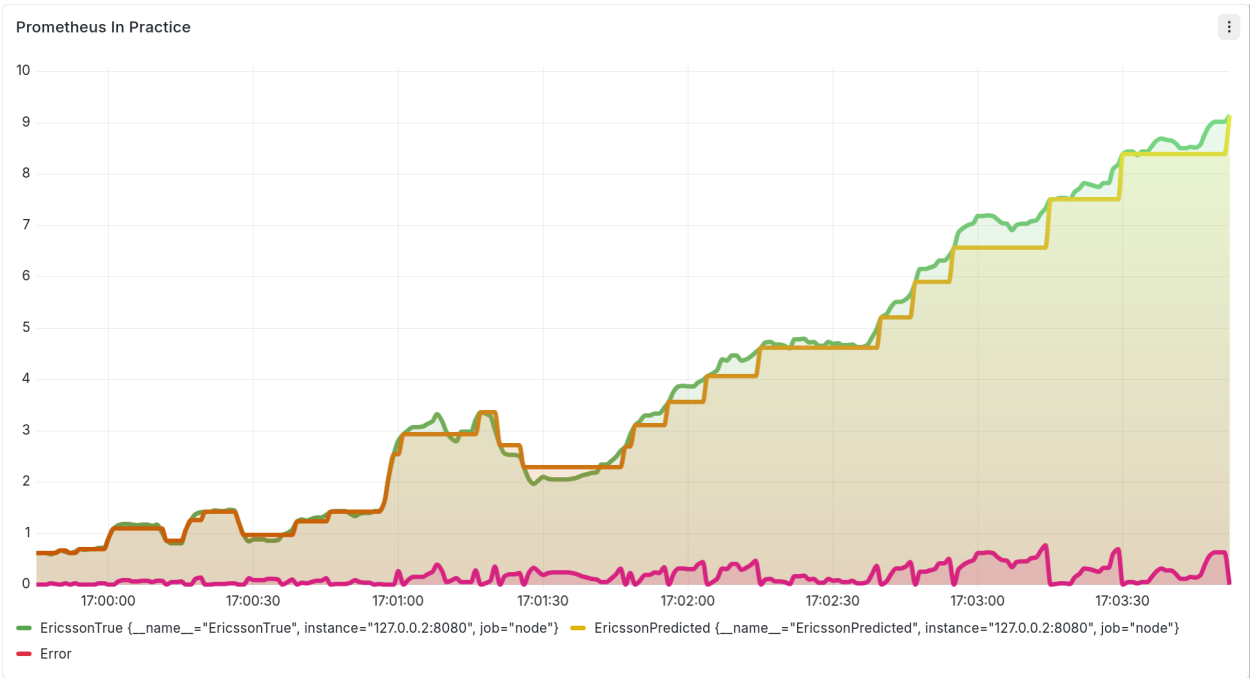

Prometheus in Practice

Communication-Efficient Monitoring on Distributed Resource-Constrained Systems

Master's thesis in Computer science and Engineering

Emmy Andreasson
Albin Hallin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Prometheus in Practice

Communication-Efficient Monitoring on Distributed Resource-Constrained Systems

EMMY ANDREASSON
ALBIN HALLIN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Prometheus in Practice
Communication-Efficient Monitoring on Distributed Resource-Constrained Systems
EMMY ANDREASSON, ALBIN HALLIN

Supervisor: Yixing Zhang, Department of Computer Science and Engineering
Advisor: Martin Forssén, Recorded Future
Examiner: Romaric Duvignau, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Prometheus in Practice
Communication-Efficient Monitoring on Distributed Resource-Constrained Systems
EMMY ANDREASSON
ALBIN HALLIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous Distributed Monitoring (CDM) aims to reduce the communication overhead associated with the monitoring of a distributed system. This thesis investigates forecast-based monitoring techniques as a practical solution to the CDM problem by applying them in a Prometheus-based monitoring framework. The investigation resulted in a proxy-based framework that interfaces with Prometheus and implements two prediction models, Simple Approximation and Piecewise Linear Approximation. Evaluation was conducted in two phases: one simulation-based phase and one hardware-based phase, in which the framework was ported to a physical test bench of Raspberry Pis. The final framework was evaluated in terms of communication ratio, mean absolute error, update latency, as well as CPU and memory usage. On average, the final results indicated a 77.4% decrease in transmitted network packets while keeping the mean error at 1.13%.

Keywords: distributed monitoring, continuous distributed monitoring, forecast-based monitoring, Prometheus

Acknowledgements

We would like to express our sincere gratitude to our supervisor Yixing Zhang, our examiner Romaric Duvignau and our company advisor Martin Forssén for their invaluable support and feedback during this project. Their contributions were essential to the completion of this thesis.

Emmy Andreasson, Albin Hallin, Gothenburg, 2026-07-03

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Problem Description . . . 1
1.2 Purpose and Objectives . . . 2
1.3 Limitations . . . 2
1.4 Report Outline . . . 3

2 Theory **5**
2.1 Distributed Monitoring . . . 5
2.2 Prometheus . . . 6
2.2.1 Server . . . 6
2.2.2 Jobs . . . 7
2.2.3 Client Libraries and Exporter Modules . . . 8
2.2.4 Prometheus Metric Types . . . 8
2.2.5 Configuration . . . 9
2.3 Error-Bounded Forecast-based Monitoring . . . 9
2.4 All Values Tracking . . . 10
2.5 Related Work . . . 10
2.5.1 The Continuous Distributed Monitoring Problem . . . 10
2.5.2 Other Monitoring Tools and Prometheus Selection . . . 11
2.5.3 Research Gap . . . 12

3 Methods **13**
3.1 Proxy-based Architecture . . . 13
3.2 Communication . . . 13
3.3 Prediction Models . . . 14
3.3.1 Simple Approximation . . . 14
3.3.2 Piecewise Linear Approximation . . . 14
3.4 Determining the Error Threshold: Epsilon . . . 15
3.4.1 Static Epsilon . . . 15
3.4.2 Dynamic Epsilon . . . 15
3.5 Evaluation Metrics . . . 15
3.5.1 Communication Ratio . . . 16

3.5.2 Mean Absolute Error . . . 16
3.5.3 Latency . . . 17
3.6 Evaluation Methodology . . . 17
3.6.1 Measuring Performance . . . 17
3.6.2 Simulation-based Evaluation . . . 18
3.6.3 Hardware-based Evaluation . . . 19
3.6.4 Configurations . . . 22

4 Implementation **25**
4.1 Proxy-based Architecture . . . 25
4.1.1 Coordinator-side Proxy . . . 26
4.1.2 Node-side Proxy . . . 26
4.1.3 TestClient . . . 26
4.1.4 Configuration of Prometheus . . . 26
4.1.5 Communication and Message Types . . . 27
4.1.6 Models and Metrics . . . 28
4.1.7 Metric Priority . . . 28
4.2 Algorithm Design . . . 29
4.2.1 Initialization Phase . . . 29
4.2.2 Warm-up Phase . . . 30
4.2.3 Standalone . . . 30

5 Results **33**
5.1 Simulation-based Evaluation Results . . . 33
5.1.1 Prediction Model Performance . . . 34
5.1.2 Impact of PLA Buffer Size . . . 35
5.1.3 Impact of Dynamic Epsilon Buffer Size . . . 36
5.1.4 Impact of Epsilon Percentage Value . . . 37
5.1.5 Multiple Metrics per Node $M > 1$ . . . 38
5.2 Hardware-based Evaluation Results . . . 39
5.2.1 Federated Learning Datasets . . . 39
5.2.2 Impact of PLA Buffer Size . . . 40
5.2.3 Impact of Dynamic Epsilon Buffer Size . . . 41
5.2.4 Impact of Epsilon Percentage Value . . . 41
5.2.5 Multiple Metrics per Node $M > 1$ . . . 43
5.2.5.1 Metrics Priority . . . 43
5.2.6 CPU and Memory Usage . . . 45
5.2.7 Live Tests . . . 45
5.3 Summary . . . 47
5.4 Latency . . . 47

6 Discussion **49**
6.1 Evaluation Results . . . 49
6.2 Live Tests . . . 50
6.3 Implementation Challenges . . . 51
6.3.1 Limitations of Push-based Monitoring . . . 51
6.3.2 Prometheus Metric Coverage . . . 51

6.3.3 gzip Compression 51
6.3.4 Communication Protocol Limits 52
6.3.5 TestClient-Induced Latency Issues 52
6.4 Dataset Considerations 52
6.5 Future Work 53

7 Conclusion **55**

Bibliography **57**

A Evaluation configurations **I**

List of Figures

2.1 Overview of Prometheus system architecture. Image licensed under Apache 2.0 [1] 7

3.1 Testbench of four Raspberry Pis connected via a switch 19

4.1 Outline of the proxy-based architecture, depicting devices and respective processes 25
4.2 Byte layout of the custom protocol header (in yellow) 27
4.3 An overview of the system's communication flow 29

5.1 Comparison of true and predicted values over time (D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=1, M=1) 34
5.2 Evaluation across PLA buffer sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, L_ϵ=1000, TP=TCP, M=1, N=40) 35
5.3 Evaluation across PLA buffer sizes using the *IntelLab* dataset (D=4000s, p_ϵ=10, L_ϵ=1000, TP=TCP, M=1, N=40) 35
5.4 Evaluation across L_ϵ sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, TP=**TCP**, M=1, N=40) 36
5.5 Evaluation across L_ϵ sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, TP=**UDP**, M=1, N=40) 37
5.6 Evaluation across p_ϵ using the *IntelLab* dataset (D=4000s, L_ϵ=300, TP=TCP, M=1, N=40) 38
5.7 Evaluation across M using the *Ericsson* dataset (D=1500s, p_ϵ=10, L_ϵ=300, TP=TCP, $N = 20$) 38
5.8 Raspberry Pi 1: CPU Usage: a) Average (150s), b) Per-core (150s), c) Average (600s), d) Per-core (600s) 40
5.9 Evaluation across PLA buffer sizes using the *CPU Usage (Averaged)* dataset (D=600s, p_ϵ=5, L_ϵ=200, TP=TCP, M=1, N=4) 40
5.10 Evaluation across L_ϵ sizes using the *CPU Usage (Averaged)* dataset (D=600s, p_ϵ=5, TP=TCP, M=1, N=4) 41
5.11 Evaluation across p_ϵ using the *CPU Usage (Averaged)* dataset (D=600s, $L_{\epsilon,\text{PLA3}}$=200, $L_{\epsilon,\text{PLA8}}$=$L_{\epsilon,\text{SA}}$=300, TP=TCP, M=1, N=4) 42
5.12 Evaluation across p_ϵ using the *CPU Usage (Per-core)* dataset (D=600s, $L_\epsilon = 200$, TP=TCP, M=4, N=4) 42
5.13 Comparison between TCP and UDP across p_ϵ using the *CPU Usage (Per-core)* dataset and model PLA-3 (D=600s, L_ϵ=200, M=4, N=4) 43

5.14 Evaluation across M using the *Ericsson* dataset (D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=4) 44
5.15 Evaluation across number of metrics with priority=1, against a baseline with only priority=1 metrics, using the *Ericsson* dataset (D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, M=150, N=4) 44
5.16 CPU and Memory Usage across M using the *Ericsson* dataset (PM=PLA-8, D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=4) 45
5.17 CPU Average Usage and Network Activity at Raspberry Pi 1 during training of CNN-Heavy (PM=SA, p_ϵ=10, L_ϵ=200, TP=UDP, M=399, N=4, δ=250) 45
5.18 Evaluation of live test across p_ϵ during training of CNN-Heavy. Each datapoint is the mean of two tests. (D=868-911s, L_ϵ=200, TP=TCP, N=4, M=19, δ=250) 46

List of Tables

3.1 Summary of *CNN-Heavy* 21
3.2 Summary of *CNN-Light* 22

5.1 Notation of the configuration parameters used in the results section . 33
5.2 Summary of the federated learning datasets 39
5.3 Mean evaluation metric measurements for the overall test set (%) . . 47
5.4 A summary of the mean (and maximum) update latency in milliseconds for different subsets of tests 48

A.1 Configurations for simulation-based evaluation II
A.2 Configurations for hardware-based evaluation III

1

Introduction

Distributed systems and services are vital pieces of our modern-day infrastructure. They power cloud computing, media streaming, financial transactions, and a myriad of other services on which we have grown reliant. As the systems grow in size and complexity, the need for effective distributed monitoring of edge devices increases. Distributed monitoring refers to the process of monitoring all components of a distributed system, granting system maintainers a comprehensive view of the system's status and enabling the collection of usage statistics and identification of potential bottlenecks. In a system with a large number of devices, this leads to a substantial communication overhead in the form of transmitted system information between nodes. To improve the performance and energy efficiency of the system, it is imperative to decrease the amount of such transmitted data without sacrificing accuracy.

Numerous open-source tools have been developed for the purpose of monitoring. One such tool is *Prometheus* [1], a systems monitoring toolkit for distributed systems where a central coordinator pulls data from instances of applications, all distributed across a network of devices. In this project, we aim to explore the possibility of increasing communication efficiency in Prometheus by building and implementing a framework that incorporates communication-efficient techniques.

1.1 Problem Description

Within large-scale distributed systems, there is a need to continuously gather information and statistics about the distributed devices in order to get a general view of the health and performance of the system. Bandwidth and energy are often constrained, which introduces a need to decrease the volume and frequency of metrics communicated while preserving statistical accuracy.

Consider a distributed system with a central coordinator, which we denote by C, along with a collection of k distributed nodes $N = \{n_0, n_1, ...n_{k-1}\}$. The system makes use of discrete monitoring rounds of duration l, where round t spans over the time interval $[t, t + l)$. The communication in the system is bidirectional and synchronous, meaning that in each round, each node in N makes a monitoring decision, deciding whether to transmit its monitoring data to C or not. The nodes may not communicate directly with each other but can communicate with C. The central task is to design and deploy methods that minimize the communication overhead

between C and N without compromising the accuracy of the metrics that C continuously aggregates and monitors. Within the literature, this is commonly known as the *Continuous Distributed Monitoring* (CDM) problem [2]. The assignment in this particular project is therefore to address the CDM problem within the confines of a Prometheus ecosystem.

1.2 Purpose and Objectives

The ultimate goal of this project is to explore and investigate the possibility of addressing the CDM problem within Prometheus by integrating and subsequently testing communication-efficient monitoring methods, focusing on forecast-based methods.

In order to achieve this, we develop a configurable framework that interfaces with Prometheus and enables easy deployment of forecast-based monitoring in the form of prediction models. The framework handles all inter-device communication with the help of a custom protocol. In parallel, we develop a comprehensive testing methodology with the responsibility of evaluating the framework and the chosen models by measuring evaluation metrics such as communication ratio, mean absolute error, memory usage, latency, and CPU usage. We compare all the results to a baseline consisting of a clean Prometheus configuration.

After an initial research phase, we implement two prediction models, selected based on implementation complexity and resource demands: Simple Approximation and Piecewise Linear Approximation. We also evaluate different variations of the custom communication protocol by varying the underlying transport protocol and different system parameters.

With the two prediction models implemented, we evaluate and analyze the framework's communication efficiency compared to that of Prometheus itself. The evaluation is conducted in two stages: first, through simulation-based evaluation, followed by hardware-based evaluation. We run the simulated tests on a single device, emulating a distributed system, whereas in the hardware-based evaluation, we deploy the framework on real-world distributed devices in the form of Raspberry Pis. After completing an extended analysis of the evaluation results, the primary aim will be considered fulfilled. To guide the research, we aim to answer the following research question:

- *How can Prometheus communication efficiency be improved by integrating forecast-based monitoring mechanisms?*

1.3 Limitations

The framework will not support the Prometheus metric types *Summary* and *Histograms*. We make no modifications to the Prometheus source code itself, again to save time and development efforts. Native integration of communication-efficient methods into Prometheus is desirable, but we leave it as part of future work. The

framework developed is to be considered a proof-of-concept product, meaning that it is not to be used in a production environment since that would require more time and effort aimed towards ensuring fault tolerance, scalability, and efficiency.

1.4 Report Outline

This report begins by explaining the concepts of distributed monitoring, how Prometheus works, and related work in forecast-based monitoring in Chapter 2. Chapter 3 outlines testing and implementation-specific details and serves as a guide on how to reproduce our results. Chapter 4 describes the implementation in more technical detail, outlining communication protocols, system phases, and more. Chapter 5 presents the results of the evaluation phase. Chapter 6 features a discussion regarding these results, and Chapter 7 finalizes the report with some concluding remarks.

2

Theory

This section serves to explain the underlying theory behind this project, from distributed monitoring to the Prometheus system to forecast-based monitoring. Finally, it concludes by examining some related work in this field of research.

2.1 Distributed Monitoring

In modern computing, systems are rarely confined to a single machine. Instead, they take the shape of a distributed system, described by Tanenbaum and van Steen [3, p.2] as "a collection of autonomous computing elements that appears to its users as a single coherent system". Thus, according to [3], they have two defining characteristics: being a collection of independent devices that can work collaboratively, and appearing to a user as a single system. Examples of distributed systems include cloud computing platforms such as Amazon Web Services and cluster computing systems such as CERN's High Performance Computing cluster [3], [4], [5].

Monitoring refers to the act of collecting and processing real-time quantitative data about a system [6]. For a single machine, this task is relatively straightforward; all monitoring data originates from the same location and can be recorded and aggregated in a centralized manner. In contrast, monitoring a distributed system requires data from multiple nodes to be transmitted and aggregated remotely, increasing the complexity and importance of the task.

Although monitoring remains a fundamental aspect in any system, the inherent collaborative nature and scale of distributed systems render it especially critical. The importance of this can be understood through three key aspects. Firstly, monitoring a distributed system is important for fault detection, as the failure of a single node has the potential to impact performance and coordination. Secondly, to fully utilize the scalability of a distributed system, monitoring can be used to provide insight into load-balancing strategies, ultimately supporting scaling decisions. Finally, monitoring can be used to support performance optimization, detecting nodes that act as bottlenecks and slow down operations.

