UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Áika: A Distributed Edge System For Machine Learning Inference

Detecting and defending against abnormal behavior in untrusted edge environments

Joakim Aalstad Alslie

INF-3981: Master's Thesis in Computer Science, December 2021

This thesis document was typeset using the *UiT Thesis LaTeX Template*.
© 2021 – http://github.com/egraff/uit-thesis

“Livet er ikke bare fylt med motbakker. Det er også nedturer.”
–Alexander Torkelsen

Abstract

The edge computing paradigm has recently started to gain a lot of momentum. The field of Artificial Intelligence (AI) has also grown in recent years, and there is currently ongoing research that investigates how AI can be applied to numerous of different fields. This includes the edge computing domain. In Norway, there is currently ongoing research being conducted that investigates how the confluence between AI and edge computing can be used to hinder fish crime, by stationing surveillance equipment aboard fishing vessels, and perform all the monitoring directly on the vessel with support of AI.

This is challenging for several reasons. First and foremost, the equipment needs to be stationed on the vessel, where actors may impose a threat to it and attempt to damage it, or interfere with the analytical process. The second challenge is to enable multiple machine learning pipelines to be executed effectively on the equipment. This requires a versatile computation model, where data is handled in a privacy preserving manner.

This thesis presents Áika, a distributed edge computing system that supports machine learning inference in such untrusted edge environments. Áika is designed as a hierarchical fault tolerant system that supports a directed acyclic graph (DAG) computation model for executing machine inference on the edge, where a monitor residing in a trusted location can ensure that the system is running as expected.

The experiment results demonstrate that Áika can tolerate failures while remaining operable with a stable throughput, although this will depend on the specific configuration and what computations that are implemented. The results also demonstrate that Áika can be used for both simple tasks, like counting words in a textual document, and for more complex tasks, like performing feature extraction using pre-trained deep learning models that are distributed across different workers. With Áika, application developers can develop fault tolerant and different distributed DAGs composed of multiple pipelines.

Acknowledgements

First and foremost I want to thank my supervisor Dag Johansen and co-supervisor Michael Riegler for their guidance, and for being available whenever I needed help during this project. Their passion for the field has truly been a great and invaluable source of inspiration.

I want to thank the other members of the Cyber Security Group for their help, discussions and insight during the last few months. I want to commend the employees at the Department of Computer Science for their openness throughout the 4,5 years I have been here. The open-door policy that many of you have has truly been appreciated. Among the employees, I want to direct a special thanks to our student advisor Jan Fuglesteg for being available at almost any time. You are truly invaluable for all students at the CS department.

To all my friends, family and fellow students that have been a part of this journey, you all have my sincerest gratitude.

I want to direct a special thanks to Randi for convincing me to move to Tromsø and start studying, to Stina for helping me name the system, to Alexander for always being there motivating me, to Eirik for sticking with me since the beginning and to Isak for helping me get through the first semester.

And finally, I want to thank Ida for her love and support during this final year, and for reminding me to take breaks when I pushed myself too hard. Without you, this final semester would not have been the same.

The last few years has truly been the best years of my life. I am eternally grateful for each and every one who has been a part of this journey.

Thank you.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii

List of Listings xv

List of Acronyms xvii

1 Introduction 1
- 1.1 Problem Definition . . . 2
- 1.2 Methodology . . . 3
- 1.3 Scope, Limitation and Assumptions . . . 4
- 1.4 Context . . . 6
- 1.5 Contributions . . . 7
- 1.6 Outline . . . 8

2 Background and Related Work 9
- 2.1 Artificial Intelligence . . . 9
  - 2.1.1 History . . . 10
  - 2.1.2 AI Areas . . . 11
  - 2.1.3 Machine Learning . . . 13
  - 2.1.4 Deep Learning . . . 16
- 2.2 Edge Computing . . . 19
- 2.3 Distributed Software Architectures . . . 20
  - 2.3.1 Controller/Agent . . . 21
  - 2.3.2 Pipeline . . . 22
- 2.4 Directed Acyclic Graph . . . 23
- 2.5 Fault Tolerance . . . 24
  - 2.5.1 Dependability . . . 25
  - 2.5.2 Failure Models . . . 26

2.5.3 Redundancy and Resilience 26
2.5.4 The CAP Theorem 28
2.5.5 Detecting Failures in Distributed Systems 29
2.6 Persistent Event Queue 29
2.7 Related Work 30
2.7.1 FRAME and CESSNA 30
2.7.2 NAP and Falcon Spy Network 31
2.7.3 Dryad and Cogset 31
2.7.4 SEDA and Vortex 32
2.8 Summary 33

3 Requirement Analysis 35
3.1 Low-Bandwidth Edge Environments 36
3.2 Fault Tolerance and Security 37
3.3 Privacy-Preserving Data 38
3.4 Laws, Regulations and GDPR 39
3.5 Incentive for Fisheries 42
3.6 Requirement Specification of Áika 43
3.6.1 Functional Requirements 43
3.6.2 Non-Functional Requirements 44
3.7 Summary 45

4 Design 47
4.1 System Overview 47
4.2 System Components Structure 50
4.3 Controllers 50
4.3.1 Local Controller 51
4.3.2 Cluster Controller 51
4.4 Agents 52
4.5 Monitor 56
4.6 Summary 57

5 Implementation 59
5.1 Implementation Specific Details 59
5.2 Testing 60
5.3 Cluster Controller 61
5.4 Local Controller 61
5.5 Agents 62
5.6 Monitor 63
5.7 Logging 64
5.8 Killer 65
5.9 Summary 66

6 Evaluation 67

6.1 Overall Experimental Setup 67
6.2 Micro-Benchmarks 68
6.3 End-to-End Evaluation 69
6.4 Distributed Word Counter 74
6.5 Distributed Deep Feature Extraction 76
6.6 Review of Non-Functional Requirements 79
6.7 Discussion 80
6.8 Summary 80

7 Conclusion and Future Work **81**
7.1 Conclusion 81
7.2 Future Work 82

A Configuration Format **85**

B Local Run-time Example **87**

C Distributed Word Counter **91**

D Distributed Deep Feature Extractor **93**

List of Figures

2.1 Areas in Artificial Intelligence 12
2.2 The Perceptron Algorithm 14
2.3 Deep Neural Network 17
2.4 The Edge Layer 19
2.5 Controller/Agent Architecture. 22
2.6 Pipeline Architecture 23
2.7 Directed Acyclic Graph 24
2.8 Triple Modular Redundancy. 28
2.9 Event Queue 29

4.1 System Overview 49
4.2 General Process Structure 50
4.3 Cluster Controller Replication 52
4.4 General Agent Structure 52
4.5 Left Worker Agent 53
4.6 Right Worker Agent 54
4.7 Double Worker Agent 54
4.8 Initial Worker Agents 55
4.9 Final Worker Agents 55
4.10 Queue Agent and Server-less Worker Agent 56

5.1 Process Killer 66

6.1 End-to-End Performance With Persistent Queues 70
6.2 End-To-End Performance With In-Memory Queues. 71
6.3 End-To-End Performance With Workload 72
6.4 End-To-End Performance With And Without Killer 73
6.5 Distributed Word Counter Experiment Result 76
6.6 Distributed Feature Extraction Experiment Results 78

