TU WIEN Informatics

Effizientes Verarbeiten von IoT-Datenströmen

Evaluierung von imperativem und deklarativem Programmieren in ressourcenbeschränkten Einsatzgebieten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Anton Oellerer, BSc
Matrikelnummer 01429853

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Víctor Casamayor Pujol

Wien, 8 Mai, 2024

Anton Oellerer	Schahram Dustdar

TU WIEN Informatics

Efficent processing of IoT data streams

Evaluating imperative and declarative programming in resource-constrained environments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Anton Oellerer, BSc
Registration Number 01429853

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Assistance: Dr. Víctor Casamayor Pujol

Vienna, 8th May, 2024

Anton Oellerer	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Anton Oellerer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8 Mai, 2024

Anton Oellerer

Danksagung

An erster Stelle möchte ich meinem Diplomarbeitsbetreuer Victor danken, der mich durch die Entstehung dieses Werkes begleitet hat, und trotz des langen Zeitraumes nie die Energie verloren hat, mir mit Rat und Korrektur zur Seite zu stehen. Ebenso möchte ich meiner Freundin Hanni für die viele Geduld danken, die sie aufgebracht hat, während ich mich wieder in meinem Laptop verloren hatte, anstatt das Wochendende gemeinsam zu verbringen. Letztlich danke ich auch meiner Familie für die moralische Unterstützung, die sie mir während meiner Studienzeit entgegengebracht haben, und, dass sie mir (hoffentlich) immer verziehen haben, wenn Familienzeit zu Gunsten meines Studiums geopfert wurde.

Acknowledgements

First, I want to thank Victor, my supervisor for this thesis, who accompanied me through this project, and, even though it span multiple years, never lost the energy to help me and opine on drafts whenever necessary. I also want to thank my girlfriend Hanni for all the patience she showed whenever I lost myself in my laptop instead of spending the weekend together. Lastly I also give thanks to my family for providing me with moral support during all of my studies, and for (hopefully) always forgiving me, whenever I sacrificed family time in favor of studying.

Kurzfassung

Um die großen Mengen an Daten, welche durch das “Internet der Dinge” produziert werden, nutzen zu können, wurde das “Edge Computing”-Softwarearchitekturparadigma entworfen, bei dem Prozessoren in der Nähe der Datenquellen deren Messungen auf das Wesentliche reduzieren. Dadurch muss weniger Information durch das Internet geschickt werden, und die Anzahl der Verarbeitungsschritte auf den zentralen Servern verringert sich. Die Geräte, auf denen die Vorverarbeitung durchgeführt wird, sind aber für gewöhlich weitaus weniger leistungsfähig, und besitzen teilweise nur eine beschränkte Stromzufuhr. Dementsprechend ist es wichtig, dass die eingesetzten Programme Rücksicht auf die Ressourcenbegrenzungen nehmen, und so effizient wie möglich implementiert werden. Für deren Umsetzung kann entweder dem imperativen oder dem deklarativen Programmierparadigma gefolgt werden. Während das deklarative Programmierparadigma aufgrund seines höheren Abstraktionslevels Vorteile bei der Parallelisierung bieten kann, ist nicht klar, inwiefern sich dies auf den Ressourcenverbrauch auswirkt.

Das Hauptziel dieser Arbeit ist es, durch Handlungsempfelungen für die Implementierung von Programmen, die “Internet der Dinge”-Daten verarbeiten, die Effizenz von zukünftigen “Edge Computing” Systemen zu verbessern. Im Zuge dieser Arbeit wurden mehrere Artefakte hergestellt, die in zukünftigen Projekten eingesetzt werden können. Im ersten Schritt wurde ein Ablauf zur Definition von Benchmarks angelegt. Diesem Ablauf folgend wurde ein Benchmark-Szenario entwickelt, mithilfe welchem die Forschungsfragen beantwortet werden konnten. Nachdem dieses Szenario implementiert wurde, wurden damit zwei Services evaluiert, wobei einer mittels des imperative Paradigmas und der andere mittels des deklarativen Paradigmas erstellt wurde. Die Messungen die dabei anfielen wurden gespeichert und zur Beantwortung der Forschungsfragen genutzt.

Die Ergebnisse dieser Messungen zeigen, dass die imperative Implementierung in dem gegebenen Kontext weniger Ressourcen verbraucht und die Daten schneller verarbeiten kann. Gründe dafür sind der größere Ressourcenverbrauch, der mit dem höheren Abstraktionslevel einhergeht und, dass die imperative Implementierung besser an das Benchmark-Szenario angepasst werden konnte.

Abstract

Edge Computing is a software-architectural paradigm which employs data preprocessors close to "Internet of Things" sensors to reduce measurements down to their vital parts before forwarding them. This lowers the amount of information which has to be transmitted over the internet, and less work needs to be done on central servers performing the final analysis. To realize Edge Computing, large numbers of data preprocessors are needed. So that the paradigm is economically viable, those preprocessors need to be cheap, which means that their performance is often constrained. Due to this, services running on those processors need to be designed with the resource limitations in mind, and run as efficiently as possible. Implementations of such services can either follow the declarative or the imperative programming paradigm. When following the imperative paradigm, data infrastructural tasks, like where temporal values are saved, are specified on the same level of abstraction as a programs business logic. In the declarative paradigm on the other hand, those two issues are handled on separate layers. While hiding the infrastructural concerns from the application developers means that parallelization can be implemented easier, it is not clear how this impacts resource consumption.

The main goal of this thesis is to increase the efficiency of future Edge Computing systems by offering insights into the effects of following the declarative or imperative paradigm when implementing a service processing "Internet of Things" data streams. Over the course of this thesis, multiple artifacts were created and made publicly available, which can be used in further work. First, a workflow to design benchmark scenarios was developed. Following this workflow, a benchmark scenario appropriate for answering the research questions was designed. Afterwards, the scenario and additional necessary tooling was implemented. Two services, one following the declarative, one following the imperative paradigm, and a library to handle the data-infrastructural tasks for the declarative implementation was created. Finally, the services were benchmarked, and their performance was measured.

The results show that, in the given context, the imperative implementation is able to process the data faster and more efficiently. Reasons for this are the additional costs coming with a higher level of abstraction, and that the imperative implementation can be tailored better to the scenario at hand.

Contents

CHAPTER 1

Introduction

Innovations in the hardware sector over the last two decades enable companies, researchers, and average consumers to deploy cheap internet-connected sensors in every imaginable place, from weather balloons to blast furnaces to thermostats. Cisco estimates that by 2030 there will be 500 billion devices connected to the internet or another network, of which a large amount will be simple sensors periodically reporting data about their environment [1, 2].

1.1 Motivation

The drastic increase in sensors employed for a multitude of purposes also introduces new communication patterns which did not exist before. First, the way autonomous sensors operate is entirely different from how user-operated appliances act. A smartphone, for example, idles until it receives user input. Once such input requires remote resources, like a web page, the phone sends a request to a server, awaits its response, and processes it on arrival. A sensor on the other hand operates without third-party input, sending its readings to a designated computing node (which might be a server) at certain intervals, not expecting a response. Other interaction models do exist, but the differences between those two highlight the issues relevant for this thesis.

Second, the pattern of the transferred data of user-operated devices is usually very unpredictable. For example, a typical interaction with a messaging service consists of downloading and uploading texts and messages, which vary greatly in size (a few kilobytes for text, a couple of megabytes for images) and in frequency (one interaction might consist of hundreds of messages in one hour, followed by silence for multiple days). All of those scenarios have to be served over the same network infrastructure and by the same servers. Sensors on the other hand send a very predictable set of messages, at very predictable times. A temperature sensor sends a tiny message only containing a few

numbers on each update, while an imaging sensor (like an infrared camera surveying an area for forest fires) always sends a picture of about the same size.

Third, the amount of devices in both scenarios differs greatly, and will only continue to diverge further in the future. The amount of user-operated devices is limited by the amount of users, about 9 billion, and the amount of devices a user can operate concurrently, which one can hardly imagine exceeding more than a couple. Conversely, the amount of existing sensors is only limited by their costs, which are decreasing constantly due to increasing production capabilities. Additionally, the decline in sensor prices makes further use cases profitable. To put it in absolute numbers, Nicolas de Loisy estimated that there were about 8 billion internet-connected end-user devices versus 20 billion Internet of Things (IoT) devices in 2019, and we can only expect this gap to have grown over the last four years [3].

1.2 Problem Statement

This rise of the Internet of Things is accompanied by a stark change in the composition and communication of the devices connected to the internet. To be able to adapt to this, professionals of all involved trades need a fitting set of tools, be it specialized hardware for device designers, assistance for distributed deployment for DevOps engineers, or efficient programming paradigms for application developers [4]. While there already exists guidance for other environments, developers lack resources on how the choice of programming paradigm influences the performance of services running on resource constrained devices.

Those devices play an important role in the development of IoT systems, as they can be employed to preprocess sensor readings at their point of production, thus reducing the amount of data sent over the internet [5, 6]. While this architectural design, known as "edge computing", is emerging as an essential tool in distributed application development, its reliance on hardware devices of limited power and computing capabilities places an additional burden on the developers, which have to design the data processing services with those constraints in mind.

When creating such a service, one of the first decisions developers have to make is about which programming paradigm they want to use for implementing it. While the list of specific paradigms which can be used is large, they are usually classified into being either *imperative* or *declarative*. The elementary difference between those two models is whether the infrastructural aspects of the data processing are implemented on the same abstraction level as the business logic. To put it into an example, a script applying two transformations to a variable in an imperative language could look like the following:

```
function transform(x):
    y = x + 1
    x = y * 2
```

while a declarative solution could be implemented like this:

```
function transform(x):
    return (x + 1) * 2
```

As one can see, in the second example we do not care about where to store the intermediate or the final value, but trust the underlying execution procedure to handle this for us.

Comparing both paradigms, there are advantages on either side. Naturally, the efficiency of declarative implementations is very dependent on how good the lower-level procedure can optimize the data infrastructural operations for the problem at hand. Furthermore, while not having to focus on how data is moved through the system can lead to cleaner code, there are cases where making it explicit allows easier understanding of the algorithm [7, 8, 9].

Connecting this with the above-mentioned changes in device communication models, one can see that it is important that developers are aware of the performance and resource efficiency implications of the considered programming paradigms.

1.3 Research Questions

To be able to assist developers creating IoT systems, the goal of this work is to determine the performance tradeoffs between using a declarative approach versus an imperative approach when implementing an Internet of Things data processing service operating in a resource constrained environment.

RQ1: How does the utilization of hardware resources of a stream processing service implemented following a declarative paradigm differ from an implementation following an imperative approach? To be able to assess the performance of the implementations, a number of readings about the runtime performance of the service will be taken. Those measurements include CPU load, processing time, and memory usage.

RQ2: How does the processing latency of a stream processing service implemented following a declarative paradigm differ from an implementation following an imperative approach? Another factor to consider, especially in systems where the timeliness of results has importance, is the duration an implementation needs to process the incoming data. To answer this question, the time difference between when a message first occurs in the system to when it arrives at its final destination will be measured.

RQ3: What do the differences in resource utilization and message latency mean for practitioners aiming to implement data stream processing on the edge? While RQ1 and RQ2 evaluate the low-level differences of the compared paradigms, RQ3 aims to summarize those learnings to make recommendations for future edge stream processing services.

1.4 Hypotheses

Sourcing from personal experience and the conducted literature review, we suspect to encounter the following results after evaluating an imperative and a declarative implementation of an IoT data processing service.

The imperative implementation is more resource-efficient As the imperative implementation can be tuned more closely to the problem, unnecessary computations, thread creations, and inter-process communication can be avoided better than in the declarative implementation. This leads to less CPU cycles and memory being wasted.

The processing latency of the declarative implementation is more constant under varying system load As declarative implementations are usually easier to parallelize, since no state needs to be shared between processing steps, the declarative solution should be able to better deal with very high system load, maintaining a steady processing latency.

The different advantages of both implementations require practitioners to make decisions about the processing technique to use on a case-by-case basis As the imperative implementation is more resource-efficient than the declarative one, but has a higher message latency, practitioners looking for a guidance on what paradigm to use for their IoT data processing service need to consider the environment and the requirements of their individual scenario.

1.5 Methodological Approach

The central methodology to evaluate the advantages and disadvantages of using a declarative versus an imperative approach to implement an IoT data processing system will be a comparison on a set of hardware measurement taken by a benchmark defined in this thesis.

The benchmark scenario will be constructed after a literature review determining the most important use cases for IoT data processing, edge computing and stream processing. It should be designed in such a way that no paradigm has initial advantages.

The services to compare will be prototypes able to perform the specified tasks efficiently, the declarative implementation using an abstraction over the data infrastructural issues, and the imperative implementation handling data processing and forwarding on the same level.

Once those services are constructed, the benchmarks will be run to determine their runtime and space efficiency.

Finally, the results will be documented, evaluated, and put into context to allow discussion about the benefits of the two paradigms in resource-constrained environments.

1.6 Contributions

Multiple contributions have been made over the course of this thesis, not only to the academic body of knowledge, but also to the tools which can be used by developers to implement services in this or other IoT-related use-cases.

The first academic contribution is the assembling of *research literature* concerning big data processing, stream processing, edge processing, and, at their intersection, IoT data stream processing in resource constrained environments. As became evident during the research done for this thesis, especially the aspect of the efficiency of different programming techniques to be used during implementation of services operating in this context has neither been examined thoroughly, not has related literature been assembled yet. This gap has now been filled by this work.

Another academic contribution is a *defined process for constructing a benchmark* for a given problem space. As benchmarks are a central instrument to determine dynamic properties of a system, a clearly documented procedure for creating them is necessary. This allows fitting a created scenario to another context, and to compare different benchmarks originating from different academic sources. Additionally, it increases the reproducibility of work, as the taken steps need to be documented explicitly.

Consequently, the *benchmark scenario* constructed by the defined process is another scientific contribution. It allows the comparison of multiple services processing data emitted by a chosen number of sensors. The goal of those services is to find irregularities in the received data, which are then reported to a cloud server. After the execution of one benchmark run, the resource consumption and message latency of the employed services are recorded, so that they can be compared later on.

At the scientific center of this thesis are the *results* obtained by the execution of the benchmark suite, the performance measurements of the declarative and the imperative data processing service, and with them the testing of the hypotheses and the answering of the research questions. Those answers serve to fill an existing gap in the field of programming paradigm efficiency, specifically looking at the aspect of their resource consumption when processing IoT data streams.

On the practical side, the *benchmark suite* which was implemented for the created benchmark scenario, and which was used to compare the two data processing services, is made publicly available. As it has been constructed in such a way that its execution is independent of the services to evaluate, it can be reused outside the context of this thesis. Services confirming to the specifications outlined in Section 3.3 can be added to the system, and after the execution of the suite its performance measurements can be retrieved. Furthermore, by splitting the different tasks of the benchmark execution into multiple services with clearly defined responsibilities, it is possible to modify both the scenario and execution modalities in the respective services, without having to do changes over the full codebase.

Multiple contributions to the open source ecosystem have been made. Just as the benchmark suite, the two data processing services evaluated are accessible in the codebase of this thesis. Furthermore, two additional prototypes evaluated at the start of this thesis are included as well[1]. Those services can be used as a reference for how IoT data stream processing can be implemented, available techniques, and how viable they currently are. Another open-source contribution is the declarative data processing library which has been used for implementing the service following the declarative paradigm[2]. It can be used for creating multithreaded data processing services, reference for other libraries implementing a declarative processing backend, or as a starting point for a production-grade library. Lastly, during the creation of this thesis, some bugs have been reported and fixed in Rust packages which provide facilities for data stream processing[3] [4] [5].

1.7 Structure

The thesis is built up in the following way:

1. The first chapter introduced the problem and stated the goals of this thesis.
2. The second chapter explores the scientific background of the central issue and the research questions. Furthermore, the state-of-the-art in related areas is documented, and solutions for similar issues are presented.
3. The third chapter is concerned with the creation and implementation of a benchmark scenario to evaluate the two programming paradigms.
4. The fourth chapter describes the implementation of the two data stream processing prototypes.
5. In the fifth chapter, the results of the benchmark are documented.
6. The sixth chapter analyzes the findings of the performance comparisons, relates them to the outcomes of previous academic work, and connects them to the research questions. Additionally, the limitations of the outcomes of this thesis are listed and explained in-depth.
7. In the seventh and final chapter, a summary of the thesis is given, and future research areas are proposed.

[1] `https://github.com/AntonOellerer/Reactive-Streaming-on-the-Edge` (visited on 2024-05-08)
[2] `https://github.com/AntonOellerer/rx_rust_mp` (visited on 2024-05-08)
[3] `https://github.com/rxRust/rxRust/pull/202` (visited on 2024-05-08)
[4] `https://github.com/SpringQL/SpringQL/pull/257` (visited on 2024-05-08)
[5] `https://github.com/SpringQL/SpringQL/pull/260` (visited on 2024-05-08)

CHAPTER 2

Background & State-of-the-Art

Data processing on the edge is an issue at the intersection of many topics such as "Internet of Things", "big data processing", and "edge computing". Explaining the relevant terms in depth to make both the problem and the solution comprehensible, and exploration of existing research in this area is the goal of this chapter.

2.1 Background

The issue of efficiently processing data streams, and employing the correct programming techniques to achieve this has already been examined from multiple angles. In this section, significant results of previous work are analyzed.

2.1.1 Definition of Terms

Two central terms used throughout this thesis are *imperative programming* and *declarative programming*. To avoid confusion stemming from different definitions of these terms in the programming vocabulary, it is important to clarify their meaning in the context of this thesis beforehand.

Imperative Programming

> *Imperative programming languages are, to varying degrees, abstractions of the underlying von Neumann computer architecture. The architecture's two primary components are its memory, which stores both instructions and data, and its processor, which provides operations for modifying the contents of the memory. The abstractions in a language for the memory cells of the machine are variables.*
>
> Sebesta
>
> *Concepts of Programming Languages* [10]

Putting this definition into plain text, a key aspect of imperative programming is that programmers have to deal with storage and retrieval of data on the same level of abstraction as they implement business logic. For clarity, the example given in the introduction is printed here again:

```
1 function transform(x):
2     y = x + 1
3     x = y * 2
```

In lines 2 and 3, additionally to defining the calculations, it is also stated where to store the results. This is in close correspondence to what the CPU (a von Neumann machine) does: increment the value stored in cell x by one and store the result in cell y, multiply the value stored in y by 2 and store it in x. As one can see, the "infrastructural" logic (where to store the results) is intertwined with the business logic of the function (incrementing a value by one and doubling the result). This means additional cognitive load for programmers trying to debug issues in programs written following the imperative paradigm, as they have to consider the evaluated logic as well as the storage places of used variables and their modification.

Declarative Programming

> *A declarative operation is independent (does not depend on any execution state outside itself), stateless (has no internal execution state that is remembered between calls), and deterministic (always gives the same results when given the same arguments).*
>
> Roy and Haridi
>
> *Concepts, Techniques, and Models of Computer Programming* [11]

The basic trait of declarative programming is that its results are produced by the application of stateless functions to the input values. Again, looking at the example presented earlier:

```
1 function transform(x):
2         return (x + 1) * 2;
```

Here, the programmer only declares the necessary logic and leaves the issues of store and load operations to the underlying procedure managing the memory. While this allows for more concise code, declarative programming also has its drawbacks, which can make it unsuitable for certain kinds of problems. One of those issues is that some tasks can be solved easier if keeping state in a function is possible. An example for this are graph searches, where imperative implementations can simply keep a list of all nodes to visit and loop over it. Implementations following the declarative paradigm have to handle the search with recursion, thus leading to code that is possibly harder to understand [12].

2.1.2 Internet of Things

The amount of data being produced by internet-connected devices is growing steadily due to cheaper sensors and expanding areas of usage such as Smart Cities, Industry 4.0 and home automation [6, 13, 14]. Those application fields can be subsumed under the term *Internet of Things (IoT)*.

The "Internet of Things" is the composition of all autonomously operating internet connected devices, which is thus made up out of both actuators (such as robot vacuums) and sensors [15], orchestrated by processing units.

As seemingly most of the colloquial terms of the IT-world, the phrase was first used in a conference presentation, but did not rise to prominence until almost a decade later. In this specific case, the first mention of "Internet of Things" was by Kevin Ashton in 1999 [16], but broad relevance was only reached around 2009 according to a Cisco estimation, coinciding with the point from which onward more internet-connected devices than people existed [2].

The concept of autonomous internet-connected devices goes even further back to 1982, and can be dated to another pacemaker of innovation, the concern of people about the status of their caffeinated beverages, solved in this case by a vending machine reporting its filling level and temperature every couple of minutes [17].

Over the years since then, the IoT sector has evolved into a market valued at about 300 billion USD in 2020, as estimated by "Fortune Business Insights"[1].

The field of IoT is often segmented into its specific use cases, of which some of the most prominent ones are:

- Smart Home: Services tightly integrated with the living space of its users, such as thermostats regulating a rooms temperature based on whether people are in it or not [18].
- Smart City: Using sensors and actuators distributed through the city to enhance the living experience of its population, for example traffic systems which can guide drivers to the least occupied lanes on busy roads, reducing overall congestion [14].
- Smart Healthcare: Employing IoT services in the public health system. One use case is tracking RFID-tagged pill bottles from production to the hospital bed, to make sure that they are given to the correct patient [19].
- Industry 4.0: This term encompasses the large area of all services related to manufacturing and maintenance. One such appliance is "Predictive Manufacturing", which enables factories to increase or decrease their output based on the estimated remaining lifetime reported by the sensors in the products they sold [20].

[1] https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307 (visited on 2024-05-08)

While for smaller applications, like a smart thermostat, a single temperature sensor might suffice, more sophisticated services have to deal with readings sent by thousands of devices, as for example in the scenario presented in „SmartSantander: IoT Experimentation over a Smart City Testbed“ [21].

Once a certain number of sensors in a system is reached, traditional methods of dealing with the emitted data can struggle to fulfill their service level objectives, as their algorithms and interaction modes are not intended for dealing with IoT sensors. Zschörnig, Wehlitz, and Franczyk identified the following points as problems frequently encountered when analyzing sensor data streams [22]:

- Large amounts of data
- High frequency of data updates
- Multiple data sources
- (Near) real time handling

To deal with those issues, multiple techniques and paradigms have emerged, which are subsumed under the term *Big Data Processing*.

2.1.3 Big Data Processing

The term “Big Data” has first been used in the 1990s and is attributed to John R Mashey in *The Origins of 'Big Data'* by Steve Lohr [23, 24]. It is concerned with all issues related to transmitting, analyzing, and storing large amounts of information, of which a good overview is given in „Big Data Processing“ by Ji, Li, Qiu, *et al.* [25]. While the exact definition of how large a data problem has to be to be regarded as “Big” has not been agreed upon, the issue has received a lot of attention in academic research over the last decade [25, 26, 27].

One possible partitioning of the big data processing field is along the requirements of the timeliness of the analytical results:

1. Real-Time, which requires the results of the data analysis close to its delivery. The meaning of “real-time” is dependent on the issue at hand. *Stream processing* is an architectural implementation for real-time analysis, as it can be employed in such a way that analysis results are generated while processing the incoming data *stream*.
2. Not requiring an immediate outcome, where a timely delivery of the data is not as decisive as other factors. For such tasks, *batch-processing* is a possible pattern, where messages are collected to sets of a certain size or timespan and then processed together. Popular frameworks implementing it are Hadoop[2] and Apache Spark[3].

[2] `https://hadoop.apache.org/` (visited on 2024-05-08)
[3] `https://spark.apache.org/` (visited on 2024-05-08)

While they generally cover two different use cases, a distinctive advantage of real-time processing is that incoming data can be reduced in size immediately, as the significant elements are extracted from the stream at its point of entry into the analyzing system. This means that less storage capacity is needed, and consequently *Stream Processing* is a relevant solution approach for the problem at the core of this thesis.

Stream Processing

Data stream processing describes the process of analyzing data packages upon arrival in the system, as opposed to first storing them and only doing an analysis at scheduled points in time or upon request. There is neither a definition of the maximum amount of time between each package, nor is it restricted to bounded or unbounded streams, although in the IoT context unbounded streams are usually assumed. An example would be a simple processor receiving temperature readings from a sensor in a fridge every 5 seconds, and alerting if the received temperature exceeds 12 °C.

In its simplest variant, stream processing consists of analyzing each data package on its own and acting based on the result, but many tasks require more complex evaluation. An algorithm determining a fridges' loss of cooling capacity, for example, might need to aggregate data over a certain timespan, to prevent false alarms when its door is opened, and multiple sensor, to be able to stay operational if one sensor fails. As the complexity of the algorithm and the data intensity grows, traditional programming paradigms, such as object-oriented programming, can not provide the support needed to the programmer [28], and more sophisticated tools are required, such as Apache Storm[4] [29] and others [26, 30, 31, 32].