Industry professionals adopt ready-made tools such as Prometheus [1], Sensu Go [7], InfluxDB [8], and Zabbix [9] to handle monitoring, each with slightly different approaches to collection, transmission, and storage. The common approach that the tools share is having a central node, the coordinator, collecting monitoring data from

edge nodes for further analysis [10], thereby establishing a foundational model for much of modern distributed monitoring. Whether the coordinator scrapes its nodes or receives pushed data from them determines whether the monitoring system follows a pull- or push-based architecture, respectively [10]. The choice between the two is a topic of considerable debate as each has its pros and cons. For example, push-based monitoring is well-suited for ephemeral environments as it avoids coordinator-side node tracking, whereas pull-based monitoring aids in fault detection and system stability.

Depending on the system, the monitored data takes different forms, with the most common types being metrics, which provide measurements of software or hardware properties, and logs, which are textual records documenting specific events [10]. As a monitoring strategy or tool is only as effective as the relevance and quality of the data it collects, it is of high importance that the monitored data is chosen with consideration to the system's most critical aspects. To assist with this, commonly used monitoring methodologies can be applied. Turnbull [10, p. 36] suggests Brendan Gregg's USE or Google's Four Golden signals, which provide useful baseline frameworks for monitoring of host-level performance and application-level performance, respectively. Metrics that these methodologies prioritize include utilization, saturation, traffic, errors, and latency.

Prioritizing and collecting suitable metrics enables operators to gain a dynamic, real-time picture of the state of their computing system. Additionally, monitoring frameworks commonly also allow alarms to be configured for specific events or thresholds [6], ensuring that critical issues do not go unnoticed. This visibility enables informed insights to be drawn regarding broader patterns such as performance, resource allocation, and overall system health. Furthermore, the observations can be applied to more specific tasks such as cost management, monitoring of usage statistics, and gaining live feedback on current tasks.

2.2 Prometheus

Prometheus is an open source event monitoring and alerting application written in *Golang* [1]. Prometheus features a client-coordinator architecture where a central coordinator scrapes clients for time series data periodically. Due to the coordinator being the active component, continuously querying applications running on distributed client devices, Prometheus is a pull-based monitoring system. An overview of the Prometheus architecture can be seen in Figure 2.1.

2.2.1 Server

The server component is at the core of the Prometheus architecture. This component is responsible for the periodic scraping of metric data, the management and persistence of this data within a time-series database (TSDB), and the hosting of an application programming interface (API) that enables the retrieval and querying of the stored metrics. It also hosts an expression browser and graphing interface that

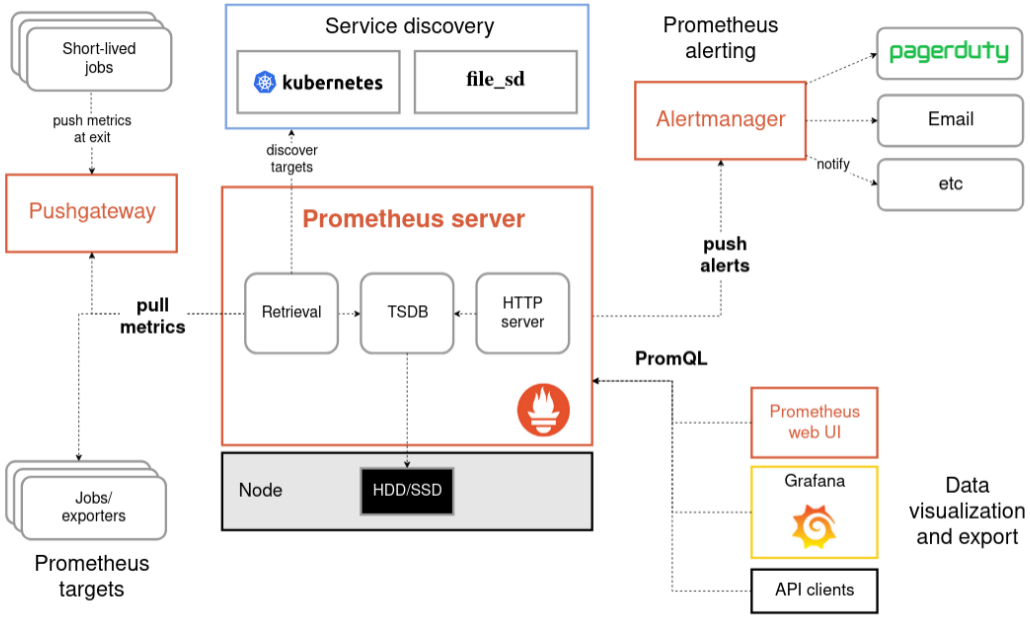

Figure 2.1: Overview of Prometheus system architecture. Image licensed under Apache 2.0 [1]

allows for a more user-friendly retrieval and visualization of metric data through a web user interface [1].

The server can be configured to periodically evaluate *rules* on the received metrics [11]. There are two types of rules: recording rules and alerting rules. The former precomputes expressions and saves the result as a new set of time series [10]. This is especially useful if the expressions are commonly used, since it saves computing power at query time as they have already been processed. The latter defines the criteria that govern when Prometheus should send an alert. One such rule could be configured to trigger an alert when the CPU temperature exceeds a certain threshold, enabling mitigation of potential future issues.

While the Prometheus server does not handle the alerting directly, it can be set up to push alert messages to an *Alertmanager*, a separate server component that can issue alerts via services like mail, *Slack*, and text messages via *Pagerduty* [12].

2.2.2 Jobs

An HTTP endpoint from which the Prometheus coordinator can scrape metric data corresponds to a single process, application, or host, and is also referred to as an *instance* [10]. Multiple instances may collectively be grouped into *jobs*, since an application might be replicated for redundancy purposes [1].

2.2.3 Client Libraries and Exporter Modules

Prometheus provides *client libraries*, which enable developers to define and expose application data through a Prometheus-compatible HTTP endpoint [13]. There currently is official support for *Python*, *Java*, and *Go* with multiple third-party libraries for other languages like *C++*, *Swift*, and *Haskell*, to name a few.

There is also a plethora of ready-made *exporter modules*, both official and third-party ones, that aid in exporting application data from a collection of well-known applications and systems such as *Apache*, *Github*, *MySQL server*, and *system information and statistics* via the Prometheus *Node Exporter* [14][15]. These are useful for situations where it is difficult to instrument data exposition through the means of client libraries.

2.2.4 Prometheus Metric Types

There are four different types of metrics that Prometheus can monitor [1]:

- Counter - A monotonically increasing numeric value. Can be used to track items such as API requests, the number of context switches, or the number of received network packets.
- Gauge - A numeric value that is not monotonically increasing, as opposed to the counter. Suitable for measuring fluctuating values like temperatures or disk usage.
- Histogram - A metric that illustrates the distribution of observations. It samples values and groups them into configurable buckets. It also includes the sum of the values measured, along with the total number of values observed.
- Summary - Similarly to histograms, summaries also sample values and provide sums of the values along with the total number of observed events. In addition, it also reveals quantile values directly on the endpoint.

Every metric is associated with a name as well as a help label that provides a short description of the metric. As shown in the example below.

```
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 5
```

This example metric is taken from Prometheus's *Node Exporter* [16], which exposes a list of system information, ranging from CPU frequency to the number of entries that reside in the device's ARP table. Furthermore, every metric may be comprised of "sub-metrics", which carry the same name but are distinguished by a set of key-value pairs. This is useful if multiple measurements of the same metric are needed, for example, the frequency scaling per CPU core as shown in the example below.

```
# HELP node_cpu_frequency_max_hertz Maximum CPU thread frequency in hertz.
# TYPE node_cpu_frequency_max_hertz gauge
node_cpu_frequency_max_hertz{cpu="0"} 4.056819e+09
```

```
node_cpu_frequency_max_hertz{cpu="1"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="10"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="11"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="2"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="3"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="4"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="5"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="6"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="7"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="8"} 4.056819e+09
node_cpu_frequency_max_hertz{cpu="9"} 4.056819e+09
```

2.2.5 Configuration

Prometheus may be configured to the administrator's preferences by editing a *.YAML* configuration file. It is in this file that static scrape-jobs, rules, and alerting behavior are defined [10].

Jobs are defined by entering the job name together with the addresses of the targets corresponding to the job. Due to the static predefined nature of the target definition, this is aptly called a *static configuration*. Additional parameters, like proxy addresses, authentication options, and job-specific scraping intervals, may also be set to further modify the behavior of the scraping.

2.3 Error-Bounded Forecast-based Monitoring

As established in 2.2, Prometheus operates by periodically scraping its targets, whereby in each scrape, all metrics for each target are retrieved. While this pull-based design entails a reliable and complete system overview, it introduces communication overhead. Addressing this limitation requires rethinking how and when communication is performed, a challenge that has motivated the development of more communication-efficient solutions.

The current state-of-the-art approach to the above is called forecast-based monitoring and is detailed in [17]. In forecast-based monitoring, predictive models are adopted and instantiated on both the coordinator and node sides. Communication savings are achieved by the node only transmitting the true value if the value predicted on both sides exceeds a certain threshold. Without an update from the node, the coordinator implicitly concludes that the value it has predicted is within the specified error threshold. That is, after each model prediction, the node performs a comparison at time t, between the predicted ($\hat{v}_t$) and the true value (v_t):

$$|\hat{v}_t - v_t| > \epsilon$$

Only if the absolute difference between $\hat{v}_t$ and v_t exceeds the error threshold ϵ, the node will transmit v_t to the coordinator.

Several different prediction models applicable to this context have been found in [2], [18], and [17]. In this work, we employ one lightweight model, along with a more complex model. The first, Simple Approximation, compares the most recent true value to the previous one, and in the case that the difference between the two exceeds ϵ, the current true value is transmitted to the coordinator. The second model is Piecewise Linear Approximation, in which the prediction is generated by calculating the slope and intercept of a linear segment from recent observations and time durations. More details on the models can be found in 3.3.

2.4 All Values Tracking

One prevalent survey by Cormode [19] outlines a continuous distributed monitoring system model similar to the one described in 1.1, where k observers monitor their own data stream $A = a_1, a_2, ..., a_n$. A bidirectional communication channel exists between a central coordinator C and every observer. The objective is for the coordinator C to calculate a function $f(A)$. Such a function can be to determine whether or not $f(A)$ has broken a threshold τ or to gather the *top K* observations made in the system. This project will instead focus on having the coordinator maintain an up-to-date view of the current value of all nodes. This is also known as the *All Values Tracking problem* (AVT) [2] and serves as a more general approach to the CDM problem where no particular aggregating function f is considered. This approach is also more suitable when building a system that aims to interface with an existing monitoring tool like Prometheus. Instead, the end user can choose what function to apply to the gathered data.

Forecast-based monitoring naturally fits the criteria of AVT, since it allows the coordinator to maintain an up-to-date view of all metrics for each node. Thus, for the purposes of this project, it is a suitable method to implement in Prometheus, which requires a complete system overview.

2.5 Related Work

Communication-efficient monitoring of distributed systems has been approached through research on the Continuous Distributed Monitoring (CDM) problem, but also through the development of practical monitoring tools.

2.5.1 The Continuous Distributed Monitoring Problem

Numerous approaches and methodologies have been studied within the research field of Continuous Distributed Monitoring (CDM), all with the common aim of reducing communication costs. These approaches utilize different system models with varying requirements concerning monitoring capabilities and functionality.

Geometric Monitoring is a family of techniques that aims to reduce communication in a network by reducing the task of monitoring complex, non-linear functions to the maintenance of local constraints [20]. As opposed to a naive central algorithm, the

geometric monitoring approach entails evaluating constraints locally and effectively filtering out data that does not affect the global monitoring outcome. Two versions of the algorithm are described: a decentralized algorithm in which distributed nodes broadcast their update messages, and a centralized one where a central coordinator is present. There have also been efforts made to minimize the size of the update messages down to a single scalar [21].

The method of utilizing prediction models on both the node and server-side is referred to as the Dual Prediction Scheme (DPS) [17] and is a common method of choice within the research field. Some examples of applications of DPS can be found in the work by Jain et al. [22] where a *Kalman filter* was deployed in a DPS fashion. The system can adapt to the characteristics of the data stream and allows caching of filter parameters instead of static data. This enables the system to produce more dynamic predictions without requiring additional information from the node. Another notable work is that of Jiang et al. [23], in which a framework is described that decides what mode a given node should take based on an analysis of the potential savings in terms of energy. It is able to completely turn off predictions and transmit every value if the deemed extra computational cost of predictions outweighs that of the communication overhead.

A central guiding framework for this work that also utilizes DPS is *FEDAMON* by Zhang and Duvignau [2]. FEDAMON proposes a forecast-based monitoring framework that addresses the problem of CDM and AVT through the use of adaptive filtering via DPS and a data-aware model selection algorithm. The framework accounts for data variation and evolution through the selection of the optimal prediction model and dynamic tuning of the model's parameters. The authors measured the communication ratio along with mean absolute error and found that the framework resulted in a 90% reduction in communication when compared to the baseline and less than 2% average error across the monitored streams.

A more hands-on study is that of Koerner et al. [18], where both adaptive filters and adaptive sampling methods were utilized. The authors ran tests on a physical test bench consisting of 39 Raspberry Pis connected in a tree network topology. The paper is a useful deployment guide and goes into great detail about how to run and subsequently evaluate the performance of the models on a physical test bench.

2.5.2 Other Monitoring Tools and Prometheus Selection

In addition to Prometheus, several other tools have been built to address similar objectives, but differ in architectural design and data collection methods.

Sensu Go [7] is an open-source observability pipeline that adopts an agent-based solution to offer a complete overview of an organization's infrastructure through integration with widely used monitoring tools. It operates using agent entities deployed on each target to be observed, which in turn are members of one or several subscriptions. Subscriptions detail the roles and responsibilities of a certain agent. Upon executing a task specified by the subscription, the agent entity pushes the result back to the Sensu backend, making Sensu Go a largely push-based system.

InfluxDB [8] specializes in time-series data and is designed to collect, store, and process such data. In this tool, data is organized into buckets and measurements, where a bucket can hold multiple measurements, allowing for logical separation of data. Typically, data is pushed directly from a device to the REST API, which is exposed directly in front of InfluxDB [24]. The pipelining of input can be further improved with the help of client libraries or InfluxData's agent-based solution Telegraf, both of which can transform incoming data to the correct format for InfluxDB and handle large volumes of data by batching it and holding on to it in case connectivity is lost.

Zabbix [9] is a monitoring tool aimed at enterprise-class systems and is, like Prometheus, based on a server-client architecture. The Zabbix server stands in the center of the system. It is the central component of the system and is responsible for polling, alerting on, and visualizing data. Deployed on the monitored targets are the Zabbix agents. They gather information about the system and applications through native system calls and report it back to the server.

Prometheus [1] was selected as the monitoring solution for this project due to its broad adoption in the industry and its extensive, active community support. Additionally, the quality of its official documentation is characterized by a more intuitive organization and a higher degree of clarity and comprehensibility relative to alternative solutions like those listed above. Furthermore, the method of exposing metrics via an HTTP endpoint and the native proxy configuration options for the Prometheus server made it a suitable foundation on which to build a proxy-based system. In addition to this, Prometheus has client libraries that further facilitate development since they simplify exposing application data as Prometheus metrics. Another advantage was that Prometheus is implemented in Golang, a programming language we are experienced with.

2.5.3 Research Gap

While there is plenty of research within continuous distributed monitoring, even within a more specialized niche such as forecast-based monitoring, there is a gap in the research when looking at the context in which the methods are implemented and evaluated. While plenty of solutions evaluate their methods in simulation-based environments or physical test benches over custom communication frameworks, we have found no research where they are evaluated by being incorporated into monitoring tools like Prometheus or others like the ones mentioned in Section 2.5.2.

The implications of deploying such methods into a real-world monitoring tool remain unknown, including how the monitoring tool's implementation itself may affect the methods, to what extent such methods can be integrated, and what parameters require particular consideration. The latter motivates the experimentation conducted in Section 5, which aims to systematically vary configuration parameters to evaluate their impact on a framework for communication-efficient methods in an existing monitoring tool.

3

Methods

This chapter serves to explain and motivate the methods employed to meet the project goals. It will start by outlining some system design choices, then describe the prediction models, and lastly explain how the evaluation phase was conducted.

3.1 Proxy-based Architecture

The implementation was designed to remain invisible to Prometheus, resulting in a proxy-based architecture that resides between the Prometheus coordinator and the nodes. It consists of proxies running on both the coordinator and the node sides, referred to as the coordinator-side proxy and the node-side proxy, respectively. Inter-device communication is handled by these proxies, and they implement and execute forecast-based prediction on the Prometheus metrics.

This design is motivated by the need for flexibility and simple implementation. Development of a proxy-based architecture removes the need to modify and add to the Prometheus source code. There are a multitude of benefits that stem from this. Firstly, it removes the possibility of introducing bugs into the core Prometheus implementation, which could have major negative impacts on the functionality of the proxy-based framework and would require great debugging efforts. Secondly, it enables end users to slot the framework into their existing Prometheus ecosystem without any major reconfiguration. Thirdly, it makes the implementation more resilient to changes and updates to Prometheus. Lastly, it improves the efficiency of the research, as there is no need to dive into the Prometheus source code.

3.2 Communication

As mentioned in Section 2.2.2, Prometheus utilizes HTTP endpoints exposed by its targets, from which metrics are collected. To realize the proxy-based architecture correctly, this HTTP scraping is preserved within the node and coordinator components. However, communication *between* proxies is handled separately and can be configured to use either TCP or UDP.

TCP [25] offers connection-oriented, reliable, and in-order communication, whereas UDP [26] provides connectionless, fast delivery of messages, without delivery or

in-order guarantees. By incorporating both of these commonly used transport protocols, the framework could be tested under more specific real-world scenarios, while covering two transport-layer protocols with fundamentally different design philosophies. This enabled systematic testing of how the full system's communication efficiency and robustness may or may not be affected by the type of transport protocol applied.

3.3 Prediction Models

We adopted two prediction models to evaluate performance under varying data stream characteristics. The prediction models were chosen due to their low complexity and implementation difficulty, in addition to being commonly used in related research [2], [18].

