual components of the diagnostics framework and the framework in its entirety. Not only to verify that the components works as intended, but also to verify their feasibility, both alone and in combination with each other. In the following sections the experiments and their settings will be presented. The experiments will be divided into three sections: ULT Computing, ULT Network, and Diagnostics framework. Where each section covers the experiments done for the respective category that represents the major components (computing probe and network sniffer) and lastly all components as a whole, i.e., the diagnostics framework. Further, the experiments are labeled with identification codes to make it easier for the reader to know which experiment a result is referring to in the result section of the report.

7.1 ULT Computing

The experiments conducted for the unobtrusive computing probe consists of the experiments to test the feasibility of the early prototype and the final prototype that is the computing probe used in the diagnostics framework. For the experiments related to the computing probe in the following sections, the identification codes will use "C" for computing, followed by a number representing the order it is presented in, e.g., C1 for the first experiments, and C2 for the second experiment. The experiment on the early, abandoned, prototype will, however, be identified as experiment C0.

Common for all the computing experiments is the simulated controller that was created to make a standardized test target for the prototypes. The simulated controller performs an arbitrary workload that can be configured to last a specific duration. In addition to the duration of the workload being configurable, the cycle time can also be configured. The combination of the two configurable parameters gives the user the ability to test different variables and compare the actual values of time metrics, such as cycle time, sleep time, and wake time, with the values that the monitoring tool computes during experiments.

7.1.1 Experiment C0 - Early prototype

The purpose of this experiment was to test the feasibility of using a procedure-based approach to measuring the sleep and wake cycles of a control application. As mentioned in Sec. 6.2.1, the early prototype was implemented in Python, but could also be implemented in C#, for which the experiment setup would be the same. The Python version used for testing was version 3.8.10, but it is likely that later as well as earlier version would work as well.

During the tests, the monitoring process was given the highest possible priority so that it would not be preempted that otherwise could be a reason for high deviation and inconsistencies in the measurements. During the tests, the simulated controller and the prototype was assigned real time priority on a Windows 10 system, and was assigned to specific cores. The machine that the test ran on was using an AMD Ryzen 7 5800X 8-core processor and had 32 GB of RAM. The experiment was conducted by starting the simulated controller in one terminal, and the prototype application in another terminal and then assigning the process priorities and processor affinities via the activity manager.

7.1.2 Experiment C1 - Simulated controller behavior

The behavior of the simulated controller was examined in experiments that tracked the time it took to complete 1000 cycles with various cycle times and workloads. The goal of the experiment is to see if the simulated controller behaves as it should. The different sets of settings are shown in Table 4. Three different set of settings was used for this experiment, resulting in three set of results that should be able to sufficiently show the behavior of the simulated controller. The experiment was run in WSL2 on a Windows 10 computer with an AMD Ryzen 7 5800X 8-core processor and had 32 GB of RAM. The simulated controller process had the default priority when it was running in the WSL2 virtual machine.

The hypothesis for this experiment is that the lower the cycle time is, the bigger the gap will be between the actual time it took to complete all cycles and the time that it should have taken in an

ideal scenario. Proving this hypothesis will set a baseline for what to expect in other experiments using the simulated controller.

Cycles	Cycle time	Control loop workload	Expected time to complete
1000	10 ms	1.5 ms	10001.5 ms
2000	5 ms	1 ms	10001 ms
10000	1 ms	0.5 ms	10000.5 ms

Table 4: Experiment settings are shown in this table. The expected time to complete is the ideal completion time calculated with an ideal wake and sleep cycles. The settings are configured so that each test runs for roughly 10 seconds, which is enough to capture any emerging patterns in the actual completion time.

7.1.3 Experiment C2 - Integration test

With all components of the ULT computing probe implemented, the feasibility of the tool needed to be tested. An experiment was set up just to see that the individual parts integrated as they should. For this part, the database connection of the tool was omitted, and instead, the data in this experiment was generated by taking a screenshot of the simulated controller output and the computing probe output. The cycle time and workload for the simulated controller were set to 10 ms and 1.5 ms respectively. Since the purpose of this experiment was just to see that the different parts of the probe integrated properly, there was no need to use different sets of settings. The results of this experiment will show if the computing probe, as it was implemented, is a feasible approach to tracing a process' sleep and wake cycles. The environment that hosted this experiment was the same as in Experiment C1: A WSL2 VM running on Windows 10, with an AMD Ryzen 7 5800X 8-core processor and 32 GB of RAM.

7.1.4 Experiment C3 - Computing stress test

To further validate the feasibility of the computing probe and to verify that the probe can capture all the sleep and wake cycles of a process in a demanding scenario, i.e., low cycle times, an experiment was set up to track how many sleep and wake cycles per second the computing probe can handle. The test was conducted for one hour, where a comparison was made between how many cycles were produced by the simulated controller and how many were handled by the computing probe per second. The simulated controller was set to run at a cycle time of 0.75 ms, with a workload of 0.1ms. The goal was to produce 1000 cycles per second, corresponding to what an actual controller would produce per second with a cycle time of 1 ms. The experiment was conducted on a Dell PowerEdge R440 running Debian 11, using a 16 core Intel Xeon Silver 4208 CPU @ 2.1GHz and 46GB Samsung DDR4 3200MHz ECC RAM.

7.2 ULT Network

All experiments that examined the network part of the diagnostics framework, i.e., ULT Network, will be presented in the following sections. The experiments are named with the same naming convention as the ULT Computing section (Section 7.1): Using identification codes, with N, for network, as the prefix instead of C, for computing.

7.2.1 Experiment N1 - Comparison test

This experiment aimed to validate that the ULT Network tool can monitor various network topologies and calculate an accurate latency over the network. A comparison was made between the newly developed ULT Network tool and the pre-existing UL3T tool over various network topologies. An overview of the experimental setup can be viewed in figure 15. The experiment consisted of several tests where an ABB PLC controller controlled an ABB PLC drive over various network topologies. Two ET2000 devices were connected in series to the network, with a laptop connected to each device running the UL3T and ULT Network script, respectively. In one of the experiments, however, another topology was used were UL3T and ULT both ran on the same computer and

only one ET2000 device was used so that the ET2000 timestamps would be identical in both sets of measurement and more detailed analysis could be performed on the data. The measurement data captured by both programs were stored locally on the computer and compared after the experiment, where the mean latency, standard deviation, and total number och packets recorded was compared. The experiment aimed to test two network types, WiFi 6, and 5G, using various protocols presented in table 5.

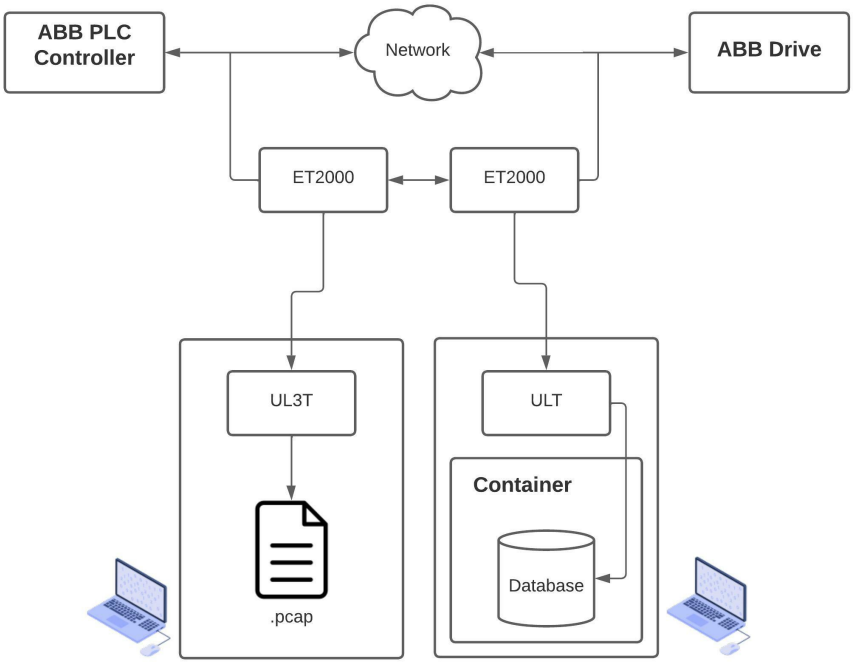

Figure 15: Overview of the setup for experiment N1. The ABB PLC controller is connected to an ABB drive over various network topologies. Two ET2000 devices are connected in series with the network, which mirrors the network traffic to two separate laptops running the UL3T and ULT Network, respectively. The UL3T script stores the processed data in a pcap file, while ULT Network stores the data in a containerised database.