While employing stream processing in data centers allows saving processing and storage resources, as the data is reduced to its important parts as soon as possible, every sensor reading still has to be sent over the internet. This can result in high load on global connections, and overwhelming services in the data centers dealing with incoming connections before they are being processed, like firewalls and proxies. One solution for this is employing *edge computing*, where IoT readings are preprocessed close to the sensor, before the relevant data is sent over the internet.

2.1.4 Edge Computing

Edge computing and fog computing are architectural concepts proposing locating data processing nodes close to where the data is generated to reduce latency and network congestion and to increase the service reliability through decentralization. This is seen in opposition to cloud computing, where the largest part of the data is sent to central servers in geographically separated locations.

The exact differences between edge and fog computing have not been agreed upon, but Francis and Madhiajagan define them as follows [33, 34]:

[4] `https://storm.apache.org/` (visited on 2024-05-08)

- Edge Computing: Small "data centers" located closer to the data emitters than the central server.
- Fog Computing: Utilizing consumer hardware (notebooks, routers, ...) in vicinity of the emitters.

While the employment of such nodes can generally be very useful from an architectural perspective, there are a couple of problems which hinder the wide-spread usage of those paradigms, as for example reported in „What the Fog?" by Gedeon, Brandherm, Egert, *et al.* [35]:

First, from a business perspective it is often desirable to possess as much user data as possible, so filtering and transforming it outside the companies servers can be seen negatively by decision-makers less concerned about architectural than monetary aspects.

Second, while reducing the data being sent to the central servers is favorable from a privacy point of view, fog nodes might be devices pertaining to untrusted persons, and processing delicate data there raises further issues.

Third, fog devices such as mobile phones may not be available for computational offloading all the time, since they can be turned off or under load for longer periods. Furthermore, those periods can start at random and unannounced, so assigning tasks to run on the devices needs a lot of coordination.

Finally, scheduling on geographically distributed machines with varying computing capabilities and network connections necessities a lot of planning, since network transfer times have to be factored in, and tasks should only be assigned to nodes capable of executing them.

Gedeon, Brandherm, Egert, *et al.* broadly categorize the solutions to alleviate the above listed problems into three types:

1. *Cloud Services on the Edge* provide cloud computing services, like AWS Lambda, on devices located closer to the data producing entities.
2. *General Computing Hardware on the Edge* locates traditional, "server-like" hardware closer to the data emitters so that established programming techniques can be used for service implementation.
3. *Dedicated Computing Hardware on the Edge* relies on specialized edge devices, running software designed with the constraints of the environment in mind.

Evaluating the above listed architectural implementations, "Dedicated Computing Hardware on the Edge" is the one most relevant for this thesis, as it allows for flexible adaption to different resource-constrained environments. While it can maximize the efficiency of the employed resources, it requires developers to be aware of the device's resource constraints.

2.1.5 Summary

Looking at the existing work in the fields concerned with data stream processing on the edge, there are a few key points to take away:

- The Internet of Things has a reached a broad prevalence, which lead to an increase in communication volume and new communication patterns.
- The large amounts of data emitted by IoT devices necessitate new solutions, as traditional programming techniques and system designs assume different interaction procedures and data intensity.
- Stream processing is one possible paradigm for implementing services handling IoT data, as the reduction of data at the point of entry into the system lessens the load on involved components.
- While there has been a lot of research regarding stream processing in data centers, the problem can not be solved there alone, since it still requires that all sensor readings are transmitted over the internet.
- Edge and fog computing can be applied to reduce the amount of data being sent over the network.
- There are multiple approaches to edge computing, their differences being in the required resources, environment, and development constraints.

2.2 State-of-the-Art

To tackle the challenges introduced by the growing number of automated, internet-connected devices, much research has been done, and multiple solutions have been proposed. Highlighting the most important results and setting them in context with the issue of edge processing in IoT systems is the goal of this chapter.

2.2.1 Declarative and Imperative Programming

In „A Comparative Study of Programming Languages in Rosetta Code“, Nanz and Furia evaluate two subtypes of the imperative programming paradigm (procedural and object-oriented) and one subtype of the declarative programming paradigm (functional) based on the performance of submissions to a coding chrestomathy[5] site. Their results show that the ones following the declarative paradigm were more concise, while the snippets implemented in a certain imperative programming language (C) were the fastest. Regarding the performance, they note that the measured differences only became significant on large input sets. [36].

[5]“Chrestomathy” is a collection of short texts to assist with language learning

In another study, Alic, Omanovic, and Giedrimas investigated performance differences when implementing three algorithms in different programming languages. They note that the fastest solutions were the ones written in an imperative language (Java), with the solutions written in a declarative language (Haskell) having a similar runtime for two of the three algorithms, and consistently less memory usage. Two other implementations, one imperative and one declarative, but both compiled to the same virtual machine code, were up to four times slower than the fastest solution. Additionally, the imperative solutions had significantly more lines of code. This implicates that both the imperative and the declarative paradigm are valid options when considering runtime and memory usage [37], and that the effects of the choice of the compilation target can be more significant.

In opposition to this, there are some studies which indicate that, using the same language or intermediate virtual machine, declarative implementations are usually slower than imperative ones. Both Fraller and Krantz conducted performance measurements of declarative and imperative solution implementations targeting the Java virtual machine (JVM) and report that the declarative solutions are typically slower [38, 39].

While not performance related, Mehlhorn and Hanenberg reports positive results regarding programming speed and code readability when employing declarative programming [40].

Summing up, the existing literature suggests that following the imperative programming paradigm can lead to better runtimes, but the results are neither conclusive, nor is runtime the only relevant metric. Additionally, the results are highly dependent on the used programming languages, with the difference between two implementations in languages of the same paradigms often being larger than the differences between one imperative and one declarative implementation. Furthermore, in the context of implementing programs for usage in resource constrained environments, there does not exist any meaningful literature yet.

2.2.2 Stream processing engines

Looking at the research area of data stream processing, most of the available literature examines the evaluation of stream processing engines (SPEs), and over the course of this the performance differences between declarative and imperative SPEs.

Declarative SPEs

In declarative SPEs, developers only specify the data transformations which should be done, and which transformations are connected to each other. How exactly those connections are constructed is decided by the compiler and runtime, which are thus free to decide where exactly each operation should be run, and whether it is more efficient to merge two transformations into one. Gedik *et al.* present “SPADE”, a declarative frontend for a stream processing engine. SPADE enables developers to focus on describing the processing graph, which is then transpiled into a series of imperative statements constructing this processing graph in the stream processing engine. The authors note that

the implementation can generate code to distribute calculations efficiently over hundreds of nodes [41].

While SPADE is linked to a proprietary system, Bonino and Corno created “spChains”, a declarative framework for data stream processing in Java. Programmers specify the operators to employ and the connections between them in XML, and spChains transforms this specification into Java bytecode, which can then be incorporated into larger programs [41]. The authors report that their system has successfully been used in data-intensive, real-world systems.

Imperative SPEs

Imperative SPEs require developers to detail how data should be moved between two operating steps. While this precludes certain optimizations relying on the stateless nature of declarative operators, it allows programmers to tailor the system more closely to the task at hand. As documented in „A Comprehensive Survey on Parallelization and Elasticity in Stream Processing“ by Röger and Mayer [43], this can lead to imperative solutions outperforming declarative ones.

Apache Storm is a well-established imperative SPE. Developers define so called “spouts” which feed data into the system, and “bolts” which contain both the business logic and the connection to other “spouts” and “bolts” [29]. It has for example been used at Twitter for word occurrence counting and tweet clustering, as reported by Toshniwal, Taneja, Shukla, *et al.* in „Storm@twitter“ [44].

Summing up, adaption indicates that despite the above listed advantages of declarative SPEs, imperative engines are more prevalent in production systems. A reason for this might be that data stream processing may mostly be employed by large companies with dedicated teams, which can afford the manual optimization offered by imperative engines over the automatic improvements possible in declarative ones. For a more in-depth overview of techniques and frameworks employed in data stream processing, see also „A Comprehensive Survey on Parallelization and Elasticity in Stream Processing“ by Röger and Mayer [43].

One more important point to note is that none of the presented SPEs was designed with resource constrains in mind. They are intended to be deployed in data centers featuring multiple processors, large amounts of memory and an unlimited supply of power. This makes them unsuitable for the problem covered in this thesis.

2.2.3 Stream Processing at the Edge

By combining data stream processing and edge computing, software architects can attempt to build systems which are able to efficiently preprocess large amounts of (IoT) data before it is transmitted over the internet, thus saving on bandwidth and cloud server demands.

In their literature review „Distributed Data Stream Processing and Edge Computing“ de Assuncao, Veith, and Buyya acknowledge that the existing work on edge stream processing is still in its infancy [45]. Most of the research analyzed by them is either concerned with virtualization and containerization, or the correct placement and work distribution in computing networks.

While containerization is necessary to automate and orchestrate interacting systems, it only facilitates running stream processing on disparate edge devices, and does not absolve developers from writing code analyzing the data, and handling the communication between different components.

What is missing, and will subsequently be the topic of this thesis, is the evaluation of different programming paradigms in the context of such resource constrained environments. The growing number of connected devices necessitate efficient implementations which can process large amounts of unbounded data streams, while the ever-increasing complexity of the business requirements can only be tackled by developers equipped with appropriate tooling.

2.3 Summary

Looking at the available research, a few key points can be taken away:

- Following the imperative programming paradigm usually yields programs with shorter runtime, although the results are very dependent on the selected programming language.
- Looking at other metrics like memory usage or readability, neither programming paradigm guarantees an advantage.
- Stream processing engines are an often-covered topic in academic literature, but they are not suited for employment in resource-constrained environments.
- Analyzing streaming data on the edge, while an important technique for reducing network load, has not yet been a frequent subject of academic evaluation.
- There does not seem to exist any literature on the employment of declarative versus imperative programming in the context of resource constrained devices.

CHAPTER 3

Benchmark Suite

To evaluate the performance of services processing IoT data streams on resource-constrained devices, a benchmark system is needed. So that it can be also be used in other research projects, and is not limited to this thesis, clear design guidelines are established, with no dependencies on the systems under evaluation. This chapter starts with research on existing work on distributed systems benchmarking, from which a benchmark is constructed and subsequently implemented.

3.1 Overview on Related Benchmarks

One of the first papers presenting a benchmark for stream processing is „Linear Road: A Stream Data Management Benchmark“ by Arasu, Cherniack, Galvez, *et al.* [46]. In this work, the authors use a conceptual model of a toll system with variable tolls, depending on certain sensor inputs, to evaluate the performance of a stream processing system connected to a relational database. The design of the benchmark consists of three key definitions: The *data inputs*, the *required outputs*, and the *execution setup*. While it is an interesting start, their focus on a single-processor system makes the benchmark not viable for our use case. As it does not assume multiple participating nodes, it lacks a system layout, which is necessary for testing the assumptions of this thesis.

Almost ten years later, „Benchmarking Distributed Stream Processing Platforms for IoT Applications“ by Shukla and Simmhan [47] introduced a benchmarking suite for evaluating SPEs, like Apache Storm and Spark, in an IoT context. Their benchmark suite consists of 27 micro-benchmarks and four larger benchmarks assembled from the smaller ones. While closer to the problem of this thesis, the benchmarks are focused on large-scale processing engines and not on solutions based on reducing data at the edge.

The Transaction Processing Performance Council (TPC) created a benchmarking suite for IoT gateways, described in „Analysis of TPCx-IoT: The First Industry Standard

Benchmark for IoT Gateway Systems“ by Poess, Nambiar, Kulkarni, *et al.* [48]. Those gateways are located at the edge of a data processing system and serve as a middle layer between sensors and cloud servers. They are often tasked with reducing the data sent over the internet, or doing a preliminary analysis to reduce the latency of time-critical results. The suite emulates sensor output and measures the software running on the gateways. It is designed to evaluate the performance and cost-effectiveness of stream processing engines. The benchmarking suite was later extended to account for the efficient handling of time-series data and the ability of the system under test to scale out on demand [49]. While the device layout per se is very close to the one desired for the thesis, the benchmark itself is not designed for evaluating the performance of an interconnected system. Its definition consists of a *use case*, the *description of the test execution*, the *inputs*, the *outputs*, the *benchmark driver*, and the *metrics to take*. TPCx-IoT relies on the Yahoo! Cloud Serving Benchmark framework (YCSB) for generating workload data. This framework is designed for evaluating the performance of databases developed for cloud data serving, so it can not really be employed for benchmarking the services implemented in this thesis [50].

Another benchmark for the components on the edge of an IoT data processing system is IoTMark[1], designed by the Embedded Microprocessor Benchmark Consortium (EEMBC). Complementary to the TPCx-IoT suite, it evaluates the energy efficiency of the nodes closest to the end-users, which are made up of a sensor, a processor, and a data transmission interface. The tests are executed by connecting the object under test to three monitoring devices (one I/O manager, one energy monitor, one radio manager), which are in turn controlled by a host system. Similar to the earlier benchmark, the suite simulates sensor readings, which are sent to the object under test, and its outputs are verified by the host system emulating a gateway. Contrary to the interests of this work, the IoTMark suite is only concerned with the energy efficiency of the hard- and software of singular edge devices. The IoTMark suite consists of the *inputs*, a *testing setup*, *test execution software*, and software to *parameterize* the benchmark. Unfortunately, it is not possible to look into the data used for the benchmark runs, since it is not available publicly.

„Explainable Artificial Intelligence for Predictive Maintenance Applications“ by Matzka introduces a synthetic dataset for emulating IoT sensor output. While the paper itself is not concerned with the domain of data stream processing (or edge processing), the virtual sensor readings created as its output are not confined to a certain use case, as they simply reflect data readings encountered in industrial applications [51].

In „Towards a Methodology for Benchmarking Edge Processing Frameworks“ by Silva, Costan, and Antoniu, the following steps to construct a benchmark are proposed: 1) *Benchmark Objectives*, 2) *Edge Processing Frameworks*, 3) *Infrastructure*, 4) *Scenario Applications and Input Data*, 5) *Experiment Parameters*, 6) *Evaluation Metrics*, and 7) *Benchmark Workflow* [52] Those steps can be taken as a basis for the framework of this thesis.

[1] `https://www.eembc.org/iotmark/` (visited on 2024-05-08)

Combining this procedure with the data made available by Matzka, a new suite can be created to evaluate data stream processing approaches in a resource-constrained environment.

3.2 The Definition of the Benchmark Suite

The constructed framework will be defined by the following points:

1. The *goal of the benchmark*
2. The *items to evaluate*
3. The *infrastructural layout*
4. The *scenario*
5. The *input data*
6. The *parameters*
7. The *metrics*
8. The *workflow of the benchmark*

In their entirety, those components form the benchmark suite which will be used to examine the performance of the imperative and the declarative programming paradigm in resource constrained environments. As it is designed separately from the service implementations, it can be reused for different comparisons in its entirety, or, if necessary, only the appropriate elements.

3.2.1 Goal of the benchmark

The objectives of the benchmark are evaluating runtime, memory consumption, and message latency of data stream processing services in resource constrained environments.

3.2.2 Items to evaluate

The items to evaluate are two implementations of an IoT data processing service. One is implemented following the imperative programming paradigm, the other following the declarative programming paradigm, both of the services will be written in Rust. The system should be designed in such a way that there are no dependencies on the items to evaluate, so that the benchmark scenario and suite can be reused for further research. Achieving this means that they can be switched out without requiring modification to the benchmark suite.

3.2.3 Infrastructural layout

For implementing the benchmarking suite, two different types of hardware are available.

1. A multi-cored single-board computer, functioning as the resource-constrained edge gateway. It contains the service to evaluate, which processes the unbounded data streams arriving from the sensors.

2. A conventional computer which is used to emulate those parts of the system which do not need to be present in a physical form, as they are not being evaluated.

3.2.4 Scenario

To answer the research questions of this thesis satisfactorily, the model scenario at the base of the benchmark suite needs to define the following features:

- Sensors, modelling IoT data sources.
- Continuous, "infinite" data streams, being created by those sensors.
- A node processing the data from multiple sensors, running on a resource constrained device.
- A task for the data processor, so that the performance of different implementations can be measured and compared.

An exemplary benchmark scenario fulfilling the requirements can be seen in Figure 3.1. The "Object under observation" is the element which needs to be monitored for errors. "Sensor one" and "Sensor two" take certain readings from this element and forward it to the "Data processor". This node is responsible for merging data from multiple sensors to detect correlations indicating a problem in the "Object under observation". If an issue is noticed, it is forwarded to the cloud server, where the error notification is dealt with. The "System under evaluation" constitutes the object which is evaluated by the benchmark, which is in the case at hand the data processing service.

Implemented Scenario

To cover the earlier listed requirements, a scenario is implemented which is built upon *motors*, *motor sensors*, a *motor monitor*, and a *cloud server*. In short, the motor sensors measure certain properties of the motor and forward the data to the motor monitor. The monitor processes the data and, in the case of it indicating motor problems, sends an alert to the cloud server. See Figure 3.2 for a graphical explanation of the scenario. The individual components of the scenario are defined as follows:

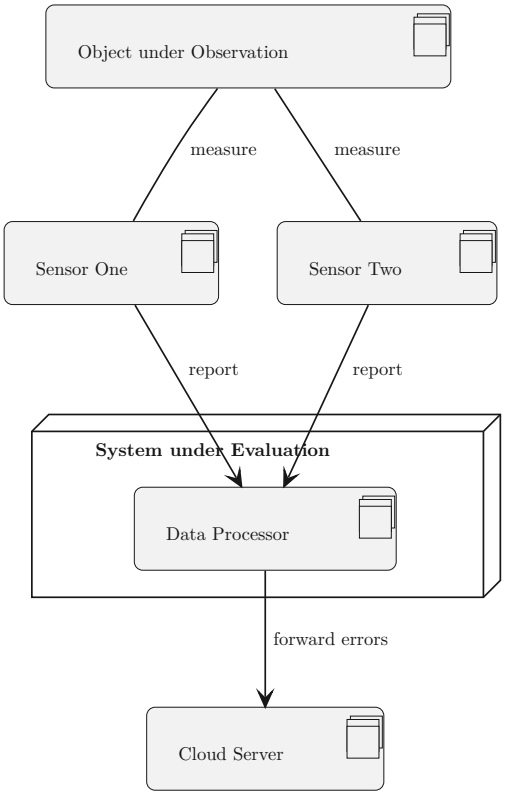


Figure 3.1: Architectural sketch of a possible benchmark scenario.

Motor The object under observation of the data stream processing system will be a motor participating in an unspecified task. Due to wear, such a motor fails at random points in time. Depending on the failure, this is expressed by combinations of abnormal temperature and mechanical readings. The status of the motor is emulated by the dataset presented in Section 3.2.5, so it only exists as a concept for visualization in this benchmark.

Motor Sensor Attached to the motor are four sensors. Each sensor reports one of the following measurements needed for detecting possible motor failures:

- Air temperature
- Process temperature
- Rotational speed
- Torque

The sensors are emulated, taking their readings from a dataset (see Section 3.2.5), and run on the consumer PC in docker containers. Four sensors together with the motor they are monitoring are considered one "motor group".

Motor Monitor Each motor group will be connected to a monitoring device (for details on the number of motor groups see Section 3.2.6 and Section 3.2.8). This device receives the data of the groups sensors, and analyzes it to detect motor failures. If the readings indicate a problem in the subject under observation, an alarm is sent to the cloud server. The motor monitor runs on the single-board computer. Since it has multiple cores, it should be able to process the incoming data streams more efficiently than if it only had one, and is thus a fitting test subject for examining the multithreaded performance of the implemented services.

Cloud Server The cloud server is responsible for dealing with the failure notifications emitted by the motor monitor. In the case of our benchmark, this component simply receives notifications and documents them so that the benchmark suite can process them further. It is also deployed on the consumer PC.

Figure 3.2 lays out the intended testing architecture for one motor group graphically.

3.2.5 Input Data

In „Explainable Artificial Intelligence for Predictive Maintenance Applications", Matzka provides a dataset for machine-learning researchers studying explainability of artificial intelligence. While it is synthetic, it reflects maintenance data encountered in industry [51]. Three properties make the dataset a fitting choice as the base of the benchmarking suite: First, it being a time series of multiple sensor readings possibly indicating an error in the observed object align closely to the use case of this thesis. Second, the data points are plain numbers, which corresponds well to the assumption about the sensor readings made earlier. Third, the readings are sampled from the normal distribution, so simple statistical analysis tools can be used to emulate gaining insight into the status of the motor.

The provisioned dataset is a comma-separated values (CSV) file of 10,000 rows, each containing the following readings:

1. Type of the tool (low/medium/high quality)
2. Air temperature
3. Process temperature
4. Rotational speed
5. Torque

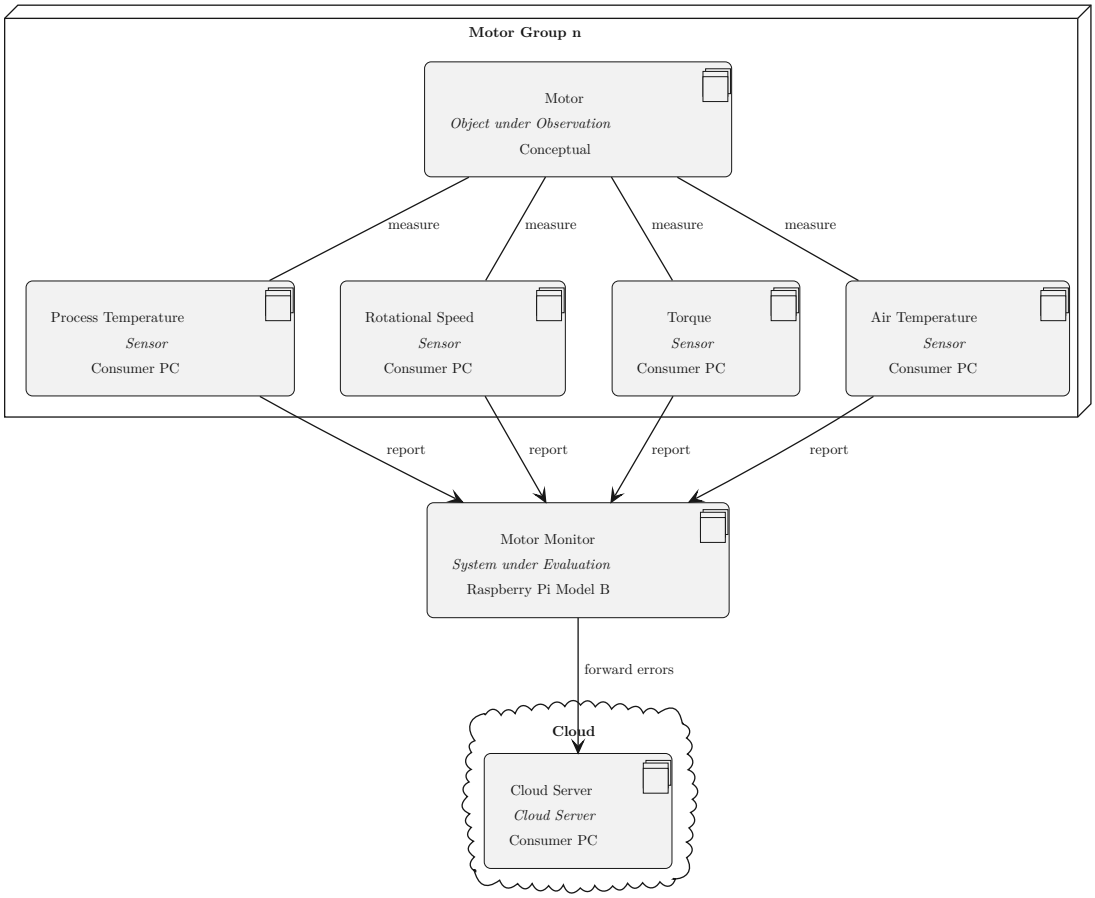


Figure 3.2: Device Assembly of the Motor Monitoring System.

6. Tool wear

Additionally, each entry also contains information of possibly occurring failure states.

1. Tool wear failure
2. Heat dissipation failure
3. Power failure
4. Overstrain failure
5. Random failure

See Table 3.1 for an excerpt of the dataset.

In the use case by Matzka, a machine learning tool is trained on the dataset to preemptively recognize failures in the object under observation. For this, a classifier receives one row

Table 3.1: An excerpt of the dataset by Matzka.