3.3.1 Simple Approximation

Simple Approximation [2] is the simplest model employed. It consists of keeping track of the last value transmitted to the coordinator, v_l. The predicted value for monitoring round t, $\hat{v}_t$, is then defined as follows:

- $|v_t - v_l| \leq \epsilon \implies \hat{v}_t = v_l$
- $|v_t - v_l| > \epsilon \implies \hat{v}_t = v_t$

where ϵ is the error threshold and v_t is the true value at round t. In the context of the proxy-based framework, the node-side proxy will compute the difference between the true and the last transmitted value and only transmit the true value if ϵ is exceeded. On the coordinator-side, the last received value is used as its prediction until it receives a new value.

3.3.2 Piecewise Linear Approximation

Piecewise Linear Approximation (PLA) [27] is a prediction model that works by approximating a non-linear function by fitting a line through past data. For a function $f(x)$ defined on the interval $[a, b]$, a piecewise linear approximation is a function $g(x)$, a sequence of linear segments of the form $\alpha + \beta x$. Although more sophisticated algorithms exist for computing g, our framework fits a line on past values as outlined in [2].

Let Y and X be vectors of length l. Y contains the last l readings from the metric and will be referred to as the PLA buffer, and X the corresponding time stamps for the readings in Y. The predicted value $\hat{v}_t$ is then calculated as follows:

$$\hat{v}_t = a \cdot t + b$$

Where a and b are defined as:

$$a = \frac{covariance(X, Y)}{variance(X)}$$

$$b = \frac{\sum_{i=1}^{l} Y_i}{l} - a \cdot \frac{\sum_{i=1}^{l} X_i}{l}$$

3.4 Determining the Error Threshold: Epsilon

The error threshold ϵ (epsilon) is a crucial component of the prediction model methodology as it decides how lenient the system should be regarding the size of the errors. The method used to determine this value is therefore of great importance.

3.4.1 Static Epsilon

In a scenario where the characteristics and range of a metric are known beforehand, such as ARP-table entries, or when conducting a test with a precollected dataset, it is possible to set ϵ statically. One possible method of determining the static threshold is to base it on the range of the used dataset.

3.4.2 Dynamic Epsilon

In a more practical setting, the system will encounter metrics whose characteristics are not known beforehand, or the system might not even know *at all* what metrics it will be monitoring. To approach this problem more dynamically, it would be favorable if ϵ could be calculated during execution based on the current range of the observed data.

To achieve this, every node maintains a buffer of length L_ϵ for every metric. The buffer works as a sliding window, continuously keeping track of the last L_ϵ true values. Let the current monitoring round be denoted as t_c and let the largest and smallest *true* value in the time frame $[t_{c-L_\epsilon}, t_c]$ be denoted as Max_c and Min_c respectively. Then ϵ can be calculated as:

$$\epsilon = |\text{Max}_c - \text{Min}_c| \cdot \frac{p_\epsilon}{100}$$

Where p_ϵ is a predetermined value in the range $[0, 100]$. Notice that if $p_\epsilon = 0$ then the prediction models are completely bypassed, and the system would function as a purely push-based monitoring system. Various values of p_ϵ will be evaluated in the testing.

In this framework, the responsibility for calculating ϵ falls solely upon the node-side proxies and is updated upon the transmission of a metric update to the coordinator-side proxy.

3.5 Evaluation Metrics

In the evaluation, three main evaluation metrics were tracked as a way of measuring performance. The first is Communication Ratio (CR), a direct indicator of the system's communication efficiency. The second is Mean Absolute Error (MAE), which

measures the average error of predictions. The third is Latency, which measures the time delay in the system. By considering all three metrics, it was possible to evaluate whether the loss in monitoring accuracy (MAE) could be minimized while simultaneously improving communication efficiency (CR) and maintaining latency at an acceptable level.

3.5.1 Communication Ratio

The communication ratio was calculated by measuring inter-proxy traffic and comparing it to the Prometheus baseline. Communication was quantified on three different levels: in the number of transmitted packets, the number of transmitted bytes, and the number of transmitted update messages.

The packet ratio was calculated by analyzing the inter-proxy traffic of both the system and the baseline tests. Let P_S and P_B be the list of captured network packets for the system and baseline, respectively. Then the packet communication ratio CR_{pkt} was calculated as:

$$CR_{pkt} = \frac{|P_S|}{|P_B|}$$

and the byte communication ratio CR_{bytes} was calculated as:

$$CR_{bytes} = \frac{\sum_{i=1}^{|P_S|} |P_{Si}|}{\sum_{i=1}^{|P_B|} |P_{Bi}|}$$

Measuring the ratio of update messages differs from the packet and byte ratios in that it is not directly based on the captured network traffic. Let U be the total number of update messages that the coordinator-side proxy receives, let R be the number of monitoring rounds completed, and let n be the number of nodes in the system. Then the update message ratio CR_m was calculated as:

$$CR_m = \frac{U}{R \cdot n}$$

This is due to the fact that in a baseline run, the Prometheus coordinator would receive one update per round from every node in the system, which totals to $R \cdot n$ update messages. Note that CR_m counts the bundled update messages containing all metrics that required an update for that round. It does not count the metric updates individually.

3.5.2 Mean Absolute Error

MAE measures the absolute difference between the value reported on the coordinator-side, $y_i^{\text{coordinator}}$, and the true value reported on the node, y_i^{true}. It is given by:

$$\text{MAE} = \frac{1}{R} \sum_{i=1}^{R} \left| y_i^{\text{coordinator}} - y_i^{\text{true}} \right|$$

where R denotes the number of rounds. We may also normalize the MAE over the range of a dataset, making it possible to compare MAE across datasets and experiments, in which case we call it *MAE over range*. If the range of the data is not known beforehand, for instance, when running the Prometheus *Node Exporter* (see 2.2.4) and receiving live system metrics, the range will be calculated after the fact by examining the true values recorded. When running tests that involve multiple nodes and/or several metrics per node, the MAE is computed as the mean of all error values aggregated across all metrics, all rounds, and all nodes.

A lower MAE indicates more accurate predictions, whereas a higher MAE indicates larger discrepancies.

3.5.3 Latency

The latency of the system, more specifically the *update latency*, is the duration between the point in time when a given node-side proxy N scrapes its endpoint to receive fresh data and the point in time when the coordinator-side proxy C is finished processing the metric update containing said data. This metric is therefore an indicator of how much earlier N needs to scrape its endpoint in relation to when Prometheus scrapes C so that C is able to receive the data in time.

3.6 Evaluation Methodology

To assess the framework, the evaluation was conducted in two stages: the first in a simulated environment and the second on a physical testbed. Two distinct stages were used in order to provide a balance between controlled testing and practical applicability.

Once the prediction models had been implemented and verified to be working correctly, simulation testing was initiated. This phase aimed to assess the framework's performance under purely virtual conditions. The simulation testing enabled systematic testing in a flexible and easily accessible testing environment. This made it possible to efficiently explore a wide range of configurations and simplified debugging.

The hardware-based evaluation followed the simulation phase, with some preparatory adjustments made in parallel with the simulation phase to facilitate the porting of the experiments to a hardware testbed. As a central focus of the project is bridging the gap between theory and practical application, hardware testing was prioritized in order to validate assumptions under more realistic conditions and evaluate the framework's applicability in practice. Additionally, it served as a way to detect technical difficulties exclusive to hardware tests.

3.6.1 Measuring Performance

To evaluate performance, three Python scripts were used to collect and process the results of each experiment. One script was created for each evaluation metric:

Communication Ratio (CR), Mean Absolute Error (MAE), and Latency.

The command-line packet sniffer *TShark* [28] was utilized to capture the network traffic, and by filtering the traffic on source and destination IP addresses using a Python packet analysis module *dpkt* [29], we were able to extract the desired inter-proxy traffic. To further ensure that only the desired traffic was captured, DNS, NTP, and SSH traffic were removed from the capture file by filtering out ports 53, 123, and 22 as outlined in RFC6335 [30]. This was compared against a baseline of a normal Prometheus run with the same dataset, node count, and runtime.

The framework itself was extended to support MAE calculation. This new functionality meant that each true value recorded by the node-side server and each value recorded on the coordinator-side was written to a file. This recording of values was made to be carried out continuously during the framework's execution. Once execution was finished, the MAE script used these values to calculate the resulting MAE over range by averaging the absolute differences between the values and normalizing this result over the dataset range.

To measure latency, timestamps were continuously recorded on both the node side and on the coordinator side, in one file per node. The first timestamp is recorded once the node-side proxy scrapes its client, and the second once the coordinator-side proxy has processed the update corresponding to the same scrape. To identify two timestamps as being related to the same round, the monitoring round number was written as well. Naturally, each scrape will not generate an update to the coordinator-side proxy, in which case the scrape timestamp is discarded.

3.6.2 Simulation-based Evaluation

To simplify the simulation testing, a bash startup script was written, which is responsible for orchestrating the tests. It launches a *Docker Compose* build that deploys the coordinator-side proxy, the Prometheus process, and a component called the *nodespawner*. The nodespawner is a single *Golang* program that is responsible for configuring and instantiating the virtual nodes, each consisting of a *TestClient* that exposes Prometheus metrics via an HTTP endpoint and a node-side proxy. It accomplishes this by creating go-routines, one for each client and proxy. The nodespawner assigns every virtual node a unique IP address within the 127.0.0.1/24 range on loopback and can therefore be logically distinguished in every output file it issues. The nodespawner is also responsible for creating a *service discovery* file that Prometheus can use to locate the spawned nodes on the network.

When the test run was completed, evaluation metrics were calculated, and the resulting data and graphs were stored in a results directory. In addition to the startup script, we also created a similar script that would execute only Prometheus and the TestClient to generate results that would be used as the baseline. As a way of enabling easy and traceable configuration of the framework, the system was equipped with the ability to read in configuration parameters from a JSON configuration file. Documentation of the tests was made simple by means of copying the configuration file into the result directory.

Two different pre-collected time-series datasets were used to evaluate the system under varying data characteristics. Each dataset contained a unique statistic, capturing different time-series behaviors and variation, in order to evaluate how the system performance may fluctuate across different types of patterns. The following datasets were used:

- Ericsson: CPU usage from a load testing procedure in an Evolved Packet Core testing infrastructure. Data range: 97.07 [31].
- IntelLab: Temperature readings from 54 IoT sensors from Intel's Berkeley Research Lab. Data range: 44.64 [32].

To save time, the datasets were obtained from the FEDAMON GitHub repository, as they had already been processed [2].

3.6.3 Hardware-based Evaluation

The hardware-based evaluation introduced a real-world distributed system, implemented as a test bench consisting of a laptop acting as the coordinator and four Raspberry Pis serving as nodes, interconnected via a network switch. This test bench is illustrated in Figure 3.1.

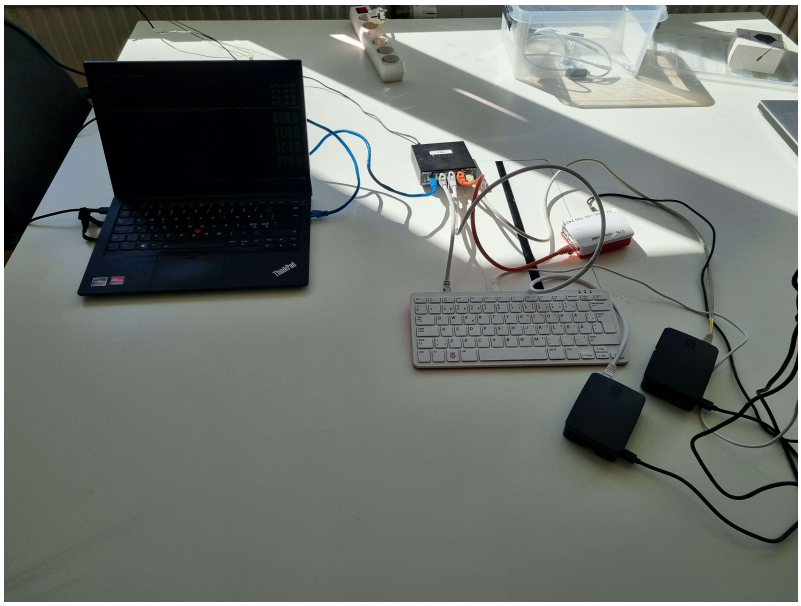

Figure 3.1: Testbench of four Raspberry Pis connected via a switch

The deployment script used in the simulation evaluation was refactored to handle the complexities that came with physical testing. This included compiling a binary compatible with the Raspberry Pi's architecture *armv7l*, transmitting this binary to each Raspberry Pi at the start of each experiment, and then accessing each Raspberry Pi via SSH in order to execute the binary. The binary was responsible for starting up the node-side client, node-side proxy, and a federated learning client. Furthermore, at the end of each experiment, each Raspberry Pi was made to transmit recorded

measurements required to calculate the evaluation metrics to the coordinator-side proxy.

To evaluate this system on a suitable real-world application, it was decided to record a brand new dataset on our test bench while running a federated learning task. *Flower* [33], a Python framework for federated learning, was used to train a convolutional neural network (CNN) to classify images from the public CIFAR-10 dataset [34]. The machine learning library *TensorFlow* [35] was utilized to train and evaluate the CNN. Additionally, TensorFlow's high-level API *Keras* [36] was used to facilitate the creation of the CNNs and their layers.

Two distinct CNN models were used with significantly different parameter counts in order to regulate the load being put on the system's computing resources and network bandwidth, and will be referred to as *CNN-Heavy* and *CNN-Light*. Summaries of their respective parameter count and Keras layers are depicted in Tables 3.1 and 3.2.

As a final assessment to evaluate how the framework compares to the Prometheus baseline, live tests were conducted at the end of the hardware-based evaluation. The purpose of these tests was to evaluate the framework's performance while it live monitors the test bench running a real distributed workload. Once again, the federated learning framework Flower was employed to train a CNN model. To simulate a resource-constrained system, the more complex model, CNN-Heavy, was utilized, and the number of metrics exported by each node was scaled up.

To manage handling live metrics, the Prometheus *Node Exporter* was employed for the live tests in order to allow the framework to monitor a large number of metrics. The node exporter has multiple *collectors*, each responsible for collecting a certain type of system metric [16]. The following collectors were used in the first live stress test, where 399 metrics were monitored:

- *bcachefs*
- *conntrack*
- *cpu*
- *cpufreq*
- *diskstats*
- *filefd*
- *filesystem*
- *hwmon*
- *kernel_hung*
- *loadavg*
- *meminfo*
- *schedstat*

- *sockstat*
- *softnet*
- *stat*
- *thermal_zone*
- *time*
- *udp_queues*

A custom exporter was written with the help of the Prometheus client libraries mentioned in 2.3 to conduct a test with a more sensible number of system metrics, since the Node Exporter could not provide that level of fine-grained control. The exporter was written in Python, and the library *psutil* [37] was used to retrieve the system metrics, which were subsequently exported. The 19 exported metrics encompassed the following:

- CPU statistics, including usage, temperature, and frequency.
- System load
- Disk usage
- RAM usage
- SWAP usage
- Network activity

The 19 metrics were chosen in particular as they represented the primary indicators for resource utilization (CPU, memory, disk, and system load) and traffic intensity (network activity), the two aspects monitored to evaluate the impact of the federated learning workload on the Raspberry Pis.

Layer (type)	**Output Shape**	**Parameters #**
Conv2D	(None, 26, 26, 32)	320
MaxPooling2D	(None, 13, 13, 32)	0
Conv2D	(None, 11, 11, 64)	18,496
MaxPooling2D	(None, 5, 5, 64)	0
Flatten	(None, 1600)	0
Dense	(None, 4096)	6,557,696
Dense	(None, 2048)	8,390,656
Dropout	(None, 2048)	0
Dense	(None, 10)	20490
Total parameters	14,987,658	

Table 3.1: Summary of *CNN-Heavy*

The command line utility *sar* from the *sysstat* package was utilized to continuously record system load with an interval of 1 second on all 4 Raspberry Pis as well as on the coordinator device for the duration of the federated learning tasks. The

Layer (type)	Output Shape	Parameters #
Conv2D	(None, 26, 26, 32)	320
MaxPooling2D	(None, 13, 13, 32)	0
Conv2D	(None, 11, 11, 64)	18,496
MaxPooling2D	(None, 5, 5, 64)	0
Flatten	(None, 1600)	0
Dropout	(None, 2048)	0
Dense	(None, 10)	20490
Total parameters	38,826	

Table 3.2: Summary of *CNN-Light*

purpose of this information was to examine the impact that the framework had on the devices in the system, as well as to collect a new dataset to be used. In particular, the following metrics were gathered:

- CPU Usage, taken as an average across all CPU cores.
- Per-core CPU Usage
- Network Usage
- Memory usage, expressed as a percentage of the total amount of memory available

The local clocks of the Raspberry Pis were synchronized via *Network Time Protocol* (NTP), meaning the nodes continuously made requests to an NTP server set up on the coordinator device. The devices were configured to send NTP requests with an interval of 32 seconds. This is a relatively small NTP request interval. The proxy-based framework remains functional even with larger NTP request intervals, despite system clocks differing by a few milliseconds. However, to obtain meaningful update-latency measurements, it is desirable to minimize clock skew as much as possible, which motivated the choice of this small interval. NTP traffic is filtered out from the packet capture, meaning it does not impact the communication ratio calculations.

3.6.4 Configurations

Experiments were conducted under multiple configurations, from highly controlled to what might more closely resemble real-life monitoring scenarios. The former allowed a closer analysis of how certain factors may impact the evaluation metrics, whereas the latter provided a more realistic assessment of how system performance may behave under actual deployment environments.

Each configuration varied across certain key factors, the main being a choice between the two implemented prediction models, SA and PLA, forming the basis for subsequent tests. Another important one was the choice of transport protocol: UDP or TCP, creating differences in communication overhead and reliability. Different numbers of nodes and different numbers of models and metrics per node were also varied, the former for more controlled analysis and the latter representing a more

realistic Prometheus monitoring scenario. Experiments considered different parameters to the error threshold, dynamic ϵ, assessing how different levels of leniency affect the final results. Specifically, in the PLA model, different buffer sizes were tested.