B.1 Directed Acyclic Graph example 87

C.1 Distributed Word Counter 91

D.1 Distributed Deep Feature Extractor 94

List of Tables

5.1 Monitor Request Table 63
5.2 Failure Log Record Table 64

6.1 Micro-Benchmark Results 68
6.2 Continuous Performance Summary 73

List of Listings

5.1 Worker Agent Implementation Example. 62

A.1 JSON Configuration Format 85

B.1 Directed Acyclic Graph Configuration Example 87

List of Acronyms

AI Artificial Intelligence

AR Augmented Reality

CCTV Closed-circuit Television

CNN Convolutional Neural Network

DAG Directed Acyclic Graph

DNN Deep Neural Network

DPIA Data Protection Impact Assessment

DSS Decision Support System

FIFO First In First Out

FUSE Filesystem in Userspace

GDPR General Data Protection Regulation

IoT Internet of Things

JSON JavaScript Object Notation

LED Law Enforcement Directive

MLP Multilayer Perceptron

NFS Network File System

NLP Natural Language Processing

OS Operating System

PCA Principal Component Analysis

POA Proof of Applicability

POC Proof of Concept

POP Proof of Performance

RDBMS Relational Database Management System

REST Representational State Transfer

SGD Stochastic Gradient Descent

SGX Software Guard Extension

SSH Secure Shell

SSL Semi-Supervised Learning

SVM Support Vector Machine

TCP Transmission Control Protocol

TEE Trusted Execution Environment

UDP User Datagram Protocol

VR Virtual Reality

Introduction

Edge computing, a distributed computation paradigm that moves computations and data storage closer to physical locations in an attempt to improve response times and save bandwidth, has gained considerable momentum recently [1]. Numerous application fields are adopting this distributed computing architecture that moves computations closer to the sources generating voluminous data [2]. Furthermore, the LF State of the Edge report of 2021 predicts cumulative capital expenditures of up to $800 billion USD will be spent on new and replacement IT server equipment and edge computing facilities between 2019 and 2028 [3].

Recent breakthroughs in the field of Artificial Intelligence (AI) have lead to a spark of new technological advancements. Currently, there is an incredible amount of ongoing research in areas that investigate how AI may be used to drive the field forward. This includes fields such as healthcare [4, 5, 6, 7], autonomous vehicles [8, 9] and business automation [10]. In attempt to decrease latency, and increase security and reliability, some of the new AI solutions are on the rise of being deployed and executed on the edge. Ion Stoica et al. [11] and Fabrizio Carcillo et al. [12], for instance discuss challenges related to the field of AI. Both edge computing and security are listed as important topics of research. The term "Edge Intelligence" is often used to describe the confluence of the edge computing and AI fields [13].

Moving AI solutions to the edge may not only be beneficial, it can also be required in environments where bandwidth is too low for effective data trans-

portation. This is relevant for systems that operate in environments where the access to high-speed internet connection is either limited, or non-existing. The systems may, for instance rely on a limited satellite connection to communicate with sources outside their environment. Some connections may be unable to transport data with an acceptable throughput without compressing it. Locations where this is a plausible assumption includes international waters, Antarctica and other remote locations and islands, such as Tristan da Cunha Island. The development of 5G networks may change this in the future, but it is highly implausible to assume that it will cover the entire globe, due to its short range [14].

The process also comes with certain challenges. This is especially the case for systems that deal with private data in untrusted environments, where untrusted sources may constantly aim to compromise the different layers of the system. It is important to build robust systems that can protect private data, tolerate failures and ensure both physical security and connectivity [2]. Even more so, an AI system that deals with private data of users also needs to satisfy the requirements of privacy-governing laws[1].

This thesis presents Áika, a fault tolerant system for executing distributed AI applications on the edge. The word "Áika" means "oak" in northern sami and was chosen as a name due to the systems hierarchical design. Áika is developed and evaluated in a scientific context as a concrete edge computing technology infrastructure with focus on AI and system problems. Of particular interest is how to provide fault tolerance while the system remains active and doing continuous analysis of data. We will thus investigate if and eventually how to provide efficient data analytics in an untrusted edge environment.

The application domain will be related to surveillance and monitoring of active fishing vessels off-shore Norway. These activities are fundamental for a sustainable management and harvesting of fish resources in the Arctic.

1.1 Problem Definition

In untrusted edge environments it is necessary to assume that the system operating in it is under constant threat due to potential actors in the environment that may try to compromise the physical hardware and/or the software of the system. It is important to have a robust and secure system that not only is able to tolerate faults, but also detect and report them. We will, in this thesis put an emphasis on fault tolerance and detection of abnormal behavior. The thesis

1. For instance, see the Norwegian constitution: § 102, first paragraph

statement is defined as:

Artificial Intelligence solutions can be executed in untrusted edge environments through a Directed Acyclic Graph computation model in a fault tolerant system that can detect, recover from and report abnormal behavior.

The statement above is investigated through the following steps:

1. Outline a set of requirements specifications based on the problem definition, related work and the problem domain.

2. Develop a proof of concept (POC) system based on the stated requirements.

3. Evaluate the resulting system through experiments and conclude to what extent the system satisfies the requirements.

1.2 Methodology

The Task Force on the Core of Computer Science presents in their final report an approach to divide the discipline of Computer Science into three distinct paradigms: *Theory*, *Abstraction*, *Design* [15].

Theory is rooted in mathematics and consist of four steps, followed in the development of a coherent and valid theory:

1. *Definition:* Characterize objects of study.

2. *Theorem:* Hypothesize possible relationships among them.

3. *Proof:* Determine whether the relationships are true.

4. Interpret the results.

Mathematicians are expected to re-iterate these steps, for example when errors or inconsistencies are discovered.

Abstraction is rooted in the experimental scientific method and consists of four steps that are followed in the investigation of a phenomenon:

1. Form a hypothesis.

2. Construct a model, then make a prediction.

3. Design and experiment, collect data.

4. Analyze the results.

A scientist is expected to re-iterate these steps, for example when a models prediction disagrees with experimental evidence.

Design is rooted in engineering and consists of four steps followed in the construction of a system to solve a given problem:

1. State requirements.

2. State specifications.

3. Design and implement the system.

4. Test the system.

An engineer is expected to re-iterate these steps, for example when tests reveal that the latest version of the system does not meet the requirements stated.

This thesis roots in both the abstraction paradigm and the design paradigm. Initially, the requirements and specifications are derived with roots in the problem definition and the application domain. A prototype of the system is designed and implemented based on the requirement specifications. Finally, the system is evaluated through experiments and an evaluation of the stated requirements.

In summary, a set of requirements is outlined from the problem definition and the application domain. The system is constructed as a POC based on the requirements, and evaluated through Proof of Performance (POP).

1.3 Scope, Limitation and Assumptions

It is necessary to make some assumptions with regards to the problem domain in order to limit the scope of the problem definition. The assumptions are documented here.

- To simplify the process of data storage and data retrieval, we assume

that the system is mounted to a distributed file system (DFS). We further assume that this is a transparent file system that abstracts away communication, storage, etc. such that any files stored in the system is retrievable by any node.

- With regards to the AI field, we narrow the scope to focus on machine learning tasks, as it seems to have the most potential for analyzing sensor data.

- Due to the low level of trust in untrusted edge environments, we avoid focusing on the training process with regards to machine learning, especially on labeled data, as model training in untrusted environments are risky. The scope is thus limited to focus on inference, where the system may retrieve pre-trained models from some available source.

- We assume that the system operates with a secure file system that ensures that disk data is stored securely and in a privacy-preserving manner. We also assume that the system fetches data that is stored on one or several secure storage units. To limit the scope of the thesis, we therefore abstract these away from the thesis implementation. Since the system may deal with private data, issues related to privacy-governing laws will be addressed. Execution on data in a privacy-preserving manner will also be addressed.

- Since we limit the thesis work to the development of a POC, we do not focus on optimizing the system with regards to power consumption. We therefore assume that the system may operate in an environment with a sufficient power source.

- With regards to the security principles, we limit the scope to focus on fault tolerance for the system.

Abnormal Behavior

As both the subtitle and the problem definition states, one of the aims of the system is to be able to detect and defend against abnormal behavior. The term "abnormal behavior" has its roots in psychology. According to the APA dictionary of psychology[2], the term is defined as:

"*Behavior that is atypical or statistically uncommon within a particular culture or that is maladaptive or detrimental to an individual or to those around that*

2. https://dictionary.apa.org/

individual. Such behavior is often regarded as evidence of a mental or emotional disturbance, ranging from minor adjustment problems to severe mental disorder."

For this thesis, we expand this definition to the computing systems domain, and use it to describe any form of unexpected behavior that may inflict damage of some form to, or disturb the system at run-time.