Type	Air Temperature [K]	Process Temperature [K]	Rotational Speed [rmp]	Torque [Nm]	Tool Wear [min]	Machine failure	TWF	HDF	PWF	OSF	RNF
M	298.1	308.6	1551	42.8	0	0	0	0	0	0	0
L	298.2	308.7	1408	46.3	3	0	0	0	0	0	0
L	298.1	308.5	1498	49.4	5	0	0	0	0	0	0
L	298.2	308.6	1433	39.5	7	0	0	0	0	0	0
L	298.2	308.7	1408	40	9	0	0	0	0	0	0
M	298.1	308.6	1425	41.9	11	0	0	0	0	0	0

of the data after the other, and, via a supervised learning process, trains predicting upcoming problems. This sequential flow, where each row is regarded as one unseparable fact, is not directly applicable to our use case. As an alternative, the motor monitor running on the edge device will use the following rules to determine whether the object under observation encountered a failure:

Temperature Failure The difference of the mean of the last ω process and air temperature readings is outside the 90% confidence interval of the population average.

Power Failure The average power (product of rotational speed and torque) over the last ω readings is outside the 90% confidence interval of the population average.

ω is defined as the number of past sensor readings regarded in each analysis.

3.2.6 Parameters

The parameters of the benchmark suite are selected in such a way that additionally to "normal" system load, different possible sources of stress are simulated. Those sources are:

1. The number of elements that should be considered for determining whether there has been a failure in an observed motor (window size ω). This influences how often the arrived data should be evaluated, and how many items should be included in each analysis. The purpose of running the evaluation with different values is to determine the performance of the implementations when working with larger batches of data.

2. The number of motor groups connected to the monitor (Number of motor groups θ). By varying this parameter, the two implementations can be evaluated based on their handling of multiple (θ) unrelated tasks in parallel.

See Table 3.2 for a listing of the different parameter values employed in the benchmark execution.

Table 3.2: The parameters of the benchmarking suite.

Parameter	Values
Processing Model ρ	{Declarative, Imperative}
Number of motor groups θ	{1, 2, 4, 16}
Window Size ω	{1, 2, 4, ..., 4096}
Total testing assemblies	2 * 4 * 13 = 104

3.2.7 Metrics

To be able to assess the advantages and disadvantages of the data processing approaches, multiple aspects of the implementations are observed. The measurements of hardware utilization will be taken on the single-board computer, as this is the object where the data analysis process runs. Additionally, the processing latency will be measured over the full system.

CPU measurements

To assess the processing efficiency of the employed implementation, the total processing time of the executed processes and the load average of the executing system should be recorded. The total processing time t_{total} is calculated as the sum of the execution time of the main process and its children

$$t_{total} = t_{process} + \sum_{i=0}^{c} t_i$$

where t_p = the execution time of process p, $process$ = the main process of the service, and c = the number of its child processes.

The load average $\bar{l}$ is defined as the average number of processes waiting to either be executed on the CPU or for disk Input/Output (I/O) over the last minute

$$\bar{l} = \frac{\sum_{i=0}^{n} w_i^{io} + w_i^{cpu}}{n}$$

where n = the number of samples per minute, w_n^{cpu} = the number of processes waiting for execution at sampling time n, and w_n^{io} = the number of processes waiting for I/O at sampling time n.

Memory measurements

Another important factor which has to be examined, especially when working with resource constrained devices, is the memory usage of the employed services. Thus, the maximum amount of RAM the process occupied should be measured and reported.

System throughput

To analyze the throughput of the implementations, and whether they can sustain it under high load, the message latency should be measured. It is defined as the difference between the time a message triggering an alert is read by a sensor and the time the corresponding alert arrives at the cloud server

$$l = t_{server} - t_{read}$$

where l = the alert's latency, t_{server} = its arrival time at the server, and t_{read} = the time the message triggering the alert is read by the sensor.

3.2.8 Workflow of the benchmark

Once the benchmark suite has been designed and implemented and the services are ready to test, the system can be put into use.

As the benchmark is executed in a consumer setting, there are many factors which might influence the measurements of the benchmarking runs, but can realistically not be avoided. Examples for this are congestion of the network by other connected devices, or housekeeping tasks of the operating system. To mitigate the effect of those influences, each set of benchmark parameters will be executed 50 times, and the results will be analyzed statistically. The employed methods are explained in Chapter 5.

Verification of the Suite

To be sure that the data emitted by the sensors is processed correctly, the failures reported by the monitor need to be verified. This is done by comparing the alerts sent by the motor monitor to a list of expected errors which has been generated independently of the testing system. As all the sensors are retrieving their readings from the same list their sequence of emissions is predetermined, and can be replicated externally. To avoid static patterns in the emitted data, each sensor uses its ID as the seed for a random number generator determining at what positions the reads from the value list should happen. Thus, the data is emitted in a pseudorandom pattern, and can still be reconstructed externally.

See Figure 3.3 for a graphical depiction of the verification process.

An important point to note is that the process can only be employed for verifying the services in a configuration where the time between sensor messages and the window size are larger than randomly introduced latencies by a large margin (those latencies can be, depending on connection and CPU speed, up to multiple milliseconds). If the duration between single readings and the window length are similar in size to external interferences, such as fluctuations in message delivery times, the alerts are not deterministic in reference to the emission time. This is because those disturbances significantly influence whether a reading is part of a window or not, and thus have an effect on the triggering of alerts. Resulting from this, it is not possible to apply the verification procedure when executing

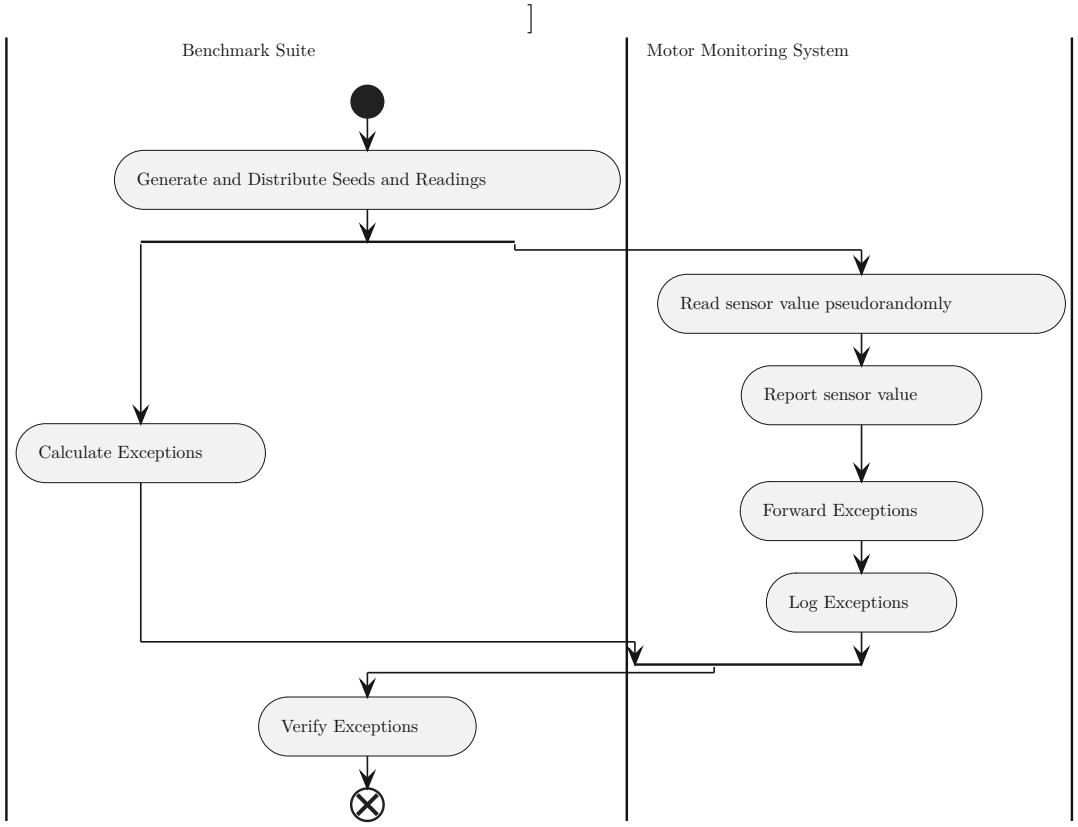


Figure 3.3: The verification workflow.

the benchmark with the sensor sampling rates and window sizes defined earlier. To nevertheless be able to assert the correctness of the services, they were tested during development with a sensor emission rate of one per second, and a window size of three seconds. As those values are much larger than possible external disturbances, their correctness could be asserted as described above. Upon execution of the benchmark, the verification was disabled.

Outcomes

The outcome of a benchmarking run is the assembly of the reported metrics:

- Runtime on the motor monitor (sum of the time the process and its children were executed on the CPU, in 1/100th of seconds)
- System load on the motor monitor (average number of processes waiting for execution or I/O during the test run)
- Memory usage on the motor monitor (maximum amount of physical amount of memory occupied, in KiB)

- A list of the recorded message latencies (in seconds)

Summing up, each benchmarking run goes through the following steps:

1. Instruct the participating elements about the details of the benchmarking run
   - The sensor about when they should start emitting data, with which frequency, and for how long
   - The motor monitor about when the run starts, when it ends, and how long the window of relevant data should be,
   - The cloud server about when the run starts, and when it ends
2. The sensors emit readings, the monitor processes them and emits alerts when appropriate
3. When the run is over, the benchmark readings on the motor monitor are recorded and sent to the test driver, and the cloud server transmits the alert delays.
4. The test driver stores the benchmark readings and the alert delays for further analysis.

3.3 Implementation

To realize the benchmark designed above, a suite of programs was implemented. It consists of six services, of which some are responsible for benchmark execution and others for emulating the components interacting with the evaluated services.

3.3.1 Motor Sensor

The motor sensors emulate the sensors taking readings from the motor and forwarding them to the data processing service. They are deployed as a docker service with the required number of replicas on the consumer PC, together with a file containing sensor readings to be read from during operation. Upon execution of a benchmarking run, each sensor receives a message from the test driver containing

- ID of the sensor (also containing the motor ID)
- Starting time of the benchmarking run
- Duration of the benchmarking run
- Sampling interval of the sensor in milliseconds
- IP address of the monitor

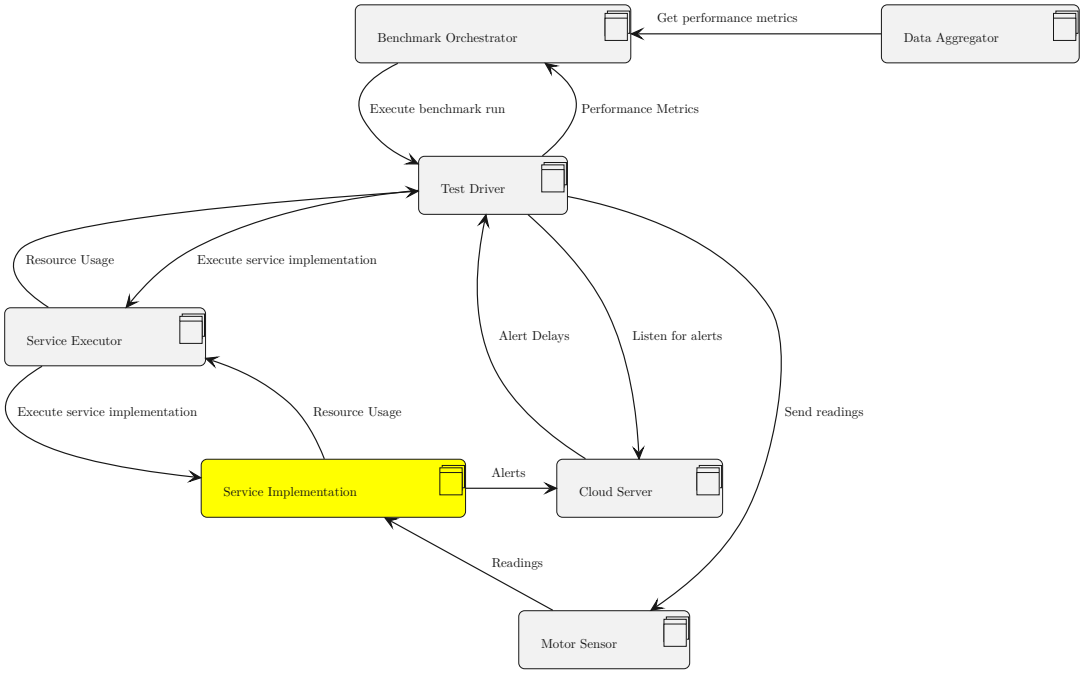


Figure 3.4: The components of the benchmark system, with the data processing service marked in yellow.

Once the information has been parsed, the sensor initializes a random number generator with its ID as a seed, opens a TCP connection to the motor monitor, and waits until the specified starting time has been reached. From this point on, until the specified duration has elapsed, the sensor uses the random number generator to pick a line from the supplied readings file in the declared intervals, and sends it to the motor monitoring service.

Each message sent from the sensor to the monitor contains the following data:

The ID of the sensor is supplied to the sensors at the start of each test run and has the size of one byte. The first six bits constitute the motor ID and the last two bits the types of readings which will be taken, where zero signifies air temperature, one process temperature, two rotational speed, and three torque. As an example, the sensor ID 13 is represented in binary as 000011|01 (leading zeros and "|" separator added for clarity) which means that the sensor is the process temperature sensor of motor number three.

The sensor readings are transmitted as floating point numbers with a length of 32 bits.

The timestamp of the reading is transmitted as a 64-bit floating point decimal specifying the number of seconds since the Unix epoch.

The messages are additionally COBS encoded and serialized to the "Postcard Wire Format"[2].

3.3.2 Service Executor

The service executor runs on the Raspberry Pi and is responsible for starting the desired version of the motor monitor on each test run, passing it the correct parameters, and returning the benchmark readings of the monitoring service to the test driver. It is also deployed as a docker service (with a replication count of one), and on each test run receives the following information from the driver:

- When the test run starts
- The duration of the test run
- Which implementation of the service should be run
- The number of motor groups
- How large each data evaluation window should be (in milliseconds)
- On which port the service should listen for incoming sensor connections

The service executor starts the desired motor monitoring process, handing it the parameters received from the driver as program arguments. It then waits until the process completes, reads the resource consumption metrics from the processes standard output, and sends it back to the test driver. The readings which are reported are:

- Time the CPU was running in user mode while executing the process
- Time the CPU was running in kernel mode while executing the process
- Time the CPU was running in user mode while executing all child processes
- Time the CPU was running in kernel mode while executing all child processes
- Maximum amount of RAM occupied by the process (resident set size)
- Average system load

For more information on the collected readings, see Section 3.2.7.

[2] `https://postcard.jamesmunns.com/wire-format` (visited on 2024-05-08)

3.3.3 Cloud Server

The cloud server component fulfills the theoretical role of a remote server that receives only relevant data from the preprocessor on the edge. It is also implemented as a docker service with a replication count of one, and runs on the consumer PC. For each test run, the cloud server receives the following information:

- When the run starts
- Duration of the run
- The port to listen on for incoming connections from the IoT data processing service.

Upon execution, the cloud server first waits for an incoming connection by the motor monitor on the specified port, accepts it, and then processes the arriving alerts. The alerts are expected to be COBS encoded, serialized to the "Postcard Wire Format" and to contain the following information:

Motor ID as an unsigned 16-bit integer.

Time of the alert is defined as the timestamp of the youngest data packet relevant for the alert. It is represented by a 64-bit floating point decimal, measuring the time from the Unix epoch.

Motor failure is an enum (encoded as an unsigned 32-bit integer) signifying the type of failure that occurred, which can either be a temperature or a power failure. For details, see Section 3.2.5.

Upon arrival of one such alert, the cloud server immediately adds a second timestamp noting when it was received, and saves all four data points in a CSV-formatted file.

Once the test run has finished, the cloud server sends the CSV file to the test driver for further analysis.

3.3.4 Test Driver

The test driver is the program responsible for executing single test runs. It runs natively on the consumer PC, and is invoked via the command line. Configuration of a test run is done via command line arguments, defining the exact parameters and procedure of the test run, and an accompanying file, listing the participating services and where they can be reached.

The options specifying the parameters of the test run are:

- Number of motor groups

- Duration of the test run
- Which implementation of the motor monitoring service should be run
- Sensor sampling interval (in milliseconds)
- How large each data evaluation window should be (in milliseconds)

The network layout passed to the test driver consists of:

- The address of the TCP port where the cloud server is expecting connections from the test driver
- The address of the TCP port where the cloud server is expecting connections from the motor monitoring service
- The address of the TCP port where the service executor is expecting connections from the test driver
- The IP addresses of all sensors
- The address of the TCP port where the motor monitoring service is expecting connections from the sensors

Upon invocation of the program, the test driver reads the configuration parameters, processes them, and sends them to the cloud server, the sensors, and the service executor as appropriate. It then sleeps until the test run duration has elapsed. Once awake again, the test driver reads the benchmarking results from the service executor and the alerts from the cloud server, persists them to a CSV file, and finally exits.

3.3.5 Benchmark Orchestrator

The benchmark orchestrator is responsible for supervising a full execution of the benchmark suite. It runs natively on the consumer PC. Upon invocation, it reads a file stating all desired parameter variations to test, and how many repetitions should be done per parameter set. The parameters which can be specified are

- Number of motor groups
- Duration of a test run
- Which implementations of the motor monitoring service should be benchmarked.
- How large each data evaluation window should be (in milliseconds)
- The sensors sampling interval (in milliseconds)

It then executes test runs with every possible combination of parameter sets with as many repetitions as specified. Whenever the number of motor groups changes, the executor updates the docker service of the motor sensors to the specified number of replications, and waits until the change has been deployed. It then writes the IP addresses of each participating service to a text file to be read by the test driver, and executes it, passing the run parameters as program arguments. Once the specified test run duration has elapsed, the orchestrator will wait for up to 30 seconds for the driver to finish. In the event of a successful process execution, the benchmark measurements are persisted and the next run starts. If the exit code of the subprocess was not zero, or it did not complete in the specified time, the orchestrator restarts the cloud server, the service executor, and the sensors, before beginning the next test run.

3.3.6 Data Aggregator

The data aggregator is not directly linked to test execution, but is used to extract and display the results of the benchmark in an accessible manner. Upon invocation, it reads the CSV files created by the benchmark executor, containing the measured performance indicators, and calculates the following values of each metric over all repetitions of one parameter set:

- minimum
- maximum
- average
- standard deviation

For both the imperative and the declarative implementation of the motor monitoring service, one file per measurement is created containing the calculated values. Additionally, further data analysis is done, and figures graphically depicting the measurements are created, as can be read in detail in Chapter 5.

3.3.7 Usage

Execution of the benchmark suite is done by first configuring the system as required, deploying the services, starting the benchmark runner, and finally evaluating the aggregated data. Below is a short overview over the four phases of the procedure, a more detailed description is delivered with the code[3]. As initially specified, this execution is not dependent on the services under evaluation, but can be run for any programs conforming to the communication specification given above.

[3] `https://github.com/AntonOellerer/Reactive-Streaming-on-the-Edge` (visited on 2024-05-08)

Configuration

The first step for executing a service evaluation is configuring the parameters of the suite accordingly. The possible tailoring is mostly done on two levels, one being the layout of the services over the employed hardware, the other being the arguments for the benchmark execution as listed in Section 3.3.5.

Deployment and Execution

Deployment of the benchmark suite happens in two phases, the first being the deployment of the services to the required hardware, and the other being their management during the execution of the benchmark suite.

The initial deployment is specified via a set of container creation and orchestration files provided with the code of the benchmark suite [4]. It is done via the usage of docker, and consists of the following steps:

1. Build the container images for the architecture of the employed devices.
2. Push the images to a registry reachable by the employed devices.
3. Create the docker services on the employed devices as specified by the deployment instructions.
4. Verify that all services have started correctly.

Once the deployment has finished, further orchestration is managed by the benchmark orchestrator, although docker is still used for executing the starting and stopping of the services.

During execution, the benchmark orchestrator has the following responsibilities:

1. Execute the test driver with the parameters specified in the configuration file.
2. When the desired number of motor groups changes, instruct docker to scale the "motor sensor" service to the required number of containers.
3. When a benchmarking run fails, reboot the system by shutting down the cloud server, the service executor, and all sensors, and restoring the system to its desired status once this has completed.

[4] `https://github.com/AntonOellerer/Reactive-Streaming-on-the-Edge` (visited on 2024-05-08)

Result evaluation

The final step of employing the benchmark suite designed in this thesis is the evaluation of the result data amassed during its execution. As described in Section 3.3.6, the data aggregator collects the data, analyzes it statistically, and creates diagrams for visual exploration of the results.

CHAPTER 4

Imperative and Declarative Data Processing Service

This chapter details the implementation of the imperative and declarative IoT data processing services, which analyze data arriving from sensors to detect failures in observed motors. They are then benchmarked to evaluate the suitability of the two paradigms when implementing IoT data processors in resource constrained environments.

During execution, the processing services proceed through multiple phases, as explained graphically in Figure 4.1. Of those steps, the argument reading and benchmark results handling always follow the same procedure (marked in yellow in Figure 4.1), while the setup and data processing phases are specific to the implementations.

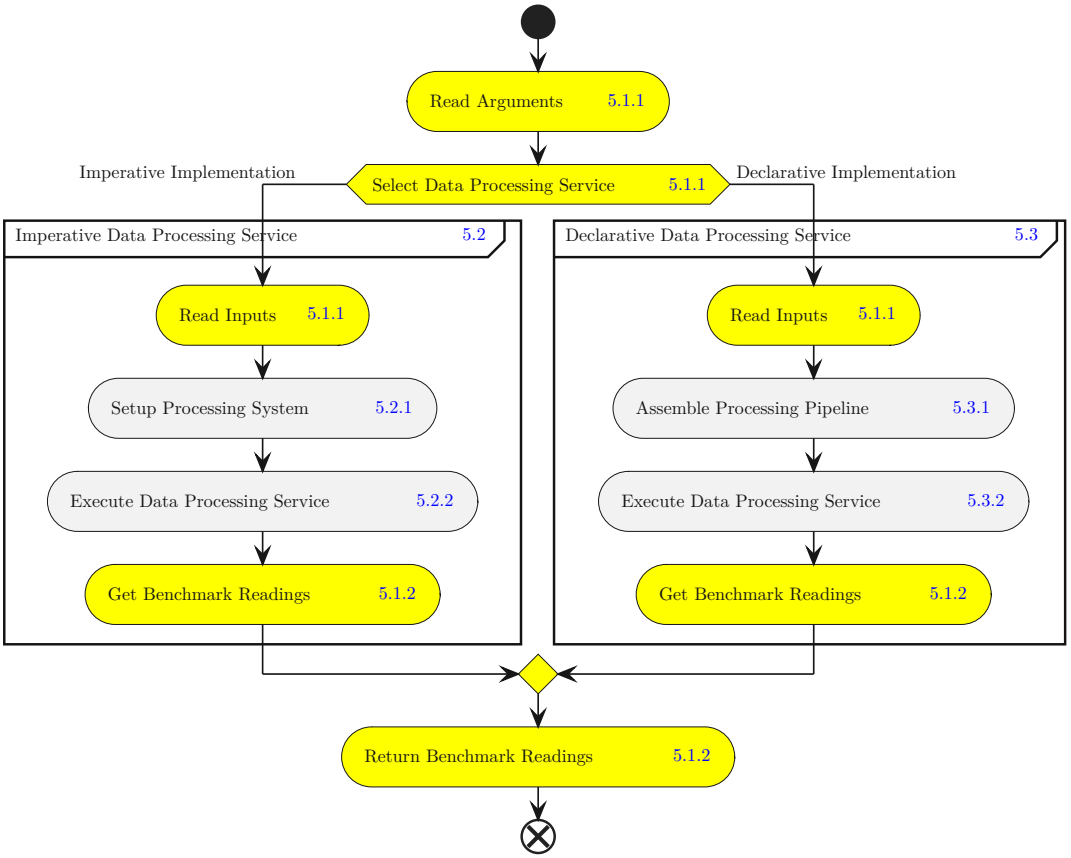


Figure 4.1: Procedure of one test run, with the implementation independent steps marked in yellow.

4.1 Independent Steps Implementation

The implementation independent steps consist of reading the program arguments and collecting and returning the benchmark measurements from the `/proc` pseudo-filesystem via the standard output.

4.1.1 Argument Reading

When the execution of the data processing service starts, as a first step the program collects the arguments it has been started with (commonly known as the "argv") and extracts from it the information it needs for running the monitoring service. This consists of

1. The start time of the execution
2. The duration of the execution

3. The number of motor groups
4. The timespan each data evaluation window should cover (in milliseconds)
5. The address to listen on for incoming sensor connections
6. The address the cloud server is listening on for incoming connections

If one of these parameters is not present, the system exits with a nonzero exit code. Once all information has been parsed and put into a data object, it is handed to the next step to start the implementation-dependent setup.