Given the wide range of possible configurations, exhaustive testing was deemed impractical. Therefore, initial configurations were focused on factors believed most likely to influence the evaluation metrics, such as model choice and transport protocol. After obtaining these initial results, we moved on to experimenting further with the remaining configuration parameters.

4

Implementation

This chapter serves to describe the proxy-based framework developed for this paper. It will give a high-level overview of the framework and how data flows, as well as provide more details on how each component is implemented.

4.1 Proxy-based Architecture

The proxy-based architecture, seen in Figure 4.1, introduces an intermediary layer between the Prometheus server and the monitored nodes. All communication is routed through these proxies, rather than relying on Prometheus' direct scraping.

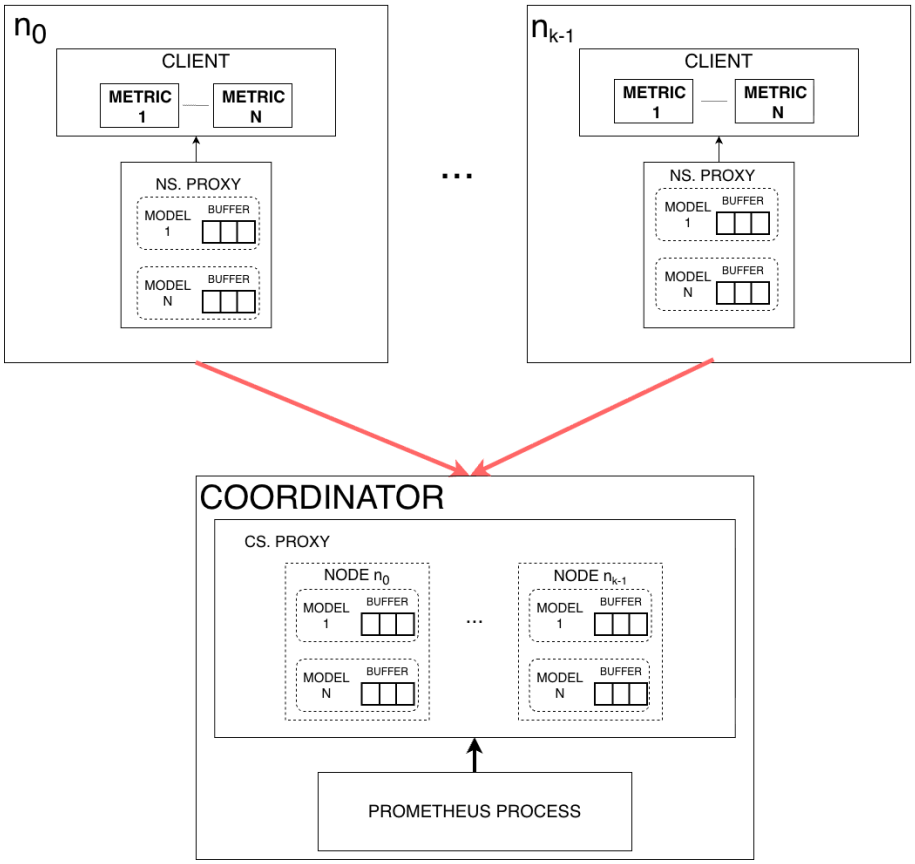

Figure 4.1: Outline of the proxy-based architecture, depicting devices and respective processes

4.1.1 Coordinator-side Proxy

This component, as can be inferred by the name, resides on the same physical device as the Prometheus server. It can be seen at the bottom of Figure 4.1, within the coordinator component. The Prometheus process is configured to relay every scrape request to the proxy. Furthermore, the coordinator-side proxy maintains the corresponding prediction models and model buffers on each monitored node and can thus fulfill every Prometheus request by returning the values predicted by the prediction models. When in operational mode, the objective is for the coordinator-side proxy to fulfill as many requests as possible for the server without needing intervention from the nodes.

4.1.2 Node-side Proxy

In the architecture, each client in the system is situated behind a proxy located on the same physical device. This proxy is referred to as the node-side proxy and resides within the node, seen at the top of Figure 4.1. From the client's perspective, this proxy acts as a Prometheus server, scraping metrics from it regularly, with the same scrape interval used in the actual Prometheus server. Internally, the proxy is responsible for running the chosen model and directly comparing the predicted value to the true value, scraped from the node. This comparison is what initiates the monitoring decision; if the difference between the two values is larger than the given threshold, the proxy will forward a buffer containing the true value to the coordinator-side proxy. As depicted in Figure 4.1, each node-side proxy contains models with their accompanying buffers, where each model corresponds to one Prometheus metric exposed by the client.

4.1.3 TestClient

Within the scope of this project, we designed, implemented, and employed a custom Prometheus HTTP endpoint called the *TestClient*, residing inside the node in Figure 4.1. Its purpose is to read values from the previously pre-collected datasets, which are provided as CSV files, and then expose these values via HTTP. The TestClient accepts two parameters: the number of columns (number of metrics) it should serve and the index of the column where it should begin. This allows us to partition a dataset into segments so that each spawned node is assigned a specific subset of the data. The TestClient transforms the raw CSV data into Prometheus metrics, enabling Prometheus to scrape them directly. It also supports gzip [38] compression, which Prometheus uses during the baseline runs. The TestClient keeps track of the current row and advances to the next row on each scrape. In this way, we can sequentially serve the dataset, treating each new row as a new value to be exposed.

4.1.4 Configuration of Prometheus

As described in section 2.2.5, Prometheus is configured by setting options in the configuration file *prometheus.yml*. Prometheus is only altered through this file, and no modifications of the Prometheus source code are therefore necessary. In this

configuration file, a new job is created, under which all configurations can be defined. First, Prometheus is configured to forward all scrape requests to the coordinator-side proxy by setting the *proxy_url* option to the IP address of the proxy process. Second, the addresses for the scrape target devices are defined in a separate host-discovery file. This file is edited automatically in the case where the *Testclient* is used. It is otherwise manually populated, as is the case when running a test on the physical test bench. Lastly, the scrape interval is set.

4.1.5 Communication and Message Types

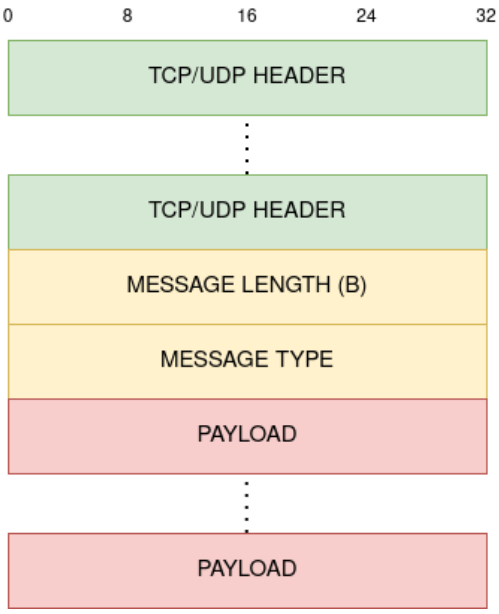

Figure 4.2: Byte layout of the custom protocol header (in yellow)

To further minimize the communication overhead of the framework, the inter-proxy communication is handled by a custom application-layer protocol carried over TCP or UDP. The custom protocol allows for versatility in regard to communication while still keeping the header overhead low.

The header of the custom protocol can be seen in Figure 4.2, colored in yellow. The first field denotes the message length and is encoded as a 32-bit unsigned integer. The second field represents the message type, also encoded as a 32-bit unsigned integer, and tells the receiver what application layer message is carried in the *payload*. The different application-layer message types are listed below:

- *INIT* is sent by the coordinator-side proxy to a given node-side proxy at startup time. It contains the timestamp at which the Prometheus scrape request arrived.
- *METRIC_INIT* is sent by a node-side proxy to the coordinator-side proxy and contains the raw form of its Prometheus metrics, an example of which is shown in Section 2.2.4. The purpose of this is to provide the coordinator-side

proxy with the necessary metric metadata, ensuring that the coordinator-side proxy can deliver the Prometheus metrics in a format suitable for Prometheus.

- *METRIC_UPDATE* is sent by a node-side proxy to the coordinator-side proxy when the prediction falls outside of the threshold ϵ. Apart from the current monitoring round, *METRIC_UPDATE* carries a list of update messages in the case that multiple models need to be updated in the current round. This bundling of updates further decreases the number of transmitted packets. The contents of a single update message vary between models.

These messages were created as Golang struct types and subsequently marshaled into byte arrays through the means of *MsgPack* [39], a convenient way of encoding struct instances to compact byte arrays. By examining the *MsgType* header field, the receiver learns what type to unmarshal the payload data into.

4.1.6 Models and Metrics

The proxy-based framework keeps track of simple numerical metrics, each distinguished by a unique label. The framework also associates a prediction model for each of the tracked metrics. It is required to keep track of the exposed metrics, perform the model calculations, update model parameters, and lastly serve them to the Prometheus server in a transparent way. To achieve this goal, there are a few differences in how the different proxies keep track of the metrics.

Each node-side proxy maintains an up-to-date view of the metrics exposed by the given client hosted on the same device. It keeps a registry of models implemented as a map that associates each metric name with an instance of *ModelContext*. This wrapper class around the type *Model* is in charge of performing the dynamic error threshold computations and managing the metrics *priority* (see 4.1.7). There are different model types, one for each prediction model listed in Section 3.3, each complying with a *Model* interface.

For the coordinator-side proxy, more data is required in order to properly present the metrics to Prometheus. In addition to this, it also needs to keep track of every node-side proxy it is monitoring, along with every metric it exposes. It therefore maintains a ledger of nodes along with their respective metrics. Each such instance contains the metrics associated with the given node along with a unique *Model* instance for said metric.

4.1.7 Metric Priority

Every metric that a node is monitoring is given a priority consisting of a positive integer starting from 1. A metric's priority determines how often its threshold will be checked by the node. The highest priority is 1, and such a metric will be checked every monitoring round. A metric with priority two will be checked *every other* monitoring round. Thus, a metric with priority $x > 1$ will only be checked if $r \bmod x = 0$ where r is the node's current monitoring round.

4.2 Algorithm Design

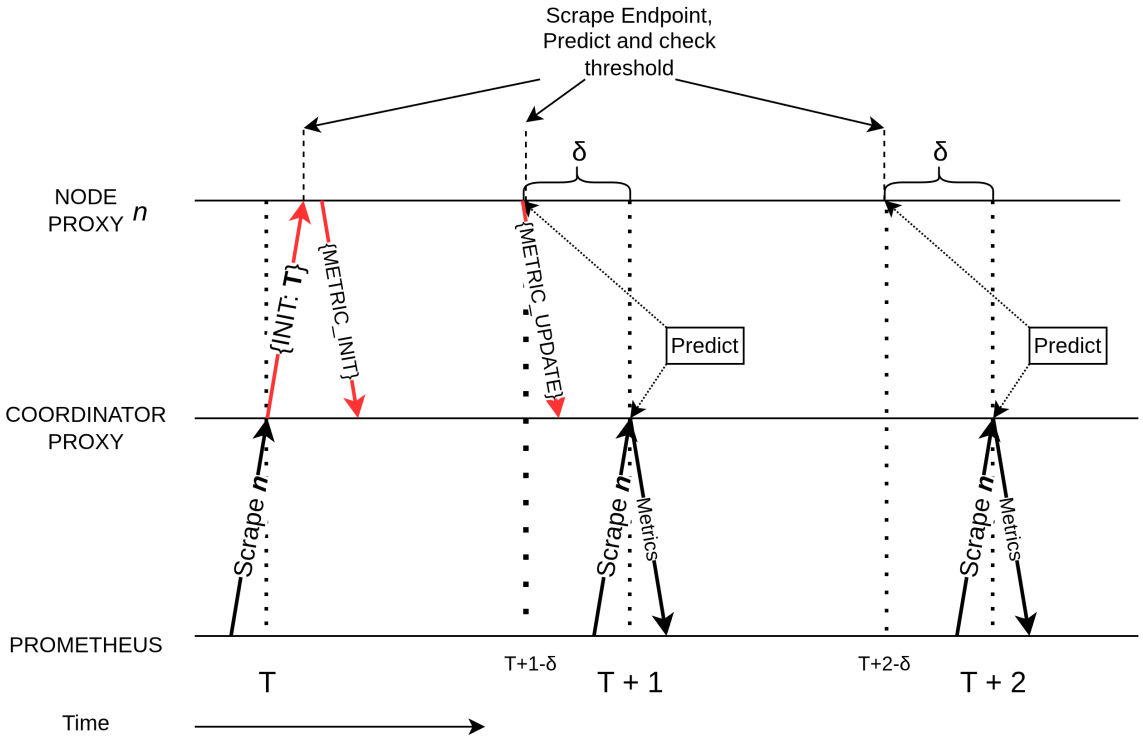

Figure 4.3: An overview of the system's communication flow

Throughout its runtime, the framework will advance through three distinct phases: the *initialization* phase, the *warm-up* phase, and the *standalone* phase. Figure 4.3 visualizes how the system communicates over time, with the red arrows depicting the inter-proxy communication that the system aims to reduce. This diagram is not to scale regarding the timeline, but is intentionally exaggerated in order to visualize the different states and relationships between messages.

4.2.1 Initialization Phase

Upon the framework's startup, node-side proxies are idle until they have been initialized. The initialization of a node begins once the Prometheus server queries the coordinator-side proxy for metrics from that node, which the Prometheus server continuously does. Once the coordinator-side proxy receives the query, *SCRAPE: n*, it checks if the node is already initialized. If not, it generates an initialization message, *INIT: T*, and sends it to the node-side proxy in front of the requested node. The initialization message contains the current monitoring round as well as the timestamp of the Prometheus scrape, both of which are used on the node-side to synchronize the proxies in their predictions.

The time is used to schedule the subsequent scrapes of the client. The future scheduled times are based on the scrape interval, which is internally configured in the node to match the Prometheus server's scrape interval. Additionally, they are shifted by

a small delta, δ, compared to the time of the Prometheus scrape intervals. This ensures that the node-side proxy has enough time to scrape its node, generate a prediction, and upload it to the coordinator-side proxy before the coordinator-side proxy handles a Prometheus scrape. With this, the initialization of the node-side proxy is complete on the node side.

The initialized node-side proxy then scrapes its client and receives the initial metrics. The initial metrics are used for two purposes. First, it is used to initialize a model per metric on the node side. Then, it is packaged inside a metric initialization message, *METRIC_INIT*, and sent to the coordinator-side proxy. Upon receiving the metric initialization message, the coordinator-side proxy uses it to capture the metric structure and initial values and to instantiate corresponding predictive models locally. Once this is done, the node-side proxy is considered initialized on the coordinator-side as well.

4.2.2 Warm-up Phase

Once a node has been successfully initialized, its instantiated model(s) must complete a warm-up phase before any reliable predictions can be made. The warm-up stage sets up the required model parameters such that models on both the node and coordinator-side are aligned and make the same reliable predictions. The warm-up process differs depending on which model is being executed.

For the Simple Approximation model, the warm-up overlaps with the initialization phase. The model requires only a single past value in its buffer to make its monitoring decision, and both proxies acquire this value during the initialization phase, enabling the model to begin regular execution immediately. Thus, the warm-up for this model takes a single monitoring round.

For the Piecewise Linear Approximation model, predictions can only be made with a full model buffer of previous values. As such, the warm-up takes as many monitoring rounds as there are entries in the buffer. In each round, the node-side proxy scrapes its client and fills each model's buffer with the newly observed metrics. Once enough warm-up rounds have been completed, a new update message containing the buffer is generated and sent to the coordinator-side proxy. With the buffer full, both proxies can make a first prediction and operate as usual.

4.2.3 Standalone

Once the warm-up phase has concluded and the first update message has been sent, the system enters its main state. In the main state, the coordinator-side can operate on its own, using its prediction models to generate metric data upon an incoming request from Prometheus. Every node predicts values in parallel and compares them against the true values, scraped from the target endpoint. If the absolute value of the difference between the predicted and true value exceeds a predetermined error threshold ϵ, an update message, *METRIC_UPDATE*, is generated. The *update latency* interval is also depicted in Figure 4.3, starting when the node-side proxy

scrapes its client, and ending when the coordinator-side proxy has received and processed the *METRIC_UPDATE* message.

The update message contains a buffer intended for use by the corresponding model on the coordinator-side. The contents of this buffer depend on the model. In the case of SA, the buffer contains only the latest true value, which is then sent directly to Prometheus by the coordinator. For PLA, the update message contains the entire buffer of length L_ϵ. The coordinator sends the true value, topmost in the buffer, in its next response to a Prometheus scrape, while the full buffer is adopted by its PLA model for all future predictions.

If a node is tracking multiple metrics and several predictions break a threshold, all generated update messages are bundled in a single message before being sent to the coordinator, saving further communication. The standalone phase is final, and the system will not revert back to either the initialization or warm-up phase.

5

Results

This chapter presents the results of the evaluation phase as described in Section 3.6. It will first describe the results of the simulation-based evaluation and then the hardware-based evaluation. Due to the large number of configurations tested, only the most insightful results are discussed.

Notation	Description
D	Test Duration
PM	Prediction Model
p_ϵ	Dynamic Threshold Epsilon Percentage
L_ϵ	Buffer Size for Dynamic Threshold Epsilon
TP	Transport Protocol used (TCP/UDP)
N	Number of Nodes
M	Metrics per Node
δ	Node Scrape Offset (ms)

Table 5.1: Notation of the configuration parameters used in the results section

Throughout this chapter, tests will be presented, and their configurational parameters will be denoted by the notation summarized in Table 5.1. For all tests, both simulation-based and hardware-based, the scrape interval S is set to 1 second, and the node scrape offset δ is fixed at 100 milliseconds (except Section 5.2.7). The priority for the Prometheus metrics in the tests is set at the highest priority 1 (see Section 4.1.7), unless stated otherwise. Occasionally, the PLA model will be referred to as PLA-X, where X denotes the size of the model's internal buffer. A complete list of the tested configurations can be found in Appendix A.