1.4 Context

This thesis is written in the context of the Corpore Sano Centre[3], which is affiliated with the Cyber Security Research Group at UiT The Arctic University of Norway. The Corpore Sano Centre undertake high-impact inter-disciplinary, inter-faculty research and innovation at the intersection of computer science, health informatics, statistics, medicine, sport sciences, and law. The long-term objective is to provide new knowledge, research tools, and innovative disruptive technologies in this convergence space. We address non-trivial challenges from the real world that integrate academic fundamentals with real-world engagement and innovations.

The Corpore Sano Centre has a deep history that dates back to 1988 with Stormcast, an expert-based distributed artificial intelligence application used for severe storm forecasting [16].

A collaboration between the Departments of Computer Science at University of Tromsø, Cornell University and University of California (San Diego) further resulted in the TACOMA (Tromsø and Cornell Moving Agents) project [17]. The focus of he project was on operating system (OS) support for mobile agents. An agent is, in this context, a piece of code that is executed on a remote computer. A moving agent is an agent that may move to other hosts within the assigned network during execution. The project proved useful in applicability, and complemented other structuring techniques that are common in distributed systems, and also lead to a re-implementation of Stormcast [18].

MapReduce is a parallel and distributed algorithm that is used in e.g. Apache Hadoop [19]. In 2012, the center further explored upon concepts of MapReduce with Cogset, where new routing and data placement mechanisms was investigated [20]. The use of a pre-determined routing scheme for data, meant that there was no need for temporary copies of data, resulting in improved data locality, and thus, performance improvements.

3. `https://corporesano.no`

The issue of intrusion tolerance in federated cloud overlay networks with a high number of nodes was addressed with Fireflies [21, 22], a secure and scalable membership and gossip service protocol which makes federated cloud overlay networks fault tolerant through intrusion tolerant-distributed hash tables or overlay network routines. Further research into cloud has been done with Balava [23], a system that enables management of computations on confidential data that span multiple clouds. Vortex [24] is an even-driven multiprocessor OS that explores how performance isolation can be used to divide resources between web servers effectively.

The Corpore Sano Centre has also conducted research on trusted execution environments (TEE), like Intel Software Extension Guard (SGX) [25] and Arm TrustZone [26]. The research conducted on SGX, in particular lead to Diggi [27], a framework that enables development of trusted cloud services with the use of TEEs.

In recent years, The Corpore Sano Centre has had a focus towards both sport, and health sciences. In sport sciences, the research has focused primarily on soccer, for instance real-time events detection [28]. Among the research done within health sciences, we can find ResUNet++ [29] NanoNet [30], which enables polyp segmentation in real-time with Video Capsule Endoscopy and Colonoscopy.

The center recently published Dutkat [31], a multimedia system used for catching illegal catchers. The paper introduces the concept and a design of a surveillance system that maintains privacy of legal actors while continuously capturing evidence-based data of illegal activities through video footage, sensory data and forensic proof. As a contribution to the Dutkat-project, the center recently published Dorvu [32], which aim to enable confidential and secure storage of data in low-bandwidth edge environments.

1.5 Contributions

This thesis is a contribution to the Dutkat-project with the design and implementation of a POC that can execute multiple machine learning pipelines with a distributed DAG computation model format in untrusted edge environments, while remaining tolerant to faults.

The resulting system prototype of Áika demonstrates that it is possible to achieve fault tolerance while executing tasks with a distributed DAG format, and thus forms a basis for future work within the group with multiple potential directions.

Some potential future work involves further developing Áika as a Proof of Applicability (POA) and directly tie the system to the Dutkat-project by investigating the systems ability to run on fishing vessels in the Arctic. This also includes integrating the system with other components of Dutkat, like the Dorvu file system.

The source code is appended to the thesis, and is provided as a part of the source files. The source code is also available on Github, but is currently placed within a private repository[4].

1.6 Outline

Chapter 2 covers relevant background information and related work within the domain of edge computing, AI and distributed systems.

Chapter 3 outlines functional and non-functional requirements of Áika.

Chapter 4 presents the design and the overall architecture of Áika.

Chapter 5 covers implementation details.

Chapter 6 provides the experimental setup, results of the experiments, an evaluation and a discussion of the results. The non-functional requirements are also reviewed based on the results.

Chapter 7 summarizes and concludes the thesis and provides some potential future work.

4. `https://github.com/Joaalslie`

2

Background and Related Work

This chapter outlines relevant fundamental concepts grounded in the thesis and some related work. Section 2.1 gives an introduction to AI, the different areas in the field and an in-depth introduction to machine learning and deep learning. Section 2.2 explains relevant background information in edge computing. Section 2.3 explains distributed software architectures that are relevant. Section 2.4 explains directed acyclic graph (DAG) as a computation model. Section 2.5 gives an overview of fault tolerance. Section 2.6 explains persistent event queues. Section 2.7 summarizes some related work relevant for the thesis.

2.1 Artificial Intelligence

The field of AI is broad. It is composed of multiple sub-fields that have been popular at different times in the history of AI. Even more so, the fields history has been rocky, with unrealistic expectations, hopes and promises. This section presents an introduction to AI fundamentals and the history of the field.

2.1.1 History

The concept of thinking machines was introduced in 1950 by Alan Turing, through the paper "Computing Machinery and Intelligence", which was published in MIND [33]. In the paper, Turing reflects on the possibility of thinking machines and how thinking machines could be distinguished from humans through a test of which he named "The Imitation Game". The paper is important for the later breakthrough of AI due to the visions and thoughts that Turing introduce regarding thinking machines. Considering the later outcome of how AI models turned out, it is also safe to say that Turing was a man ahead of his time.

Unfortunately, Turing was unable to realize his work, as computers was extremely expensive in the early 1950s, where leasing a computer could cost up to $200,000 (inflation adjusted per 2010) per month [34]. The early computers also imposed severe limitations with regards to memory, as they could only execute commands, not store them.

The term "Artificial Intelligence" was later coined in 1956 by John McCarthy. He defines artificial intelligence as:

The science and engineering of making intelligent machines, especially intelligent computer programs [35].

As computers became faster and able to store more and more information, the field of AI started to grow. The Perceptron algorithm was invented already in 1958 by Frank Rosenblatt [36]. It is a fundamental building block in neural networks today and Rosenblatt has in the aftermath been labeled as the father of deep learning, due to his invention. The research in the 1960s focused on solving geometrical theorems and mathematical problems. In the later part of the decade, researchers started working on machine vision, in robotics.

Despite the increase in computational power and memory, early AI still had difficulties processing large amounts of data. The field of AI eventually started to halt due to this limitation and combined with false hope resulted in the first AI winter in 1974. An AI winter is a period of time during the history of AI with cut funding and loss of interest [37, p. 21].

The salvation for the AI winter turned out to be an increased growth of interest in expert systems. Expert systems are computing systems that aim to emulate decision making of a domain expert [37, p. 80-81]. Expert systems are composed of two main components:

1. An **inference engine** which execute rules on known facts to reach a

conclusion. This may also include explanation of the conclusion and/or abilities for debugging.

2. A **knowledge base** composed of the rules and the facts.

Expert systems was originally invented in the 70s by Edward Feigenbaum (now known as "the father of expert systems") through the DENDRAL project, which started in 1965 [38]. DENDRALs main objective was to hypothesize the substance's molecular structure and the system even managed to rival chemists that where labeled expert on this specific task.

The popularity and funding within AI started to grow in the 1980s and more expert systems started to form, such as MYCIN [39] and the already mentioned Stormcast (see Chapter 1, section 4). The increase in hype was, however partly due to exponents of the field creating false promises and unrealistic claims. An example of this is how the press marketed MYCIN as a general purpose diagnosis tool, when it in fact, was limited to diagnosis of blood infections and was never used on real patients [37, p. 21]. Such unrealistic claims and false promises lead to a second AI winter, which struck the field in the late 1980s and lasted until the late 1990s.