Network type	**Protocol(s)**
WiFi 6	Profinet, Modbus TCP
5G	Profinet, Modbus TCP

Table 5: Overview of the network types and protocols used during the experiment. Both WiFi 6 and 5G were tested using the Profinet and Modbus TCP protocol.

7.2.2 Experiment N2 - Long-term test

When testing the stability of controllers and drives at ABB, a common practice is to conduct long-term tests over several days to measure the performance of the devices and network used. Thus, the measurement tool should be able to handle running for more extended periods. This experiment aimed to test the long-term stability of the ULT Network tool by having it monitor an ABB PLC controller and drive over three days. The experiment setup was the same as Experiment N1, with 5G Profinet as network topology. A comparison with the UL3T at the end of the three-day measurement was conducted to evaluate how well the ULT Network performed.

7.2.3 Experiment N3 - Network stress test

Similar to Experiment C3, the ULT Network tool also went through a stress test to validate the tool's feasibility in capturing all packets in a demanding scenario. The experiment was set up using an ABB PLC controller and drive over an Ethernet connection using the Profinet protocol. The cycle time of the controller was set to 1ms to produce 1000 packets per second. Since the ULT Network tool handles five queues for various latency data, it would have to handle 5000 packets per second. The experiment was conducted for one hour. Then, a comparison was made between the expected and actual number of packets recorded per second and the total number of packets recorded for the whole experiment.

7.3 Diagnostics framework

Some experiments had the purpose of testing all components together as a whole, instead of individually, and they will be presented here, in the following sections. Just like for the sections above, identification codes will be used here as well, but with DF, for diagnostics framework, as the prefix to the number.

7.3.1 Experiment DF1 - Framework test

This experiment aimed to validate the functionality of the diagnostics framework as a whole. The experiment used the ULT Network and Computing probe, connected to a centralised database, to simultaneously monitor a controller and drive. The setup of the experiment can be viewed in figure 16. A softPLC program residing in a Debian KVM on a server was used to control a ball-and-beam device over ethernet using a cycle time of 4ms. The control signals were sent to a RevPi, that acted as a drive which sent response messages back to the controller. The ULT computing probe was used inside the controller KVM, which sent the measurement data to a centralised database running inside a Docker container. An ET2000 device was connected over the Ethernet connection between the nodes, which mirrored the packets to the ULT Network program running on a laptop. The processed network packets were then sent to the centralised database on the server. The experiment was set to run for one hour with the controller running at a cycle time of 4ms. This would generate 900 000 system calls for the process waking up and going to sleep, respectively. During three occasions of the experiment the Ethernet cable between the controller and the drive was shortly disconnected to simulate packet loss. The reason to these intentional disconnections was to validate if the ULT Network tool would register those occasions as dropped packets.

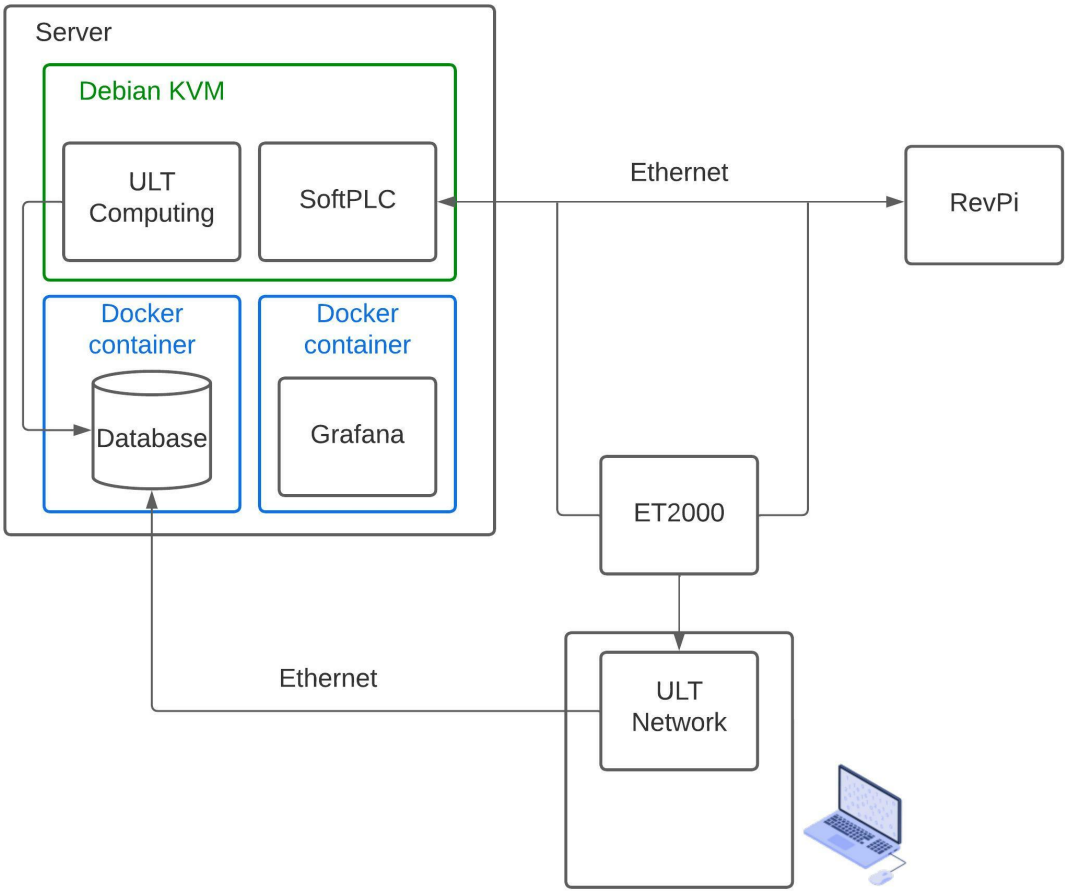

Figure 16: Setup overview of Experiment DF1. The controller consisted of a softPLC in a Debian KVM, which controlled a RevPi device over Ethernet. The RevPi acted as a drive which sent response messages back to the controller. Inside the same KVM as the softPLC resided in the ULT computing probe was also installed, which sent measurement data of the softPLC to a centralised database on the server running in a Docker container. An ET2000 device was connected to the network to mirror packets sent between the controller and drive. These packets were fed to the ULT Network running on a laptop, which measured the latency and sent the data to the centralised database on the server.

7.3.2 Experiment DF2 - Database stress test

Since the measurement data of the ULT Network and computing tool was stored in a time-series database, a stress test would validate the performance of the database. Thus, an experiment was conducted where the measurement data read using both measurement tools was uploaded to a database running in a Docker container. The experiment was conducted in two parts. The first part used the ULT computing to stress the database, which used the same environment as Experiment C3. The second part used the ULT Network to stress the database, which used the same environment as in Experiment N3. In both parts, the database resided in a local Docker container. After completing the tests, an analysis of how many results the database could record was done.

8. Results

In this chapter, the results from the performed experiments will be presented. The same naming convention is used here as in the experiment section so that the reader can refer the results to the corresponding experiment with ease. The chapter is divided into three main sections that each present the results for ULT Computing, ULT Networking, and the Diagnostics framework respectively. The diagnostics framework section will present results from the experiments that used all parts of the framework.

8.1 ULT Computing

The results from the experiments that were performed on the ULT Computing tool, or the parts closely related to the tool, will be presented in the following sections.

8.1.1 Experiment C0 - Early prototype

The results from this experiment are the authors own observations from the performance of the prototype. Due to the quick conclusion drawn from the observed performance, no extensive tests were conducted to numerically record the performance and compute KPIs from.

During the experiment, it was shown quickly that, despite the efforts put in place to make the main loop as fast as possible, the granularity was not nearly as high as it needed to be. The data that the prototype created had too high deviation and too many inconsistencies for it to be useful. The actual sleep and wake cycles of the simulated controller was monitored from inside the simulated controller and that gave consistent values with insignificant deviations. The prototype missed around 1% of the context switches that happened and could give time intervals that were over twice as long as they should have been, e.g., sleep intervals of around 38 ms instead of 13 ms.

8.1.2 Experiment C1 - Simulated controller behavior

The results for Experiment C1, which was presented in Section 7.1.1, showed that the lower the cycle time is, the bigger the gap is between the expected completion time and the actual completion time. The results are shown in Table 6. The first setting had an expected completion time of 10001.5 ms, but the actual completion time was 10116.4 ms that is 1.15% longer. Similarly, with the second setting, the expected completion time was 10001 ms, with an actual completion time of 10207.8 ms, 2.07% longer. Lastly, the final setting yielded 9.03% longer completion time than what was expected: 10000.5 ms expected completion time, and 10903.05 actual completion time. The experiment showed that the wake-ups of the process was, on average, delayed by roughly 0.1 ms for each cycle. The actual completion time can then, for this specific configuration, be estimated as the expected completion time plus the total delay, which is estimated to be 0.1 ms multiplied by the number of cycles. In the first, second, and third setting the estimated actual completion time would then be estimated to 10100 ms (1% longer), 10200 ms (2% longer), and 11000 ms (10% longer) respectively.