For all evaluation criteria, namely communication ratio, mean absolute error, and latency, lower values are preferred.

5.1 Simulation-based Evaluation Results

The simulation-based evaluation was initiated following the implementation and verification of the framework in order to evaluate the system in a controlled environment. The primary experiment variables, which are varied across the simulation tests, include the prediction model choice, the dataset choice, and the transport

protocol choice. For each configuration, the framework was tested across system-specific parameters such as the PLA buffer size, the dynamic epsilon variables, and different numbers of metrics at each node. The remainder of this section is structured as follows: Section 5.1.1 demonstrates model behavior, Sections 5.1.2, 5.1.3 and 5.1.4 explore system parameter variations, and Section 5.1.5 investigates the effect of increasing the number of metrics per node.

5.1.1 Prediction Model Performance

Figure 5.1 offers an overview of the characteristics of the two employed prediction models and how they perform over time, measured in monitoring rounds. Figure 5.1a depicts PLA-8 and 5.1b, SA. The error between the predicted and true values can be seen at the bottom of the figures. Initially, the error is close to 0 but increases gradually with time. This is a consequence of the dynamic error threshold calculation. As described in Section 3.4.2, the range is calculated using values from a sliding window buffer where the last L_ϵ true values are stored. As time passes, more values are inserted, and the system adapts to the range. Since this particular portion of the dataset is steadily increasing, the error threshold grows along with it.

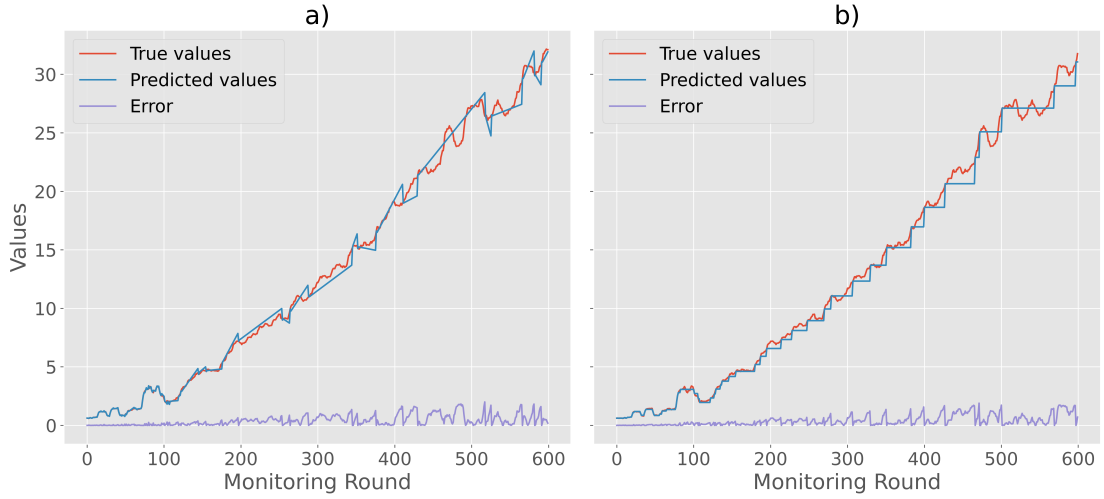

Figure 5.1: Comparison of true and predicted values over time
(D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=1, M=1)

The difference in the models' prediction technique can clearly be seen between Figures 5.1a and 5.1b. Figure 5.1a, representing PLA-8, can be seen attempting to approximate a best-fit line with the help of the data previously observed, continuously correcting itself. Figure 5.1b, on the other hand, has a much more jagged appearance. It displays the SA model maintaining the same predicted value across rounds until the threshold is broken and the true value is transmitted.

5.1.2 Impact of PLA Buffer Size

To establish a baseline for subsequent tests, the initial experiments examined the impact of the PLA buffer size. Varying the PLA buffer size determines how many historical values are used to compute a best-fit line for prediction. Tests were run across sizes ranging from 4 to 14, moving from a reactive model toward a more stable one.

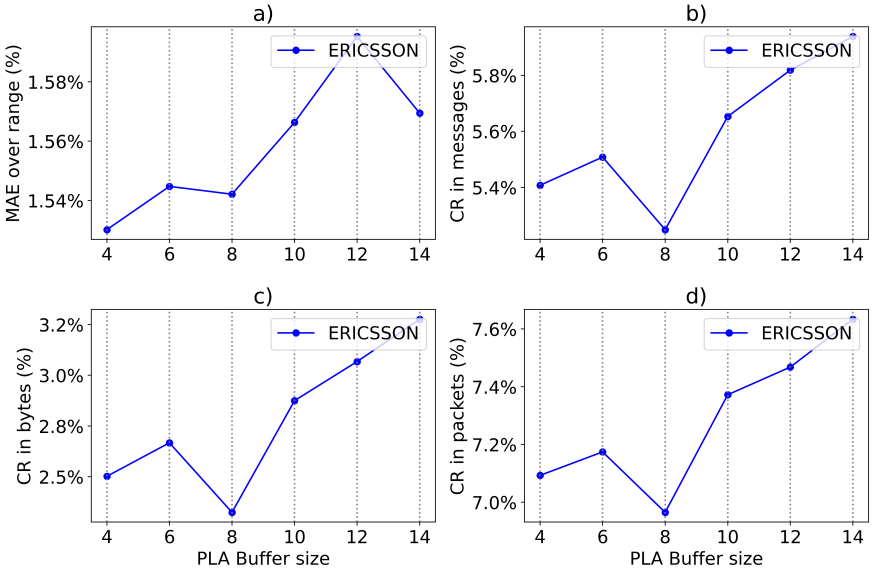

Figure 5.2: Evaluation across PLA buffer sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, L_ϵ=1000, TP=TCP, M=1, N=40)

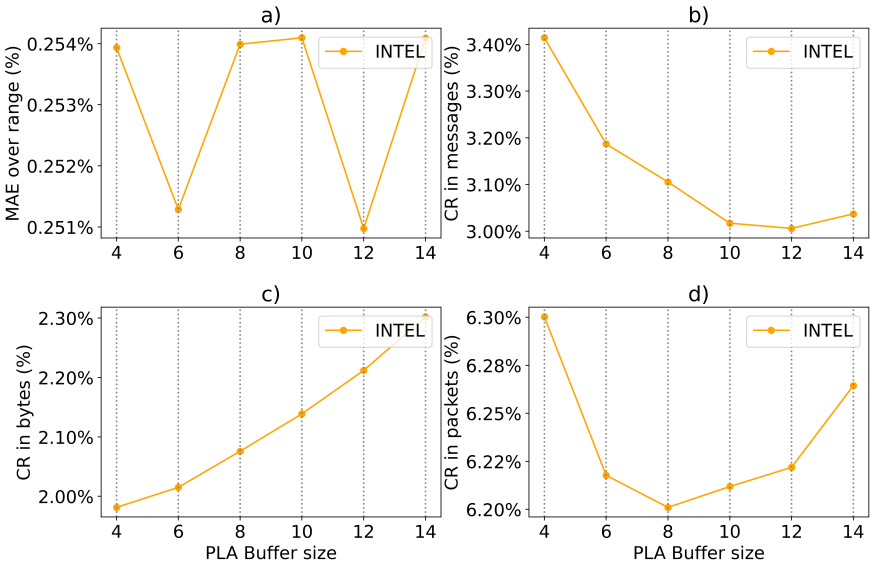

Figure 5.3: Evaluation across PLA buffer sizes using the *IntelLab* dataset (D=4000s, p_ϵ=10, L_ϵ=1000, TP=TCP, M=1, N=40)

Across both datasets, as shown in Figures 5.2 and 5.3, increasing the PLA buffer size over the chosen range does not seem to produce a consistent overall trend, although a more noticeable trend can be observed in the Ericsson dataset. A few larger PLA buffer sizes were evaluated, but were omitted as they consistently resulted in increased results across all evaluation metrics. Notably, for the Ericsson dataset, there is a considerable dip in the communication ratio at a PLA buffer size of 8, while the error also remains low at approximately 1.54%. For this reason, the PLA buffer size of 8 was utilized for the majority of the simulation tests. For the IntelLab dataset, the most favorable choice of PLA buffer size is not as evident. Nevertheless, the buffer size of 8 was adopted for future IntelLab tests for consistency. For context, an *MAE over range* of 0.254% represents an MAE of $\sim 0.11°C$ in the IntelLab dataset, while an *MAE over range* of 1.54% represents a MAE of $\sim$ 1.49% CPU usage in the Ericsson dataset.

5.1.3 Impact of Dynamic Epsilon Buffer Size

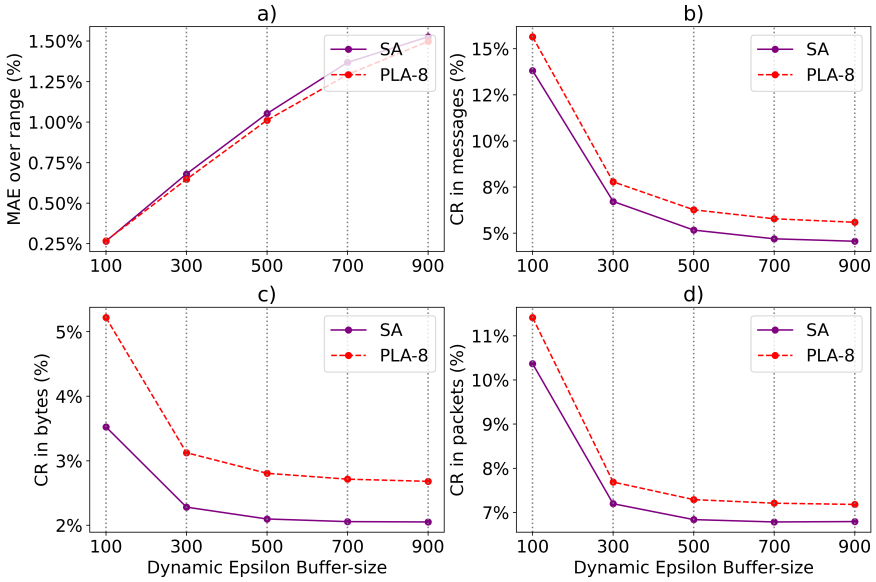

Figure 5.4: Evaluation across L_ϵ sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, TP=**TCP**, M=1, N=40)

Next, the impact of different sizes of L_ϵ was investigated (see Section 3.4.2). The size of L_ϵ determines how much of the observed data will be used to compute the error threshold ϵ, and so the impact will be determined by the dataset. Several buffer sizes ranging from L_ϵ=100 to L_ϵ=900 were explored. Such tests were conducted across both datasets and both transport protocols. The results obtained from the Ericsson dataset are presented in Figures 5.4 and 5.5.

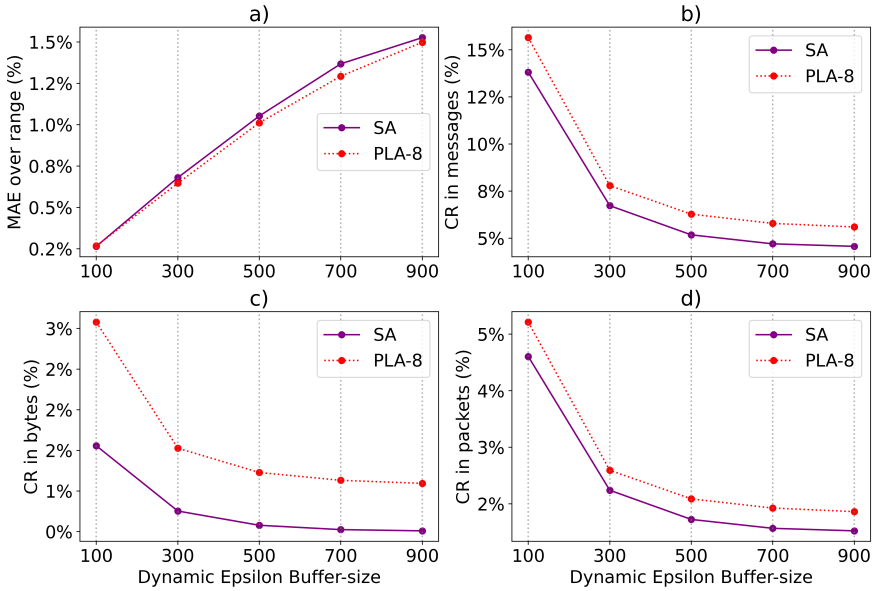

Figure 5.5: Evaluation across L_ϵ sizes using the *Ericsson* dataset (D=1500s, p_ϵ=10, TP=**UDP**, M=1, N=40)

Figure 5.4 presents the results of simulations where the TCP protocol was used. As can be seen in the figure, there is a sizable drop in the communication ratio between sizes 100 and 300, while the error grows linearly. Given that the error increases as the buffer grows in size, subsequent tests were conducted with the buffer size 300, as it appeared to strike a fair balance between error and communication ratio. This held for both the Ericsson and the IntelLab datasets. Illustrated in Figure 5.5 is the further testing in which the transport protocol was switched out from TCP to UDP. Compared to Figure 5.4, Figures 5.5c and 5.5d show a decrease in communication ratio in terms of bytes and packets by a few percentages, as is to be expected with UDP being a more lightweight transport protocol option.

5.1.4 Impact of Epsilon Percentage Value

Closely related to L_ϵ is the percentage parameter p_ϵ. This parameter essentially decides the rigidity of ϵ by scaling the range of its buffer. In the upcoming tests, the value p_ϵ ranges from 2.5 to 17.5. In these experiments, the transport protocol was fixed as TCP, while the dataset itself was varied.

Figure 5.6 shows the results of the tests across the IntelLab dataset. It shows an expected trend of the error rising and the communication ratio decreasing as p_ϵ increases. The Ericsson results are omitted for brevity, but show the same behavior, with a near-identical communication ratio across all levels. In terms of error, the Ericsson dataset ranged between $\sim 0.20\%$ to 1.25%, a higher average error rate than IntelLab.

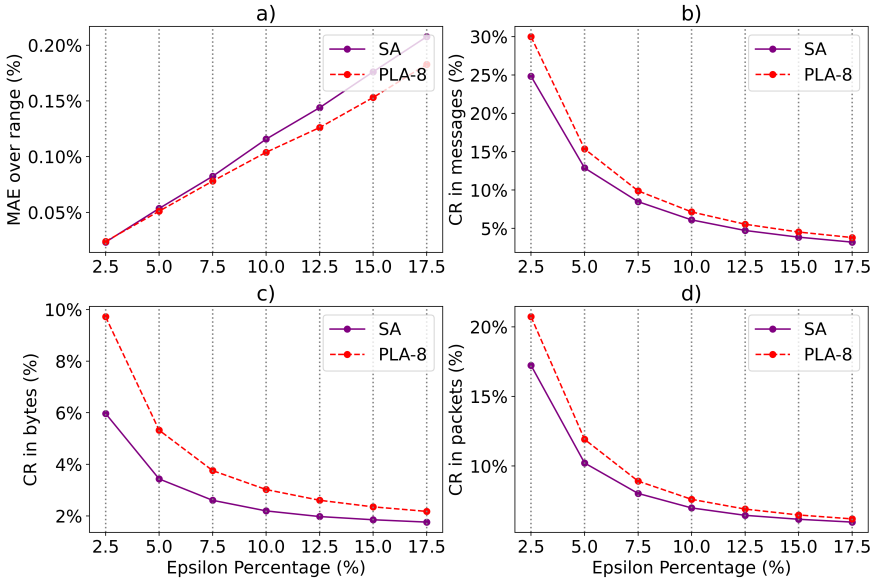

Figure 5.6: Evaluation across p_ϵ using the *IntelLab* dataset (D=4000s, L_ϵ=300, TP=TCP, M=1, N=40)

5.1.5 Multiple Metrics per Node $M > 1$

The system was subjected to increased load by raising the number of metrics per node M, resulting in multiple models being executed on both the node-side proxies and the server-side proxy. In the conducted tests, the number of metrics per node varied from 5 to 20.

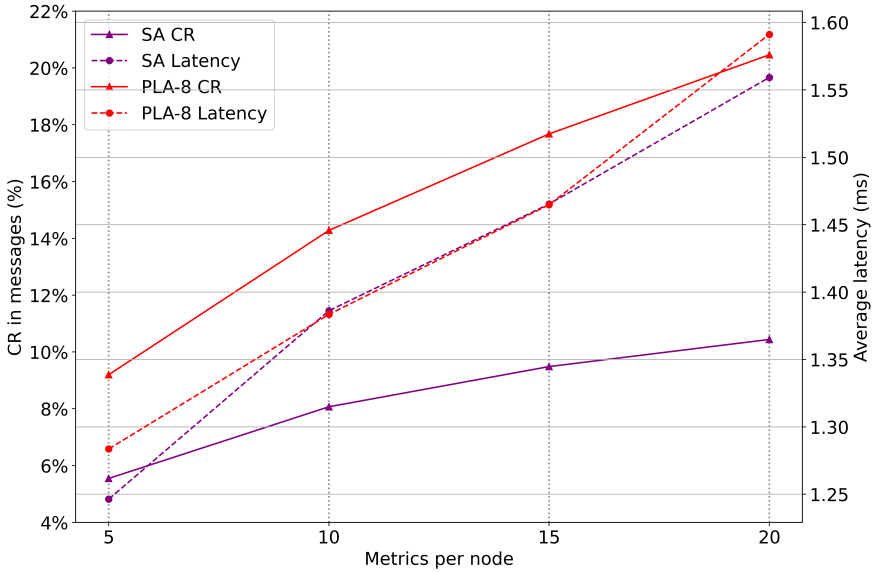

Figure 5.7: Evaluation across M using the *Ericsson* dataset (D=1500s, p_ϵ=10, L_ϵ=300, TP=TCP, $N = 20$)

The results shown in Figure 5.7 demonstrate that both latency and communication ratio increase as the number of metrics per node-side proxy increases. Although the models' difference in latency is very small at intermediate values of M (M=10 and M=15), PLA has a higher latency at M=5 and M=20, by about 40 microseconds. Overall, the communication ratio ranges from 5% to 21%, with PLA consistently exhibiting a higher communication ratio than SA, as observed in previous results.