Since then, AI has again started to prosper, with an increased interest growth in sub-fields like machine learning. Computers recently started to gain even more increased computational power, which lead to the rise of artificial neural networks in around 2010-2012.

2.1.2 AI Areas

AI is in general a vague term, since the word "intelligence" can be interpreted in different ways. Throughout the history of AI, numerous of sub-fields, or areas have sprung out from the general field. An illustration that shows the different areas can be found in Figure 2.1. As seen in the figure, AI is often separated into eight different areas [37, p. 22-26]:

- **Natural Language Processing (NLP):** This area involves parsing and/or generation of text or audio in the form of speech [40].

- **Decision support systems (DSS), knowledge based systems and expert systems:** These type of systems aim to assist in decision making by providing conclusions or advice by reasoning [41, 42].

- **Agent-based systems and multi-agent systems:** These type of systems are composed of one or more components that are able to do tasks

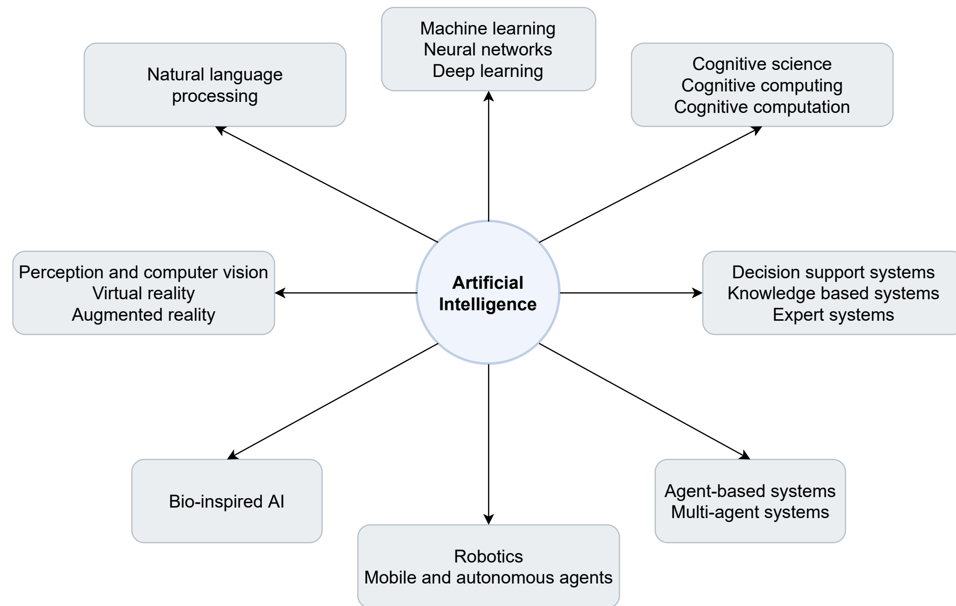

Figure 2.1: Illustrates the different areas in AI [37, p. 22]

individually, or as a group [43, 44]. A great example is the Boids simulation, where multiple components act together with a few simple rules to simulate flock behavior [45].

- **Robotics and mobile and autonomous agents:** This interdisciplinary area roots in electrical and mechanical engineering, as well as computer science. The area resolves around the dual development of physical devices and complementary software that carry out certain tasks [46, 47].

- **Bio-inspired AI:** This area revolves around using mechanisms from natural systems and/or evolution as a source of inspiration for developing AI models [48].

- **Perception and computer vision:** This area involves capturing and interpreting digital images and videos. Augmented reality (AR) and virtual reality (VR) are different ways of altering reality [49].

- **Cognitive science, computing and computation:** This area is slightly related to Bio-inspired AI, but instead uses human cognition within cognitive science as inspiration to simulate thought processes [50].

- **Machine learning, neural networks and deep learning:** This area of AI involves training of models to find patterns in data, with algorithms

that often have roots in statistics and mathematics [51, 52, 53]. This area will be discussed further in Subsection 2.1.3.

Note that the areas introduced here are by no means definite and some of them are closely related. Some AI techniques fall under multiple of these areas.

2.1.3 Machine Learning

As briefly explained in Section 2.1, machine learning revolves around training models on datasets with different types of algorithms. One fundamental thought regarding intelligent systems is that they should have the ability to adapt and learn when the environment it operates in changes. Machine learning, on a fundamental level, is to program computers using example data or actual data from past experiences in order to optimize a performance criterion. The resulting computer program may be either:

- **Predictive:** It aims to make a prediction based on example data or previous experience. An example of this is, for instance classification.

- **Descriptive:** It aims to gain and extract knowledge from data. An example of this is, for instance feature extraction or dimensionality reduction.

A machine learning model can also be both of these two [54, p. 1-4].

Machine learning is traditionally divided into three main categorizes: Supervised learning, unsupervised learning and reinforcement learning. Reinforcement learning is concerned with training software agents to act in environments in order to maximize some kind of reward without specifying how the task is going to be achieved [55]. This area will not be explained further due to relevance and we put a focus on the differences between supervised and unsupervised learning instead.

Supervised Learning

Supervised learning is a category of machine learning concerned with optimization of a model through labeled input data. Labels refers to the expected outputs obtained from feeding input data to the supervised machine learning model. The aim of a supervised learning algorithm is to use training samples of input feature vectors with corresponding labels to obtain relationship information between the inputs and labels. The resulting mapping created from training can then be used to predict the output of new feature vectors [56].

Supervised learning serves many purposes, but the most known is perhaps for classification tasks, which can be done with linear classifiers like the Perceptron algorithm [36] or the Support Vector Machine (SVMs) [57], which works well with linearly separable data sets. For more complex data sets that cannot be separated by a single line, there also exist non-linear classifiers, like decision trees [58] and kernel-based methods, for instance a kernalized SVM [59].

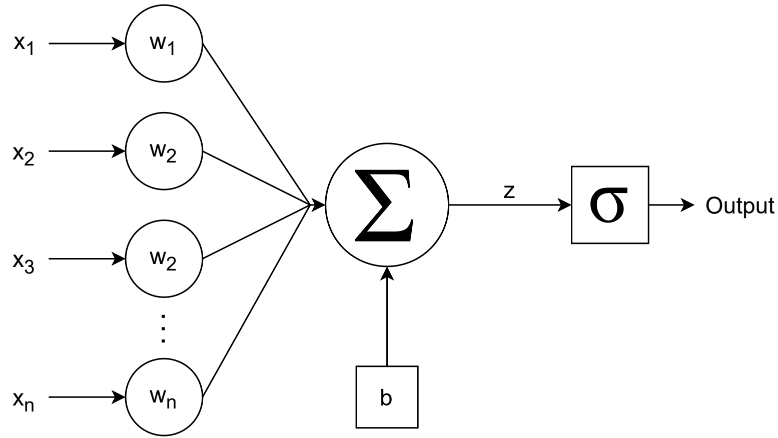

Figure 2.2: The Perceptron algorithm.

The Perceptron algorithm is particularly important due to its further development to the Multilayer Perceptron (MLP), leading to the further development of artificial neural networks. Figure 2.2 shows an illustration of the Perceptron algorithm. The input feature vector x is initially multiplied element-wise by an equally large vector w. The resulting vector is summed together, before a bias value b is added to the final sum. Mathematically, this procedure can be written as:

$$z = b + \sum_{i=1}^{n} w_i x_i$$

Which can be further generalized to:

$$z = W^T x + b \tag{2.1}$$

Where W is a matrix of weights, x is vector of the input data to the perceptron, and b is the bias.

The result may be propagated through an activation function in order to introduce some form of non-linearity. The activation function can be a simple, linear function where the input is equal to the output, or for instance the Sigmoid function, which can be seen in Equation 2.2:

$$\sigma(z) = \frac{1}{1 + e^{-az}} \tag{2.2}$$

Where z is equal to the output from Equation 2.1 and a determines the steepness of the curve. With the following structure, the Perceptron can be trained by randomly initializing the weights w and the bias b, then pass data through the Perceptron and compute the error with regards to the labels in the data. The loss function can then be optimized to be as low as possible through an optimization algorithm, like Stochastic Gradient Descent (SGD).