4.1.2 Benchmark Results Handling

After finishing the motor monitoring task, the desired measurements are taken and returned to the service executor via the standard output.

The measurements are retrieved from the `/proc` pseudo-filsystem, an interface to kernel data structures, which can be used to read process-bookkeeping data. For each process there exists a directory with the name of the processes ID (`/proc/[pid]`), and for each of its threads a directory with the name of the tasks ID (`/proc/[pid]/task/[tid]`). Those directories contain further files, of which for this thesis the `stat` and `status` files are important, as they contain information about the resources used during process execution. As the files get deleted once the process exits, the data has to be fetched by the motor monitoring service at the end of each test run.

The values listed in Section 3.2.7 are retrieved as documented in Table 4.1.

Table 4.1: The metrics and how they were recorded.

Metrics	Calculated as
Time CPU was running in user mode while executing the process	`/proc/[pid]/stat.utime`
Time CPU was running in kernel mode while executing the process	`/proc/[pid]/stat.stime`
Time CPU was running in user mode while executing all child processes	Sum of all `/proc/[pid]/tasks/[tid]/stat.utime`
Time CPU was running in kernel mode while executing all child processes	Sum of all `/proc/[pid]/tasks/[tid]/stat.stime`
Maximum amount of RAM occupied by process	`/proc/[pid]/status.vmhwm`
System load	`/proc/loadavg`

While there exist two fields `/proc/[pid]/stat.cutime` and `/proc/[pid]/stat.cstime`, which indicate a processes children `utime` and `stime`, this value only measures the number of ticks the parent process has waited for their completion. This means that in a parallel processing environment, where the main thread may also be busy while the child threads are executing, this number does not provide the desired measurements.

Once all necessary data has been retrieved, it is COBS encoded, serialized into the "Postcard Wire Format"[1], and written to the standard output so that the service executor can access it.

[1] `https://postcard.jamesmunns.com/wire-format` (visited on 2024-05-08)

4.2 Imperative Data Processing Service

The imperative version of the data processing service analyzing sensor data to detect motor failures is implemented following the imperative programming paradigm. This means that the data-infrastructural tasks of the service are implemented on the same level of abstraction as the business logic. For a detailed explanation, see Section 2.1.1.

The processing structure is constructed at the start of the service, where each node is assigned to one thread, communicating over Rusts `std::sync::mpsc::channels` as required. As data processing and forwarding only happen upon arrival of new messages, the system can also be considered an "event-driven" system, where each incoming sensor reading starts a procedure which in turn might start other procedures, until finally an alert gets emitted, if necessary.

4.2.1 Initialize Data Processing Service

On startup, the data processing service constructs one "sensor handler" per sensor connected to it, and one "data combiner" per motor group. Four "sensor handlers" connected to the sensors of the same motor group are passed the sending end of a `std::sync::mpsc::channel`, where the receiving end is held by a "data handler" which analyzes the received data.

See Figure 4.2 for a graphical depiction of the construction and the processing of the service during one benchmarking run.

Sensor Handler

Each sensor handler gets assigned a TCP stream connected to a sensor and one sending end of a "multi-producer, single-consumer" queue. Additionally, it receives information on how long each examined window should be, and how often it should send data updates. It uses this data to construct a data structure for keeping the incoming messages, containing a vector for storage, the specified duration a message should be relevant, and the timestamp of the last time the window average was sent to the data processor. The sensor handler then opens the specified socket and waits for an incoming connection, which signals the start of the test run.

Data Combiner

The data combiner receives a TCP connection to the cloud server, and the receiving end of a "multi-producer, single-consumer" queue.

4.2.2 Process Data Streams

After all data combiners and sensor handlers have been constructed, their event loops are submitted to a thread pool, with the service keeping handles to all the processes. Using the handles, the service then waits until all the submitted processes have completed.

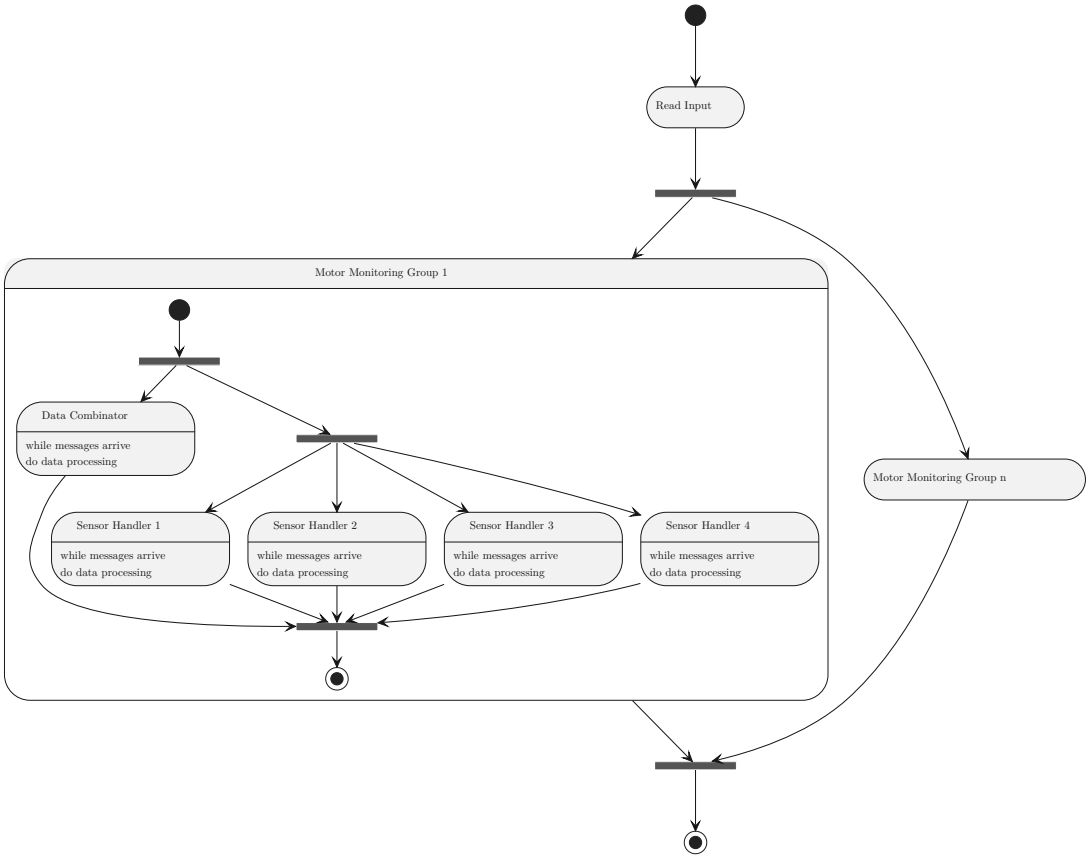


Figure 4.2: The high-level states the imperative data processing service is going through in one test run.

Sensor Handler

When a message arrives at the sensor, it is put into the storage vector. If the time since the last reported average has exceeded the desired update interval, the list of relevant values is filtered for expired readings and its average is calculated, which is then forwarded to the data processor. After the "last sent" timestamp is updated, the sensor handler again starts waiting for the next message from its assigned sensor. Once the sensor closes the stream, the handler knows that the run has ended and in turn closes its writing end of the queue to the data combiner and exits its event loop. See Figure 4.3 for a graphical overview of the processing flow of the sensor handlers.

Data Combiner

The data combiner listens on the receiving end of the channel it holds for new messages. Once a message arrives, its sensor type is checked and then stored for further evaluation. If averages from each sensor type are available, the processor calculates whether the temperature difference or the power output are outside the expected confidence interval.

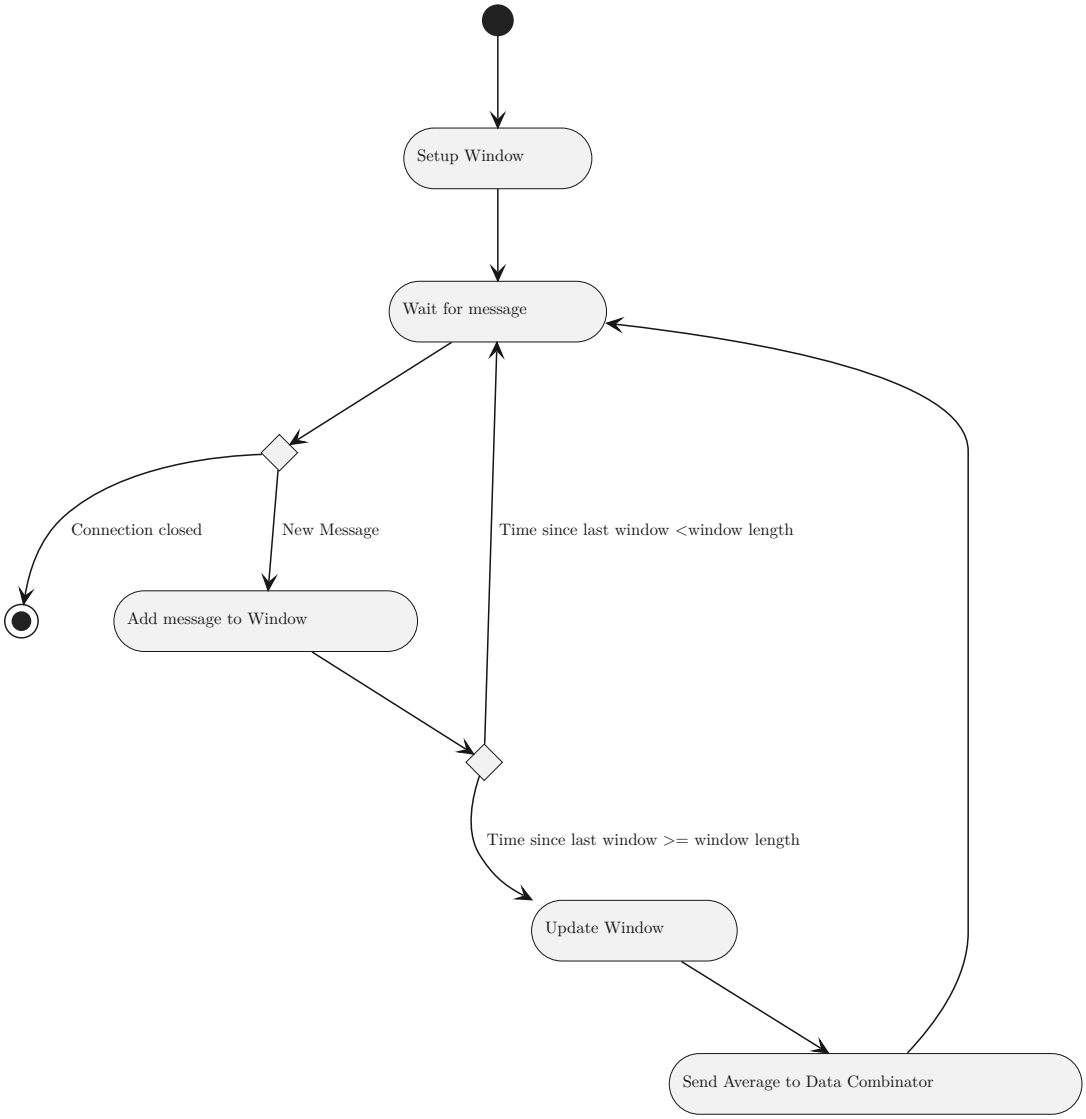


Figure 4.3: The processing flow of the sensor handlers.

If this is the case, the data processor sends a message to the cloud server containing the motor the problem is occurring in, the type of the alert, and a timestamp of the last message influencing the alert. Afterwards, independent of whether an alert has been sent or not, the data combiner again starts waiting for a message to arrive at the channel to the sensors. If it receives the signal that all writing ends have closed, the data combiner exits the event loop. See Figure 4.4 for a depiction of the internal processing flow of the data combiners.

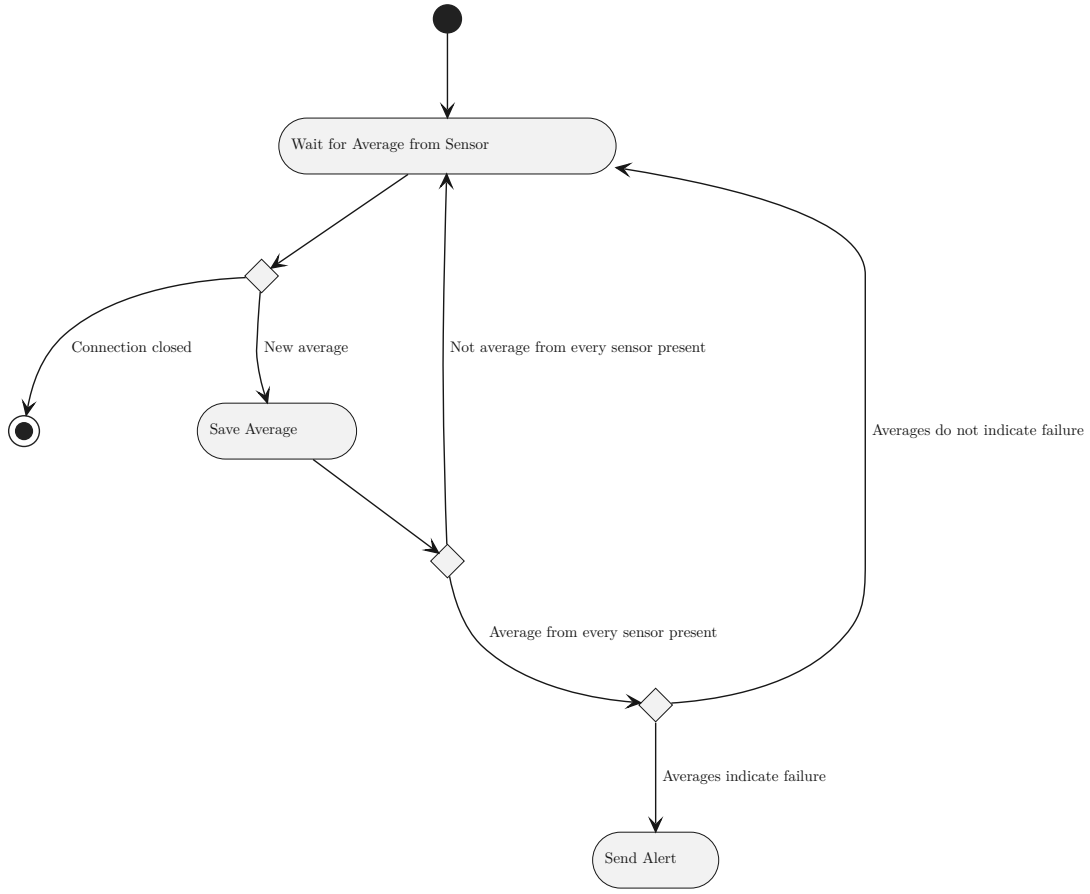


Figure 4.4: The processing flow of the data combiners.

Finally, after all sensor handlers have closed their ends and exited, and all data combiners have exited as well, the main process is notified that the test run has ended. It then collects the benchmark data, writes it to its standard output, and exits as well.

4.3 Declarative Data Processing Service

Following the declarative programming paradigm, the implementation of the declarative data processing service happened on two layers: One handling the higher-level business

logic of how the data should be evaluated, and one handling the data infrastructural part of the service. Due to this different architecture and resulting flow, a division into "initialization phase" and "data processing phase" as in the imperative implementation is not appropriate, but in general the business logic layer is responsible for assembling the processing pipeline before the test run starts, and the data infrastructural layer realizes its correct execution.

4.3.1 Business-Logic Layer

The business logic layer defines, via a series of function applications on the received data, how the incoming sensor readings should be processed. A graphical representation of the processing steps can be seen in Figure 4.5.

The pipeline is defined in multiple layers, with the first level being made up of the following three stages (See also the left lane in Figure 4.5):

Sensor connection initialization

In the first stage, the pipeline opens a socket on the specified TCP port, and forwards each incoming connection to the next processing step, where it is polled for data. After parsing it into sensor messages objects, they are forwarded to the next step, which concludes the first stage.

Data processing

The first step of the data processing stage starts is generating windows of the incoming readings. As necessary, their size and emission frequency can be specified externally. From the aggregated measurements, the average value per sensor is first calculated by grouping the emitted values by sensor ID, then adding up their readings, and finally dividing by the number of relevant messages (right lane in Figure 4.5).

Those averages are again emitted into a single stream, which is then grouped per motor ID. In the grouped stream, the averages of all four sensors are collected into a single object, which is then checked for problem indicators (middle lane in Figure 4.5). If an error is found, an alert is emitted into the last stage.

Alert forwarding

The final stage receives the alerts detected by the previous stage, serializes and encodes them, and finally sends them to the cloud server.

Remarks

As can be noted, the business logic layer also deals with data infrastructural tasks, those being the handling of the incoming TCP connections in the first step, and of the outgoing ones in the last step. This is because those issues are on another layer of

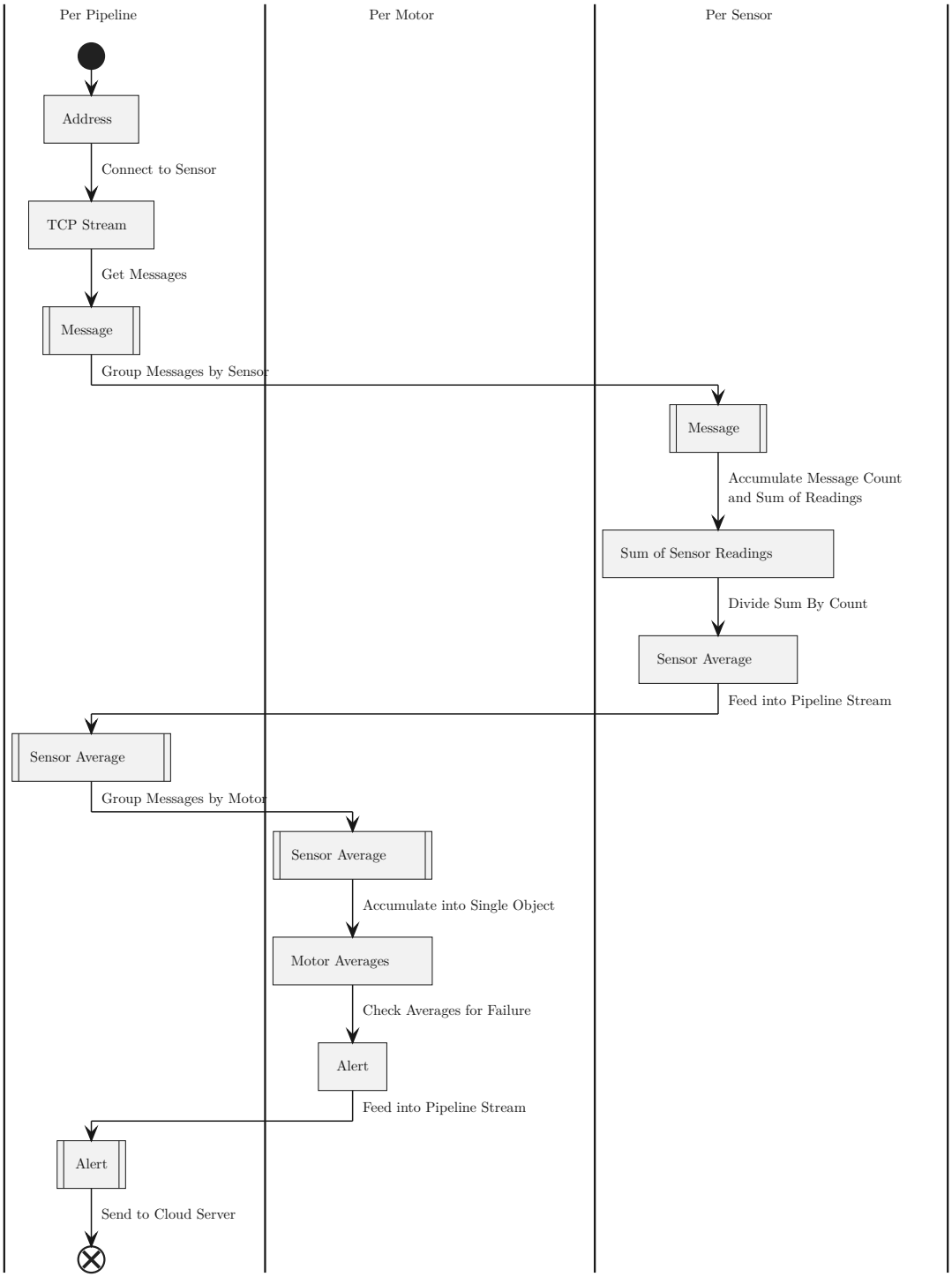


Figure 4.5: The data flow in the declarative data processing service. The elements with a vertical bar left and right symbolize collections.

abstraction - the communication between different machines - than the communication regarding the processing of the received data, which happens between multiple threads on the same machine. In an implementation designed for production use, the inter-device communication would also be hidden in a library providing only the required abstractions, so that the pipeline definition consisted of the second stage of the three listed above. As the creation of a corresponding software package has been deemed out of scope for the thesis, it was decided to forego this and handle the TCP connections in the business logic layer.

4.3.2 Data Infrastructural Layer

The data infrastructural layer provides a set of operations for data transformation to developers, which they can use to set up a processing pipeline at the business logic level. Upon execution, it is responsible for executing the specified operations, and passing the results from one step to the next.

Implementation

The implementation of the data infrastructural layer is done in a separate library. The operators provided by it can be classified into three types:

- Create a stream
- Transform a stream or the items in it
- Assemble and execute a stream

When using the library to implement and execute a data processing pipeline, its employment happens in three phases — creation, assembly and execution. See Figure 4.6 for an exemplary graphical depiction of the procedure of the different phases.

Creation Creation of the processing pipeline starts with a stream creation function, like `create`, which returns an object offering all possible stream transformation and finalization methods. Invoking a transformation returns a new one object which again offers all possible functions and internally contains a reference to the old object. By chaining those procedures, the data processing pipeline is constructed, where each step holds a reference to the step before it. Upon application of a finalization function, the pipeline is completed and the assembly phase begins.

Assembly Assembly of the pipeline starts at the finalization invocation of the last element of its definition. In it, a method called `actual_subscribe`, provided by the transformation step contained in the finalization object, is invoked, passing it the writing end of a communication channel. During assembly, all transformation objects follow the same procedure:

- Create a communication channel
- Set up the specific data transformation, which reads objects from the newly created channel and writes them to the channel passed to it.
- Invoke the `actual_subscribe` function of the step the objects holds a reference to, passing it the created communication channel.

Once the `actual_subscribe` function of the stream creation object has been reached, it starts to emit items, and the execution phase begins.

Execution During execution, the creation object generates items as specified, and writes them to the channel it received in its `actual_subscribe` function. The transformation functions wait for items arriving from the upstream step, apply their respective operations, and send them to the downstream object. The finalization object does not participate in this phase.

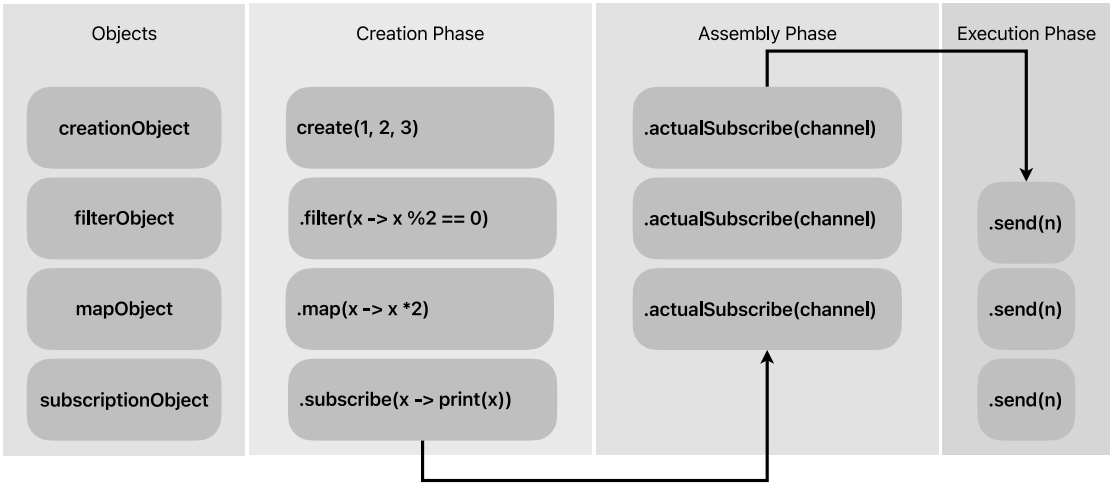


Figure 4.6: The procedure of the phases of an exemplary data processing pipeline. The black arrow marks the processing flow during pipeline execution.