5.2 Hardware-based Evaluation Results

To validate framework behavior under more realistic conditions, nodes were deployed on Raspberry Pis, new datasets were collected (see Section 3.6.3), and live tests were conducted. The primary experiment variables and system-specific parameters that were varied remain the same as in Section 5.1. The remainder of the section is structured as follows: Section 5.2.1 describes the new datasets, Sections 5.2.2, 5.2.3 and 5.2.4 explore system parameter variations, Section 5.2.5 investigates the effect of increasing the number of metrics per node, Section 5.2.6 analyzes the CPU and memory usage, and finally, Section 5.2.7 describes the results of the live tests.

5.2.1 Federated Learning Datasets

To generate new datasets, the CNN-Light model (see 3.6.3) was trained for 10 epochs, lasting for about 20 minutes. Executing this task on the Raspberry Pis resulted in two new datasets. The first one consisted of averaged CPU usage across the four cores, and the second contained per-core CPU usage. Table 5.2 captures the key characteristics of each dataset.

DATASET		
Name	**CPU Usage (Averaged)**	**CPU Usage (Per-core)**
Range	0.0 - 92.70	0.0-100.0
Mean	25.10	25.08
Entries	4	16
Datapoints per node	1300	1300
Measuring interval (s)	1	1

Table 5.2: Summary of the federated learning datasets

Figure 5.8 presents samples from each dataset, from the measurements at a single Raspberry Pi. Figures 5.8a and 5.8c show the averaged CPU usage over 150 and 600 seconds, whereas Figures 5.8b and 5.8d illustrate the CPU usage per core over the same respective time periods. The data exhibits a recurring pattern, which can clearly be observed in Figure 5.8c. The pattern corresponds to the iterative training epochs of the federated learning task and a realistic workload with significant fluctuations in CPU usage.

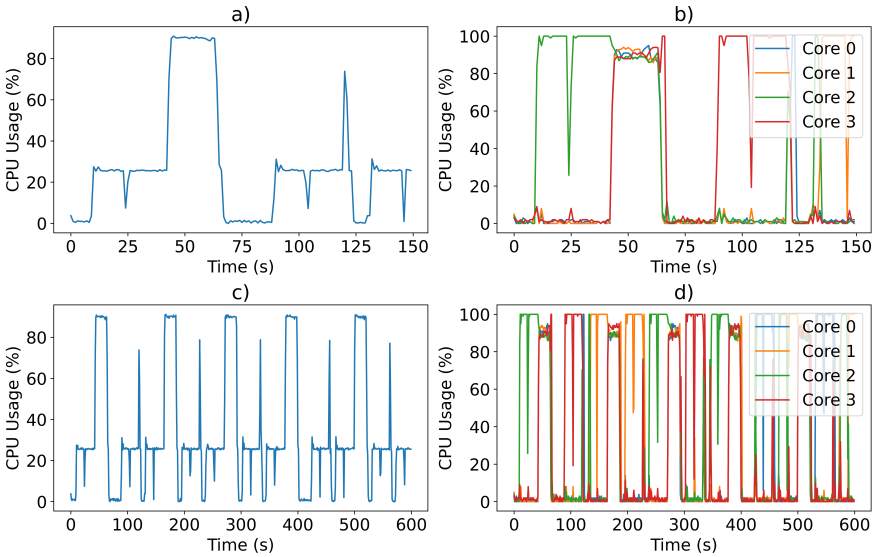

Figure 5.8: Raspberry Pi 1: CPU Usage:
a) Average (150s), b) Per-core (150s), c) Average (600s), d) Per-core (600s)

5.2.2 Impact of PLA Buffer Size

Building on the simulation tests, the hardware-based evaluation phase was initiated with PLA-specific tests across the model's buffer size. The tests were necessary to identify a suitable PLA buffer size for the newly collected datasets. The buffer sizes tested were slightly adjusted in comparison to the simulation-based evaluation, to sizes ranging from 3 to 20.

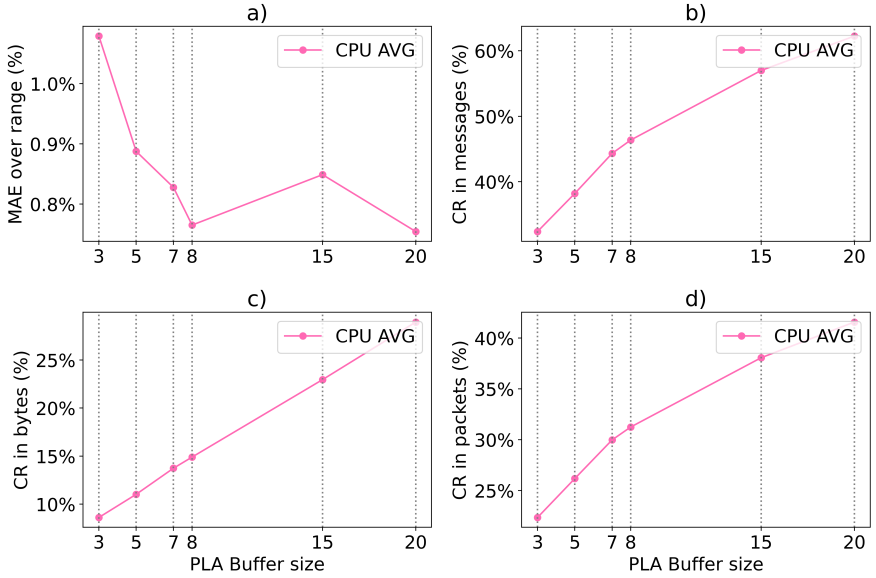

Figure 5.9: Evaluation across PLA buffer sizes using the *CPU Usage (Averaged)* dataset (D=600s, p_ϵ=5, L_ϵ=200, TP=TCP, M=1, N=4)

The test results, which can be seen in Figure 5.9, indicate that a PLA buffer size of 3 displays the highest error of 1.08%, while achieving the lowest communication ratio in all three aspects. This makes it the most communication-efficient option, but also the least accurate among the tested buffer sizes. Although size 8 shows a considerable dip in error, the communication ratio appears to quickly rise with the buffer size, making PLA-8 less suitable than in previous tests. Despite this, both PLA-3 and PLA-8 were experimented with in upcoming tests.

5.2.3 Impact of Dynamic Epsilon Buffer Size

Following the PLA buffer size tests, tests were done across different L_ϵ values. Sizes from 100 to 400 were tested to accommodate the smaller dataset size. The results can be seen in Figure 5.10 and seem to indicate minimal changes in all evaluation metrics across the buffer sizes.

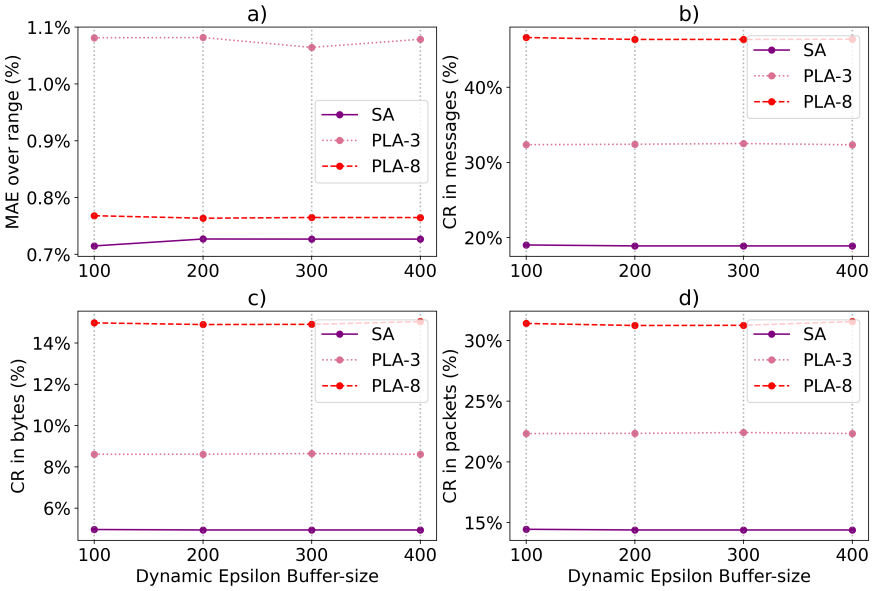

Figure 5.10: Evaluation across L_ϵ sizes using the *CPU Usage (Averaged)* dataset (D=600s, p_ϵ=5, TP=TCP, M=1, N=4)

5.2.4 Impact of Epsilon Percentage Value

To once again further explore the parameters of ϵ, the percentage value p_ϵ was varied across multiple tests. These tests were made across both new datasets, both transport protocols, and using three different PLA buffer sizes. The same p_ϵ values were used as in the simulation-based evaluation, 2.5% to 17.5%.

The subplots of Figure 5.11 show the results of trying different p_ϵ values across the CPU Usage (Averaged) dataset. The PLA-8 model stands out as consistently having the highest communication ratio, while both PLA models maintain a higher communication ratio than SA at all levels.

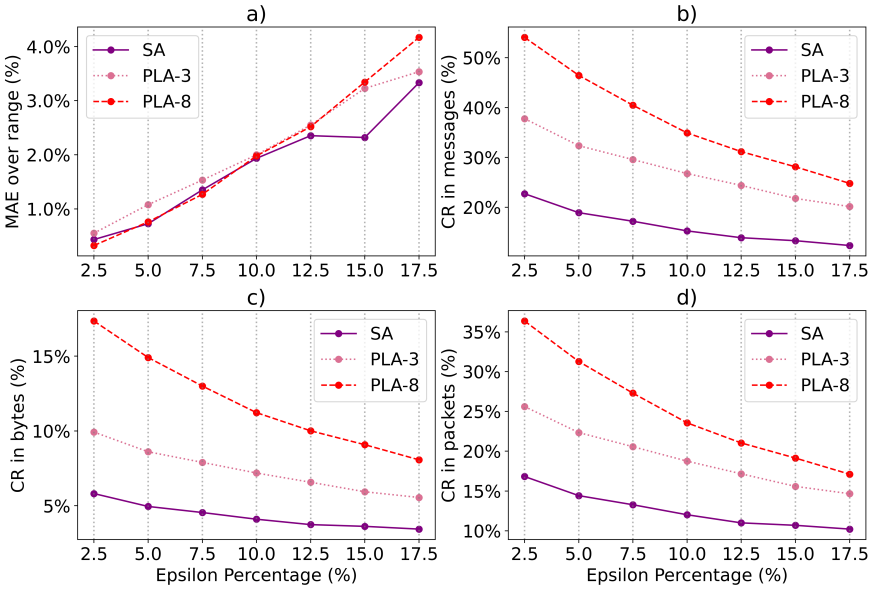

Figure 5.11: Evaluation across p_ϵ using the *CPU Usage (Averaged)* dataset (D=600s, $L_{\epsilon,\text{PLA3}}$=200, $L_{\epsilon,\text{PLA8}}$=$L_{\epsilon,\text{SA}}$=300, TP=TCP, M=1, N=4)

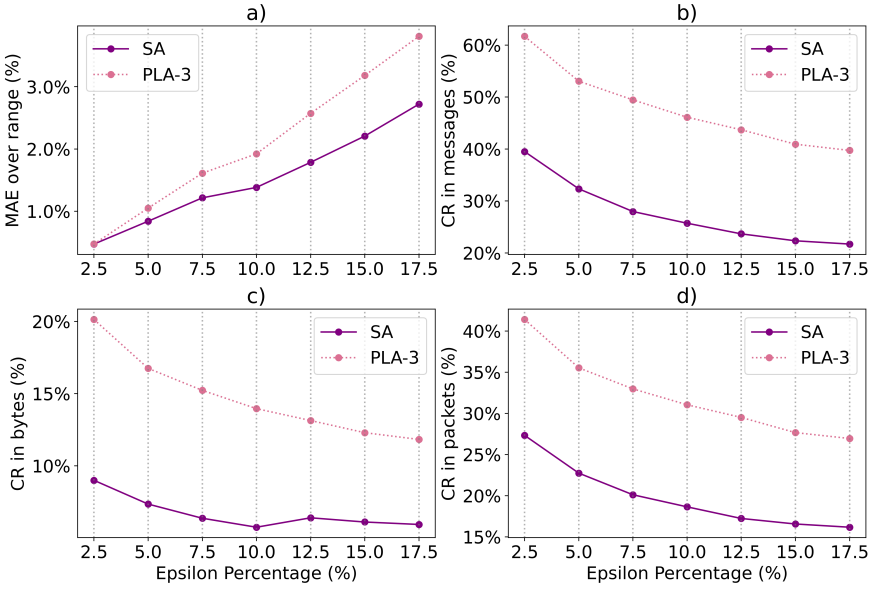

Figure 5.12: Evaluation across p_ϵ using the *CPU Usage (Per-core)* dataset (D=600s, $L_\epsilon = 200$, TP=TCP, M=4, N=4)

Figures 5.12 and 5.13, on the other hand, show the tests across the CPU Usage (Per-core) dataset. The former compares the SA model against PLA-3, showing similar trends as previously observed, but with a generally higher communication ratio. This is to be expected as each node maintains multiple metrics, one for each CPU core. Figure 5.13 instead compares the performance of TCP and UDP when running PLA-3. The results overlap in terms of error and message-based

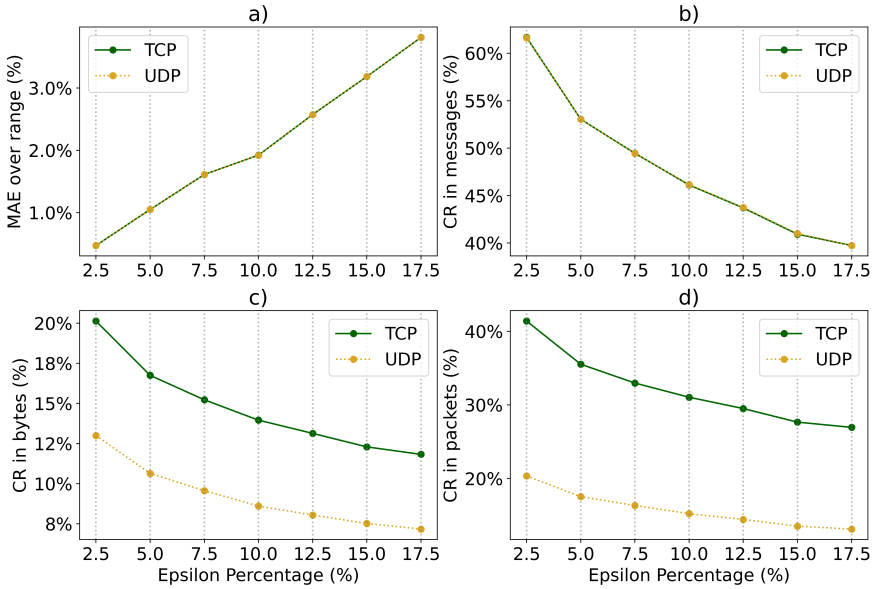

Figure 5.13: Comparison between TCP and UDP across p_ϵ using the *CPU Usage (Per-core)* dataset and model PLA-3 (D=600s, L_ϵ=200, M=4, N=4)

communication ratio across different p_ϵ, while the communication ratio in terms of bytes differs by approximately 8% and in terms of packets by about 20%.

5.2.5 Multiple Metrics per Node $M > 1$

To further validate the results of the simulation-based evaluation, tests were run across different numbers of metrics per node in the hardware-based evaluation as well. Due to the limited size of the federated learning datasets, the Ericsson dataset was employed in order to place further strain on the system. As an additional experiment, the priority-based solution explained in Section 4.1.7 was experimented with at this point.

As we raise M further, its impact on both latency and the communication ratio becomes more apparent, as seen in Figure 5.14. The communication ratio measured in terms of messages seen in Figure 5.14b suffers greatly, with the ratio quickly approaching 1 as M grows. In terms of packets and bytes, the communication ratio resides at 67% and 59% respectively for $M = 150$, shown in Figures 5.14c and 5.14d.

5.2.5.1 Metrics Priority

The aforementioned priority system was subsequently tested to investigate the possibility of reducing the communication ratio for large M. The framework was configured to assign every Prometheus metric the priority=5, except for a select number assigned the highest priority=1, effectively reducing the scrape interval for a large portion of the monitored metrics. The result of which can be seen in Figure 5.15, where the number of high-priority metrics ranges from 5 to 20. The communication ratio measured in messages is the most remarkable result, with a mean decrease in

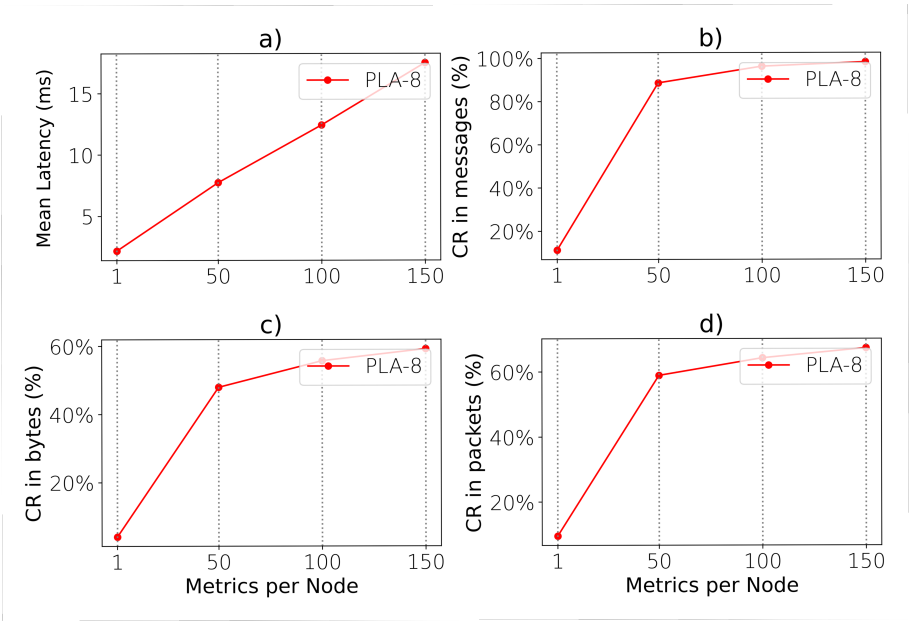

Figure 5.14: Evaluation across M using the *Ericsson* dataset (D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=4)

communication ratio of 40.25% when compared to the framework running with the same parameters but with *only* priority=1 metrics. The bytes and packets saw a mean reduction in communication ratio by 37.5% and 39.61%, respectively.