Unsupervised Learning

While supervised learning often provides satisfactory results, it also requires resources in order to label the data properly. Other challenges includes hindering the model from becoming biased and dealing with lack of training data. Unsupervised learning is a category of machine learning that counters this issue by aiming to detect patterns in unlabeled input data. It therefore does not rely on using error signals to evaluate solutions [60]. Classification tasks can still be done through clustering algorithms, a type of machine learning algorithm that puts the feature vectors from the input data into groups based on some pre-determined criterion.

The most used clustering algorithm is K-means clustering [61], which assigns feature vectors to a given number (k) of clusters. This is done by randomly initializing k feature vectors, known as *centroids*. Each feature vector is assigned to the closest centroid, before the centroid is moved to the average location of its members. This process is repeated for a specific number of times, or until the centroids movement converges towards 0. The disadvantage of k-means is that it is limited to be used for spherically shaped data. There is also a risk that one or more centroids become stuck between clusters during the training process.

Unsupervised learning can also be used to extract features from the input data through dimensionality reduction. An example of an unsupervised dimensionality reduction algorithm is Principal Component Analysis (PCA) [62, 63, 64]. PCA selects the optimal features by performing a linear transformation to a new feature space, where the principal components (the selected features) are linear functions of the original ones. They are also uncorrelated and PCA selects features with the highest variance as principal components. This is done by computing the covariance matrix of the full data set, before the eigenvector and eigenvalues are computed and sorted based on in decreasing order with regards to the eigenvalues.

Semi-Supervised Learning

machine learning relies on high-quality data to obtain high-quality machine learning models. As previously mentioned, a problem with supervised learning is to obtain correct labeling for large data sets. Semi-supervised learning (SSL) is a combination of supervised and unsupervised learning, where the machine learning algorithm is provided with some information for supervision, but not for all data points. In standard SSL, the data set used during training can be divided into two distinct parts: one where labels are provided and one where they are not [65, p. 2-3].

The idea of using unlabeled data for classification training has existed since the 1960s with self-learning [66]. In the initial stages of the training process, the model is trained on labels only and unlabeled data will gradually be labeled based on the decision function in the current step. The model is then retrained based on the previously used data and the newly labeled data. In the 1970s, semi-supervised learning started to gain increased momentum through estimation of the Fisher linear discriminant rule using unlabeled data. The term "semi-supervised learning" was, however not used before 1992, when the term was introduced by Merz et. al [67] The interest in SSL increased further in the 1990s due to its potential application in text classification and NLP problems [65, p. 3-4].

The SSL paradigm is becoming even more important today as we are gradually generating more and more data in large volumes. Labeling such large quantities require time and resources and we therefore rely on SSL techniques to use the data in machine learning tasks. In the recent years, SSL has for instance been utilized in neural networks, in clustering methods and virtual adversarial training [68].

2.1.4 Deep Learning

Deep learning is a sub-field of machine learning concerned with recognizing patterns in or extracting information from data through deep artificial neural networks. As mentioned previously, the Perceptron is a fundamental building block in neural networks. After the Perceptron algorithm was introduced, researchers found that multiple Perceptrons could be stacked together in layers to approximate non-linear functions. In addition to this, several layers could also be put together to allow more complex analysis of the data. For Deep Learning, the network is typically built up of at least 2 hidden layers. This structure is today called Multilayer Perceptron (MLP) and it is also one of the most frequently used today [69].

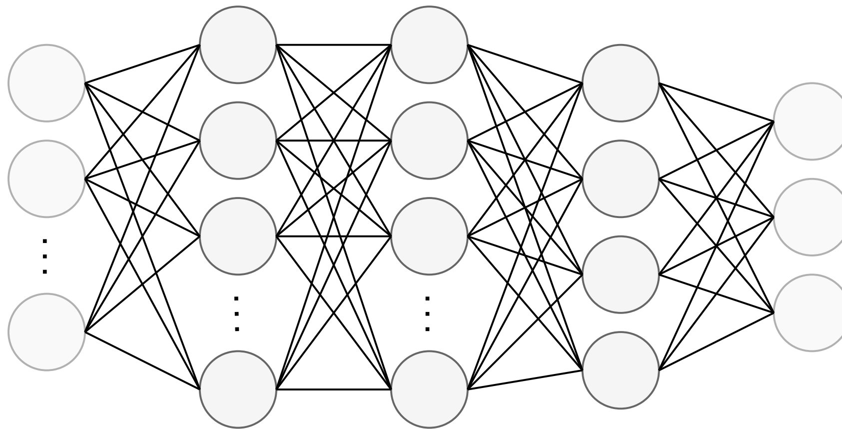

Figure 2.3: The typical structure of neural networks. This network is composed of five layers in total, where each circle represent a neuron (or a perceptron). For this network, three of the layers are hidden layers. During inference, data is propagated into the input layer (to the left) and through the network until the output result is received at the output layer (to the right).

Figure 2.3 illustrates how the general structure of neural networks. The network receives input data, which is fed into each neuron (perceptron). The output from the neurons in the first layer is then propagated forward as input to the next layer. This is done layer by layer, until the output of the final layer has been computed. A loss function can be used to compute the error between the provided output and the given label. Since the network is composed of many layers, with weights in each layer, the optimization routine is a bit more complex compared to using a single perceptron. Optimization is done by propagating backward in the network, layer by layer and computing the gradients with respect to the weights and the biases in each neuron. The weights and biases are adjusted slightly depending on the optimization algorithm used. This procedure of forward-propagating data and back-propagating is done iteratively with the training data. Data can be propagated through the network as single data points, in batches, or by sending the entire data set through each time. The procedure may be repeated for a specific number of times (epochs) as well.

Neural networks are not limited to MLPs. An example of a different type of neural network is Convolutional Neural Networks (CNNs). This type of network maintains the same structure as the MLP (seen in Figure 2.3), but utilizes discrete convolution, often in combination with max-pooling. The convolution operation functions by using a moving kernel of values to compute a weighted sum between the overlapping structured data and the kernels values. The kernels values constitute the weights that are trained in the convolutional layers. The operation is often used in computer vision tasks on image data.

The convolution operation is described in Equation 2.3:

$$z = W * x = \sum_{c=1}^{C} \sum_{k=-\lfloor F_H/2 \rfloor}^{\lfloor F_H/2 \rfloor} \sum_{m=-\lfloor F_W/2 \rfloor}^{\lfloor F_W/2 \rfloor} W_{c,k,m} x_{c,i+k,j+m} \tag{2.3}$$

Where C denotes the number of channels in the image, F_H denotes the height of the kernel, F_W denotes the width of the kernel, W denotes the kernel, and x denotes the image.

Specific kernels can be used to blur images, detect edges or corners, etc. With CNNs, we seek to train the networks to learn the kernels themselves. Convolution layers are often used as initial layers in neural networks that extract features from the input data.

Max pooling is often applied along with convolution in order to reduce the dimensionality of the data and extract the most relevant features. The max pooling operation performs feature reduction by selecting the highest valued feature from each cell in a grid that overlaps with the input data. The cell sizes in the grid are pre-determined.

Convolution is usually used for extracting features from the data and an MLP is therefore often used afterwards to classify the data based on the features extracted by the convolution layers [70]. Some advantages of using the CNN is that they utilizes few weights compared to a single, large MLP and the convolution operation does not require the data to be of a fixed size [71, p. 326-366].

When a network model is done training, it is also possible to detach the classifier and use the remaining part of the network as a feature extractor on other data sets. This procedure is known as *transfer learning* [72].

In general, neural networks can be shaped in many ways, from relatively simple architectures like the convolutional neural network LeNet5 [73] to complex networks like the semantic segmentation network U-NET [74]. Despite this, they often tend to maintain a similar structure, where a network is organized into multiple layers composed of neurons. Note that there also exist many other types of neural networks than the ones mentioned here, for instance LSTMs [75] that is used for NLP tasks, or Autoencoders [76] that can be used for noise removal and/or compression of data. We will not go into these types here, as they are not particularly relevant for the thesis.