Communication

Communication between the threads of the different processing steps is realized with Rust's `std::sync::mpsc::channels`. Those are multiple producers, single consumer, first-in first-out queues which are one of the programming language's main construct for facilitating coordination of parallelized tasks. Upon channel creation, one object for sending and one object for receiving is created, which can then be passed to different processes. The sending object can be used in both a synchronous form (upon sending data, processing is only resumed once it has been read at the other end) and an asynchronous form (processing can resume immediately, and the data is buffered by the communication

channel). Independent of the sending mode, the object for receiving can also be used either synchronously (processing stops until data has been read from the channel) or asynchronously (reading immediately returns pending data or an empty object if the buffer contained none).

For this library, the main mode of exchange is asynchronous sending, as the execution of the sending thread should not be slowed down by how fast the receiver can process the data, and synchronous reading, so that the receiver does not require processing time when there is nothing to do.

The usage of `std::sync::mpsc::channels` also directly supports communication about termination of the processing stream. Once one side of the channel closes the connection (either the sender because all elements have been processed, or the receiver because the processing execution has been aborted prematurely), any further attempts to write to or read from the channel fail. Upon noticing this, the still active thread can shut down its processing, close all outgoing connections, and then be removed by the scheduling system.

Operators

The operator set provided by the library has been selected from the ones available for most stream processing libraries such as RxJava[2], Project Reactor[3] and Akka Streams[4]. From the list of the functions provided by them, those that would be necessary to implement the required functionality of the motor monitoring service have been chosen. The implemented operators are:

1. `map(x → f(x))`: Apply the function `f` to each object `x` sent by the previous operator, and emit the result.

2. `filter(x → p(x))`: Evaluate the predicate `p` with each received object `x`. Only emit the object `x` if the result is `true`.

3. `reduce(C, x → f(c, x))`: Sequentially apply the function `f(c, x)` to each received object and use the result as `c` for the next application, starting with `C` as the first `c`. Once the previous operator closes the connection, emit the last calculated `c`.

4. `group_by(x → g(x))`: Emit each item `x` into a stream corresponding to the result of `g(x)`.

5. `flatten()`: Assuming that each item sent by the previous operator is a stream by itself, emit all values emitted by them into a single stream.

[2] `https://github.com/ReactiveX/RxJava` (visited on 2024-05-08)
[3] `https://projectreactor.io/` (visited on 2024-05-08)
[4] `https://akka.io/` (visited on 2024-05-08)

6. `flat_map(x → f(x))`: A combination of `map` and `flatten`. Apply the stream-emitting transformation `f` on each element, and flatten the resulting streams into a single one.

7. `average()`: Collect all elements arriving from the previous operator and emit their (arithmetic) average when it exits.

8. `sliding_window(t1, t2, x → t(x))`: Collect all arriving elements, and emit all collected elements which are not older than `t1` every `t2` seconds. Applying `t(x)` on an object determines its age.

9. `merge(s)`: Emit all received items and the items received from the stream `s` into a single stream.

Multithreading

As the communication between the processing steps is done in a thread-safe way, they can be scheduled freely and independently by the operating system.

A defining characteristic of Rust is that scheduling, and asynchronicity in general, is not a fixed part of the standard library, or any other library, but is usually defined by the developers using them in their executable program. This is done as the desired mode of scheduling varies greatly depending on the environment an executable should be run in. Separating the library implementation from how its independent tasks are executed allows users to tailor the execution mode to their specific context, be it single-threaded embedded processors or large multiprocessor systems.

In the library implemented for this master's thesis, this freedom of choice is provided through the finalization function `subscribe`. As argument, it expects a scheduling structure, which, together with the communication channel, is passed upwards the processing stream through the `actual_subscribe` function. This scheduling structure needs to provide two functions:

- `schedule(f()) → h`: Schedule a function `f` to run, returning a handle `h` to the execution, which can be used to wait on the completion of the submitted task.
- `schedule_repeating(f(), d) → h`: Repeat the function `f()` periodically every `d` seconds. The returned handle `h` can be used to terminate the scheduled repetition.

As each system has a limited amount of processors, and thread creation uses a lot of operating system resources, many scheduling libraries limit the amount of tasks which can be executed concurrently by creating a fixed number of operating-system threads, and assigning tasks to them whenever possible. This prevents the unnecessary creation of threads which cannot be scheduled by the processor, as it only has the resources to execute a limited amount of jobs anyway. While this is a fitting system for tasks which

only do data processing, it runs into problems when there are tasks which communicate with other components and thus need to execute I/O operations. Those operations are so-called "blocking" calls, as a thread submitting such an operation needs to wait on the response of the external component before it can continue processing. Thus, programs which submit blocking tasks run at risk of thread contention, meaning that all operating-system threads are waiting on the result of I/O calls and thus cannot process other submitted tasks, bringing the system to a halt. There exist schedulers which can prevent some cases of thread contention via internal mechanisms, but to not limit users of the presented library in their choice, an additional mechanism at the level of the pipeline creation is provided: The function `subscribe_on(s)` can be invoked on any creation or transformation object, and receives a scheduler, like the `subscribe` function. Upon pipeline assembly, when the operators `actual_subscribe` function is invoked, it passes the scheduler it received upon construction to the next processing step, instead of the one handed to it by the downstream operator. This can be used to provide a dedicated set of workers to a processing step which executes blocking calls, thus preventing thread contention.

In the data processing service created for the evaluation, the futures-rs[5] library is used for task scheduling. It has been selected as it is thoroughly documented, can be executed on a broad range of platforms, and the thread pool provided by it does not use any internal heuristics for scheduling. This was assessed as important, as having a more sophisticated scheduler could heavily influence the results of the benchmarks and thus further limit the transferability of the learning of this thesis. Other evaluated schedulers were Tokio[6], Actix[7], and async-std[8].

[5] `https://github.com/rust-lang/futures-rs` (visited on 2024-05-08)
[6] `https://github.com/tokio-rs/tokio` (visited on 2024-05-08)
[7] `https://github.com/actix/actix-net` (visited on 2024-05-08)
[8] `https://github.com/async-rs/async-std` (visited on 2024-05-08)

CHAPTER 5

Results

In this chapter, the results of executing the benchmark as defined in Chapter 3 are presented. For easier reference, a short summary of the methodology is given beforehand.

To answer the research questions listed in Section 1.3, a benchmark scenario and suite implementing it are constructed.

The scenario consists of θ motors, which are observed by sensors, and a monitor combining those observations to detect failures. In the case of failures, an alert is sent to a cloud server for further processing. See Figure 3.2 for a graphical depiction of the benchmark setup.

To compare the performance of the declarative and the imperative paradigm in the given setting, the service processing the incoming sensor data is implemented once per paradigm.

The scenario is then executed on two similar, but not equal, machine setups (see Table 5.1). The execution on two distinct systems is done to test whether the service implementations are influenced significantly by the underlying hardware.

The benchmarking runs were done with differing parameter sets, chosen to evaluate the services under different load (number of motors θ) and processing requirements (window size ω). See Table 3.2 for a list of all parameter values the benchmark was executed with.

On each run the execution time and memory consumption of the benchmarked service, and the CPU load on the motor monitor are recorded. Additionally, the message latency between the sensors and the cloud server receiving the alerts is measured. Per parameter set 50 repetitions were executed, and the measurements stored for analysis.

As each test run is influenced by a multitude of external factors, the results have a certain degree of variance. To determine the parameter sets where the differences stem from internal properties of the implementations, and are not due to chance, a statistical analysis

Table 5.1: The hardware makeup of the two benchmark systems.

		System A	**System B**
System-On-A-Chip	Model	Raspberry Pi 3B	Raspberry Pi 4B
	Base Processor Speed	1.2Ghz	1.8GHz
	Cores	4	4
	RAM	1GB	2GB
	Linux Kernel Version	6.2	6.1
	Rust Version	1.72 (stable)	1.72 (stable)
Benchmark runner	CPU Model	AMD Ryzen 7 4800H	Intel Xeon Gold 6152
	Base Processor Speed	2.9 GHz	2.1 GHz
	Cores	16	16
	RAM	32GB	32GB
	Linux Kernel Version	6.2	5.14
	Rust Version	1.72 (stable)	1.72 (stable)

of the measurements is done. In this thesis it was decided to conduct paired sample, one-tailed t-tests with a significance level α of 0.05 for assessing the differences between the calculated averages of the measurements. To cover all possible results, each test will be conducted two times, with the null hypothesis always assuming equal performance, and the two alternative hypotheses assuming better performance of respectively one implementation.

In the figures shown below, each diagram represents the measurements of the benchmark of one system size configuration. The x-axis of the diagrams represents the window size, the y-axis the dependent parameter. The results of the declarative implementation are displayed in red, the ones of the imperative implementation in gray. The left column are the results of Benchmark System A, the right of Benchmark System B. From the top to the bottom, the system size increases.

It is important to note that while a boxplot visualizes the quartiles of the recorded data, the t-test is based off its average, which is much more influenced by outliers. This means that the tests results are not necessarily visible in the diagrams.

5.1 Runtime

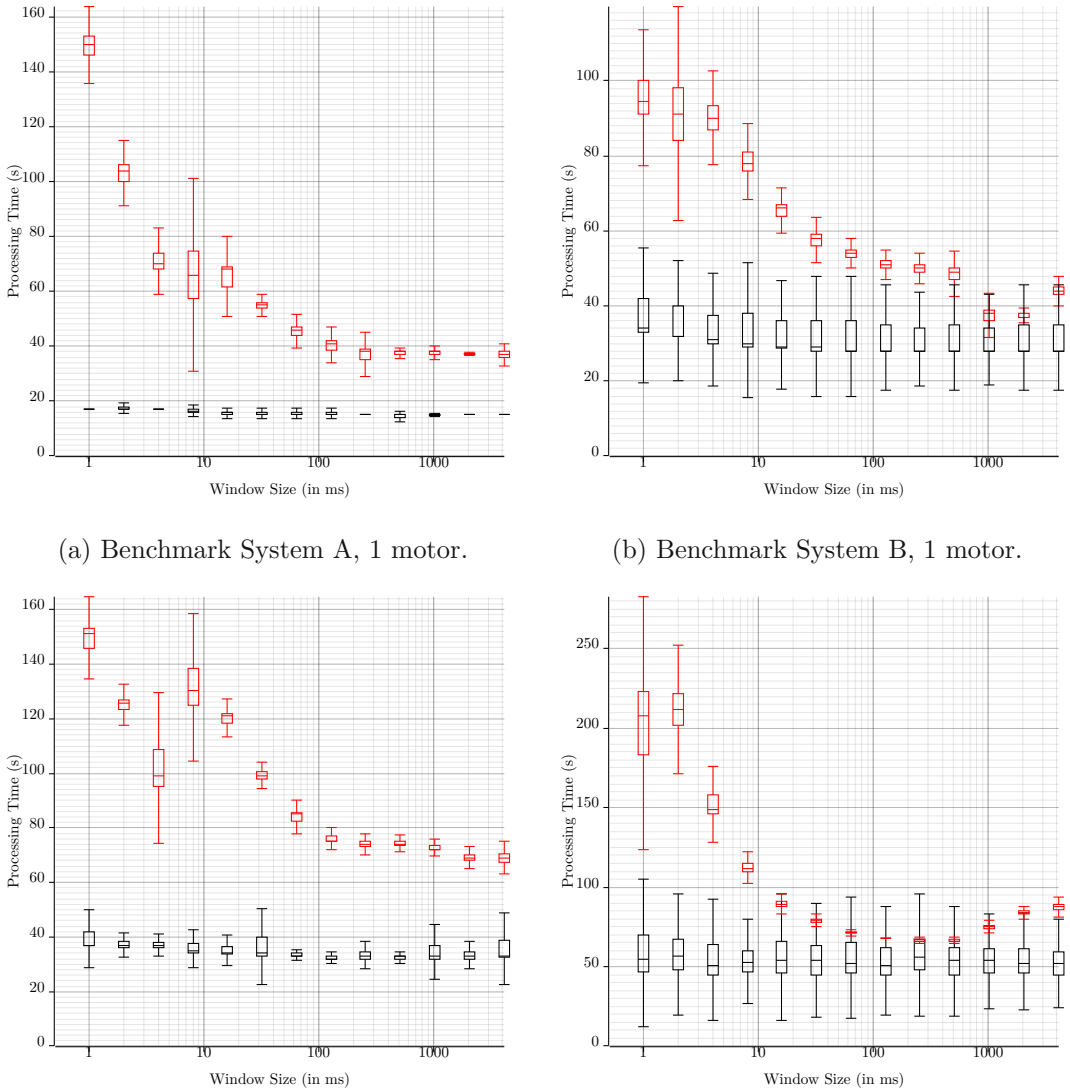
The runtime, measured to test Hypothesis 1, was recorded as the total time the monitoring process and its children were executed in user and kernel space. A graphical summary of the measurements can be seen in Figure 5.1.

In Figures 5.1a to 5.1f, we can see that for a system size of one, two, and four motors, the imperative implementation outperforms the declarative implementation consistently on both benchmark systems, although the speed differences decrease with increasing window size. For a system size of eight motors, the declarative implementation is still notably faster on Benchmark System A (Figure 5.1g), but on Benchmark System B

(Figure 5.1h), the performance seems about equal for lower window sizes (<128ms). For the benchmark runs using systems of 16 motors, Figure 5.1i and Figure 5.1j show a very different performance on the two benchmark systems. While on Benchmark System A, the processing times of the two services indicate better performance of the declarative implementation in only two cases (window sizes of 4ms and 512ms), on Benchmark System B the declarative data processing service seems to have executed faster for all tested parameter sets.

For Benchmark System A, over all 65 different executions of the t-test, the statistical tests confirmed a shorter processing time of the imperative implementation in 61 cases, an equally long in three cases (16 motors, windows of 8, 128 and 256 milliseconds length) and a shorter processing time of the declarative implementation in one case (16 motors, windows of 4 milliseconds length).

For Benchmark System B, the tests confirmed a shorter processing time for the imperative implementation in 52 cases, and a shorter processing time for the declarative implementation in 13 cases (16 motors, all window sizes).



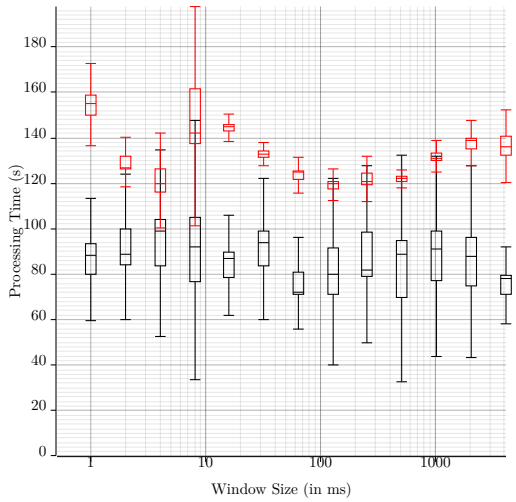
(a) Benchmark System A, 1 motor.

(b) Benchmark System B, 1 motor.

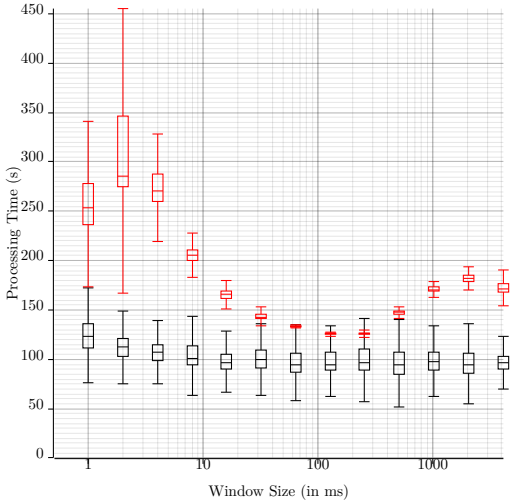
(c) Benchmark System A, 2 motors.

(d) Benchmark System B, 2 motors.

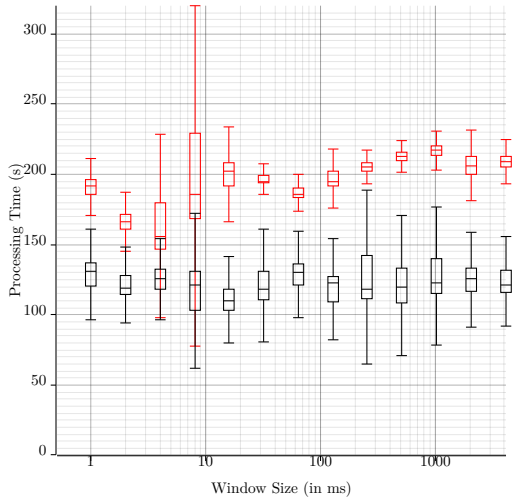
Figure 5.1: Boxplots of the recorded Processing Times (in s).
■ Declarative Implementation ■ Imperative Implementation



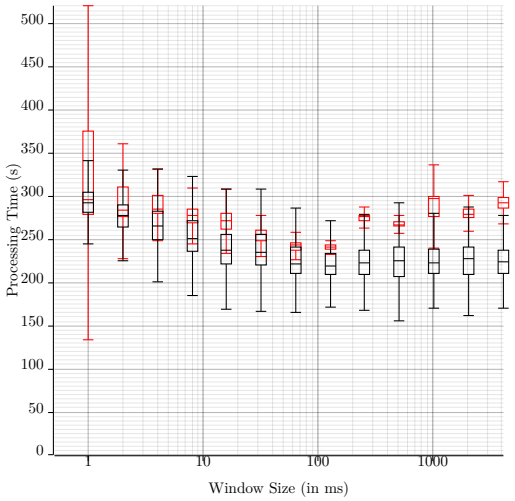
(e) Benchmark System A, 4 motors.



(f) Benchmark System B, 4 motors.

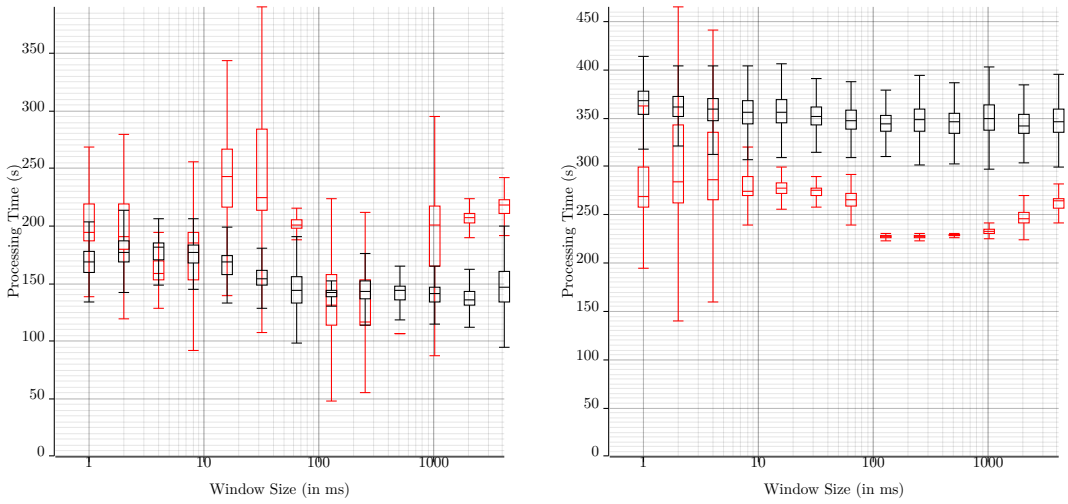


(g) Benchmark System A, 8 motors.



(h) Benchmark System B, 8 motors.

Figure 5.1: Boxplots of the recorded Processing Times (in s).
■ Declarative Implementation ■ Imperative Implementation



(i) Benchmark System A, 16 motors.

(j) Benchmark System B, 16 motors.

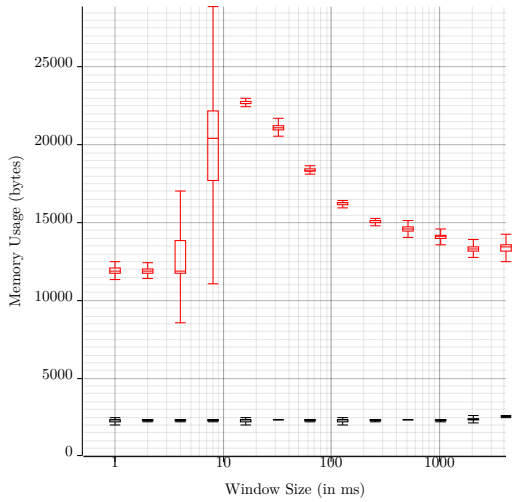
Figure 5.1: Boxplots of the recorded Processing Times (in s).
■ Declarative Implementation ■ Imperative Implementation

5.2 Memory Usage

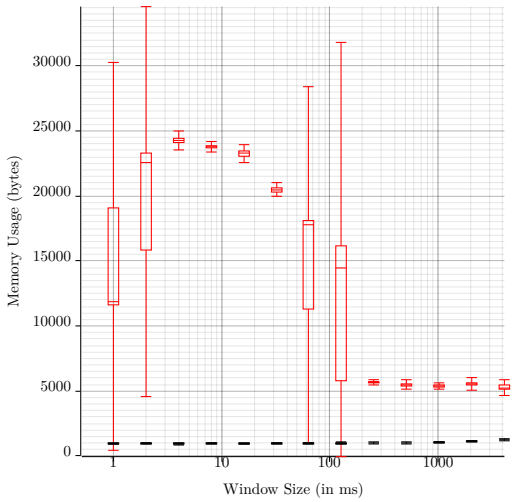
As indicator for the memory usage of each implementation, also measured for testing Hypothesis 1, the peak resident set size was recorded during the benchmark execution. A graphic overview of the recordings can be seen in Figure 5.2.

In the figures, one can clearly see that the declarative implementation consistently uses more memory than the imperative implementation in all evaluated parameter configurations. Generally, the memory consumption of the imperative implementation stays the same with growing window sizes. The declarative implementation on the other hand seems to consume more memory when increasing the window size from one to 16 milliseconds, but for larger sizes the further development depends on the number of motor groups. Up until eight motor groups (Figures 5.2a to 5.2h), the memory consumption decreases in all cases until around a window size of 265ms, and then grows again. For the setups with 16 motor groups (Figure 5.2i and Figure 5.2j), the consumed memory does not decrease with larger window sizes, but stays the same or continues to grow.

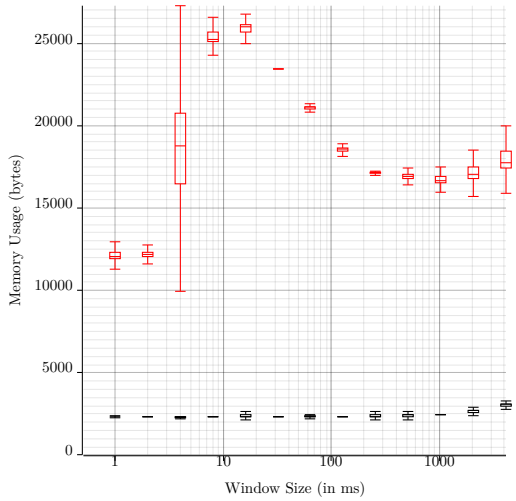
The t-tests, testing the memory usage of the implementations, showed that the imperative implementation used less memory in all the cases for both Benchmark Systems.



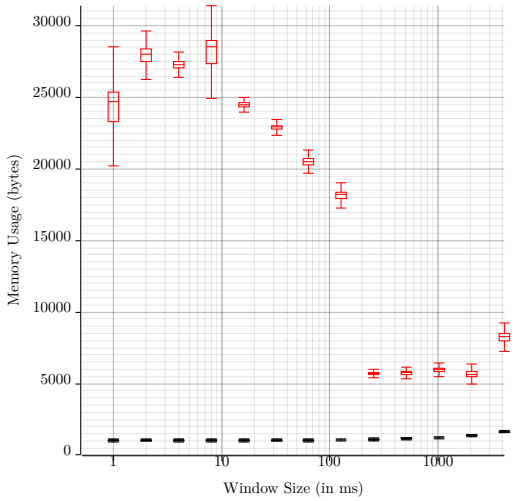
(a) Benchmark System A, 1 motor.



(b) Benchmark System B, 1 motor.