			Completion time	
Cycles	**Cycle time**	**Control loop workload**	**Expected**	**Actual**
1000	10 ms	1.5 ms	10001.5 ms	**10116.4 ms**
2000	5 ms	1 ms	10001 ms	**10207.8 ms**
10000	1 ms	0.5 ms	10000.5 ms	**10903.05 ms**

Table 6: Results from Experiment C1 that aimed to show the behavior of the simulated controller. The table is showing the experiment settings, and the results from the experiment that is written in bold font in the Actual completion time column. Lower cycle times yielded a bigger gap between the actual and expected completion time.

8.1.3 Experiment C2 - Integration test

The implementation of the proposed unobtrusive computing probe resulted in a computing probe that performed as expected: The probe did not appear to affect the controller application in any way, and the measurements from testing the probe showed that the probe seemed to be as accurate as it can get, considering that even the most fine grained clock in a computer is inaccurate to some extent. In Fig. 17, the integration test that was run to verify the feasibility of the computing probe is shown. The left side shows the simulated controller application running, and the right side shows the computing probe printing out the sleep and awake moving mean over 50 values in milliseconds.

By examining the left side Fig. 17 it can be seen that the simulated controller attempts to go to sleep for roughly 8.5 ms, which is too be expected since the simulated workload is executed for roughly 1.5 ms and the cycle time is 10 ms. The time that the simulated controller should sleep for is calculated by subtracting the time spent in the control loop (1.5) from the cycle time (10), which results in 8.5 ms. Due to some internal overhead in the control loop, the expected sleep time is always slightly lower than 8.5 ms. The actual time that the controller sleeps for, however, is roughly 0.1 ms longer than it is supposed to. This behavior is captured in the measurements from the computing probe that can be seen by looking at the right side of the figure. The moving mean over 50 values is showing that the simulated controller was sleeping for around 8.59 ms and was awake for around 1.575 ms.

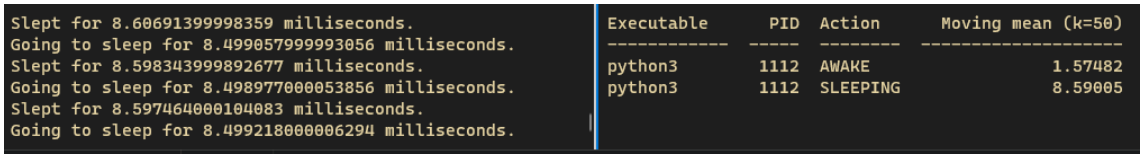

Figure 17: A screenshot of the simulated controller application (left) and the ULT computing probe running in parallel. The figure shows that the controller's actual time slept and the time it is attempting to sleep differ by around 0.1 ms. Further, the computing probe that is displayed on the right side shows that it is able to capture the real behavior of the controller: The moving mean over 50 values for time sleeping is at the time of capturing the screenshot 8.59 ms, which is close to the actual time slept value printed on the left side. It is worth keeping in mind that the left side is showing individual values and the right side is showing moving mean over 50 values, which means that the values will differ.

8.1.4 Experiment C3 - Computing stress test

The results for Experiment C3, presented in Section 7.1.4, are shown in table 7. A cycle time of 0.75ms and a workload of 0.1ms were used for the simulated controller, which produced 1000±2 cycles per second. The computing probe recorded 2000±2 packets per second, corresponding to roughly 1000 calls to enter and exit sleep, respectively. The simulated controller ran for 3 608 292 cycles, and the computing probe registered 7 216 584 system calls.

	Simulated controller		**Computing probe**	
	Cycles/s	**Total cycles**	**Captured system calls/s**	**Total captured system calls**
Expected	1000	3 600 000	2000	7 200 000
Actual	1000±2	3 608 292	2000±2	7 216 583

Table 7: Results from experiment C3. The table is divided into the simulated controller and time-series database with two rows each to showcase the expected and actual results. The simulated controller consists of two columns that present cycles per second and total number of cycles produced during the experiment. The computing probe consists of two columns that present the captured system calls per second and the total number of captured system calls during the experiment.

8.2 ULT Network

In the following section, the results of the experiments performed on the ULT Network tool are presented.

8.2.1 Experiment N1 - Comparison test

Results for experiment N1 are presented in tables 8, 9, 10, and 11. The results are structured into mean latency, standard deviation, and the total number of packets processed from the controller to the drive and vice versa. In most cases of the four tested network topologies, the mean latency, standard deviation, and the total number of packets differed around 0.001-0.01%. In the experiment using WiFi 6 and Modbus TCP presented in Table 9, the deviating topology was used, with the two applications running on the same computer with only one ET2000 device instead of two. With this experiment, the data could be precisely aligned and the result yielded an exact match in captured packets between ULT and UL3T. The mean latency and standard deviation for that setup, however, showed very different numbers between the two tools.

	Controller to drive			**Drive to controller**		
	Mean latency	**Std**	**Total processed**	**Mean latency**	**Std**	**Total processed**
UL3T	2.7826497 ms	0.6720196 ms	256000	2.7776510 ms	0.6111273 ms	256000
ULT Network	2.7814451 ms	0.6719275 ms	255999	2.7764969 ms	0.6112934 ms	255998

Table 8: Results for WiFi 6 using the Profinet protocol. The table is divided into controller to drive and drive to controller metrics, with two rows each to showcase the results for UL3T and ULT Network. Both columns consist of three sub-columns with mean latency, standard deviation, and total number of processed packets.

	Controller to drive			**Drive to controller**		
	Mean latency	**Std**	**Total processed**	**Mean latency**	**Std**	**Total processed**
UL3T	9.82837 ms	12.1929 ms	505653	10.2132 ms	11.9862 ms	505652
ULT Network	2.931 ms	5.40226 ms	505653	2.70839 ms	4.27178 ms	505652

Table 9: Comparison results for WiFi 6 using Modbus TCP protocol. The table is divided into controller to drive and drive to controller metrics, with two rows each to showcase the results for UL3T and ULT Network. Both columns consist of three sub-columns with mean latency, standard deviation, and total number of processed packets.

	Controller to drive			**Driver to controller**		
	Mean latency	**Std**	**Total processed**	**Mean latency**	**Std**	**Total processed**
UL3T	9.9712920 ms	1.6618322 ms	224995	9.9944380 ms	1.6908298 ms	224991
ULT Network	9.9721828 ms	1.6617099 ms	224993	9.9955145 ms	1.6907945 ms	224989

Table 10: Results for 5G using Profinet protocol. The table is divided into controller to drive and drive to controller metrics, with two rows each to showcase the results for UL3T and ULT Network. Both columns consist of three sub-columns with mean latency, standard deviation, and total number of processed packets.

	Controller to drive			**Drive to controller**		
	Mean latency	**Std**	**Total processed**	**Mean latency**	**Std**	**Total processed**
UL3T	6.9965597 ms	1.5432179 ms	392000	3.1370157 ms	0.8176261 ms	392000
ULT Network	6.9954348 ms	1.5432419 ms	391997	3.1358189 ms	0.8176564 ms	391996

Table 11: Results for 5G using Modbus TCP protocol. The table is divided into controller to drive and drive to controller metrics, with two rows each to showcase the results for UL3T and ULT Network. Both columns consist of three sub-columns with mean latency, standard deviation, and total number of processed packets.

8.2.2 Experiment N2 - Long-term test

The long-term test was set to run for 72 hours but ran for roughly 60 hours instead. Table 12 presents the results for the long term test. The table shows the mean latency, standard deviation, total number of processed packets, and recorded packet drops in both network directions for the ULT Network and UL3T. Note that the dropped packets that were recorded in this experiment does not represent the packets sometimes disappearing in the database due to the Packet Loss Problem (PLP), but instead it represents packets that are sometimes dropped in normal network communication in a distributed control system. It is also worth mentioning that a packet is regarded as dropped by both ULT and UL3T if the packet does not arrive at the other side of the network within 300 ms. Mean latency, standard deviation, and the total number of processed packets remained fairly similar in both network directions, with a difference varying between 0.001%-0.056%. UL3T reported no dropped packets during the measurement. In contrast, the ULT Network reported 32 dropped packets from the controller to the drive and 19 from the drive to the controller. Figure 18 shows the dropped packets on a timeline over the experiment period. All packets that were dropped in both network directions occurred at the start of the measurement session.