2.2 Edge Computing

Edge computing is a distributed programming paradigm where computational processes are moved closer to the data source at the edge of a network. The aim is to reduce the response time and to save bandwidth, but can also be used for privacy. Figure 2.4 illustrates how the Edge Layer connects the IoT layer, where sensor data is collected, in relation to the cloud layer [77].

There are several additional motivations behind edge computing, such as enabling a more sustainable energy consumption by offloading cloud centers. Another important motivation is that edge computing can be used to deal with data explosion due to the increase in IoT devices and thereby decrease the network traffic. Edge computing does also, however has many challenges. Enabling general-purpose computing on edge nodes is considered a severe challenge due to the variety of edge nodes that exists. A second challenge involves how to partition and offload tasks efficiently and automatically to improve performance. Enabling the use of edge nodes in a secure and privacy preserving manner is also an important challenge, particularly for alternative devices, like switches, routers and base stations [77].

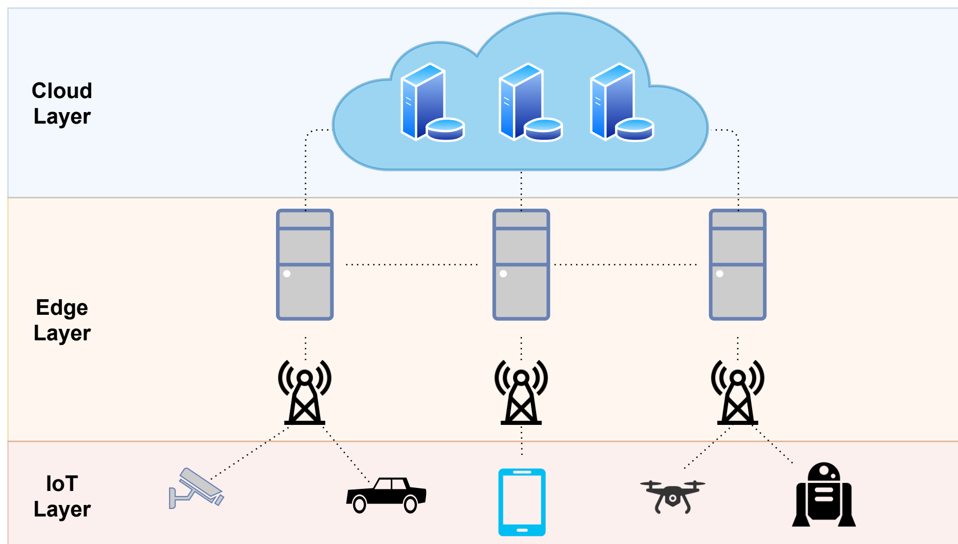

Figure 2.4: The Edge Layer. The figure illustrates how the edge layers can be used as a connection between the IoT Layer and the Cloud Layer. Sensor data is collected in the IoT layer, while computation is carried out in the Edge Layer. Data collection, results and additional computing that does not require a quick response time can be moved to the Cloud Layer.

Centralized and Decentralized Edge Computing

While the Edge layer is placed between the IoT Layer and the Cloud Layer, the physical edge nodes may still be placed at different locations. A centralized edge will have the edge nodes placed close, typically in the same local network. A decentralized edge will, however have nodes scattered across several locations. It is also possible to mix these two into a partly centralized and partly decentralized edge, where some nodes are connected in the same local network and some nodes are scattered across multiple locations. Decentralized edges has for instance been used to create decentralized storage systems [78].

Untrusted Edge Environment

Edge devices may sometimes operate in environments where people, processes or physical equipment are at risk of harming the devices. For example in surveillance systems where people are being monitored, there is a risk that malicious actors will try to damage the system hardware, or interact with the processes. We refer to an environment like this as an *untrusted edge environment*. When operating in these types of environments, it is important that processes within the system and the equipment itself is protected in such a way that the system is able to withstand a certain amount of damage and to be able to detect various types of failures.

Edge Intelligence

A rapid development in communication technology has lead to a surge in mobile device usage. This has further lead to the rise of edge computing, which is gaining a constant increase in popularity today. At the same time, the increased computing power has lead AI to thrive more than ever before. Current ongoing research aims to investigate how some AI solutions can be moved to an edge environment to reduce the response time and save bandwidth. The term "Edge Intelligence" is often used to describe the confluence between edge computing and artificial intelligence [13].

2.3 Distributed Software Architectures

Distributed systems are characterized by dispersing components across multiple machines, which may lead to a high amount of complexity within each individual system. When working with such complex system, it is crucial to organize the system in a proper manner. The aim of *distributed software archi-*

tectures is to describe how the components in a distributed system is organized and how they interact [79, p. 55]. Distributed software architectures can be categorized into four main styles: Layered, event-driven, object-oriented and resource-centered architectures.

Layered architectures organize components into layers, where components at a layer above can make down-calls to layers below and layers below can make up-calls to above layers. This type of architecture can typically be found in layered communication protocols, like Transmission Control Protocol (TCP), or in application layering [79, p. 57-62].

In object-oriented architectures, objects correspond to defined distributed system components and each object is connected through a procedural mechanism [79, p. 62-64].

Resource-based architectures see a distributed system as a collection of resources, where each resource is managed by one of the systems components. An example of a resource-based architecture can be found in the web and is today known as Representational State Transfer (REST) [79, p. 64-66].

Event-driven architectures is based on detection, consumption, production and reaction to events, where events often are considered as significant changes in a state. This architectural style is for instance used in publish-subscribe architectures [79, p. 66-71].

2.3.1 Controller/Agent

Often known as *master/slave* architecture, the term has recently caused some controversy due to its improper naming scheme [80, 81, 82]. The problem was also addressed already in 1997 by Johansen et al. [83], where they used controller and worker as terms. It is possible that the rise of the Black Lives Matter-movement has shed light on this in more recent times. For this thesis, we have decided to use an alternative to the previous and will therefore refer to this architecture as *Controller/Agent architecture*, which is now used by e.g. Jenkins[1].

In distributed systems, the controller/agent architecture is a model for communication where a single process or device (controller) controls one or more other processes or devices (agents) [84]. This is illustrated in Figure 2.5, where a single controller assigns tasks to four agents. Once a task has been completed,

1. `https://cd.foundation/blog/2020/08/25/jenkins-terminology-changes/`

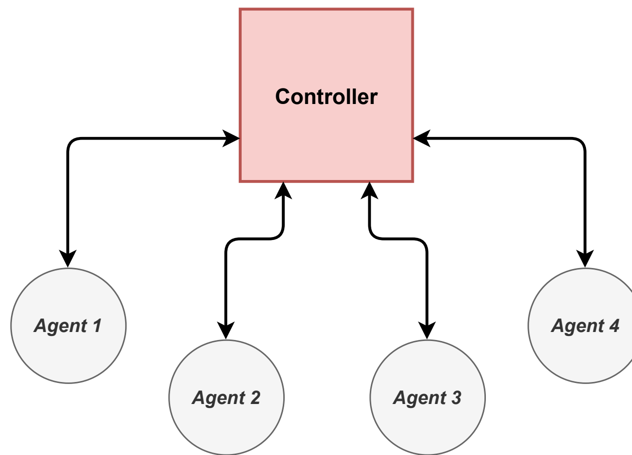

Figure 2.5: Controller/agent architecture. Illustrates how a single controller orchestrates several agents.

the agent may refer back to the controller to receive another task. The controller may also collect the result from the agent, if necessary. The architecture is common due to its simplicity and has been used in several systems, such as the Google File System [85].

There are several reasons for using the controller/agent architecture. It is practical to avoid deadlocks as the controller is the only process that synchronizes with the agents. The system can thus avoid typical synchronization pitfalls. The architecture also scales well horizontally, as more agents can be added to carry out work. This will be the case as long as the controllers workload does not increase by too much, as such would lead to it becoming a bottleneck in the system. The disadvantage of this model is that the controller is at risk of becoming a single point of failure, meaning that if the controller shuts down, the entire system will shut down.