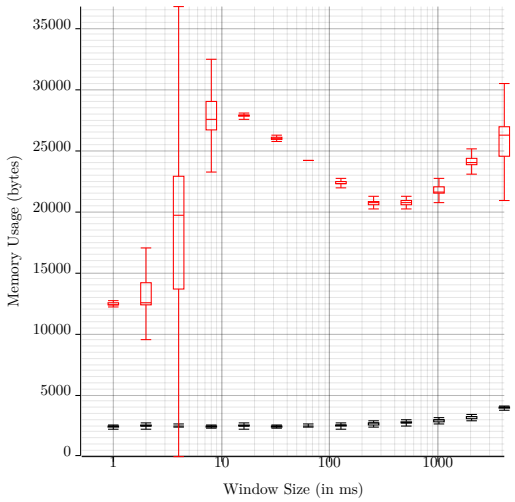


(c) Benchmark System A, 2 motors.

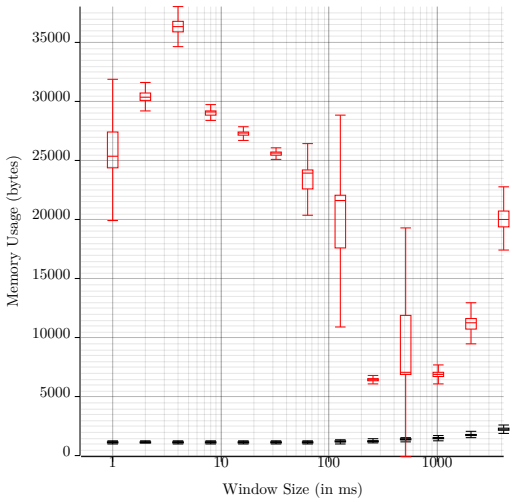


(d) Benchmark System B, 2 motors.

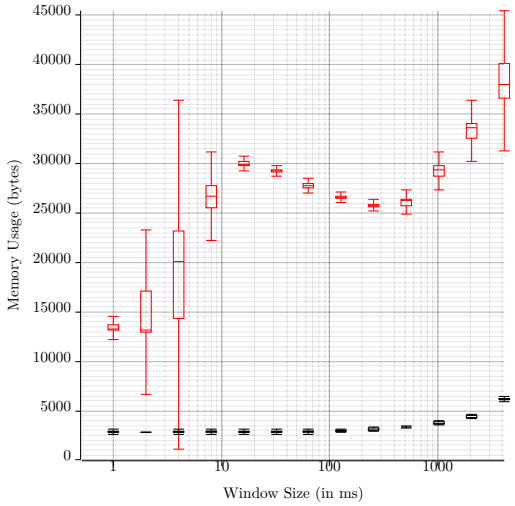
Figure 5.2: Boxplots of the recorded Memory Usage (in kB).
■ Declarative Implementation ■ Imperative Implementation



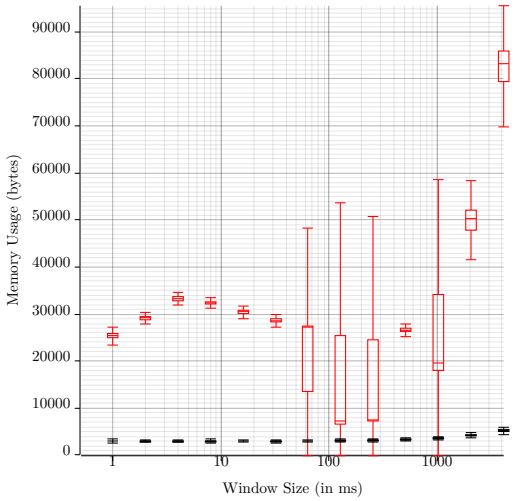
(e) Benchmark System A, 4 motors.



(f) Benchmark System B, 4 motors.

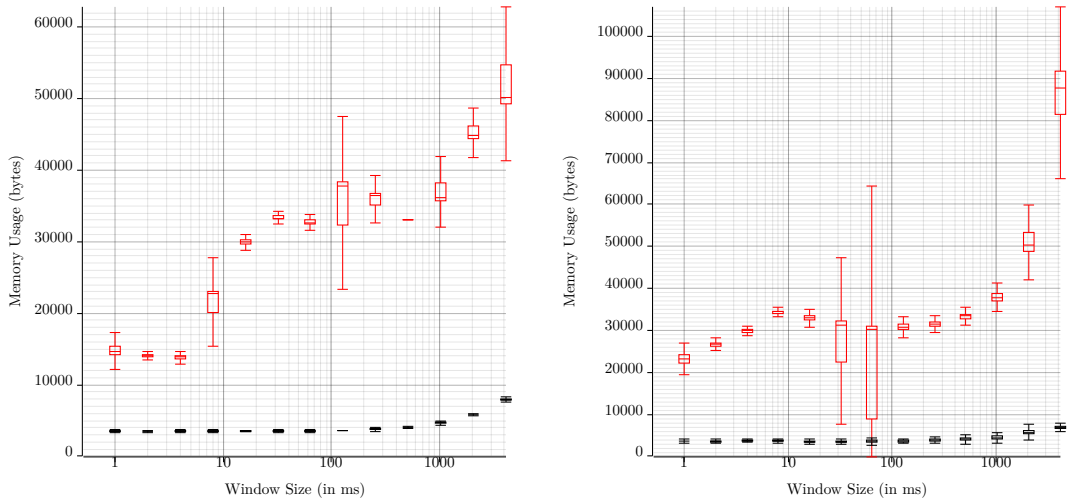


(g) Benchmark System A, 8 motors.



(h) Benchmark System B, 8 motors.

Figure 5.2: Boxplots of the recorded Memory Usage (in kB).
■ Declarative Implementation ■ Imperative Implementation



(i) Benchmark System A, 16 motors.

(j) Benchmark System B, 16 motors.

Figure 5.2: Boxplots of the recorded Memory Usage (in kB).
■ Declarative Implementation ■ Imperative Implementation

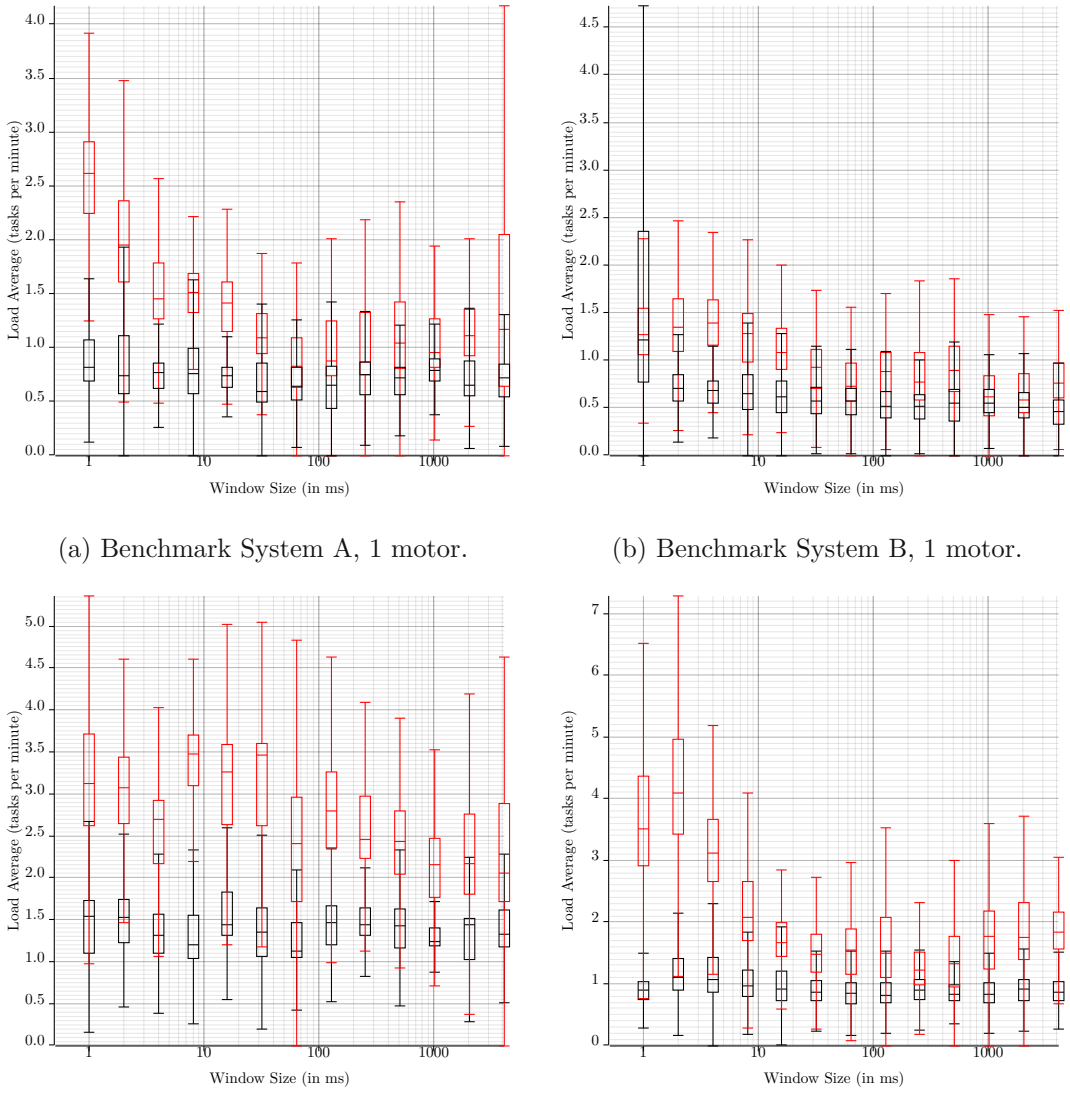
5.3 Load Average

"Load Average" is defined as the average length of the run queue (how many tasks are waiting for execution or an I/O operation) over the last minute. This number is also relevant for verifying Hypothesis 1. A graphic summary of the measurements of the benchmark runs is given in Figure 5.3.

In the figures, it is immediately visible that the variance of the load average is very high in all cases. Furthermore, the imperative implementation seems to be able to sustain about the same load average with increasing window size for all tested configurations except on Benchmark System B with 16 motors (Figures 5.3a to 5.3i). The declarative implementation on the other hand mostly has a higher load average than the imperative one, but the boxplots often overlap, which indicates that the differences might not be significant. When looking at the load averages of the declarative implementation running on Benchmark System A, one can see that for system sizes of one and two motors (Figure 5.3a and Figure 5.3c), it generally seems to decrease with larger window sizes. For the systems with four and eight motors (Figure 5.3a) and Figure 5.3c), the load average grows with the sizes of the windows, and for the system with 16 motors (Figure 5.3i), the load average changes from growing to shrinking multiple times. In Benchmark System B, the load average of the declarative implementation seems to grow for the first window sizes, then decreases, until it finally grows again. The length of the growth and shrinking phases is different from system size to system size.

Executing the t-tests for Benchmark System A showed that the imperative solution had a lower load average in 63 cases. In the other two cases (16 motors, window sizes of 128 and 256 milliseconds), equal performance of both services was determined.

For Benchmark System B, the tests revealed a lower system load with the imperative solution for 64 parameter sets, and with the declarative solution for the system with one motor and a window size of one millisecond.



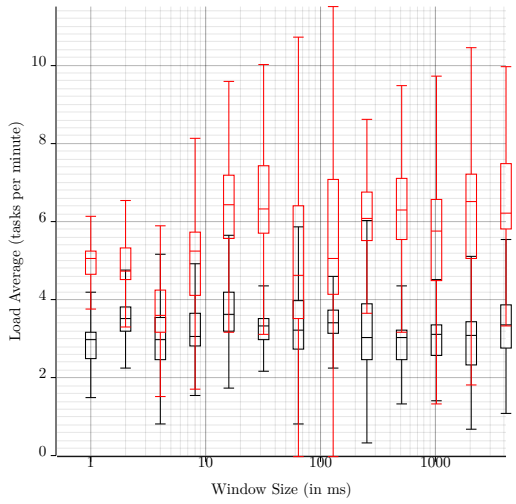
(a) Benchmark System A, 1 motor.

(b) Benchmark System B, 1 motor.

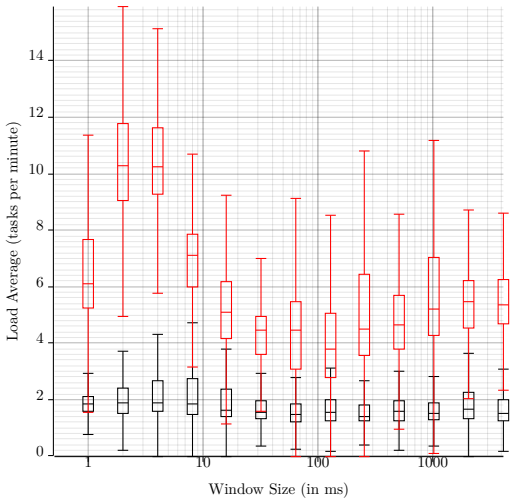
(c) Benchmark System A, 2 motors.

(d) Benchmark System B, 2 motors.

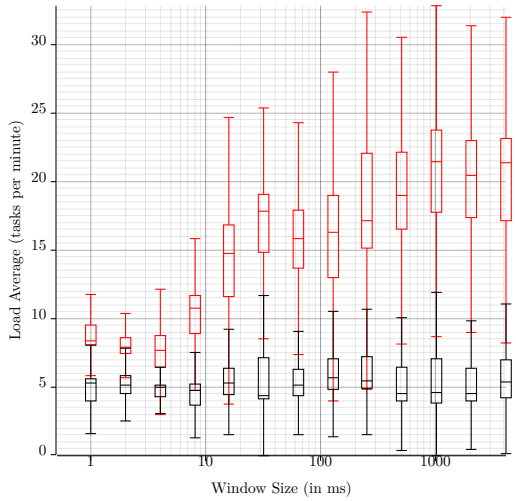
Figure 5.3: Boxplots of the recorded Load Averages.
■ Declarative Implementation ■ Imperative Implementation



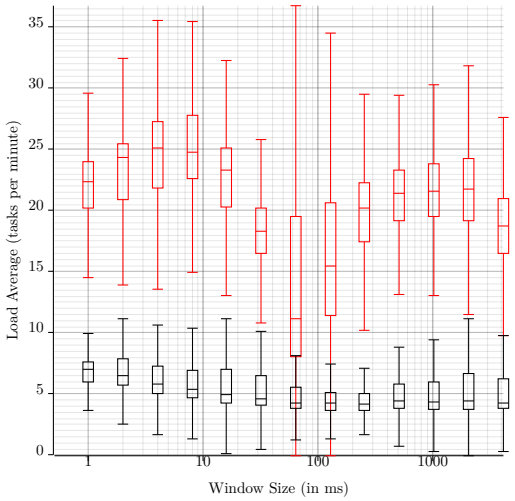
(e) Benchmark System A, 4 motors.



(f) Benchmark System B, 4 motors.

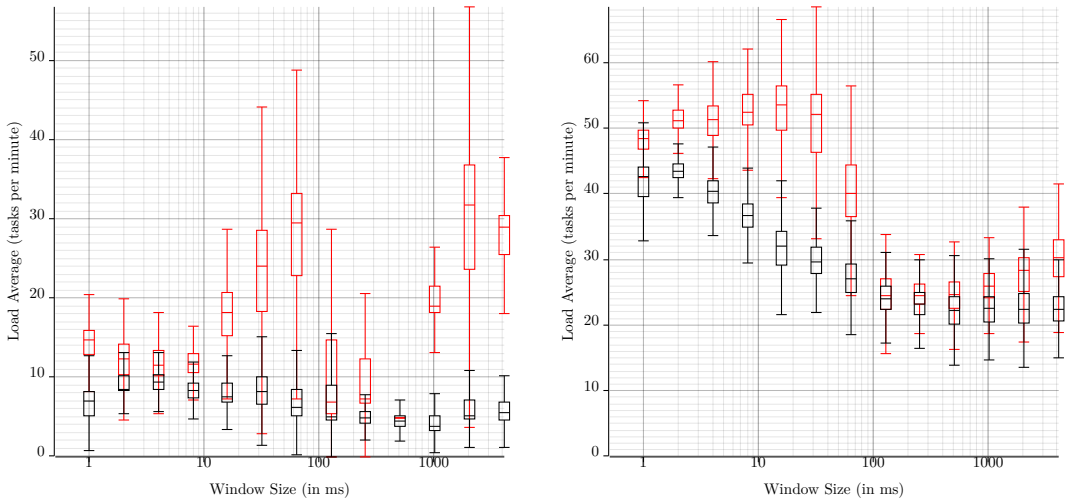


(g) Benchmark System A, 8 motors.



(h) Benchmark System B, 8 motors.

Figure 5.3: Boxplots of the recorded Load Averages.
■ Declarative Implementation ■ Imperative Implementation



(i) Benchmark System A, 16 motors.

(j) Benchmark System B, 16 motors.

Figure 5.3: Boxplots of the recorded Load Averages.
■ Declarative Implementation ■ Imperative Implementation

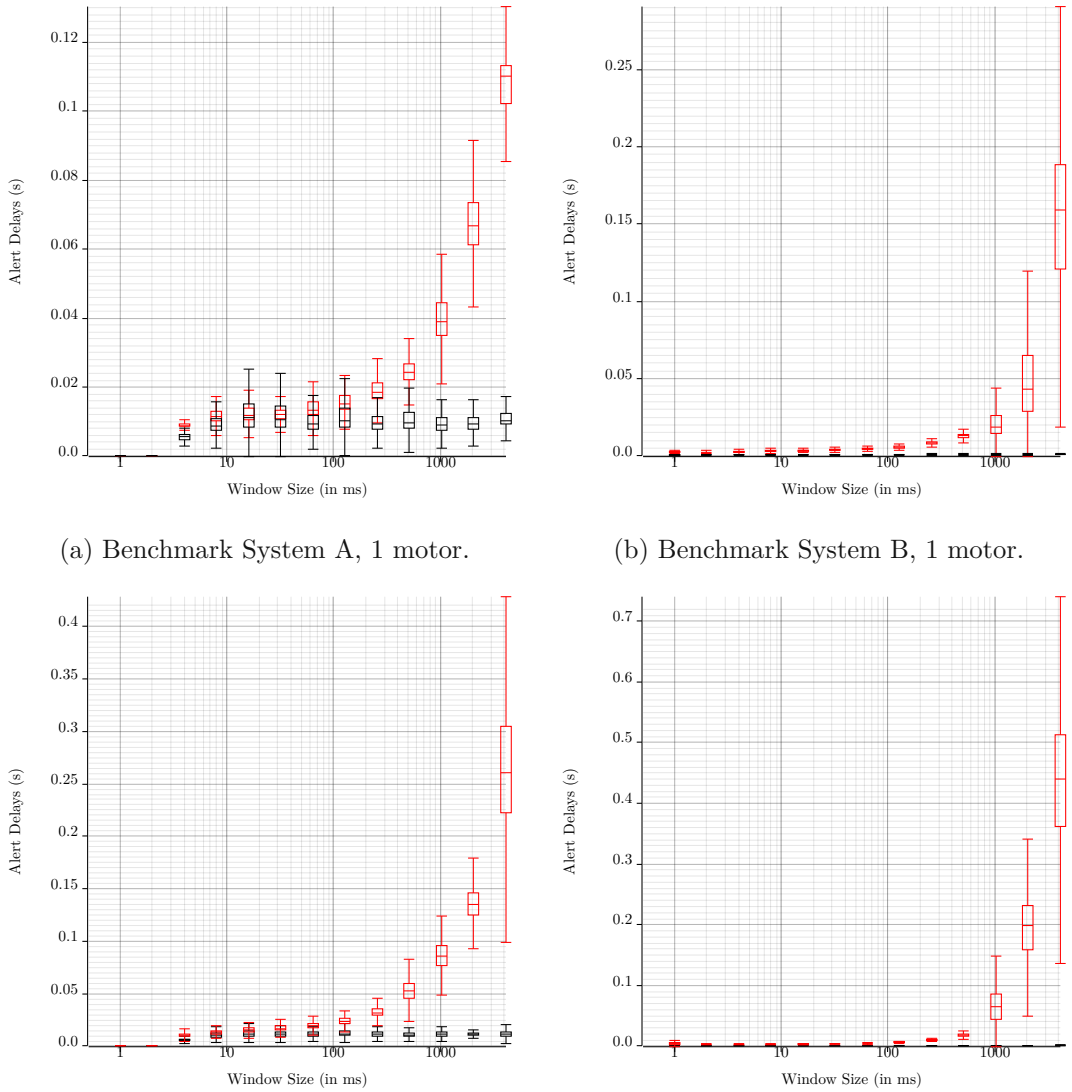
5.4 Alert Delay

"Alert Delay" describes the time difference between the emission of a message triggering an alert from a sensor and the arrival of the corresponding alert at the cloud server. It has been measured to test Hypothesis 2. A graphic roundup of the measurements taken during the benchmark can be seen in Figure 5.4.

The figures show that the imperative implementation can generally sustain a lower alert delay than the declarative implementation. As with the load average, the imperative implementation manages to keep a consistent alert delay with growing window sizes. Contrary to this, the alert delay of declarative implementation is close to the one of the imperative implementation for short window sizes, but as the window size increases, the delays grow more than linearly.

Evaluating the retrieved data, the t-tests confirmed a lower alert delay for Benchmark System A with the imperative solution in 63 cases, and an equal load average in the configuration with one motor and with two motors, both times with window sizes of 128 milliseconds.

For Benchmark System B, the data always indicated a lower alert delay of the imperative solution.



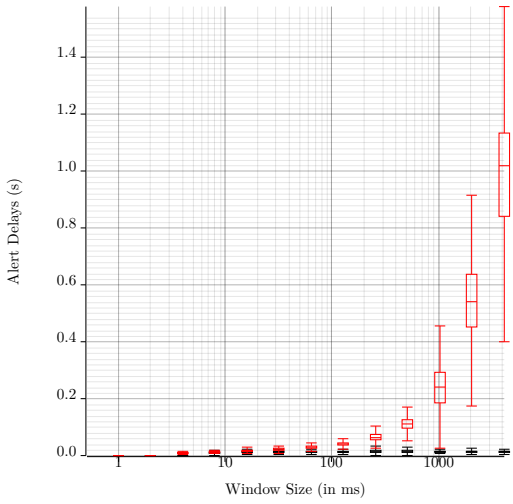
(a) Benchmark System A, 1 motor.

(b) Benchmark System B, 1 motor.

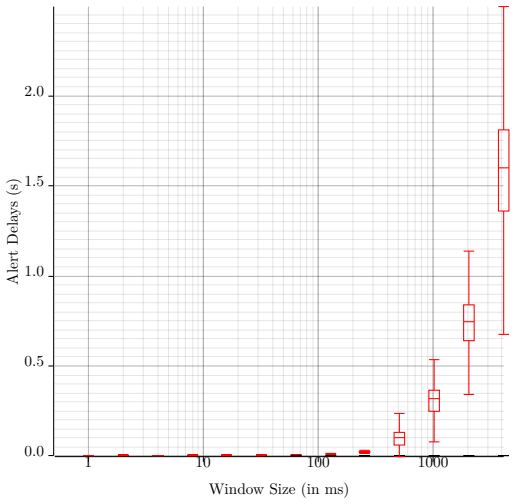
(c) Benchmark System A, 2 motors.

(d) Benchmark System B, 2 motors.

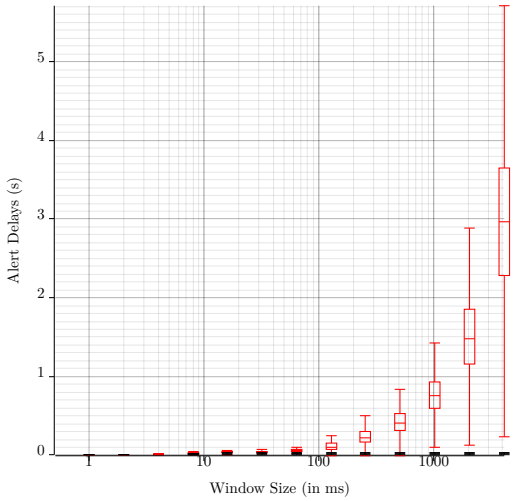
Figure 5.4: Boxplots of the recorded Alert Delays (in seconds).
■ Declarative Implementation ■ Imperative Implementation



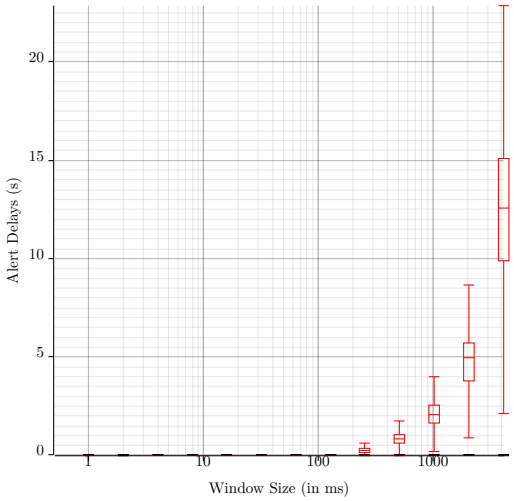
(e) Benchmark System A, 4 motors.