	Controller to drive				Drive to controller			
	Mean latency	**Std**	**Total processed**	**Packet drops**	**Mean latency**	**Std**	**Total processed**	**Packet drops**
UL3T	9.9738694ms	1.6737733ms	13 396 000	0	9.9889337ms	1.6928732ms	13 395 000	0
ULT Network	9.9795217ms	1.6735716ms	13 396 147	32	9.9896830ms	1.6923548ms	13 395 937	19

Table 12: Results for long-term experiment using 5G Profinet topology. The table is divided into controller to drive and drive to controller metrics, with two rows each to showcase the results for UL3T and ULT Network. Both columns consist of four sub-columns with mean latency, standard deviation, total number of processed packets, and packet drops.

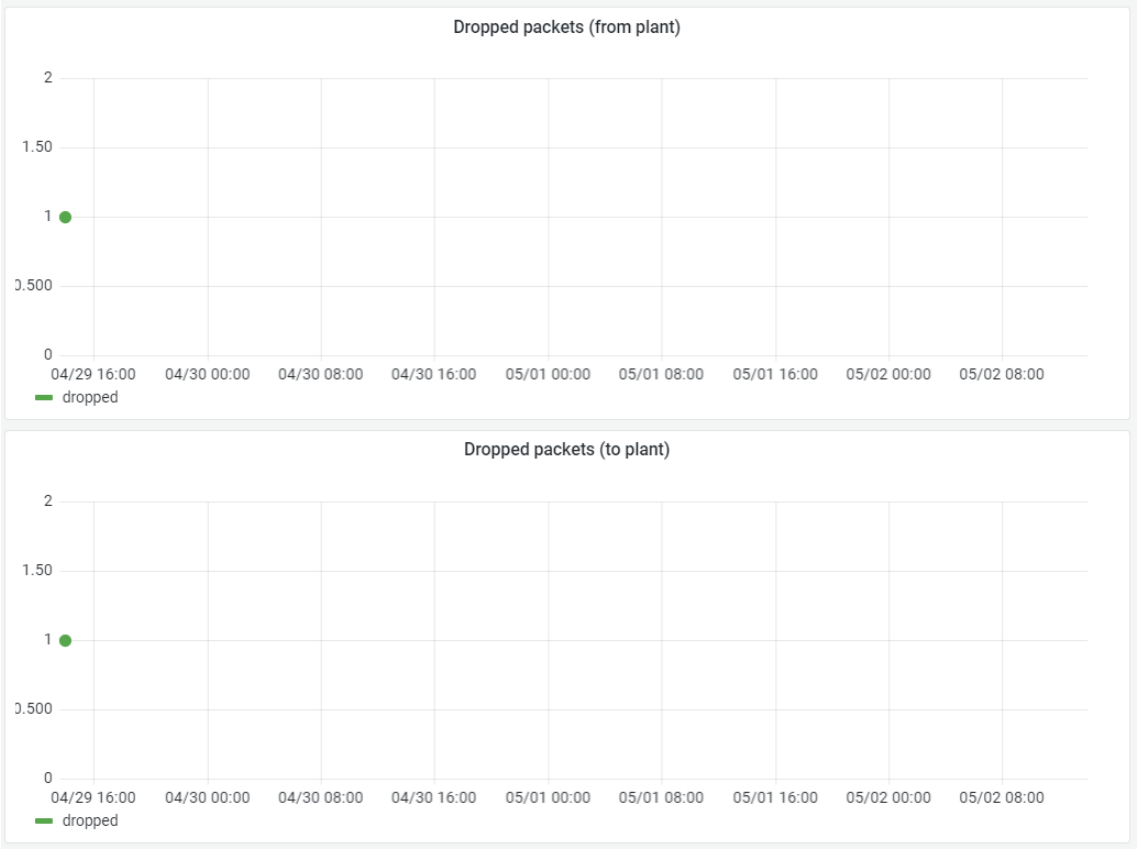

Figure 18: Plot of the dropped packets during Experiment N2. Each occurrence of a dropped packet is registered as one point on the graph with a value of 1. All dropped packets that were registered from the controller to the drive and vice versa occurred at the start of the experiment.

8.2.3 Experiment N3 - Stress test

Results for the network stress test are shown in table 13. The ABB PLC controller was expected to run at a 1ms cycle time to produce 1000 cycles per second, accumulating to 3 600 000 cycles in total over the experiment period. The actual values of these could not be verified during the experiment. The expected amount of captured packets per second for the ULT Network was 5000, with the total expected captured packets being 18 000 000. Actual results show that the ULT Network captured 5000±1 packets per second, with 18 001 226 packets recorded.

	ABB PLC Controller		**ULT Network**	
	Cycles/s	**Total cycles**	**Captured packets/s**	**Total captured packets**
Expected	1000	3 600 000	5000	18 000 000
Actual	-	-	5000±1	18 001 226

Table 13: Results from experiment N3. The table is divided into the ABB PLC controller and ULT Network with two rows each to showcase the expected and actual results. The ABB PLC controller consists of two columns that present cycles per second and total number of cycles produced during the experiment. The ULT Network consists of two columns that present the captured packets per second and the total number of captured packets during the experiment.

8.3 Diagnostics framework

In the following section, the results related to the experiments that investigated the diagnostics framework more broadly, i.e., not the computing or network probe specifically, will be presented.

8.3.1 Experiment DF1 - Framework test

The results for Experiment DF1 are displayed by the dashboards in Grafana, depicted in figure 19 and 20. In the ULT Computing dashboard the total number of system calls shown for both sleep and awake cycle was 900 000. The mean awake time was 0.1473911ms and the mean sleep time was 3.8526088ms, which add up to a cycle time of 3.9999999ms. For the ULT Network dashboard the total number of packets shown from the controller to the drive was 899 949, while the reversed was 899 977. The latency in both network directions was 0.6µs. The mean send interval of the controller was 4.0000272ms, and for the plant 3.9999304ms. The ULT Network dashboard also showed dropped packets during three occasions of the experiment, which corresponded to when the network was intentionally disconnected.

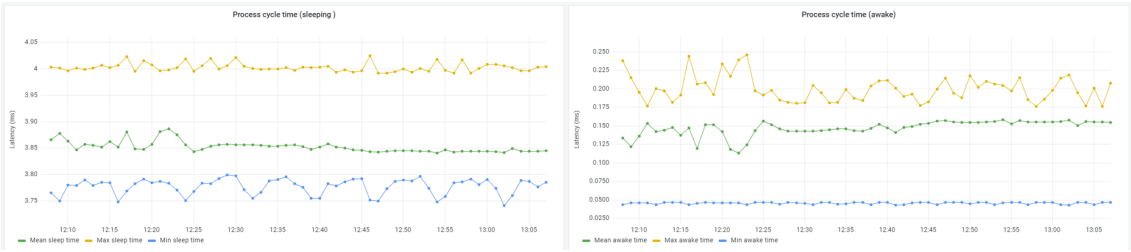

Figure 19: Overview of the Grafana computing dashboard showcasing the readings of the process sleep and awake time during experiment DF1.

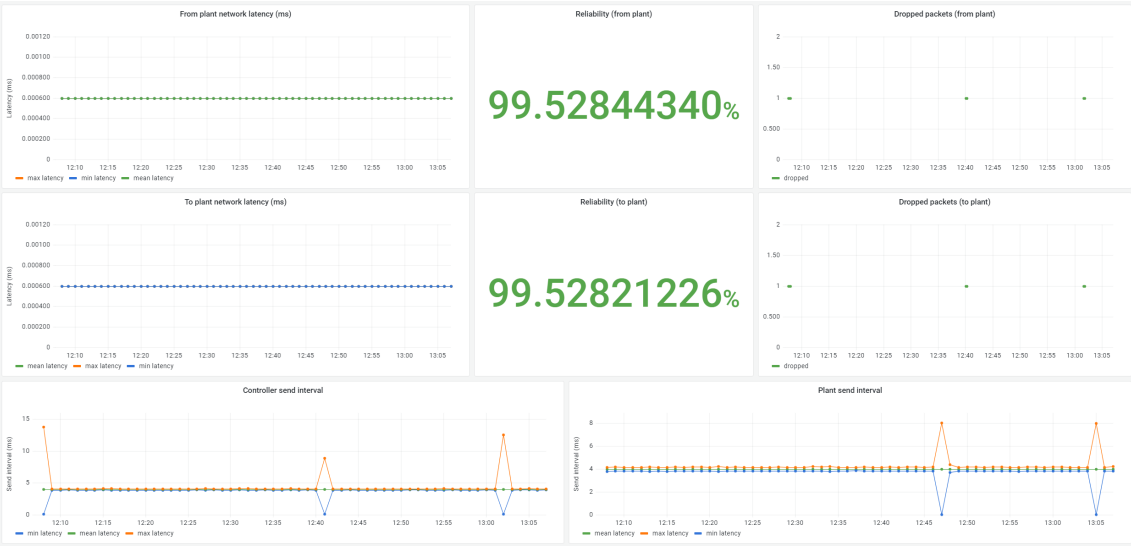

Figure 20: Overview of the Grafana network dashboard showcasing the readings of the network latency, reliability, dropped packets, and send interval in both directions.