2.3.2 Pipeline

In the 1960s the need for most cost-effective and high-performing systems became critical. This lead to the development of the pipeline architecture. Pipelining is a form of embedding parallelism or concurrency where a computation process is segmented into stages. These stages are executed by autonomous units in an overlapped mode, almost like an industrial assembly line [86]. A loose definition of pipelining, according to C. V. Ramamoorthy and H. F. Li is:

The technique of decomposing a repeated sequential process into sub-processes,

each of which can be executed efficiently on a special dedicated autonomous module that operates concurrently with the others [86].

An illustration of the pipeline architecture can be found in Figure 2.6, where a sequential process is split into N stages. The pipeline is constructed as a chain of stages. The initial stage fetches or receives input from some kind of source and performs a task with respect to the input. The result is then passed on to the next stage before a new input element is retrieved.

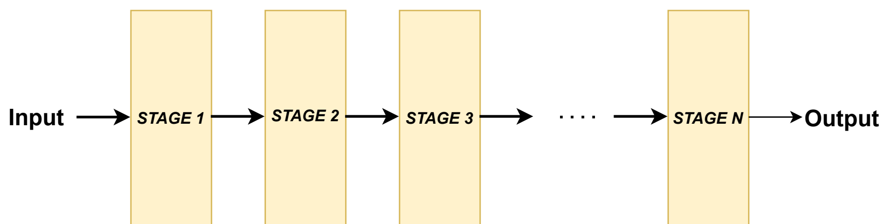

Figure 2.6: Pipeline Architecture. Illustrates a pipeline consisting of N stages.

The pipeline architecture has gained considerable attention since its initial usage in the 1960s and it has been used in a wide number of areas, in hardware architectures like MIPS [87], in operating systems like Vortex [24], in distributed deep learning inference [88], or even relatively simple tasks like the *Farmer Task* [79, p. 202-203].

The downsides of the pipeline architecture is that it requires a very strict, sequential processing format to be able to work. Another challenge is how a process can be divided into stages with an even work load in each stage. If the stages does not have an even work load, the system is at risk of being stalled by one of the stages, which can lead to a producer/consumer problem [89] if not handled correctly.

2.4 Directed Acyclic Graph

In the field of graph theory within mathematics and computer science, a directed acyclic graph (DAG) is a graph where the edges point in only one direction. This is typically illustrated with arrows that point in the direction that the information is flowing.

The term "acyclic" means that there is no way for a vertex to cycle back to itself (there are no cycles in the graph). This means that once a node has been visited, it cannot be revisited. DAGs are great to use for modeling dependency tasks, where one task needs to be carried out before another [90]. Figure 2.7

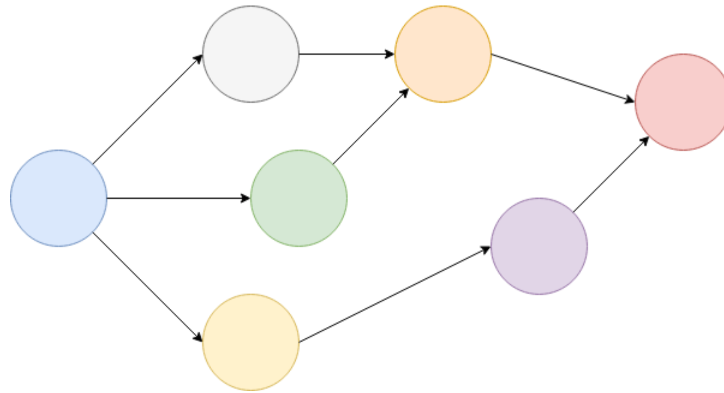

Figure 2.7: An example DAG consisting of 7 vertices and 8 edges.

illustrates an example DAG. The graph is composed of 6 vertexes and a total of 8 edges.

In distributed systems, DAGs can be used to illustrate the connection between multiple processes within a system. Each vertex typically represents a process (or execution), while the edges represents communication between them. The implementation of the communication between the vertices may vary from system to system. In for example Dryad (see Subsection 2.7.3), the communication can come in the form of either file, TCP pipes, or shared memory FIFO queues [91].

The DAG is often considered as a general computation model. In AI, DAGs are particularly used to model neural networks, due to their practical, layered structure. This can for example be seen in deep learning frameworks like DistBelief [92], Caffe [93] and Tensorflow [94], where the computation is performed at the vertices of the graph. The values that flow along edges are referred to as tensors. Tensors are multi-dimensional arrays that contains data. The DAG computation model is not limited to deep learning and is also used in other types of computational frameworks, like Cogset [20] and MapReduce [19].

2.5 Fault Tolerance

The notion of partial failure occurring is an important characteristic in distributed systems. Partial failure refers to the event where a component of the system has failed, while the other components remain operable. Whenever such a failure occurs, the system should continue to operate while the failing component of the system is being repaired. The ability to recover from

such failures while the system operates is referred to as *fault tolerance* [79, p. 423].

2.5.1 Dependability

Fault Tolerance is strongly related to dependability. We often refer to systems that can satisfy the dependability characteristic as dependable systems. These type of systems typically cover four requirements associated with dependability[95]:

- **Availability:** The system is ready for use immediately. It refers to the probability that the system is operating correctly at any given moment. Availability is defined in terms of an instant in time.

- **Reliability:** The system can continuously run without failure. It is different from availability by being defined in terms of a time interval.

- **Safety:** No catastrophic action will occur, despite temporary failures. This is a very important requirement in safety-critical systems, where failure can lead to loss of lives or generally extreme damages.

- **Maintainability:** A failed system can easily be repaired. It relates to availability, as a maintainable system tends to show a high degree of availability.

There is, however not a complete agreement on this. Some researchers choose to include three additional requirements [96]:

- **Confidentiality:** data and other information should not be made available without intent and authorization

- **Survivability:** The systems services are robust enough to withstand attacks.

- **Integrity:** Ensures that data is not modified or deleted without intent and authorization.

It is important to note that these requirements are generally strongly related to each other. This can be seen in particular for the four base requirements. A system that can be repaired easily (and ideally automatically) will make the system available for use quicker. A reliable system ensures that the system runs for longer periods of time without failing, which compliments safety, which reduces the probability that something catastrophic happens when a failure

finally occurs.

2.5.2 Failure Models

A failing system indicates inability in providing its services. If a failure occurs, it is important to know what type of failure it is and the implications of the failure. The seriousness of failures can be classified based on a classification scheme [79, p. 428]:

- **Crash Failure:** Permanent halt, but working correctly until halt.
- **Omission Failure:** Failure of responding to incoming requests. Could either fail to receive message (*receive-omission failure*) or fail to send message (*send-omission failure*).
- **Timing Failure:** The response lies outside a specific time interval.
- **Response Failure:** The response is incorrect. The value can be wrong (*value failure*) or the response may deviate from the correct control flow (*state-transition failure*).
- **Byzantine Failure:** Arbitrary responses may be produced at arbitrary times and may remain undetected. In addition, output may be outside expected value limits.

Byzantine failures are often considered the most serious, as the failure are at risk of not being detected at all. They tend to occur if a component fails to perform an action which it should have, or if it perform an action which it should not have.

There are a wide range of potential consequences due to failures within a system. They may for instance affect consensus in asynchronous systems. The FLP impossibility result shows that it is impossible to achieve deterministic consensus in asynchronous distributed systems where one or more processes are unreliable [97]. Despite this, the case can be by-passed through randomized consensus algorithms [98], where liveness and safety, even under worst-case scenarios is achieved with high probability.

2.5.3 Redundancy and Resilience

Both Redundancy and Resilience play important roles when attempting to achieve fault tolerance in a system.

Redundancy

The term "redundancy" tends to have a negative meaning, but in the case of distributed systems, we use redundancy as a means for masking failures [79, p. 431]. Failure masking by redundancy is typically divided into three categories:

- **Information Redundancy:** Extra bits are added to allow recovery from garbled bits.
- **Time Redundancy:** An action is performed again if it timed out or failed.
- **Physical Redundancy:** Extra equipment or processes are added to make it possible for the system to tolerate loss/malfunctioning of some components.