(f) Benchmark System B, 4 motors.

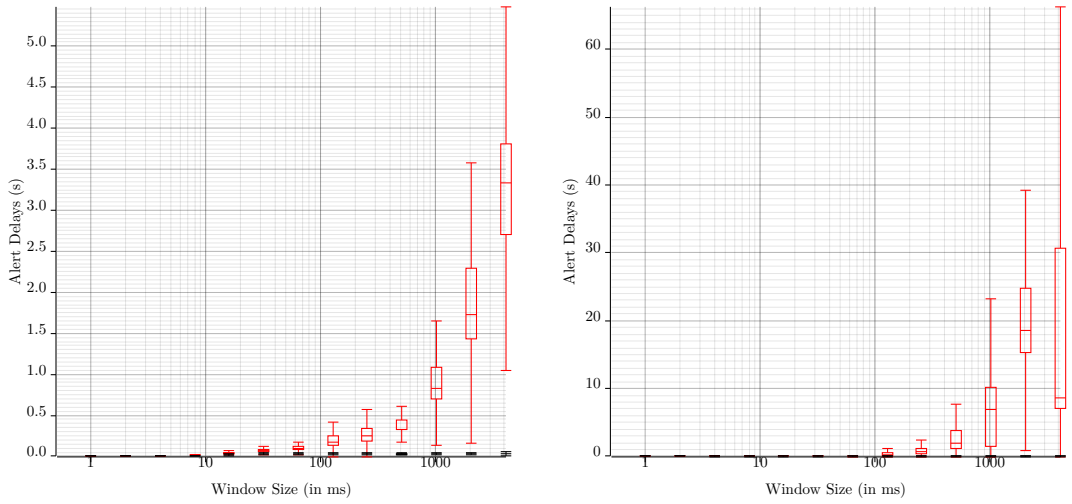


(g) Benchmark System A, 8 motors.



(h) Benchmark System B, 8 motors.

Figure 5.4: Boxplots of the recorded Alert Delays (in seconds).
■ Declarative Implementation ■ Imperative Implementation



(i) Benchmark System A, 16 motors.

(j) Benchmark System B, 16 motors.

Figure 5.4: Boxplots of the recorded Alert Delays (in seconds).
■ Declarative Implementation ■ Imperative Implementation

5.5 Summary

Table 5.2 and Table 5.3 list the results of the statistical tests performed on the collected data. On System A, the imperative implementation outperformed the declarative implementation on 252 of the 260 parameter sets. On System B, this was the case for all but 14 parameter sets. As can be seen, the imperative implementation generally seems to outperform the declarative implementation, and the underlying hardware does not seem to influence the relative performance difference significantly[1]. An in-depth analysis of those findings and possible influencing factors will be done in the next chapter, which will also explore the limitations of this work.

[1] As evaluating hardware is out of the scope of this thesis, only the relative performance differences are analyzed.

Table 5.2: How many times the t-tests over the measurements on System A indicated better performance for each implementation.

	Imperative	**Equal**	**Declarative**
Runtime	61	3	1
Memory	65	0	0
Load Average	63	2	0
Alert Latency	63	2	0

Table 5.3: How many times the t-tests over the measurements on System B indicated better performance for each implementation.

	Imperative	**Equal**	**Declarative**
Runtime	52	0	13
Memory	65	0	0
Load Average	64	0	1
Alert Latency	65	0	0

CHAPTER 6

Discussion

Looking at the results of the test runs, it is clear that the declarative implementation generally requires more resources in the scenarios selected for evaluation. To allow a more in-depth discussion of the findings, this chapter interprets the outcomes and sets them in relation to previous work and the research question, so that the implications of the results can be used as a basis for decision-making when realizing similar projects. Furthermore, the limitations of the findings of this thesis are laid out, to make it clear in which scenarios they can be considered applicable.

6.1 Analysis of the Results

Evaluating the measurements, the foremost aspects to analyze are of course the performance differences between the two implementations, with the goal of answering the research questions posed in the introduction. Another aspect, which will be looked at in the second subsection, is the overall impact of the benchmark parameters on the metrics, meaning for example how larger window sizes influenced the runtimes of the services.

6.1.1 Performance of the implementations

From the obtained data, one can see that the performance of the imperative approach is generally better for every selected metric. While it is not possible to determine the exact reasons for this without in-depth analysis of all participating components, which is out of the scope of this work, there are a some factors which probably influenced the results significantly. Those factors are

Task management overhead One large probable factor is connected to the efforts needed to create and manage tasks. First, on task creation it needs to be submitted to the task queue, and assigned to a thread once it has moved to the front of the queue.

Second, during execution of the task, the scheduler regularly needs to check whether the task has finished, in which case the next one is submitted for execution. While the imperative implementation only creates five tasks per motor group, the number of tasks created by the declarative implementation can not really be upper-bounded. This is because additionally to growing with the amount of motor groups, it also grows linearly with the number of messages relevant for determining whether a failure has occurred. This fact probably has a large influence on the longer runtime and alert delay of the declarative implementation.

Task coordination overhead Related to this is the much higher amount of messages sent in the declarative implementation. Both implementations rely on `std::sync::mpsc::channel` for transmitting data between tasks, but as the declarative implementation employs many more tasks, it also needs to send and receive a lot more messages. While the channels are designed for cheap and fast data transmission, they nevertheless incur some amount of overhead. Its accumulation could serve to explain a part of the longer runtime and alert delay of the declarative implementation.

Context switching Another cost factor of the message transmission are the additional efforts needed for context switching. Context switching is an operation necessary to elevate the privileges of a process, so that, in the implementations at hand, the program can read from and write to channels. As the declarative implementation requires much more of those operations, their added costs could significantly influence the outcome of the benchmarks, especially the required runtime and alert delay.

Thread execution indeterminism A further influence on the results of the benchmark could be that the time of execution of a single task submitted to the thread pool is highly dependent on the kernel. The library handling the execution of the data processing pipeline submits many tasks with differing interdependent workloads - some CPU bound, some I/O bound - to the processing queue. This queue is processed by a number of threads, which are in turn scheduled by the kernel. As most of the data transformations depend on the outcome of some earlier steps, decisions on when the kernel schedules a thread for execution by the CPU have large effects on the runtime of multiple tasks. One such example would be a task listening for a message on a socket which gets suspended shortly before the message would have arrived. The processing step depending on the just suspended step now has to wait for the full time the task is suspended, and the same could happen to the step after it. Due to the nature of the implementations, the task dependence chains can get much longer in the declarative implementation, thus increasing the impact of unfortunate scheduling decisions. This effect could explain the longer alert delays and the large variations in the benchmark results of this implementation.

Locality Another influencing factor which could have significant effect on the performance of the implementations is the issue of the temporal and spatial locality of multiple accesses to the same memory region. Each CPU core has multiple levels of caches, where

small parts of the program memory are stored to enable faster data access, with larger levels being slower to read and write than smaller ones. Whether a program can utilize them efficiently is influenced by multiple factors, of which in our case temporal and spatial locality are the ones which might have the largest influence. Spatial locality describes the issue of making multiple accesses to the same memory region from the same CPU core, or group of cores. This is important as cores can have individual smaller caches, and may share larger caches with other cores. Depending on this, better spatial locality can lead to faster memory access times. Temporal locality describes the concept of how much time there is between accesses to the same memory region, with lower durations meaning a higher chance that the relevant memory still resides in a smaller cache. In the case of the implementations of this thesis, it can be supposed that the imperative implementation has better temporal and spatial locality, and gains a significant speed advantage from this. In the spatial case, this could be because the buffers, used for storing the sensor messages and the averages, are only accessed by one thread each. In the declarative implementation on the other hand, during sorting the data is distributed to many tasks, which are allocated to threads at random. Neither the compiler nor the kernel can optimize for spatial locality when assigning tasks to threads and threads to cores, as they do not know what memory they will access during its execution. Linked to this is also the influence of temporal locality. As in the imperative implementation accesses for processing the buffer, or the averages, are happening in one dedicated task each, the only way how the complete reading and processing of one memory region can be suspended is by the kernel actively interrupting the thread executing the task. In the declarative implementation on the other hand, the same data is read and written by many tasks, which are all exiting and thus relinquishing the CPU after doing their data transformation. After a thread exits, the next task in the task queue is selected for execution, which might need a completely different set of data, meaning that the cache of the previous task is deleted and a new, appropriate one, has to be crated, influencing the runtime and alert delay of the implementation negatively.

Windowing implementation Furthermore, the realization of the window aggregation could also be an influencing factor. While in the imperative implementation each sensor message is only kept once, the library used in the declarative implementation creates a copy of the buffered sensor messages every time it sends a window to the next processing step. Thus, each message has to be stored as often as it occurs in a window, resulting in larger memory usage of the implementation, and additional processing time.

Fine-tuning An additional possibly influencing factor is found on the level of the actual algorithmic implementation. While the imperative implementation is designed in such a way that incoming data is collected in a buffer which is processed at the point of arrival, and only the necessary data is forwarded to the next node, which only receives the averages relevant to it, the declarative implementation could not be implemented in such a direct way. Specifically, as the concept of the sensor handlers and data combiners can not be expressed with the declarative operators, the implementation has to rely on

a number of steps which alternately emit all data into one stream, distribute it into designated buffers, perform an operation on each buffer, and again reemit the results into a single stream. While this is a valid approach for large data processing engines such as Apache Hadoop[1], in smaller use cases it means that a lot of processing power and memory is spent on sorting data to get it into the appropriate form for the next processing step. Those different approaches influence all measured indicators.

There is one series of "outliers" which deserves separate consideration: The processing times of the system with 16 motor groups are always shorter for the declarative implementation on Benchmarking System B, and in 4 cases equally long or shorter on Benchmarking System A. A reason for this might be that the large number of messages arriving every instant overwhelms the imperative implementation, which reads and processes them on the same thread. In the case of a message backlog, the sensor processor is not able to analyze the incoming data efficiently, as even expired messages trigger the execution of the algorithm. The declarative implementation on the other hand distributes reading and processing over multiple tasks, meaning that incoming readings do not necessarily stall the processing of one message window. Unfortunately, simulating even larger systems exceeded the capabilities of the employed hardware, which is why this assumption could not be tested further in this work.

Summary Summing up, it seems as if the key reason of the better performance of the imperative implementation is that it could be modeled closer to the problem of the benchmark scenario. Due to this, it requires a much smaller amount of tasks to process the incoming data, leading to a lower management overhead and better cache locality. Additionally, fewer messages are needed to coordinate the lower amount of tasks in the imperative implementation.

6.1.2 Influence of the parameters

The benchmark is influenced by two parameters, those being the number of sensors in the system θ, and the window size ω, meaning how many messages should be considered relevant for checking whether there has been a failure in the system, which also influences how often the data evaluation should be conducted.

Runtime Evaluating the processing time, multiple insights can be made. First, the runtime of the imperative implementation seems not to be significantly influenced by the window size, as the variance of the runs of one parameter set is mostly larger than its differences to other runs with the same sensor count. Second, for the imperative implementation, the runtime seems to grow linearly with the amount of motors, but the growth rate diminishes for larger counts. This indicates that in this version of the service, the effort of creating many small windows balances the additional work required for analyzing larger ones.

[1] `https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html` (visited on 2024-05-08)

Looking at the declarative implementation, one can see that for lower amounts of sensors ($\theta \leq 2$ in System A, $\theta \leq 4$ in System B), the total processing time decreases the larger the relevant windows are, while with larger systems, the runtime is less affected by this. Further examining this reveals that the larger the window size ω is, the higher the growth with increasing system size θ. This indicates that for smaller windows, a relatively longer time is spent on the overhead of creating the processing tasks, while with larger windows the greater amount of time is spent on actually calculating the relevant statistical properties of the sensor readings in each window.

Memory Usage Examining the memory usage, one can again notice that the imperative implementation does not seem to be influenced by the number of sensors or window size significantly. The declarative implementation on the other hand requires increasing memory with a higher number of sensors, and again seems to be impacted contrarily by larger window sizes. For systems with one, two, and four motor monitoring units, the maximum amount of memory is consumed at window sizes of eight and sixteen milliseconds for System A, and for System B at two, four, and eight milliseconds. For the larger systems ($\theta \geq 8$), the effect leading to this seems to be displaced by another factor, which causes the required memory to increase with longer window sizes. This might also be due to the factors mentioned above.

Latency When looking at the alert delays, one can notice that the two implementations react completely differently to an increase in the number of sensors or the size of the windows. While the imperative implementation generally keeps the same latency, unaffected by changes in the parameters, the declarative implementation is much more exposed to them, as the latency increases about linearly with the number of motor monitoring units θ, and with the sizes of the windows ω.

Load average Lastly, looking at the load average, verdicts cannot be made as easily due the large variance in the results. Nevertheless, it can be noted that it seems to increase linearly with the number of motor monitoring units in the system θ for both implementations. While the window size ω does not seem to influence the load of the imperative implementation, it might have two counteracting effects on the declarative implementation, as shorter windows correspond to a higher load with fewer sensors, but to a lower load in benchmarks with more sensors. This might be due to the ratio of tasks used for processing incoming messages to tasks used for processing the windows influenced by this parameter.

Summary Summing up, it can be said that the two implementations are influenced differently by variations in the system size θ and the window size ω. The measured performance indicators largely grow steadily with the system size both in the imperative implementation and in the declarative implementation. The window size on the other hand seems to have counteracting influences in the declarative implementation, which causes the performance to sometimes increase, and sometimes decrease with its growth.

The imperative implementation apparently is less affected by variations in this parameter, as it does not influence the taken measurements significantly in most cases.

6.2 Limitations

To clarify the area of validity of this thesis, this section is concerned with stating the limitations of the findings, meaning which aspects influencing the results could not be controlled for due to the chosen methodology and scope of this thesis. Generally, the limitations can be split into three areas, one being the constraints arising from the design of the research scenario, one being the limitations introduced by the decisions made during implementing the benchmarking suite and the data processing service, and one being the restrictions stemming from the employed hardware.

6.2.1 Design constraints

Starting at the top, the outer border of the area of validity is defined by the design of the benchmark scenario and system.

One-way communication As one of the core topics of this thesis are systems processing incoming sensor data, only one-way communication has been regarded. No message sending from the cloud server to the motor monitor or from the motor monitor to sensors is implemented, as the use case did not require it. Two-way communication increases the complexity of the deployed systems, and thus the results of the evaluation cannot be transferred directly to such scenarios. Implementers looking for input in their decision-making process should evaluate further literature covering this topic, or extend the benchmark given in this thesis and evaluate the paradigms anew. If the two communication streams are not interrelated (for example sensor results going in one direction, and new configuration going into the other direction), the results of this thesis apply to some degree, as the implementation can be realized by two separate services (possibly on the same hardware), each doing one-way communication as evaluated in this work.

Message size Looking at the data sent by the sensors, the transmitted messages have a size of 136 bits (reading of 32 bits, sensor ID of 32 bits, timestamp of 64 bits, encoding information of 8 bits). While it can be assumed that the results of the evaluation are valid for scenarios with sensor messages of similar size, the same is not true for systems where the data sent by the sensors is much larger than that, which would for example be the case if the information is transmitted in the form of multimedia. Such differences in data size could influence the results of the benchmark significantly, as a larger portion of the total expended resources would be spent on receiving and decoding the messages, a task which is done the same in both services.

Message frequency Another property of the message exchange limiting the validity of the results is the message frequency. For benchmarking the two implementations of this thesis, a message frequency of one per millisecond was chosen. This was done because preliminary testing revealed it to be a fitting size for utilizing the processor without overstraining it, thus being a good base for further evaluation. Consequently, when looking to implement a sensor data processing service with a large difference in message frequency, the relevant parts of the benchmarking system would need to be updated and evaluated accordingly to be able to make an informed decision about which processing model to follow.

Message homogeneity As all sensors were conceived to be of equal makeup, the incoming data is generally homogenous, meaning that, additionally to the same arrival frequency, the messages were practically equal in size and what data they contained (one integer and two floating point numbers). While the implementations would be able to handle more heterogeneous interactions, it is not clear whether the results of the conducted evaluation also apply to such scenarios. If a data processing service needs to be designed for such a problem, the benchmark suite should be adapted and executed anew.

Processing procedure Similar to the homogenous data packets, it also needs to be remarked that in the scenario evaluated in this thesis all messages follow the same processing path. Introducing additional algorithms for processing only a part of the data, like taking the average of some readings and the median of some other, or combining data from two sensors in one way and from the other two in another way would increase the complexity of the processing model. It is not clear what effect this would have on the performance differences between the declarative and imperative implementation. To be able to make an informed judgement when comparing the two paradigms in such a setting, the scenario would need to be extended, the implementations updated, and the benchmark run again.

Summary Summing up, it can be said that the results of the benchmark should be regarded as highly dependent on the scenario design, until the contrary has been proven. A large part of this is due to the creation, transmission, and processing of the messages, which form a core part of the service evaluation system.

6.2.2 Software constraints

Following the design of the benchmarking system, multiple choices had to be made during implementation which also limit the application of the conclusions of this thesis.

Programming language First, a limitation is of course given by the choice of programming language. Although one purpose of programming paradigms is offering design patterns which can be realized in many programming languages (some interdependence

exists between paradigms and languages), for the declarative and imperative paradigm this transferability is only given on the level of code writing, and not for resource consumption. Reasons for this are, among others, very differing choices in data structure design, memory handling, and multitasking. Due to this, it cannot be assumed that the results of the evaluation of the data processing services implemented in Rust also translate to the same implementations written in, for example, Python. If practitioners need to design a system for a similar context as in this thesis, with a programming language already selected that is not Rust, it is necessary to execute the benchmark again with motor monitoring prototypes implemented in the predetermined programming language. An exception has to be made for programming languages which are, like Rust, using LLVM-IR as a compilation target. Due to this similarity, it can be expected that the results of this thesis are applicable, to a limited degree, to such languages.

Inter-process communication One specific limitation, partly arising from the choice of programming language, is how communication between separate threads is handled. Put very simply, two tasks can either communicate by sending messages to each other (called "message-passing"), or agreeing on a memory region both can read and modify, each interpreting and rewriting the content as they see fit (called "shared memory"). In Rust, the recommended way to handle communication between threads is via `std::sync::mpsc::channel`, which is a message-passing construct[2,3]. Consequently, both implementations evaluated in this work are using a "message-passing" approach. This choice might influence the outcome of the evaluation of the two services significantly, as the dedicated implementation has a much larger inter-task communication volume than the imperative one. If implementers need to rely on "shared memory" communication for their service, it would be necessary to adapt the two services accordingly, execute the benchmark suite, and evaluate the results anew.

Threading model Another implementation choice, which possibly influences the performance of the two services significantly, is the choice of thread pool to use for scheduling the tasks for execution. As detailed in Section 4.3.2, there are multiple libraries offering this functionality for Rust. The differences between those libraries are not only of ergonomic nature (how they are used), but also in how they handle multithreading, synchronization, and related issues. This is due the various facilities operating systems offer for handling those tasks, each with its distinct advantages and disadvantages. Libraries are available to hand off tasks to separately spawned processes or to threads, and further solutions to decrease the required overhead exist. Additionally, thread pools processing can be implemented in a blocking or in a non-blocking way.[4]

[2] `https://doc.rust-lang.org/book/ch16-02-message-passing.html` (visited on 2024-05-08)

[3] Looking at `std::sync::mpsc::channel` on a very low level, it is again a "shared memory" construct, which is inevitable on any machine resembling a "Von Neuman architecture". This includes practically every consumer device with computing capabilities currently available to consumers.

[4] Non-blocking processing means that if a task is running on a processor, but needs to wait for external input, the thread pool recognizes this and schedules another task to avoid wasting processor time.

Lastly, the efficiency of the creation, processor submission, and deletion of the tasks influences the performance of the implementations, especially if a lot of tasks are submitted to the pool. Consequently, the choice of what thread pool to use in the implemented service has a possibly significant effect on the services runtime. As the declarative service creates a lot more tasks to schedule, the selected thread pool influences the results of the evaluation. Due to this, persons looking to implement a similar service to the one staged in this thesis should evaluate which thread pool is best suited for their use case, and update and reevaluate the benchmark suite as necessary.

Tool versions A factor out of the decisional power of developers trying to implement a similar service is the passing of time, and the updates to the programming languages and libraries coming with it. As changes in one or the other can increase (or decrease) the efficiency of a selected approach, the evaluation should be run anew whenever its results are required.

Implementation correctness Finally, the results of this benchmark are of course heavily dependent on the correctness of the implementations. Due to this, decision-makers consulting this thesis for evaluating the performance of the declarative and the imperative programming paradigms in a resource constrained environment should personally verify the implemented services, so that they can be sure of the validity of the results.

Summary As can be seen, the choice of the programming language and the tools to implement the data processing services can have a large influence on the results of this thesis. Due to this, the insights should not be transferred to other programming languages or different implementation approaches without proper validation.

6.2.3 Hardware constraints

Constraints imposed by the hardware deployed for running the services and the evaluation are the third factor limiting the validity of the results.

Single-board computer choice It is important to note that the measurements of the resource consumption of the two implementations was only done on two similar single-board computer of the "Raspberry Pi B" model. While it can generally be expected that the results for similar hardware would not be too different, especially the number of cores of the CPU could have a large impact on the alert latency of the implementations, as a higher number of cores means that more threads can be processed in parallel.

CPU architecture Furthermore, the Raspberry Pi B line has a CPU following the AArch64 chip design. As the chip architecture could influence how long it takes to create a thread and switch from one to another, it is possible that the resource consumption of the two services changes significantly if executed on a CPU of a different architecture. If a data processing service should be developed which is intended to run on a platform of different

hardware makeup than the one employed in this thesis, the benchmarking suite should be executed again, using the intended platform for running the two implementations.

Testing architecture Another, possibly significantly, influencing factor is the execution of the benchmarking suite on a single machine, meaning that the benchmark executor, the test driver, and the sensors are all running on one CPU, and all communication is done via one network link. While the one-millisecond delay between each message sending round should be enough time so that all sensors can open the measurement file, read a value from it, and send the value to the monitor, this setup is not an exact replica of a real-world setting, which would consist of many sensors, each on separate hardware. Due to this, developers which are implementing a system which consists of a large amount of separate sensors should first run the benchmark in the actual setting the service would be deployed in.

Summary As the benchmark was only executed on a certain type of hardware, it should be repeated in the correct environment whenever insights need to be gained about the performance of the services on different devices.

6.3 Relation to Previous Work

The findings of this thesis align with the conclusions regarding speed and memory consumption drawn in „A Comparative Study of Programming Languages in Rosetta Code“ by Nanz and Furia, which compared many small problem-solving snippets of different programming languages. They also find that machine-compiled imperative languages[5] have a better runtime performance and use less memory.

„Comparative Analysis of Functional and Object-Oriented Programming“ by Alic, Omanovic, and Giedrimas compares the performance of multiple programming languages on three more complicated problems. Contrasting the outcomes of this thesis to the results they obtained, the conclusion regarding runtime performance is generally the same, as the fastest imperative implementations consistently outperform the fastest declarative implementations. Regarding memory consumption, the findings of Alic, Omanovic, and Giedrimas differ to the results of this work, as they report that the fastest imperative implementations consume twice as much memory on average as the fastest declarative implementations. Their explanation is that the immutable data structures used in declarative languages are more space efficient than the mutable data structures employed by imperative languages. The findings made in this work do not support this assumption, so the differences reported by them might be more influenced by their choice of languages[6] than their general assumption about space efficiency.

The slower runtime performance of the declarative implementation also confirms the measurements of *Analyse von Lambda-Ausdrücken in Java* by Fraller, which analyzes

[5] designated “procedural languages” in their work

[6] in their work, the evaluated imperative languages are all virtual-machine based languages

lambda expressions in the Java programming language, and „A Performance Comparison of Clojure and Java“ by Krantz. Both works report that the speed of their evaluated declarative implementations was worse than the speed of the imperative implementations.

As theorized earlier, the services implemented for this work operate on a lower level of abstraction than stream processing engines (SPEs), such as the one presented in „SPADE“ by Gedik, Andrade, Wu, *et al.* or Apache Storm[7], so a comparison on performance related measurements is not really possible. Nevertheless, the implementations can serve as small-scale references for examining how a stream processing service implemented in a chosen paradigm could be built up, and what features a SPE of the corresponding choice should provide.

Finally, this thesis contributes towards closing the research gap identified in „Distributed Data Stream Processing and Edge Computing“ by de Assuncao, Veith, and Buyya regarding the missing academic work in the field of edge stream processing.

6.4 Relation to the Research Questions

Tying the results back to the start of the thesis, they can now be used to answer the initially posed research questions, and compare them to the stated hypotheses.