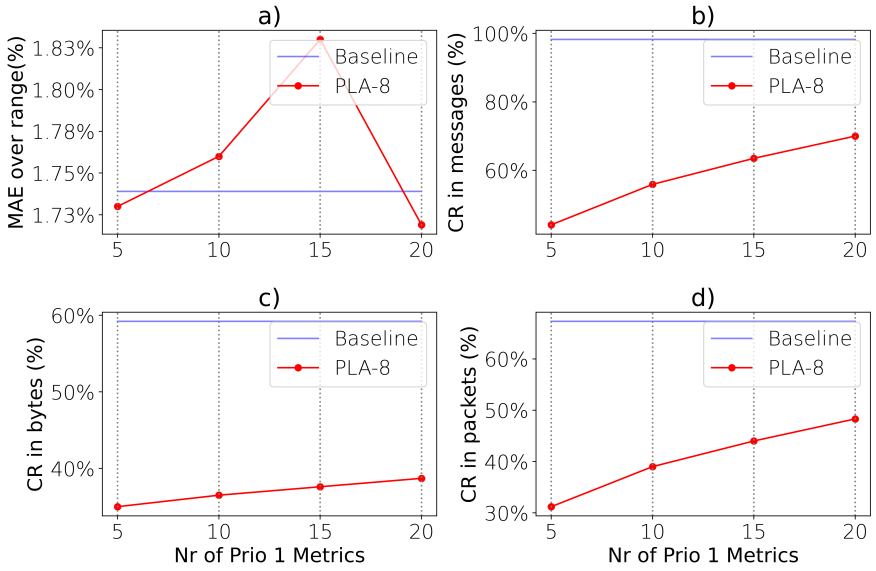

Figure 5.15: Evaluation across number of metrics with priority=1, against a baseline with only priority=1 metrics, using the *Ericsson* dataset (D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, M=150, N=4)

5.2.6 CPU and Memory Usage

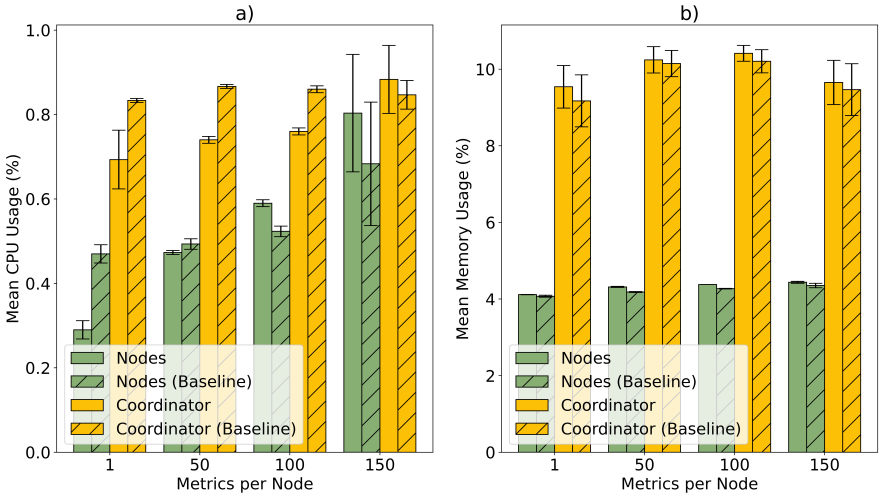

Figure 5.16: CPU and Memory Usage across M using the *Ericsson* dataset (PM=PLA-8, D=600s, p_ϵ=10, L_ϵ=300, TP=TCP, N=4)

Next, we measured the framework's impact on CPU and memory usage. This was tested on an otherwise idle system, meaning that the devices had no additional user workloads running other than standard operating system services. As can be seen in Figure 5.16a, the CPU usage *decreased* on average by 3.01% for the nodes and by 9.67% for the coordinator device. Memory usage, expressed as a percentage of a given device's total memory capacity as seen in Figure 5.16b, saw an average increase of 2.18% and 2.23% for the nodes and coordinator device, respectively.

5.2.7 Live Tests

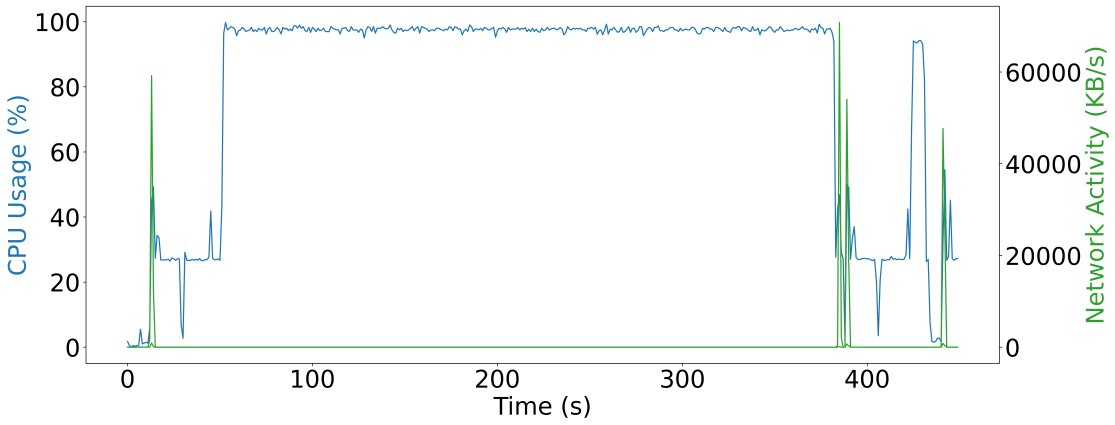

Figure 5.17: CPU Average Usage and Network Activity at Raspberry Pi 1 during training of CNN-Heavy (PM=SA, p_ϵ=10, L_ϵ=200, TP=UDP, M=399, N=4, δ=250)

Figure 5.17 shows the CPU usage averaged across all cores and network activity at a single Raspberry Pi during a live test. As 3.6.3 describes, in the live tests, the framework monitors as the test bench simultaneously executes the federated learning task. Compared to Figure 5.8, which utilized the simpler model, CNN-Light, an increased workload can be observed. In Figure 5.17, the CPU usage remains higher for a longer period of time, around 350 seconds. Spikes in network activity can be observed at the beginning of the recorded data, when the nodes are initialized, as well as shortly before the 400-second mark, when CPU activity declines and data is transmitted from the node. Figure 5.17 illustrates one of the two epochs.

In the first live test conducted, each node was made to export 399 metrics, and both UDP and TCP were evaluated. This intensified load on the system led to significant pressure, resulting in metric updates being lost. This is believed to be the result of packet loss. Inspecting the UDP traffic more closely showed that the points at which the suspected UDP packet loss occurred correlated with the network spikes caused by the transmission before and after a round of federated learning. As for TCP, high latency was observed, with a maximum reading of 421 milliseconds, resulting in updates being delayed rather than lost.

Despite the suspected packet loss and delays, the framework remained functional throughout the test. In the TCP and UDP tests, the framework achieved a communication ratio of about 20-30% in bytes and packets. Meanwhile, the error remained around 0.5%-0.9%. A reduction in the communication ratio measured in message count was not observed, which was expected given the large number of exported metrics.

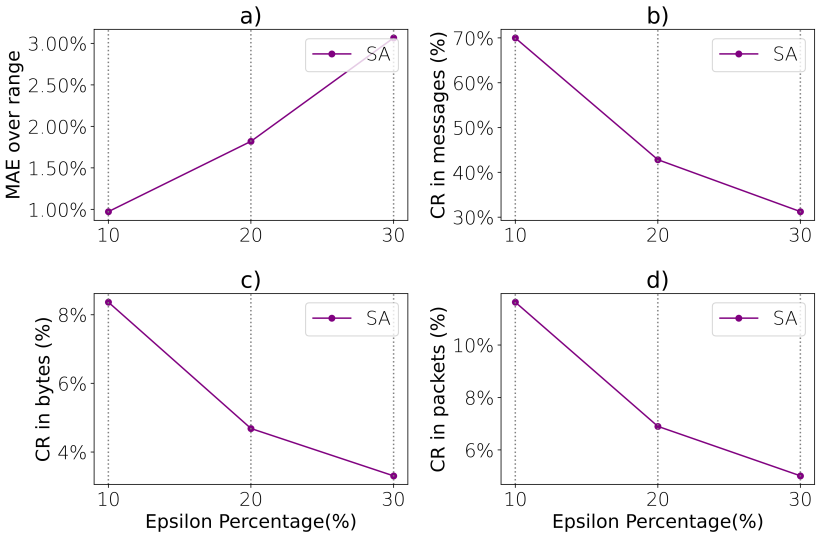

Figure 5.18: Evaluation of live test across p_ϵ during training of CNN-Heavy. Each datapoint is the mean of two tests. (D=868-911s, L_ϵ=200, TP=TCP, N=4, M=19, δ=250)

The second live test featured a more realistic monitoring task considering the devices' resource constraints. This time, only 19 system metrics were monitored at each node,

and the test was evaluated across different p_ϵ values. As shown in Figure 5.18b, we see an improvement in message-based communication ratio, ranging from 70% to 31% as the error tolerance grows. The trade-off is the error increasing from 1% to 3%.

5.3 Summary

Metrics	Error Tolerance (%)				
	5	**10**	**15**	**20**	**30**
MAE over range	0.85	1.13	2.45	1.8	3.1
CR (Messages)	45.3	48.2	28.05	49.8	31.3
CR (Packets)	27.2	22.66	18.0	7.8	5.0
CR (Bytes)	12.1	15.05	8.95	5.8	3.3

Table 5.3: Mean evaluation metric measurements for the overall test set (%)

Table 5.3 shows a summary of the mean results across all conducted tests. Note that the test configurations are not identical across all values of p_ϵ. For instance, the only tests that featured an error tolerance of 20% and 30% were the second live stress tests depicted in Figure 5.18. As a result, the trade-off between error and communication becomes more difficult to distinguish, compared with analyzing standalone error-tolerance experiments as presented earlier.

The summary shows a generally favorable improvement in the communication ratio in terms of messages, ranging from 28.05% to 49.8%. Even when the message communication ratio is high, for instance, when $p_\epsilon = 5\%$, the improvements in communication for bytes and packets remain substantial. The mean error stays within 0.85% and 3.1% across all tests. For $p_\epsilon = 10\%$, the mean error is 1.13%, with a communication ratio of 22.6% in terms of packets, corresponding to a reduction of transmitted packets by 77.4%.

5.4 Latency

For the framework to function as intended, the update latency must be smaller than the node scrape offset (δ), else Prometheus will not receive the updated value in time. The update latencies for the different categories of tests are listed in Table 5.4. Across all simulation tests, the maximum latency never exceeded 50 milliseconds, which means that the predetermined $\delta = 100$ milliseconds is safe and could even be decreased. The maximum of 40 milliseconds can be considered an outlier, since the mean latency resides at 1.35 milliseconds. For the hardware tests, the latency

also resided within the confines of the set δ, with a maximum reported latency of 65 milliseconds, although the mean latency is higher than that of the simulation tests, clocking in at 6.86 milliseconds. For the live tests, the δ was increased to 250 milliseconds as some updates reached above 300 milliseconds. Furthermore, the live tests showed higher latencies overall, with a mean update latency of 50.39 milliseconds.

	Simulation	Hardware	Live Test
Overall	1.35 (40.57)	6.86 (65.32)	50.39 (421.5)
$M = 1$	0.97 (38.20)	2.19 (65.32)	Not tested
$50 > M > 1$	1.82 (40.57)	3.16 (29.63)	17.86 (236.22)
$M \geq 50$	Not tested	15.23 (52.67)	98.55 (421.5)

Table 5.4: A summary of the mean (and maximum) update latency in milliseconds for different subsets of tests

6

Discussion

This chapter discusses the results and their implications, a few implementation-specific challenges, dataset considerations, and finally, possible areas of future work.

6.1 Evaluation Results

The results of the simulation and hardware-based evaluation proved the PLA buffer size to have a strong dependence on the chosen dataset. This is likely due to the significant differences in the characteristics of the datasets used in the respective evaluation phases. The Ericsson dataset consists of gradually increasing CPU measurements, and IntelLab contains temperature data, while the federated learning CPU datasets exhibit pronounced round-based fluctuations. The results suggest that a smaller buffer size is required for efficient performance on the federated learning datasets, likely in order to keep the model reactive enough to adapt to the sudden changes.

The conducted experiments reveal a trade-off between the two prediction models, with PLA often outperforming SA in terms of error, while SA performs better in terms of communication ratio. This can be observed, for example, in Figures 5.4, 5.6, and 5.12. Though these results may be largely dependent on the dataset and exact configuration used, one factor that may contribute to the PLA model's high communication ratio could be the approach taken to update the coordinator. When the PLA model is applied and a threshold is broken, the update message sends the entire PLA buffer, and with that, both sides can calculate the slope and intercept a and b. A possible improvement to the PLA model would be for the node-side proxy to calculate a and b using the buffer and then directly transmit a and b to the coordinator. If identified earlier in the design phase, this optimization would likely have been integrated into the implementation to further reduce communication overhead.

As indicated in Figures 5.4 and 5.5, included in the simulation-based experimentation, the size of the dynamic epsilon buffer generally appears to increase the error while decreasing the communication ratio. However, this trend is not evident in Figure 5.10, part of the hardware-based evaluation. This may be attributed to the periodic structure of the CPU usage dataset used in the given experiment. As observed in Figure 5.8, the dataset exhibits a repeated pattern with an approximate period of about 100 samples. This indicates that the dynamic epsilon buffer may

already cover the full range of the dataset once it has 100 samples, meaning that any sizes over 100 would have a reduced effect on the evaluation metrics.

A given node-side proxy transmits a metric update when there is a metric whose predicted value violates the error threshold ϵ. Thus, the chance of a node-side proxy *not transmitting* a metric update on a given monitoring round decreases as the number of metrics per node increases, since it requires that all predictions remain within their respective thresholds. As can be seen in Figure 5.14b, the communication ratio in terms of messages approaches one quickly. Meanwhile, the system still reduces the total number of packets and bytes sent. The diminishing reduction in update messages can be thwarted by utilizing the priority system, as presented in Figure 5.15. It enables manual assignment of a priority to the individual metrics, and therefore allows them to balance the scale between data accuracy and communication ratio.

Additionally, increasing the number of metrics per node shows an increase in latency, demonstrated in Figures 5.7 and 5.14. This can be attributed to the higher workload at each node and the coordinator, as they each execute more models and process more metrics.

We found evidence that the CPU usage in fact decreased when the system was in use, especially when considering the coordinator device and with $M < 150$. This is hypothesized to be caused by the omission of gzip compression [38] in the proxy-based framework. However, as illustrated in Figure 5.16, one could argue that the small difference between the system and the baseline, combined with the generally low CPU usage, suggests that the impact of this difference is negligible.

6.2 Live Tests

The results of the live tests show some limitations in the system's network behavior. The suspected UDP packet loss is an expected trade-off of the protocol's lightweight design. In this context, packet loss would lead to updates not reaching the coordinator-side proxy. If this were to happen, the coordinator-side proxy would proceed as normal, unaware of the update. It would continue to serve the Prometheus server with its erroneous predictions, leading to an increased overall error. Thus, we can conclude that in an environment where network congestion and overall activity are higher, UDP should preferably only be used when accuracy is not of utmost importance, since packet loss could introduce a higher loss in accuracy than what the predictions themselves cause.

The TCP packet delays are less desirable, but another result of the high load on the system. The latency, occasionally reaching 421 milliseconds, exceeds the live test δ of 250 milliseconds. As a result of this, the Prometheus server would receive stale data from the coordinator-side proxy until the given update arrives. This situation would also increase error, but unlike the UDP issue, the delayed TCP packet would eventually arrive. So, unlike UDP packet loss, where updates are entirely lost, TCP delays primarily affect the timeliness of the update message. Assuming the update is not replaced by a subsequent update before Prometheus scrapes, it should lead to a lesser increase in error than a packet loss.

6.3 Implementation Challenges

During development, several challenges were encountered that interfered with certain Prometheus capabilities or the potential of the proxy-based framework. The challenges either relate to differences between the implemented framework and the standard Prometheus design or reflect oversights in the implementation stage.

6.3.1 Limitations of Push-based Monitoring

According to Prometheus' documentation [1], one of the main reasons for Prometheus being a pull-based system is that it enables easy fault detection. Unfortunately, this feature is lost in the proxy-based framework, as the forecast-based methodology depends on the node-side making monitoring decisions, reflecting a push-based architecture. In other words, if the entire endpoint, including the node-side proxy, fails, the coordinator-side proxy will proceed as usual, serving the Prometheus server stale values. In fact, in all communication, the other components are assumed to be alive, so any failed communication interrupts the intended program flow. Moreover, the framework does not handle nodes running out of data to transmit. Thus, despite it not being a main priority of this thesis, a main weakness of the framework is its lack of fault detection and fault tolerance, introduced by the double layer of proxies and the switch to a push-based architecture.

6.3.2 Prometheus Metric Coverage

Forecast-based monitoring is, within the context of our proxy-based framework and the capabilities of this particular implementation, incapable of properly accommodating every metric type that Prometheus offers. As mentioned in Section 1.3, it was decided not to implement support for the metric types *Histogram* and *Summary*. Both of these metric types count observations and group them into configurable buckets. At the endpoint, the values for these buckets, as well as the total sum of the observations, are exposed. There is a mathematical invariant that needs to be preserved where bucket N must be larger than or equal to bucket $N - 1$. While the buckets could be treated as normal time-series data and predicted accordingly, there is no guarantee that the invariant will hold with every bucket being predicted with individual prediction models. This would call for a more sophisticated solution. Due to this and the additional implementation complexity required, it was decided not to include support for these types in the framework. This means that when using the Prometheus Node Exporter, any histograms or summaries were excluded from processing by the framework.