8.3.2 Experiment DF2 - Database stress test

The results for Experiment DF2, that were presented in Section 7.3.2, are shown in table 14 and 15. For the first part using ULT computing, the expected cycle time of the simulated controller was 1000 cycles per second, which would result in 3 600 000 cycles in total over an hour. Actual results show that a cycle time of 1008±2 cycles per second was achieved, with 3 628 482 cycles in total after one hour. The expected number of captured system calls per second for the database was 2000, i.e. 1000 wakeup and sleep system calls, respectively. The total number of captured system calls over one hour was expected to be 7 200 000, i.e. 3 600 000 wakeup and sleep system calls, respectively. Actual results show that the database registered 2011±2 system calls per second, with 7 256 000 captured system calls over one hour. In the second part, using the ULT Network,

the expected cycle time for the ABB PLC controller was 1000 cycles per second, which would correspond to 3 600 000 cycles in total over an hour. Actual results for the ABB PLC controller could not be verified during the experiment. The database was expected to handle 5000 packets per second, with 18 000 000 captured packets over an hour. Actual results show that it handled 5000±1 packets per second, with 17 997 303 captured packets over one hour.

	Simulated controller		**Time-series database**	
	Cycles/s	**Total cycles**	**Captured system calls/s**	**Total captured system calls**
Expected	1000	3 600 000	2000	7 200 000
Actual	1008±2	3 628 482	2011±2	7 256 000

Table 14: Results from experiment DF2 part one, using the ULT computing tool. The table is divided in to the simulated controller and time-series database with two rows each to showcase the expected and actual results. The simulated controller consists of two columns that present cycles per second and total number of cycles produced during the experiment. The time-series database consists of two columns that present the captured system calls per second and the total number of captured system calls during the experiment.

	ABB PLC Controller		**Time-series database**	
	Cycles/s	**Total cycles**	**Captured packets/s**	**Total captured packets**
Expected	1000	18 000 000	5000	18 000 000
Actual	-	-	5000±1	17 997 303

Table 15: Results from experiment DF2 part two, using the ULT Network tool. The table is divided in to the ABB PLC controller and time-series database with two rows each to showcase the expected and actual results. The ABB PLC controller consists of two columns that present cycles per second and total number of cycles produced during the experiment. The time-series database consists of two columns that present the captured packets per second and the total number of captured packets during the experiment.

9. Discussion

Discussions of the results will be presented in this chapter. The chapter follows the same structure as the Experiment chapter, as well as the Results chapter: Divided into three main sections that cover ULT Network, ULT Computing, and the diagnostics framework respectively, and a final section that discuss the thesis work as a whole. Each main section, except for the final section, will be divided into separate sections for each experiment that was performed, and the results of each experiment will be discussed in these sections. The final section will reflect upon all the work that was done during this thesis, such as how it could be improved and lessons learned.

9.1 ULT Computing

In the following sections, the results for the ULT Computing probe that were presented in Section 8.1 will be discussed. The discussion is divided into per-experiment discussions, meaning that the result for each experiment will be discussed individually.

9.1.1 Experiment C0 - Early prototype

The prototype was not tested on cycle times lower than 15.6 ms since Windows 10 does not easily support having processes voluntarily sleep for periods of time lower than 15.6 ms. The reason for this is that the default resolution for timers in Windows is around 15.6 ms. However, since it would prove more challenging to capture accurate measurements on lower cycle times, it was not necessary to test the lower cycle times since the prototype could not perform well enough with the 15.6 ms cycle time.

The conclusion from the tests is that the proposed method used in the early prototype is not a feasible approach to getting accurate and reliable time interval values. The computing probe must be able to capture all the context switches as well as give more accurate results than the tested prototype did. In addition to the low accuracy and missed context switches, the method used in the first prototype uses a lot of processing power. In fact, the method uses as much processing power as it can get since it consists of a loop that continuously polls to get the information it needs. Another negative aspect of this method is that its performance is limited to the speed of the CPU.

9.1.2 Experiment C1 - Simulated controller behavior

The results proved the hypothesis that the lower the cycle time is, the larger the gap between the actual and expected completion time would be. This information is important to consider in experiments that are using the simulated controller, especially if low cycle times are used. We are confident that the cause of this effect is the simulated controller not waking up in time, likely due to the algorithm used by operating system's scheduler. It is possible that if the priority of the simulated controller was higher, e.g., the highest possible, the gap would be smaller than it was in this experiment. However, we think that the gap still would be there if the highest priority was used since the experiment setup did not use a real-time scheduler that wakes processes more precisely than other schedulers.

The implication of the results from this experiment is that the actual measured values should be preferred over the theoretical values for the other experiments using the simulated controller. The lower the cycle time is, the bigger the effect of the tardy wake ups will be. For example, the number of sleeps and wake ups over a 10 second period, with a 1 ms cycle time, would in theory be 20 000, but in practice, the number of sleeps and wake ups is around 18 350, which is almost 10% lower.

9.1.3 Experiment C2 - Integration test

The integration test acted as both a simple integration test for the bpftrace program and the python wrapper, and a way to prove the feasibility of the ULT computing probe in a simple environment—using a simulated controller application. The results showed that it is, indeed, a feasible approach to some extent. Seeing as this experiment was using a rather simple setup, not

too much can be said about the feasibility of the probe in a more complex environment, e.g., on a server running a more complex architecture, or in containerized environments. The integration of the python wrapper and the bpftrace program, however, seemed to work as it should, without fail. Although integration and feasibility gave positive results, more experiments are encouraged to further verify the feasibility of the computing probe.

9.1.4 Experiment C3 - Stress test

The results show that the ULT computing program can successfully capture all system calls produced by a process running at 1000 cycles per second. However, as presented in table 7 there is some deviation between the expected and actual results. The actual cycles per second deviated with ±2 cycles, which resulted in a slightly higher number of total cycles. Due to the findings from Experiment C1, these deviations in cycles are expected. This deviation is also why the simulated controller's configured cycle time was lower than 1ms. Although the simulated controller did not operate at the exact desired speed, the results should be sufficient to validate that the ULT computing program can handle cycle times of 1ms.

9.2 ULT Network

Here, the results for the ULT Network tool that were presented in Section 8.2 are discussed. As in the previous discussions, the discussions here are also divided into per-experiment discussions, meaning that the result for each experiment will be discussed individually. One common factor for all network experiments, however, was that packets would sometimes sporadically disappear in the network transmission to the database, resulting in measurements not being entirely accurate. Since this disappearing packet phenomenon is common for all experiments that use the database, it will be discussed first in the section below.

9.2.1 The Packet Loss Problem

During all experiments that involved the database, there were some inexplicable packet losses that led to extensive investigation of the phenomena. From this point on, the phenomenon will be referred to as the PLP. By analyzing the number of packets that were processed by ULT Network and the previous UL3T tool used in previous thesis work [10], a difference was spotted. At first, the difference was thought to be caused by the two datasets, from ULT Network and UL3T, not being completely aligned, but after further investigation, it was clear that this was not the case. The difference between the datasets was computed and the resulting packets were analyzed in an attempt to find a pattern that could trace the problem to the specific type of packets. No pattern was found, however, and the suspicion that it was caused by malformed or packets not supported by the database was discarded.

Next, the program logic of the ULT Network tool was tested by running an offline test with a PCAP file that contained the network traffic. By counting the number of packets that were supposed to be processed, and was actually processed, it was confirmed that the program logic worked as intended and all packets were accounted for. The number of packets was also compared to the number of packets present in the database. Here, the number differed: There were fewer packets in the database than the number of packets sent to the database, which means that the packets either got lost in transmission or some points were overwritten. In InfluxDB a point is considered unique if it has a unique tag set or timestamp. In the case of ULT Network, no tags were used, so the uniqueness of a point is solely based on the timestamp. If a point is not unique based on the aforementioned criteria, the two points are merged, replacing the values of the first point, with the values of the second point. The timestamps that are used in the points are based on the PCAP timestamp that is provided by the pcapy library. The granularity of the timestamps depends on many factors including the configuration of the operating system, hardware, and performance of the system, among other things. This means that there is a chance, although relatively low, that the timestamps that are used for uniqueness in the database are not unique, and that the number of packets with the same timestamp depends on the system in use. Some of the missing packets in the PLP were, indeed, explained by this fact. A handful of the packets were confirmed to have the same timestamps, meaning that the same amount of packets would be overwritten, and, in turn, be

missing from the dataset. After this realization, the timestamps in ULT Network were augmented in a way that ensured uniqueness as long as not more than 1000 packets were timestamped within 0.001 ms.