RQ1: How does the utilization of hardware resources of a stream processing service implemented following a declarative paradigm differ from an implementation following an imperative approach?

The imperative stream processing service generally uses less resources than the declarative service. This is not just because it can be fitted more closely to the problem at hand, but also because it uses less resources during operation. The lower efficiency of the declarative implementation seems to result from the additional costs accompanying the higher level of abstraction, more specifically the larger amount of employed tasks and communication between them.

RQ2: How does the processing latency of a stream processing service implemented following a declarative paradigm differ from an implementation following an imperative approach?

The imperative implementation both has a lower latency than the declarative one, and it manages to sustain it under higher system load. Contrary to what was initially assumed, the higher level of parallelism possible in the declarative implementation did not allow it to scale better than the imperative one. A suspected reason for this is the longer chain of separate tasks necessary for generating an alert from the received data, which is more exposed to scheduling decisions by the kernel.

[7] `https://storm.apache.org/` (visited on 2024-05-08)

RQ3: What do the differences in resource utilization and message latency mean for practitioners aiming to implement data stream processing on the edge?

As in the current setting the imperative implementation consistently outperforms the declarative one, it is recommended to follow the imperative paradigm when designing a service in a similar environment. Naturally, this consequence is limited by the design and execution of the benchmark created in this thesis. The employed setting consists of up to 64 individual sensors sending data every millisecond to a monitor combining those messages in windows of up to 4096 milliseconds in size. This should be considered when evaluating this work for guidance on designing an IoT data stream processing service. To provide more insights concerning the applicability of the results of this thesis, see Section 6.2.

6.5 Summary

Summing up there are multiple factors, both on a lower and a higher level of the designed systems, leading to a generally worse performance of the declarative implementation when compared to the imperative implementation over the developed benchmark. Drawing from those findings, it can be said that imperative approaches should be preferred when aiming for lower resource consumption in a context similar to the one covered in this thesis.

This largely aligns with previous research done in related topics comparing the two paradigms, which note similar performance differences in their fields of examination. Related to this, this thesis narrows the existing research gap regarding the suitability of programming paradigms in resource constrained environments.

Nevertheless, there are many limitations which should be regarded when trying to apply the findings of this thesis to other areas. Those are based on limitations of the work stemming from the design of the benchmark, the implementation of the services, and the hardware the monitor is running on.

CHAPTER 7

Conclusion

To conclude the thesis, this chapter gives a summary of the contributions made over its course and the insights gained from conducting the evaluation of the imperative and declarative data processing service. Furthermore, future possible research areas are proposed which build on the results to further explore the space of efficient IoT stream analysis.

7.1 Contributions & Results

This thesis examined the performance implications of following the declarative versus the imperative programming paradigm when creating a service for processing unbounded IoT data streams on resource-constrained devices.

To ensure repeatable comparison, it was decided to perform this comparison with the help of a benchmark suite. A preliminary literature review revealed that there existed neither an off-the-shelf benchmark suite appropriate for the problem at the center of this thesis, nor a framework for creating benchmarks tailored to a specific use-case.

Due to this, the first step towards answering the research question was to create such a framework, consisting of guidelines to create a benchmark scenario, and to define core components and input data.

In the next step, this framework was used for creating a fitting benchmark, which was subsequently implemented. The benchmark describes an IoT scenario consisting of multiple sensors sending motor measurements to a central processing device, which scans the received data for anomalies, and forwards an alert if such an anomaly is found. The measurements to base the comparison on are defined as the total runtime, consumed memory, average number of processes waiting for execution, and the total delay between the emission of a message triggering an alert and its arrival at the final destination. Two parameters influencing the benchmark layout were selected, allowing to observe the

evaluated services under different conditions: One is the number of messages which are considered relevant for alert detection, the other the number of motors which should be observed.

During execution of the benchmark, the performance of the central processing service is documented, so that it can be evaluated afterwards. The suite implemented in this thesis supplies all necessary components, data, and orchestration for its execution. In particular this means that motor sensors sending data to the central processing devices have been implemented, data has been sourced from a paper about simulating motor observation, and scripts to handle single benchmark runs and full execution of the benchmark suite have been created.

Once the benchmark suite was in place, two services were implemented to fulfill the tasks of the central processing device, one in an imperative, one in a declarative manner. As the declarative implementation required an abstraction for handling data infrastructural tasks, the Rust package ecosystem was searched for a suitable library. As none has been found, it was decided to create a basic version of such a service, and make is accessible to the Rust community[1].

With the two services implemented, the benchmark suite could be used to compare their performance. To have a reliable base of data, the evaluation was conducted on two similar testing systems, and each parameter configuration was executed 50 times per machine.

Visualizing the collected data, it was revealed that the imperative implementation performed better than the declarative implementation in most of the evaluated parameter configurations. To confirm the visual analysis, t-tests were executed, which generally supported the previous insights.

Potential factors leading to those results have been made out, with the possible leading cause being the closer alignment of the imperative implementation to the problem at hand. Stemming from this, fewer tasks and messages are necessary for processing and coordination, thus leading to improved performance.

To be able to verify the results of this thesis, test the performance of the services in different environments, or modify the benchmark for other applications, the source code of the implemented components is available on GitHub[2].

7.2 Future Work

The limitations given in Section 6.2 constitute a possible area where future work can contribute to the extension of the body of knowledge. It would definitely be valuable to evaluate the two programming paradigms in other environments, so that further insights

[1] `https://github.com/AntonOellerer/rx_rust_mp` (visited on 2024-05-08)

[2] `https://github.com/AntonOellerer/Reactive-Streaming-on-the-Edge` (visited on 2024-05-08)

into their effects can be made. Aside from this, there are a number of other possible areas to explore.

While this thesis is concerned with the efficient processing of IoT data streams, close to the sensors and on resource-constrained hardware, it would also be valuable to evaluate the implemented services in a setting resembling a cloud processing architecture. This would mean that they would be executed on performant hardware with a larger number of cores, more RAM, and larger caches. Additionally, the sensor data would be sent over the internet, changing the ratio of processing time to transmission time possibly significantly. Following this, it would then be interesting to compare the edge-processing with the cloud-processing service on different metrics, to gain insights on the advantages and disadvantages of the two architectural paradigms, and how they are influenced by the evaluated programming paradigms.

While the measurements in this thesis were all taken automatically, partly from system information files, partly from measuring message delay, it would be valuable to conflate them with other measurements, extending the window of evaluation beyond just the performance criterion. One possible examination could focus on certain properties of the code, such as readability and speed of implementation to allow including those factors when deciding on a programming paradigm to follow.

Another possible measurement would be the power consumption of the data processors during service execution. This would complement the existing insights, verifying whether higher CPU-load corresponds to higher energy usage, and survey the viability of battery-powered devices for IoT data stream processing. Especially the last point is important for future architectures incorporating Edge Computing, as it extends its applicability even further.

Continuing the examination of the resource efficiency into the other direction, one could also compare how different hardware compositions influence the performance of the IoT data processing services. In the area of single-board computers, there are many boards of varying costs and capabilities, bringing another dimension to economic considerations.

An additional possible application scenario for the benchmark system is that of automated performance testing. It could be valuable to see how the system can be employed in software engineering environments to evaluate different versions of one program, making sure it adheres to defined performance criteria.

Lastly, another interesting aspect would be to compare two less abstract programming paradigms from the ones evaluated here, to explore how large the internal performance disparity can be. This could also provide further insights into the significance of different factors influencing the performance evaluation as listed in Section 6.1. Two examples of this would be functional versus logic-oriented programming for the declarative paradigm, and object-oriented versus procedural programming for the imperative paradigm.

List of Figures

List of Tables

Acronyms

CSV comma-separated values 22, 31–33

EEMBC Embedded Microprocessor Benchmark Consortium 18

I/O Input/Output 18, 25, 27, 50, 59, 68

IoT Internet of Things 2–7, 9–11, 13, 15, 17–20, 31, 37, 78, 79, 81

SPE stream processing engine 14–17, 77

TPC Transaction Processing Performance Council 17

YCSB Yahoo! Cloud Serving Benchmark framework 18

Glossary

COBS (Consistent Overhead Byte Stuffing) is an algorithm for encoding data so that it can be transmitted reliably. 30, 31, 39

CPU The central processing unit (CPU) is the main processor of any computer, most of the times responsible for executing the procedures passed to it. 4, 25, 26, 30, 39, 51, 68, 69, 75, 76, 81, 89, 90

Data Intensity The aggregation of data volume and the frequency of arrival 11

DevOps DevOps, a convocation of Software *Dev*elpment and IT *Op*erations, is a branch of computer engineering concerned with deploying new software versions to devices automatically, as opposed to installing them manually. 2

JVM The Java virtual machine (JVM) is a program which can execute Java bytecode. Due to this, programs compiled to Java bytecode can be run on any processor which can execute a Java virtual machine. 14

kernel The program which is, among other things, responsible for allocating tasks to the CPU and interrupting them after some time, so that a computer can execute multiple tasks in parallel. 68, 69, 77

kernel mode One of the two modes in which a process can be executed by the linux kernel (user mode being the other one). Applications running in this mode have unrestricted access to system resources. 30, 39

LLVM-IR A low-level programming language which can be used as compilation target for higher-level programming languages. The LLVM intermediate representation (LLVM-IR) is then in turn compiled to machine code. This enables new programming languages to profit off established optimizations and error checking mechanisms. 74

Object-Oriented Programming A programming technique which decomposes problems into the objects making them up and the relationships between those objects. 11

RAM (random access memory) is a type of memory which is used for fast data access by the CPU. 25, 30, 39, 52, 81, 90

Resident Set The sum of the anonymous memory, file memory, and shared memory in the RAM 30, 56

single-board computer A type of computer where all necessary components like power supply, CPU, and RAM are located on the same circuit board. As they are often running a linux-based operating system, they can be used in situations where more features are needed than a microprocessor can provide, but a small form-factor and low energy consumption are still required. 20, 22, 25, 75, 81

TCP (Transmission Control Protocol) is one of the major protocols used for transmitting data over the internet. Its main properties include reliable, ordered and error-checked delivery of the sent data packages. 29, 32, 40, 44, 46

Unix epoch Jargon for first of January 1970, 00:00:00, Coordinated Universal Time. IT systems can communicate about time by reporting how many seconds (or milliseconds) have passed since this point in time. 29, 31

user mode One of the two modes in which a process can be executed by the linux kernel (user mode being the other one). This is the default, and does not allow access to any hardware devices or memory of other processes. To achieve such tasks, the program needs assistance by services running in kernel mode, such as the kernel itself or device drivers. 30, 39, 89, 90

Bibliography

[1] Y. B. Zikria, R. Ali, M. K. Afzal, and S. W. Kim, „Next-Generation Internet of Things (IoT): Opportunities, Challenges, and Solutions“, *Sensors*, vol. 21, no. 4, p. 1174, Feb. 7, 2021, ISSN: 1424-8220. DOI: 10.3390/s21041174. [Online]. Available: https://www.mdpi.com/1424-8220/21/4/1174 (visited on 02/24/2022).

[2] D. Evans, „How the Next Evolution of the Internet Is Changing Everything“, *CISCO white paper*, vol. 1, pp. 1–11, 2011.

[3] Nicolas de Loisy, *Transportation and the Belt and Road Initiative*, 2nd ed., 1 vols. Supply Chain Management Outsource Limited, Feb. 10, 2020, vol. 1, 286 pp., ISBN: 978-988-799-122-9.

[4] Malte Scheuvens, „Datengetriebene Prädiktive Instandhaltungsmethoden“, Institut für Managementwissenschaften, TU Wien, Vienna, Austria, Project Report, Feb. 18, 2019.

[5] L. Peng, A. R. Dhaini, and P.-H. Ho, „Toward integrated Cloud–Fog networks for efficient IoT provisioning: Key challenges and solutions“, *Future Generation Computer Systems*, vol. 88, pp. 606–613, Nov. 2018, ISSN: 0167739X. DOI: 10/gd8f6c. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X1830596X (visited on 01/04/2022).

[6] Y.-K. Chen, „Challenges and opportunities of internet of things“, in *17th Asia and South Pacific Design Automation Conference*, Sydney, Australia: IEEE, Jan. 2012, pp. 383–388, ISBN: 978-1-4673-0772-7. DOI: 10.1109/ASPDAC.2012.6164978. [Online]. Available: http://ieeexplore.ieee.org/document/6164978/ (visited on 02/25/2022).

[7] P. L. Van Roy, „Can logic programming execute as fast as imperative programming?“, Ph.D. dissertation, EECS Department, University of California, Berkeley, Nov. 1990. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1990/5686.html.

[8] J. W. Lloyd, „Practical Advantages of Declarative Programming.“, in *GULP-PRODE (1)*, 1994, pp. 18–30.

[9] Q.-C. To, J. Soto, and V. Markl, „A survey of state management in big data processing systems“, *The VLDB Journal*, vol. 27, no. 6, pp. 847–872, Dec. 1, 2018, ISSN: 0949-877X. DOI: 10.1007/s00778-018-0514-9. [Online]. Available: https://doi.org/10.1007/s00778-018-0514-9.

[10] R. W. Sebesta, *Concepts of Programming Languages*, 12. ed. University of Colorado, Colorado Springs: Pearson, 2018, ISBN: 978-0-13-510226-8.

[11] P. van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. Cambridge, Mass. [u.a.]: MIT Press, 2004, ISBN: 0-262-22069-5.

[12] D. J. King and J. Launchbury, „Lazy depth-first search and linear graph algorithms in Haskell“, in *Glasgow Workshop on Functional Programming*, 1994, pp. 145–155.

[13] Y. Ge, X. Liang, Y. C. Zhou, Z. Pan, G. T. Zhao, and Y. L. Zheng, „Adaptive Analytic Service for Real-Time Internet of Things Applications“, in *2016 IEEE International Conference on Web Services (ICWS)*, San Francisco, CA, USA: IEEE, Jun. 2016, pp. 484–491, ISBN: 978-1-5090-2675-3. DOI: 10/gn7dvg. [Online]. Available: http://ieeexplore.ieee.org/document/7558038/ (visited on 01/04/2022).

[14] M. Babar and F. Arif, „Smart urban planning using Big Data analytics to contend with the interoperability in Internet of Things“, *Future Generation Computer Systems*, vol. 77, pp. 65–76, Dec. 2017, ISSN: 0167739X. DOI: 10/gfsqf9. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X17308993 (visited on 01/04/2022).

[15] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, „Internet of Things (IoT): A vision, architectural elements, and future directions“, *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013, ISSN: 0167739X. DOI: 10/f427k4. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X13000241 (visited on 01/12/2022).

[16] Kevin Ashton. „That ‘Internet of Things’ Thing“, RFID Journal. (Jun. 22, 2009), [Online]. Available: https://www.rfidjournal.com/that-internet-of-things-thing (visited on 03/16/2022).

[17] The Carnegie Mellon University Computer Science Department Coke Machine, „The "Only" Coke Machine on the Internet“, Carnegie Mellon University Computer Science Department, Pittsburgh, Pennsylvania, Experience Report, Jun. 15, 1998. [Online]. Available: https://www.cs.cmu.edu/~coke/history_long.txt (visited on 03/16/2022).

[18] J. Kelly and W. Knottenbelt, „The UK-DALE dataset, domestic appliance-level electricity demand and whole-house demand from five UK homes“, *Scientific Data*, vol. 2, no. 1, p. 150 007, Dec. 2015, ISSN: 2052-4463. DOI: 10/ggbsxw. [Online]. Available: http://www.nature.com/articles/sdata20157 (visited on 01/19/2022).

[19] E. Wu, Y. Diao, and S. Rizvi, „High-performance complex event processing over streams“, in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data - SIGMOD '06*, Chicago, IL, USA: ACM Press, 2006, p. 407, ISBN: 978-1-59593-434-5. DOI: 10/d7xzdm. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1142473.1142520 (visited on 01/05/2022).

[20] B. Nikolic, J. Ignjatic, N. Suzic, B. Stevanov, and A. Rikalovic, „Predictive Manufacturing Systems in Industry 4.0: Trends, Benefits and Challenges“, in *DAAAM Proceedings*, B. Katalinic, Ed., 1st ed., vol. 1, DAAAM International Vienna, 2017, pp. 0796–0802, ISBN: 978-3-902734-11-2. DOI: 10.2507/28th.daaam.proceedings.112. [Online]. Available: http://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2017/112.pdf (visited on 03/16/2022).

[21] L. Sanchez, L. Muñoz, J. A. Galache, *et al.*, „SmartSantander: IoT experimentation over a smart city testbed“, *Computer Networks*, vol. 61, pp. 217–238, Mar. 2014, ISSN: 13891286. DOI: 10.1016/j.bjp.2013.12.020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128613004337 (visited on 03/18/2022).

[22] T. Zschörnig, R. Wehlitz, and B. Franczyk, „IoT Analytics Architectures: Challenges, Solution Proposals and Future Research Directions“, in *Research Challenges in Information Science*, F. Dalpiaz, J. Zdravkovic, and P. Loucopoulos, Eds., vol. 385, Cham: Springer International Publishing, 2020, pp. 76–92, ISBN: 978-3-030-50316-1. DOI: 10.1007/978-3-030-50316-1_5. [Online]. Available: http://link.springer.com/10.1007/978-3-030-50316-1_5 (visited on 01/27/2022).

[23] John R Mashey, „Big Data and the Next Wave of InfraStress“, Apr. 25, 1998. [Online]. Available: https://static.usenix.org/event/usenix99/invited_talks/mashey.pdf (visited on 03/16/2022).

[24] Steve Lohr. „The Origins of 'Big Data': An Etymological Detective Story“, Bits Blog. (Feb. 1, 2013), [Online]. Available: https://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/ (visited on 03/16/2022).

[25] C. Ji, Y. Li, W. Qiu, *et al.*, „Big data processing: Big challenges and opportunities“, *Journal of Interconnection Networks*, vol. 13, p. 1 250 009, 03n04 Sep. 2012, ISSN: 0219-2659, 1793-6713. DOI: 10.1142/S0219265912500090. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0219265912500090 (visited on 03/16/2022).

[26] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, „A Survey of Distributed Data Stream Processing Frameworks“, *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2946884.

[27] K. Yasumoto, H. Yamaguchi, and H. Shigeno, „Survey of Real-time Processing Technologies of IoT Data Streams“, *Journal of Information Processing*, vol. 24, no. 2, pp. 195–202, 2016, ISSN: 1882-6652. DOI: 10/gd4trh. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjjip/24/2/24_195/_article (visited on 01/04/2022).

[28] E. Mehmood and T. Anees, „Challenges and Solutions for Processing Real-Time Big Data Stream: A Systematic Literature Review“, *IEEE Access*, vol. 8, pp. 119 123–119 143, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3005268.

[29] M. H. Iqbal and T. R. Soomro, „Big Data Analysis: Apache Storm Perspective“, *International Journal of Computer Trends and Technology*, vol. 19, no. 1, p. 6, 2015.

[30] D. J. Abadi, D. Carney, U. Cetintemel, *et al.*, „Aurora: A new model and architecture for data stream management“, *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, Aug. 1, 2003, ISSN: 1066-8888, 0949-877X. DOI: 10/dw3hzt. [Online]. Available: http://link.springer.com/10.1007/s00778-003-0095-z (visited on 01/13/2022).

[31] D. J. Abadi, Y. Ahmad, M. Balazinska, *et al.*, „The Design of the Borealis Stream Processing Engine“, in *Proceedings of the 2005 CIDR Conference*, Asilomar, CA, Jan. 2005, pp. 277–289.

[32] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, „Apache Flink™: Stream and Batch Processing in a Single Engine“, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, p. 12, 2015.

[33] T. Francis and D. M. Madhiajagan, „A Comparison of Cloud Execution Mechanisms: Fog, Edge and Clone Cloud Computing“, in *Proceeding of the 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI 2017)*, vol. 4, Yogyakarta, India, 2017, pp. 446–450. DOI: 10/gn938x.

[34] E. Marín-Tordera, X. Masip-Bruin, J. García-Almiñana, A. Jukan, G.-J. Ren, and J. Zhu, „Do we all really know what a fog node is? Current trends towards an open definition“, *Computer Communications*, vol. 109, pp. 117–130, Sep. 2017, ISSN: 01403664. DOI: 10.1016/j.comcom.2017.05.013. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0140366416307113 (visited on 03/30/2022).

[35] J. Gedeon, F. Brandherm, R. Egert, T. Grube, and M. Mühlhäuser, „What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges“, *IEEE Access*, vol. 7, pp. 152 847–152 878, 2019, ISSN: 2169-3536. DOI: 10/gn938w.

[36] S. Nanz and C. A. Furia, „A comparative study of programming languages in rosetta code“, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 778–788. DOI: 10.1109/ICSE.2015.90.

[37] D. Alic, S. Omanovic, and V. Giedrimas, „Comparative analysis of functional and object-oriented programming“, in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2016, pp. 667–672. DOI: 10.1109/MIPRO.2016.7522224.

[38] B. Fraller, *Analyse von Lambda-Ausdrücken in Java*. Wien, 2018.

[39] G. Krantz, „A performance comparison of Clojure and Java“, M.S. thesis, KTH, School of Electrical Engineering and Computer Science (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2020, p. 58.

[40] N. Mehlhorn and S. Hanenberg, „Imperative versus declarative collection processing: An RCT on the understandability of traditional loops versus the stream API in java“, in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1157–1168. DOI: 10.1145/3510003.3519016.

[41] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, „SPADE: The system s declarative stream processing engine“, in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08*, Vancouver, Canada: ACM Press, 2008, p. 1123, ISBN: 978-1-60558-102-6. DOI: 10/br2whb. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1376616.1376729 (visited on 01/19/2022).

[42] D. Bonino and F. Corno, „spChains: A declarative framework for data stream processing in pervasive applications“, *Procedia Computer Science*, vol. 10, pp. 316–323, 2012, ISSN: 1877-0509. DOI: 10.1016/j.procs.2012.06.042. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050912003997.

[43] H. Röger and R. Mayer, „A comprehensive survey on parallelization and elasticity in stream processing“, *Acm Computing Surveys*, vol. 52, no. 2, Apr. 2019, ISSN: 0360-0300. DOI: 10.1145/3303849. [Online]. Available: https://doi.org/10.1145/3303849.

[44] A. Toshniwal, S. Taneja, A. Shukla, *et al.*, „Storm@twitter“, in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, New York, NY, USA: Association for Computing Machinery, 2014, pp. 147–156, ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641. [Online]. Available: https://doi.org/10.1145/2588555.2595641.

[45] M. D. de Assuncao, A. d. S. Veith, and R. Buyya, „Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions“, *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, Feb. 1, 2018, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.12.001. arXiv: 1709.01363. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804517303971 (visited on 03/24/2022).

[46] A. Arasu, M. Cherniack, E. Galvez, *et al.*, „Linear Road: A Stream Data Management Benchmark“, in *Proceedings of the 30th International Conference on Very Large Data Bases*, Aug. 2004, pp. 480–491.

[47] A. Shukla and Y. Simmhan, „Benchmarking distributed stream processing platforms for IoT applications“, in *Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things*, R. Nambiar and M. Poess, Eds., Cham: Springer International Publishing, 2017, pp. 90–106, ISBN: 978-3-319-54334-5.

[48] M. Poess, R. Nambiar, K. Kulkarni, C. Narasimhadevara, T. Rabl, and H.-A. Jacobsen, „Analysis of TPCx-IoT: The first industry standard benchmark for IoT gateway systems“, in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1519–1530. DOI: 10.1109/ICDE.2018.00170.

[49] Y. Zhu, Y. An, Y. Zi, Y. Feng, and J. Wang, „IoTDataBench: Extending TPCx-IoT for compression and scalability“, in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds., Cham: Springer International Publishing, 2022, pp. 17–32, ISBN: 978-3-030-94437-7.

[50] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, „Benchmarking cloud serving systems with YCSB“, in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, New York, NY, USA: Association for Computing Machinery, 2010, pp. 143–154, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. [Online]. Available: https://doi.org/10.1145/1807128.1807152.

[51] S. Matzka, „Explainable Artificial Intelligence for Predictive Maintenance Applications“, in *2020 Third International Conference on Artificial Intelligence for Industries (AI4I)*, Irvine, CA, USA: IEEE, Sep. 2020, pp. 69–74, ISBN: 978-1-72818-701-3. DOI: 10.1109/AI4I49448.2020.00023. [Online]. Available: https://ieeexplore.ieee.org/document/9253083/ (visited on 05/15/2022).

[52] P. Silva, A. Costan, and G. Antoniu, „Towards a methodology for benchmarking edge processing frameworks“, in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 904–907. DOI: 10.1109/IPDPSW.2019.00149.