6.3.3 gzip Compression

Due to an oversight in the initial implementation phase, the TestClient only applies gzip [38] compression in baseline tests if it exposes several metrics. Although running Prometheus natively with the Prometheus Node Exporter always applies gzip compression, as we discovered through inspection of packets in Wireshark, we chose

not to rerun all tests. Instead, we prioritized only the cases where the payload was large enough to benefit from compression. Furthermore, applying gzip to payloads with only a single metric proved ineffective as it *inflated* the payload size while only adjusting the size by a minimal amount. We concluded that Prometheus may not be optimized for transmitting such small payloads and that omitting compression in these cases is preferable. While this introduces a slight deviation from the true baseline, its impact on the overall conclusion is likely limited due to the relative insignificance of compression on such small payloads.

6.3.4 Communication Protocol Limits

Another detail that was not fully considered before entering the testing phase was the size of the fields in the custom communication protocol (see Figure 4.2). Allocating four bytes for message length and message type each is excessive for the use case of the framework. The entire header could be reduced to four bytes, which may itself still be larger than necessary. The reason for allocating a large size was to allow for the future introduction of additional fields if needed, such as protocol or sender identification. Once the testing phase had begun, the 8-byte header remained unchanged, and correcting its size and rerunning the tests was deemed unjustified due to time constraints.

6.3.5 TestClient-Induced Latency Issues

Initially, abnormally high latencies were measured, with some readings reaching up to 600 milliseconds, even for low values of M. This greatly exceeds the set $\delta = 100ms$ used in the simulation-based tests and was diagnosed to stem from the TestClient and the node-side proxy. It was hypothesized that the problem was caused by slow I/O and garbage-collection operations. This was subsequently remedied by parallelizing writes, reusing components to appease the garbage collector, and pre-allocating space for the dynamic arrays used in both the TestClient and node-side proxy. The tests that displayed the high latencies were repeated, and they no longer suffered from this problem. That being said, the live tests with high values of M had occasional high readings, leading to lost updates. Although this happened under heavy system load, additional optimizations should still be implemented to resolve the problem.

6.4 Dataset Considerations

The time constraints of the work meant that evaluation across more configurations was prioritized over evaluation across further datasets. Although this gave more nuance to the analysis of the results gathered across the selected datasets, more variety in the chosen datasets would have been beneficial. This could better represent the framework's true performance by capturing a wider range of edge cases and data distributions.

For the majority of the hardware-based evaluation, tests were conducted using pre-

collected federated learning datasets. Using pre-collected datasets even at this stage was prioritized mainly for the reason of streamlining the hardware-based evaluation, as the framework was already configured to handle pre-collected data. At the same time, the method retained the advantage of utilizing real data collected on the same devices used in each test. Additionally, having pre-collected data meant having an understanding of the data's appearance before each test, and created consistency across several tests, enabling easier comparison.

The evaluation phase showed significant variation in error across different datasets. In particular, a clear difference can be seen between the Ericsson and IntelLab tests. This variation can be attributed to differences in the underlying characteristics of the datasets, such as their ranges (Ericsson: 97.07, IntelLab: 44.64), among other factors. Furthermore, the IntelLab dataset is significantly larger than the Ericsson dataset, meaning that the full dataset has not been utilized in any test. As a result, normalizing the error over the full dataset range may be somewhat misleading. However, the approach was chosen to enable comparability between tests across the same data.

6.5 Future Work

There exist several opportunities to further build upon this work. One direction could be to further develop the current framework into a more robust middleware solution, generalized to facilitate integration with other monitoring tools. Alternatively, the framework's functionality could be directly built into Prometheus to investigate how removing the added complexity of the proxies might affect the results.

Regardless of direction, several possible improvements are applicable to both approaches. To further build upon the work of FEDAMON [2], data-aware model selection could be implemented, such that the framework would adapt model choice based on each data stream. In order to make the framework more applicable to real-life scenarios, fault tolerance and detection should be prioritized, which has been out of the scope of this project. Introducing fault tolerance and detection would most likely require some communication overhead, such as a heartbeat. The challenge of this task would therefore be to devise a communication-efficient solution such that the fault handling stays aligned with the framework's objectives. Finally, the framework could be extended with self-configuring properties, so that it could automatically tune many of the configurational parameters that currently must be set manually.

7

Conclusion

The proposed proxy-based framework proved effective in reducing communication overhead in Prometheus through forecast-based monitoring. By incorporating the prediction models Simple Approximation and Piecewise Linear Approximation in combination with Dynamic Error Threshold calculation over a custom application-layer protocol, the number of transmitted network packets was reduced by 77.4% on average while maintaining a mean error of 1.13%. The CPU usage did not increase when running the idle tests; instead, it decreased by 3.01% on the nodes and by 9.67% on the coordinator device. The framework is also suited for Prometheus in the sense that it accommodates the possibility for a given node to monitor several metrics at once, maintaining a prediction model for each. The possibility of extending the framework with metric priority levels was explored, showing potential for more fine-grained control over the trade-off between communication ratio and error.

The framework was able to be thoroughly evaluated using a custom evaluation methodology, measuring mean absolute error, communication ratio, update latency, as well as CPU and memory usage. Furthermore, the framework was evaluated in two different settings, namely in a simulation-based environment and a physical test bench consisting of Raspberry Pis.

The framework functions as a proof of concept, and work remains before it can be employed in a production environment. Improvements in fault detection, handling, and efficiency are necessary. The framework also currently relies upon manually configuring parameters for the prediction models and dynamic epsilon, limiting its adaptability. For future work, a data-aware model selection should be considered to fully realize the potential of utilizing forecast-based monitoring within Prometheus.

Bibliography

[1] *Overview | Prometheus — prometheus.io*, [Accessed 03-12-2025], 2025. [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[2] Y. Zhang and R. Duvignau, "Fedamon: A forecast-based, error-bounded and data-aware approach to continuous distributed monitoring," in *Proceedings of the 19th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '25, Association for Computing Machinery, 2025, pp. 39–50, ISBN: 9798400713323. DOI: 10.1145/3701717.3730544. [Online]. Available: https://doi.org/10.1145/3701717.3730544.

[3] A. S. Tanenbaum and M. v. Steen, *Distributed Systems*, 3rd ed. Marten Van Steen, 2020.

[4] I. Amazon Web Services, *What is cloud computing?* [Accessed: 2026-05-22]. [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/.

[5] CERN, *HPC clusters at CERN - ABP Computing at CERN*, [Accessed: 2026-05-22]. [Online]. Available: https://abpcomputing.web.cern.ch/computing_resources/hpc_cern/.

[6] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016, Accessed: 03-12-2025. [Online]. Available: https://sre.google/sre-book/introduction/.

[7] *Sensu go*, [Accessed 08-12-2025]. [Online]. Available: https://docs.sensu.io/sensu-go/latest/.

[8] *Influxdb oss v2 documentation*, [Accessed 08-12-2025]. [Online]. Available: https://docs.influxdata.com/influxdb/v2/.

[9] *Zabbix documentation*, [Accessed 08-12-2025]. [Online]. Available: https://www.zabbix.com/documentation/current/en/.

[10] J. Turnbull, *Monitoring with Prometheus*. Brooklyn, NY: Turnbull Press, 2018.

[11] *Defining recording rules | Prometheus — prometheus.io*, [Accessed 24-02-2026]. [Online]. Available: https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/.

[12] *Alertmanager | Prometheus — prometheus.io*, [Accessed 27-03-2026]. [Online]. Available: https://prometheus.io/docs/alerting/latest/alertmanager/.

[13] *Client libraries | Prometheus — prometheus.io*, [Accessed 30-01-2026]. [Online]. Available: https://prometheus.io/docs/instrumenting/clientlibs/.

[14] *Monitoring Linux host metrics with the Node Exporter | Prometheus — prometheus.io*, https://prometheus.io/docs/guides/node-exporter/, [Accessed 29-06-2026].

[15] *Exporters and integrations | Prometheus — prometheus.io*, [Accessed 27-05-2026]. [Online]. Available: https://prometheus.io/docs/instrumenting/exporters/.

[16] *GitHub - prometheus/node_exporter: Exporter for machine metrics — github.com*, [Accessed 27-03-2026]. [Online]. Available: https://github.com/prometheus/node_exporter.

[17] Y.-A. Le Borgne, S. Santini, and G. Bontempi, "Adaptive model selection for time series prediction in wireless sensor networks," *Signal Processing*, vol. 87, no. 12, pp. 3010–3020, 2007, Special Section: Information Processing and Data Management in Wireless Sensor Networks, ISSN: 0165-1684. DOI: https://doi.org/10.1016/j.sigpro.2007.05.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165168407001879.

[18] J. Körner, S. Bach, A. Karlsson, L. Sundkvist, and R. Duvignau, "Efficient monitoring of cps and iot systems: A deployment guide for empirical evaluations," in *2024 13th Mediterranean Conference on Embedded Computing (MECO)*, 2024, pp. 1–6. DOI: 10.1109/MECO62516.2024.10577823.

[19] G. Cormode, "Continuous distributed monitoring: A short survey," in *Proceedings of the First International Workshop on Algorithms and Models for Distributed Event Processing*, ser. AlMoDEP '11, Rome, Italy: Association for Computing Machinery, 2011, pp. 1–10, ISBN: 9781450309226. DOI: 10.1145/2031792.2031793. [Online]. Available: https://doi.org/10.1145/2031792.2031793.

[20] I. Sharfman, A. Schuster, and D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams," *ACM Trans. Database Syst.*, vol. 32, Nov. 2007. DOI: 10.1145/1292609.1292613.

[21] Y. Alfassi, M. Gabel, G. Yehuda, and D. Keren, "A distance-based scheme for reducing bandwidth in distributed geometric monitoring," Apr. 2021, pp. 1164–1175. DOI: 10.1109/ICDE51399.2021.00105.

[22] A. Jain, E. Y. Chang, and Y.-F. Wang, "Adaptive stream resource management using kalman filters," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04, Paris, France: Association for Computing Machinery, 2004, pp. 11–22, ISBN: 1581138598. DOI: 10.1145/1007568.1007573. [Online]. Available: https://doi.org/10.1145/1007568.1007573.

[23] H. Jiang, S. Jin, and C. Wang, "Prediction or not? an energy-efficient framework for clustering-based data collection in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1064–1071, 2011. DOI: 10.1109/TPDS.2010.174.

[24] *Write data to influxdb*, [Accessed 13-02-2026]. [Online]. Available: https://docs.influxdata.com/influxdb/v2/write-data.

[25] J. Postel, *Transmission Control Protocol*, Sep. 1981. DOI: 10.17487/RFC0793. [Online]. Available: https://www.rfc-editor.org/info/rfc793.

[26] J. Postel, *User Datagram Protocol*, Aug. 1980. DOI: 10.17487/RFC0768. [Online]. Available: https://www.rfc-editor.org/info/rfc768.

[27] S. Z. Tianhong Tan, *Piecewise linear approximation - Cornell University Computational Optimization Open Textbook - Optimization Wiki — optimization.cbe.cornell.edu*,

[Accessed 18-02-2026], 2021. [Online]. Available: https://optimization.cbe.cornell.edu/index.php?title=Piecewise_linear_approximation.

[28] Wireshark Foundation, *Tshark(1) manual page*, [Accessed: 2026-05-25]. [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html.

[29] D. Song, *Dpkt 1.9.2 documentation — dpkt.readthedocs.io*, [Accessed 30-03-2026]. [Online]. Available: https://dpkt.readthedocs.io/en/latest/.

[30] M. Cotton, L. Eggert, D. J. D. Touch, M. Westerlund, and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry," Tech. Rep. 6335, Aug. 2011, 33 pp. DOI: 10.17487/RFC6335. [Online]. Available: https://www.rfc-editor.org/info/rfc6335.

[31] R. Duvignau, M. Papatriantafilou, K. Peratinos, E. Nordström, and P. Nyman, "Continuous distributed monitoring in the evolved packet core," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '19, Darmstadt, Germany: Association for Computing Machinery, 2019, pp. 187–192, ISBN: 9781450367943. DOI: 10.1145/3328905.3329761. [Online]. Available: https://doi.org/10.1145/3328905.3329761.

[32] S. Madden, *Intel Lab Data — db.csail.mit.edu*, [Accessed 30-03-2026], 2004. [Online]. Available: https://db.csail.mit.edu/labdata/labdata.html.

[33] *Flower 2.0.0 documentation — flower.readthedocs.io*, [Accessed 12-05-2026]. [Online]. Available: https://flower.readthedocs.io/en/latest/.

[34] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:18268744.

[35] *API Documentation TensorFlow v2.16.1 — tensorflow.org*, [Accessed 15-05-2026], 2024. [Online]. Available: https://www.tensorflow.org/api%5C_docs.

[36] *Keras: The high-level API for TensorFlow TensorFlow Core — tensorflow.org*, [Accessed 27-05-2026]. [Online]. Available: https://www.tensorflow.org/guide/keras.

[37] Giam, *Psutil: Process and System Utilities for Python — psutil.readthedocs.io*, https://psutil.readthedocs.io/latest/index.html, [Accessed 26-06-2026], 2026.

[38] *The gzip home page*, [Accessed: 2026-05-25]. [Online]. Available: https://www.gzip.org/.

[39] *MessagePack: It's like JSON, but fast and small. — msgpack.org*, [Accessed 27-03-2026]. [Online]. Available: https://msgpack.org/.

A

Evaluation configurations

Table A.1: Configurations for simulation-based evaluation

ID	Dataset	Duration	Model	p_ϵ
0	Intel	4000	PLA-8	10
1	Ericsson	1500	PLA-8	10
2	Ericsson	1500	PLA-8	2.5,5,7.5,10,12.5,15,17.5
3	Intel	1500	PLA-8	2.5,5,7.5,10,12.5,15,17.5
4	Ericsson	1500	PLA-(4,6,8,10,12,14)	10
5	Intel	4000	PLA-(4,6,8,10,12,14)	10
6	Ericsson	1500	PLA-8	10
7	Intel	4000	PLA-8	5
8	Ericsson	1500	SA	10
9	Intel	4000	SA	10
10	Ericsson	1500	SA	2.5,5,7.5,10,12.5,15,17.5
11	Intel	4000	SA	2.5,5,7.5,10,12.5,15,17.5
12	Intel	4000	SA	5
13	Ericsson	1500	SA	10
14	Intel	4000	SA	10
15	Ericsson	1500	PLA-8	10
16	Ericsson	1500	PLA-8	10
17	Intel	4000	PLA-8	5
18	Ericsson	1500	SA	10
19	Ericsson	1500	SA	10
20	Intel	4000	SA	5

ID	L_ϵ	TP	N	M	δ
0	100,300,500,700,900	TCP	40	1	100
1	100,300,500,700,900	TCP	20	1	100
2	300	TCP	40	1	100
3	300	TCP	40	1	100
4	300	TCP	40	1	100
5	1000	TCP	40	1	100
6	100,300,500,700,900	UDP	40	1	100
7	100,300,500,700,900	UDP	40	1	100
8	100,300,500,700,900	TCP	40	1	100
9	100,300,500,700,900	TCP	40	1	100
10	300	TCP	40	1	100
11	300	TCP	40	1	100
12	300	TCP	2	5,10,15,20	100
13	100,300,500,700,900	UDP	40	1	100
14	100,300,500,700,900	UDP	40	1	100
15	300	TCP	20	5,10,15,20	100
16	300	UDP	20	5,10,15,20	100
17	300	TCP	2	5,10,15,20	100
18	300	TCP	20	5,10,15,20	100
19	300	UDP	20	5,10,15,20	100
20	300	TCP	2	5,10,15,20	100

Table A.2: Configurations for hardware-based evaluation

ID	Dataset	Duration	Model	p_ϵ
1	CpuAvg	600	PLA-3	5
2	CpuAvg	600	PLA-8	5
3	CpuAvg	600	SA	5
4	CpuAvg	600	PLA-8	2.5,5,7.5,10,12.5,15,17.5
5	CpuAvg	600	PLA-3	2.5,5,7.5,10,12.5,15,17.5
6	CpuAvg	600	SA	2.5,5,7.5,10,12.5,15,17.5
7	CpuAvg	600	PLA-(3,5,7,8,15, 20,25,29,39)	5
8	CpuPerCore	600	PLA-3	2.5,5,7.5,10,12.5,15,17.5
9	CpuPerCore	600	PLA-8	2.5,5,7.5,10,12.5,15,17.5
10	CpuPerCore	600	PLA-8	2.5,5,7.5,10,12.5,15,17.5
11	CpuPerCore	600	SA	2.5,5,7.5,10,12.5,15,17.5
12	CpuPerCore	600	SA	2.5,5,7.5,10,12.5,15,17.5
13	CpuPerCore	600	PLA-3	2.5,5,7.5,10,12.5,15,17.5
14	Ericsson	600	PLA-8	10
15	Ericsson	600	PLA-8	10
16	NodeExporter	14m38s-15m6s	SA	10
17	NodeExporter	14m18s-16m1s	SA	10
18	CustomExporter	14m28s-15m11s	SA	10,20,30

ID	L_ϵ	TP	N	M	δ
1	100,200,300,400	TCP	4	1	100
2	100,200,300,400	TCP	4	1	100
3	100,200,300,400	TCP	4	1	100
4	300	TCP	4	1	100
5	300	TCP	4	1	100
6	300	TCP	4	1	100
7	200	TCP	4	1	100
8	200	UDP	4	4	100
9	200	TCP	4	4	100
10	200	UDP	4	4	100
11	200	TCP	4	4	100
12	200	UDP	4	4	100
13	200	TCP	4	4	100
14	300	TCP	4	1,50,100,150	100
15	300	TCP	4	150	100
16	200	TCP	4	399	250
17	200	UDP	4	399	250
18	200	TCP	4	19	250