Since only a small amount of packets had the same timestamp, however, the duplicate timestamps did not explain all missing packets, which means that this was not the single cause for the PLP. Extensive investigation of the packets lost in transmission to the database led to the discovery of an unknown bug in the InfluxDB python client. Another possible cause for the PLP. When writing data to a database in batches, it is beneficial to have a flush interval that determines how long time that should elapse before sending the batch, regardless of whether the desired batch size is achieved or not, i.e., flushing the batch. This exact implementation is used in the InfluxDB python client. Sometimes when a batch is flushed, however, it seems that packets get lost in transmission. A test that had an unreachable flush interval, i.e., the batch was always filled before the interval end was reached, showed that no packets went missing. Whereas, a test that had a flush interval that triggered almost every time before a batch was completely filled, several packets were missing in the database. These tests led to the reporting of the bug to the developers of the InfluxDB python client, which confirmed that the newly discovered bug was reproducible on their end as well.

By testing ULT Network on an offline PCAP file, using an unreachable flush interval, it was confirmed that no packets were lost in transmission. When using a live session, however, some packets were still lost in transmission. This means that the cause of the PLP is still not entirely clear. The usefulness of the diagnostics framework, however, is still great. The only real impact of the PLP problem is that network reliability cannot be guaranteed, since a packet registered as dropped can be lost in transmission, making the network seem more reliable than it is in reality. The chances of a dropped packet being lost in transmission, however, are extremely low, since the fraction of dropped packets compared to packets not dropped is extremely small. Further, the number of data points lost in transmission is also extremely small compared to the number of data points that are successfully transmitted, making the odds of a dropped packet data point being a victim of the PLP, extremely low. With that said, the PLP did affect the results of the experiments that used the database and still requires more investigation, suggestively after the flush-interval bug has been resolved.

9.2.2 Experiment N1 - Comparison test

The comparison test showed a low deviation between UL3T and ULT network. In almost all test cases, the number of processed packets in both network directions is differentiated by 1-4 packets. This difference was likely caused by the unsynchronised timestamp of the first and last measurement data point in the UL3T and ULT Network. Thus, either one of the data files could have fewer or more packets relative to the other. Another possible cause could be that the ULT Network consumes the first pair of packets to get the correct mac addresses for filtering. Attempts were made to synchronise the measurement data by picking the same timestamp for the first and last measurement data point. However, since there were delays when the packets got registered in each program, and because the ET2000 timestamps could not be used due to the two different ET2000 devices not having synchronized timestamps, it was not possible to get a perfect synchronisation for the start and endpoints between the datasets. The difference in mean latency and standard deviation was most likely caused by having two ET2000 devices connected in series. Hence, it added a delay to packets being registered in either program.

To combat the aforementioned issue with dataset synchronization and latency deviations caused by the two ET2000 connected in a series, one of the experiments was repeated with a different topology. In the WiFi 6 with Modbus TCP experiment, this new topology was used and a more detailed analysis of the results could then be made. To remove the effects that the PLP could have on the results, this experiment also used text files stored locally, which verified that some packets were, indeed, lost during the transmission to the database. The timestamps were exactly synchronized since the exact ET2000 timestamps could be used, which gave the same number of packets processed for the two tools if the local text files were used instead of the data in the database. The latency mean and standard deviation, however, differed a lot. From an extensive analysis of the data from the two tools, it was shown that the data in UL3T had some unnatural

latency patterns that are very likely to be caused by an unknown bug in UL3T, making the latency calculations in UL3T unreliable. In the data produced by ULT, a more natural pattern was shown in the places where the unnatural patterns occurred in UL3T, further strengthening the belief that the UL3T latency calculations are inaccurate and in turn would explain the rampant latencies values reported by the UL3T tool in some of the experiments, such as the experiment with the result presented in Table 9 and some other earlier experiments that did not make it to the report. The extensive analysis also showed that UL3T registered more dropped packets than ULT, which could be an effect of the aforementioned bug.

Since the experiment with the deviating topology proved to be much more useful than the other experiment setups, it would be very beneficial to redo all experiments using the same topology: Running the two tools in parallel on the same computer with only one ET2000 as a source for the sniffed packets. Due to time constraint this was not possible during this thesis work, thus making it a suitable task for future work.

Overall, the ULT Network tool has shown in these experiments that it can monitor various network topologies and calculate an accurate latency compared to UL3T. However, only a handful of network topologies were tested, and more extensive testing would be required to see how well the tool can perform in other cases.

9.2.3 Experiment N2 - Long-term test

During the long-term test, the ULT Network stopped recording packets after around 60h had elapsed. This was cross-checked with UL3T to eliminate any faults caused by the measurement tool. In both the ULT Network and UL3T, the recording of data had stopped simultaneously. The root cause had been an issue in the 5G network that disconnected the controller and the drive. Even though this abruption occurred in the network, the ULT Network managed to record packets for an extensive period. When comparing the results shown in table 12, the most notable difference is the total amount of processed packets and packet drops. Due to the measurement being started manually for the UL3T and ULT Network, there would be some difference in the total amount of packets recorded, which is present in this case. However, there is a more significant difference in total processed packets on the drive to controller side. The most likely explanation for this difference is the PLP. Another less likely explanation that could account for some of the packet difference for ULT Network could be the database write buffer not being flushed before the application was stopped, which would result in a few packets not being sent to the database. Another noticeable difference is the number of packet drops, where the ULT Network recorded 32 and 19 packet drops in either direction while UL3T recorded none. These packet drops occurred at the start of the experiment, as shown in figure 18. One likely explanation could be a hiccup in the ULT Network tool or on the host computer. This could be the case because if there would be a fault in the ULT Network that caused sporadic packet drops, then it should be likely to have occurred throughout the experiment and not only at the start.

For the overall performance of the experiment, the ULT Network seemed to handle it well, with a varying difference between 0.001%-0.056% in most metrics compared. However, more extensive testing would be beneficial to validate the tool even further.

9.2.4 Experiment N3 - Network stress test

The results of the network stress test showed that the ULT Network could capture the number of expected packets per second. However, as shown in table 13, there are no reported results for the actual cycles per second and total number och cycles for the ABB PLC controller. This was because there was no way to verify the variation of the cycle time in the controller used. The difference between the expected and actual number of packets captured derived from the ULT Network program not stopping its execution at the exact time of the one hour mark. Thus, the program captured and reported some extra packets, which was expected. The conclusion drawn from this test is that the ULT Network tool should be able to capture all network packets generated by a controller and drive running at a 1 ms cycle time. However, future stress tests with controllers that can have their cycle times verified could add more confidence to the feasibility of the ULT Network, and the results of future tests.

9.3 Diagnostics framework

In the following sections, the results from experiments that are related to the diagnostics framework as a whole will be discussed. The results that are individually discussed were presented in Section 8.3.

9.3.1 Experiment DF1 - Framework test

The results from experiment DF1 showed that both the ULT Computing and Network probe could accurately measure the controller's cycle time and network latency. However, the most notable difference in the results was the total number of recorded packets by the ULT Network probe compared to the expected by the controller. A likely cause of these differences was due to the PLP, covered in section 9.2.1. Furthermore, observing the dropped packets in both network directions shows that the ULT Network probe recorded timed out packets on three occasions, which corresponds to the intentional disconnections of the network. Thus, the timed out packets were correctly captured and displayed in the Grafana interface. Observing the recorded send interval of the controller and the drive shows that the ULT Network probe estimated a mean value close to the expected cycle time of the controller process. Overall the diagnostics framework has shown to be capable of recording computing and network data and displaying the measurement data live without any difficulties.

9.3.2 Experiment DF2 - Database stress test

In the stress test of the diagnostics framework, more precisely, the database, the results showed that the database was able to capture both system calls and network packets at the expected rate. There were, however, some findings that need to be addressed.

In the first part, the simulated controller's actual number of cycles per second was 1008 ± 2 cycles, most likely due to the simulated controller not operating at the expected speed. Experiment C3 gives a more in-depth explanation of this phenomenon. The actual number of cycles of the simulated controller would amount to 7 256 964 system calls. However, the database reported only 7 256 000 captured packets. The most likely cause is the PLP, covered in Section 9.2.1. For future work on the ULT computing tool, it would be beneficial to ensure that all data in the query buffer gets uploaded before the program shuts down.

In the second part, the actual number of cycles per second and the total number of cycles by the ABB PLC controller could not be verified. The cause was the same as in the findings of Experiment N3. The database reported that it captured 5000 ± 1 packets per second, which is in the expected range. The total number of captured packets shows that the database registered 2696 fewer packets than expected. This could be due to the ABB PLC controller not operating at exactly 1ms. Another cause could also be the PLP. Since it was not possible to verify the actual cycle time of the controller, this remains uncertain. Further testing would be needed to strengthen the cause of these results.

9.4 Thesis work

The purpose of the experiments performed in this thesis work was two-fold: First and foremost it was to validate the feasibility of the proposed method, which is very important to answering the research question **RQ2**. Secondly, it was to prove that the diagnostics framework is working as intended, especially the ULT Network tool in the framework, which is very important to ABB CRC. It is fortunate that the two purposes were aligned so that most of the work done during the thesis benefits both academia and the industry. A healthy symbiosis.

By studying the literature and identifying the current state-of-the-art in evaluation of latencies in distributed systems, the first research question can be answered. To the best of the author's abilities, no established practice for measuring latencies in distributed control systems was found, not in the industry or academia. There are, however, several established practices in performance evaluations of networks and processes that can be applicable to the problem when combined, after some adaptations. Cilium [11], for example, is one of the more widespread analysis tools native to Linux systems. Another example is Sysdig [7], which also is capable of monitoring and analysis of

processes and containers. Both Cilium and Sysdig are based on eBPF which seems to be the most mature and feature-rich foundation when it comes to, among other things, unobtrusive monitoring of processes. Further, the literature study showed that Linux is, by far, the more favorable platform for process monitoring, mainly due to eBPF support on Windows being so lackluster compared to the well-established integration in Linux. When it comes to unobtrusive network analysis, the state-of-the-art and the state-of-the-practice seem to be fairly aligned: In research, it has become more common to investigate software-based solutions running on commercial off-the-shelf hardware to measure latencies in a network [26]. The more favored approach, however, seems to be based on FPGAs with customized software [26]. In the industry, a common method for network analysis is using network probing hardware, such as the ET2000, with customized software, which also has been adopted in research conducted in the industry such as in previous thesis work [10], and the work done in this thesis.

From the results and discussions presented here and in the previous chapter, the second research question can be answered. The proposed method is, indeed, a feasible way to unobtrusively evaluate the performance of a distributed control system live. An eBPF program that computes the duration of sleep and wake cycles by utilizing certain system call tracepoints proved to be a feasible approach to monitoring computing latencies in the controller application. Wrapping the eBPF program in a Python application that parses the data produced by the eBPF program and then sends it to an InfluxDB time-series database—a task too complex for eBPF—was also a feasible approach that did not impede the fast and lightweight eBPF program in any way. The computing probe handled large amounts of data from a 1 ms cycle time controller application without any issues, just as the InfluxDB database did once the samples were sent in batches instead of individually. The ULT Network tool, which was inspired by the network probe from previous thesis work, also proved to be a feasible approach from the extensive testing that was conducted with the tool. Since the proposed ULT Network tool utilized horizontal scaling, several metrics could be computed simultaneously, even while cycle times as low as 1 ms in the controller application was used. When the entire diagnostics framework was tested, the database did not have any problems handling all the data, and the visual interface succeeded in presenting the data live, with only a few seconds between each update, further proving the feasibility of the InfluxDB database and Grafana visualization.

Seeing as both research questions were satisfyingly answered and the thesis work resulted in a working prototype of a diagnostics framework that can be used as a research infrastructure for future work on the CFA testbed, the goals of this thesis have been achieved. The novel parts of the diagnostics framework were proven to work by the experiments that were conducted, and the network probe that was similar to the probe used in previous thesis work [10] did also prove to be feasible through experiments evaluating its performance, and verification tests with the old network probe.

Early in this thesis work, the scope of the expected work was not well defined and it was a bit unclear what work was needed and how that work would contribute to the thesis. Unfortunately, the first few weeks were spent on various tangents that had little to no contribution to the thesis work. The good side of it is that despite the fact that knowledge attained from that period was useless to the thesis, it was still interesting knowledge that could be used in other work in the future. The time spent on tangents was very low compared to work that directly contributed to the thesis, so the final results of the thesis did not suffer from the deviating work.

10. Conclusions

The goal of this thesis work was to propose a method for unobtrusive live evaluation of latencies in distributed control systems. The proposed method would be used in the CFA testbed, which is an emerging research platform for the evaluation of technologies used in distributed control systems at ABB CRC. Although the testbed would consist of many different technologies and systems, only a subset was focused on when considering the main goal of this thesis. For the purpose of achieving the main goal, two research questions were asked in this thesis: (i) "What is the state-of-the-art related to measurements of latencies in distributed control systems?", and (ii) "How can the performance of a distributed control system, consisting of the defined building blocks, be unobtrusively evaluated live?". The first question was answered by a literature study. The conclusion of the study was that, for process monitoring, eBPF based solutions are the commonly used state-of-the-art, and for network monitoring, commercial off-the-shelf systems hosting a software-based solution, or hardware-based solutions using dedicated monitoring hardware, often in the form of an FPGA, was commonly used. The second research question was answered with a system development research method where an unobtrusive diagnostics framework was developed for the CFA testbed and was then evaluated by several experiments that confirmed the feasibility of the proposed diagnostics method. In the diagnostics framework proposed in this thesis, a hardware-based solution in the form of a multi-channel Ethernet probe was used with custom-made diagnostics software to monitor the network traffic and compute KPIs for a distributed control system. Further, the sleep and wake cycles of the controller application in the distributed control system are also monitored by the proposed CFA Diagnostics framework by utilizing an eBPF application developed for this purpose. The measured data is sent to an InfluxDB database so that a graphical interface, which was also integrated into the framework, can present the data in real-time. The CFA Diagnostic framework and its components were extensively tested in several experiments that proved the feasibility of the proposed diagnostics method. The experiments also led to the discovery of a bug related to the InfluxDB database, which caused some packets to sporadically disappear in the transmission to the database. Although the bug affected the verification of the framework and results of the experiments, the feasibility of the proposed method was still confirmed and the usefulness of the CFA Diagnostics framework as a research foundation was proven.

11. Future Work

In this section, the authors will propose future work that can be done in the field, and more precisely, in the proposed diagnostics framework. The proposed future work is mostly based on ideas and wishes that were not within the scope of this thesis but, nonetheless would be a great contribution to the diagnostics framework and, in turn, the CFA testbed.

Although extensive work has been done in evaluating the diagnostics framework, more evaluations are beneficial to further guarantee that the data produced by the framework is valid. Suggestively, testing the framework on more CFA scenarios could reveal which protocols that require more development before they can be officially supported by the framework. Together with testing more scenarios, more network protocols must be incorporated into the ULT Network tool and then validated. Similarly, it could be interesting to implement more metrics into the ULT Network tool, e.g., network bandwidth or packet rate to identify how heavy the load is for the network. On the topic of metrics, the plant response latency metric needs to be investigated before it can be relied upon in any real tests. Currently, too little is known about the behavior of the metric and the different CFA scenarios for it to be relied upon. What is known, however, is that the usefulness of the metric is very dependent on the characteristics of the communication.

Containerization of the ULT Computing probe would be highly beneficial for its portability, thus making it something that is well worth investigating. Some small unofficial tests were done with the probe containerized, but not enough data was collected to draw any conclusions. The scenarios that would be of interest, in this case, are as follows: (i) ULT Computing and controller application in two separate containers, (ii) ULT Computing in a container and controller application running directly on the OS, and (iii) vice versa. Container orchestration also needs to be considered and evaluated. Does the ULT Computing probe behave the same in a stand-alone container as it does in an orchestration framework?

Since a reliable method for incorporating eBPF applications in a diagnostics framework has been proposed in this thesis, more individual computing probes can be developed and incorporated into the diagnostics framework in a similar fashion to the ULT Computing probe. The possibilities are almost endless when it comes to what data can be produced by eBPF-based computing probes. At least when running on Linux-based systems. Different CFA scenarios may require different probes, meaning that for each scenario, it is beneficial for the researchers conducting the test to carefully consider what computing metrics could be of interest and then implement a probe that produces such metrics. Some computing probes that could be of interest to several CFA scenarios are page faults and last-level cache misses for the controller application process. Since those metrics could reveal information about how efficiently the controller application is running, which is interesting from both an energy consumption perspective and an execution time perspective, implementing said probes and incorporating them in the diagnostics framework, with the proposed method, is a suitable task for future work. Not only will it improve the capabilities of the framework, but also further verify the proposed method of incorporation and be a learning opportunity for the researcher or engineer that will use the framework.

When it comes to data visualization, the framework would benefit from more KPIs being presented in Grafana. Currently, only a few basic metrics are illustrated, but since the raw data is available to the visualization tool, more complex KPIs that are based on the available data can be illustrated. Some examples are latency bound and network availability. Further, the status of the 5G network is also something that would be of interest in displaying, seeing as the 5G network sporadically degrades to 4G LTE sometimes, which can highly impact the test results. The 5G provider has incorporated means of retrieving such information directly from the 5G base station, so it is suggested that a small application that continuously monitors the status of the station and, for record-keeping, stores the information directly in the InfluxDB database, is implemented. The Grafana front-end can then easily retrieve and display the network status, and test intervals with degraded performance can immediately be checked against the network status graph to identify obvious causes and relationships visually.

Last, but not least, is the packet loss problem (PLP), presented in Section 9.2.1. As mentioned in the section that presented the problem, further investigation is needed. The flush-interval bug was reported to the developers of InfluxDB Client for Python, and it is recommended that the investigation of the PLP is postponed until the issue has been resolved. The issue is labeled number

436 and can be followed on the InfluxDB Client GitHub. Further work can be determined on, if, and how, the issue is resolved. Another alternative is to investigate a different database solution that might be free of the inconvenient bug that is present in the InfluxDB client for Python.

References

[1] M. L. Massie, B. N. Chun and D. E. Culler, 'The ganglia distributed monitoring system: Design, implementation, and experience,' *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[2] E. Imamagic and D. Dobrenic, 'Grid infrastructure monitoring system based on nagios,' in *Proceedings of the 2007 workshop on Grid monitoring*, 2007, pp. 23–28.

[3] *Datadog - cloud monitoring as a service*, 2022. [Online]. Available: `https://www.datadoghq.com/`.

[4] *New relic - application performance monitoring*, 2022. [Online]. Available: `https://newrelic.com/products/application-monitoring`.

[5] *Prometheus - monitoring system time series database*, 2022. [Online]. Available: `https://prometheus.io/`.

[6] L. Deri, S. Sabella, S. Mainardi, P. Degano and R. Zunino, 'Combining system visibility and security using ebpf.,' in *ITASEC*, 2019.

[7] *Sysdig, security tools for containers, kubernetes, cloud*, 2022. [Online]. Available: `https://sysdig.com/`.

[8] *Weaveworks - weave scope*, 2022. [Online]. Available: `https://www.weave.works/oss/scope/`.

[9] J. F. Nunamaker Jr, M. Chen and T. D. Purdin, 'Systems development in information systems research,' *Journal of management information systems*, vol. 7, no. 3, pp. 89–106, 1990.

[10] K. Bhimavarapu, 'Performance analysis and improvement of 5g based mission critical motion control applications,' 2022. [Online]. Available: `https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1642372`.

[11] *Cilium - linux native, api-aware networking and security for containers*, 2022. [Online]. Available: `https://cilium.io/`.

[12] *Bpftrace*, 2022. [Online]. Available: `https://bpftrace.org/`.

[13] *Github - microsoft/ebpf-for-windows: Ebpf implementation that runs on top of windows*, 2022. [Online]. Available: `https://github.com/microsoft/ebpf-for-windows`.

[14] *What are industrial communication networks? an overview*, 2022. [Online]. Available: `https://www.electricaltechnology.org/2016/12/industrial-communication-networks-systems.html`.

[15] B. Drury, *Control techniques drives and controls handbook (2nd Edition)*, 35. IET, 2009.

[16] *An introduction to modbus tcp/ip*, 2022. [Online]. Available: `https://www.rtautomation.com/technologies/modbus-tcpip/`.

[17] *Ibm - containerization explained*, 2022. [Online]. Available: `https://www.ibm.com/cloud/learn/containerization`.

[18] *Docker*, 2022. [Online]. Available: `https://www.docker.com/`.

[19] K. Murillo, *Containerization explained: What it is, benefits and applications*, 2019-03-26 2019. [Online]. Available: `https://www.masterdc.com/blog/what-is-containerization-benefits-explained/`.

[20] L. Kang, C. Richard, B. Hans-Peter, C. Dave, P. Zhibo and V. Inaki, *Reliable, high-performance wireless systems for factory automation*, en, 2020-09-18 2020. [Online]. Available: `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930630`.

[21] S. Gudi, S. Ojha, J. Clark, B. Johnston and M.-A. Williams, *Fog robotics: An introduction*, en, 2017-09-24 2917. [Online]. Available: `https://opus.lib.uts.edu.au/handle/10453/120748`.

[22] P. Skarin, W. Tärneberg, K.-E. Årzen and M. Kihl, 'Towards mission-critical control at the edge and over 5g,' pp. 50–57, 2018. DOI: `10.1109/EDGE.2018.00014`.

[23] P. Skarin, 'Control over the cloud: Offloading, elastic computing, and predictive control,' Ph.D. dissertation, Lund University, 2021.

[24] O. Edfors, 'Lumami-a flexible testbed for massive mimo,'

[25] M. S. Aslanpour, S. S. Gill and A. N. Toosi, 'Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research,' *Internet of Things*, p. 100 273, 2020.

[26] M. Ruiz, J. Ramos, G. Sutter, J. E. Lopez de Vergara, S. Lopez-Buedo and J. Aracil, 'Accurate and affordable packet-train testing systems for multi-gigabit-per-second networks,' *IEEE Communications Magazine*, vol. 54, no. 3, pp. 80–87, 2016. DOI: 10.1109/MCOM.2016.7432152.

[27] J. E. López de Vergara, M. Ruiz, L. Gifre *et al.*, 'Demonstration of 100 gbit/s active measurements in dynamically provisioned optical paths,' *IET Conference Proceedings*, 4 pp.–4 pp.(1), Sep. 2019. DOI: 10.1049/cp.2019.1187.

[28] N. Ruiz and D. Mario, 'On the exploration of fpgas and high-level synthesis capabilities on multi-gigabit-per-second networks,' *Department of Electronic and Communications Technology*, Jan. 2020.

[29] R. Brondolin, M. Ferroni and M. Santambrogio, 'Performance-aware load shedding for monitoring events in container based environments,' *SIGBED Rev.*, vol. 16, no. 3, pp. 27–32, Nov. 2019. DOI: 10.1145/3373400.3373404. [Online]. Available: https://doi.org/10.1145/3373400.3373404.

[30] R. Brondolin and M. D. Santambrogio, 'A black-box monitoring approach to measure microservices runtime performance,' *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Nov. 2020, ISSN: 1544-3566. DOI: 10.1145/3418899.

[31] A. Noor, D. N. Jha, K. Mitra *et al.*, 'A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments,' in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 156–163. DOI: 10.1109/CLOUD.2019.00035.

[32] F. Pina, J. Correia, R. Filipe, F. Araujo and J. Cardroom, 'Nonintrusive monitoring of microservice-based systems,' in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548311.

[33] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin and J.-Y. Jeng, 'A kubernetes-based monitoring platform for dynamic cloud resource provisioning,' in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254046.

[34] D. Masouros, S. Xydis and D. Soudris, 'Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2021. DOI: 10.1109/TPDS.2020.3013948.

[35] *Strace - linux syscall tracer*, 2022. [Online]. Available: https://strace.io/.

[36] *Dtrace - dynamic tracing*, 2022. [Online]. Available: https://docs.oracle.com/cd/E37670_01/E37355/html/ol_about_dtrace.html.

[37] Å. Anand, *An intro to using ebpf to filter packets in the linux kernel*, 2017. [Online]. Available: https://opensource.com/article/17/9/intro-ebpf.

[38] *Pcapplusplus, a multiplatform c++ library for capturing, parsing and crafting of network packets.* 2022. [Online]. Available: https://github.com/seladb/PcapPlusPlus.

[39] *Therouter is a software packet router based on dpdk an npf libraries.* 2022. [Online]. Available: https://github.com/alexk99/the_router.

[40] *Bpf filter syntax taken from tcpdump man page*, 2022. [Online]. Available: https://biot.com/capstats/bpf.html.

[41] *Influxdb*, 2022. [Online]. Available: https://www.influxdata.com/products/influxdb-overview/.

[42] *Timescale database*, 2022. [Online]. Available: https://www.timescale.com/.

[43] *Griddb*, 2022. [Online]. Available: https://griddb.net/en/.

[44] *Opentsdb*, 2022. [Online]. Available: http://opentsdb.net/.

[45] *Openstack*, 2022. [Online]. Available: https://www.openstack.org/.

[46] *Github*, 2022. [Online]. Available: https://github.com/.

[47] *Grafana*, 2022. [Online]. Available: https://grafana.com/grafana/.

[48] *Open benchmark of influxdb*, 2022. [Online]. Available: https://openbenchmarking.org/test/pts/influxdb&eval=a25dcb4af79b1201aa71d10f3cc27565ebf52226#metrics.

[49] *Kibana*, 2022. [Online]. Available: https://www.elastic.co/kibana/.

[50] *Flux - query language in influxdb*, 2022. [Online]. Available: https://docs.influxdata.com/flux/v0.x/.