Resilience

Resilience is defined as the capacity to recover quickly from failures. In the context of distributed systems, it is a set of strategies used to achieve high availability. The resilience property is often added to distributed systems through process groups. When transmitting a message, it can then be sent to the entire group of identical processes instead of a single one. That way, the system may handle the message, as long as the entire group has not failed [79, p. 432-433].

Triple Modular Redundancy

Triple modular redundancy (TMR) is a technique used to achieve fault tolerance. In TMR, a task is performed by three modules (or processes) and the result is processed by a majority-vote system that produces a single output [79, p. 431-432].

Figure 2.8 illustrates how this can be carried out in practice, where three modules receive input from one source, while a voter collects the result from the modules and produces a single output. The voter provides healing, such that an error in stage n will be masked away for all of the pipelines. This technique illustrates how redundancy is used to group identical processes to achieve resilience in a system. Triple Modular Redundancy can be generalized to N-Modular Redundancy case, where the input is sent to a pre-determined number of modules.

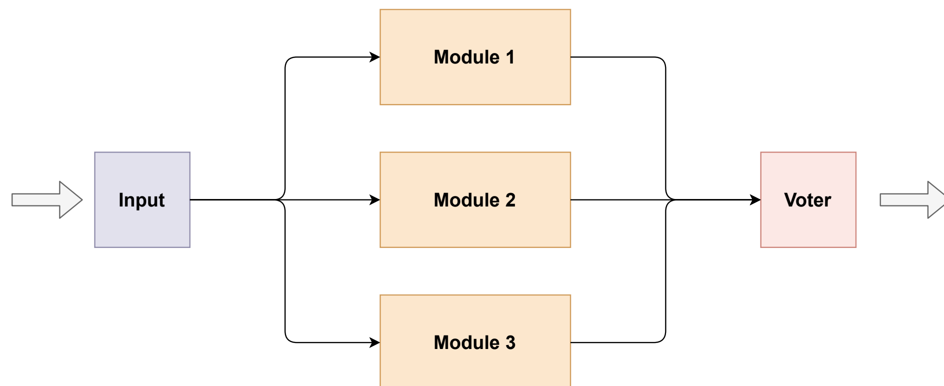

Figure 2.8: Triple Modular Redundancy. The figure illustrates how the same input source can be forwarded to multiple modules. A voter receives the result from each module and decide which of the results to forward.

2.5.4 The CAP Theorem

The CAP theorem was first proposed in 1999 [99], but it had not yet been proven and was therefore referred to as the "CAP Principle". The principle was proven in 2002 [100] and has afterwards been referred to as the CAP Theorem. A more formal definition of the CAP Theorem is as follows [79, p. 461]:

CAP Theorem: *Any networked system providing shared data can provide only two of the following properties:*

- **C:** *Consistency, by which a shared and replicated data item appears as a single up-to-date copy.*
- **A:** *Availability, by which updates will always be eventually executed.*
- **P:** *Tolerant to the partitioning of process group (e.g. because of a failing network).*

In other words, in a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request [101].

The theorem illustrates the trade-offs between safety and liveness in distributed systems and it shows that achieving both at the same time is impossible.

2.5.5 Detecting Failures in Distributed Systems

Failure detection is an important part of implementing a fault tolerant system, as faults in a system also must be detected in order to be handled. This can be difficult, there is only two ways of checking if a process has failed: Either send a ping message and wait for a response, or wait passively for a message. The problem with the assumption that a process has failed after a timeout is that it may simply be slow at responding. The base solutions can be expanded such that processes perform information exchange through gossiping when communicating. That way they can always know when a process was last heard from by other nodes [79, p. 462-464].

2.6 Persistent Event Queue

A computer is a complex machine that runs many processes and threads simultaneously. Sometimes these processes and threads need to communicate with each other through inter-process or inter-thread communication. Distributed systems are composed of multiple computers that communicate with each other and also rely on communication protocols over a network in order to pass messages between them.

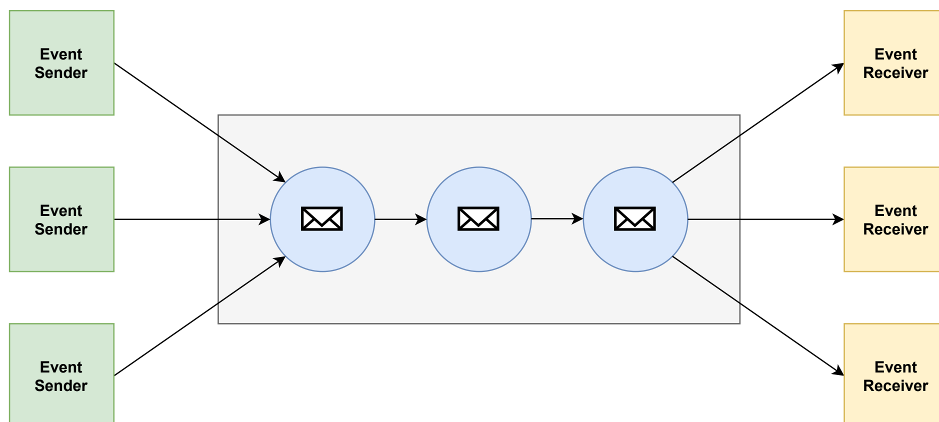

Figure 2.9: Illustration of an event queue. Senders passes events into the queue and receivers dequeues individual events which are then handled.

An event queue is a storage unit where events from a process are held prior to being processed by a receiving process. They are stored in first in first out (FIFO) order [102]. The processes that utilize the queue are typically split into two groups:

- **Senders:** Are responsible for enqueuing events.

- **Receivers:** Are responsible for dequeuing events and consume them.

Figure 2.9 illustrates how senders and receivers interact with a event queue. The communication may vary, depending on if the sender/receiver is local or remote in relation to the queue.

Persistent queues provides a certain guarantee for data integrity of the items they hold, as long as the data they hold is persisted to non-volatile memory. In the event of software failure the queue will be completely recoverable, as long as the files and the file system is not damaged or corrupted.

2.7 Related Work

A tremendous amount of research has been done in the recent years that can be related to Áika. In this section we present some of this related research. We present FRAME and CESSNA in Subsection 2.7.1, NAP and Falcon Spy Network in Subsection 2.7.2, Dryad and Cogset in Subsection 2.7.3 and SEDA and Vortex in Subsection 2.7.4.

2.7.1 FRAME and CESSNA

FRAME is a fault tolerant and real-time messaging architecture for edge computing [103]. It is developed through a publish-subscribe architecture where the broker is duplicated to avoid single point of failure. It leverages timing bounds to schedule message delivery and replication actions to meet needed levels of assurance. The timing bounds are thus able to capture relation between traffic/service parameters and loss-tolerance/latency requirements. The architecture is implemented on top of the TAO real-time event service [104]. FRAME is implemented with a publish-subscribe messaging model, consisting of three components: publishers, subscribers and brokers. They enforce fault tolerance by using a backup broker to avoid single point of failure. Toleration of broker failure is heavily weighted in the system, as the broker is responsible for accommodating all message streams. It can therefore also become a bottleneck in the system.

CESSNA (Client-Edge-Server for Stateful Network Applications) is an initial proposed protocol developed to achieve resilient networking on a stateful edge computing system [105]. The protocol provides consistency guarantees for stateful network edge applications and allows offloading computation from clients and servers. This leads to a reduction in response latency, backbone bandwidth and computational requirements for the client. CESSNA has a design

built of two layers. These layers represent different types of edge recovery: local and remote recovery. They are executed based on the distance the failed edge has to the recovered edge.

2.7.2 NAP and Falcon Spy Network

NAP (Norwegian Army Protocol) is a scheme for fault tolerance and recovery of itinerant computations [106]. The runtime architecture resolves around having a landing pad thread and a failure detection thread within each process. The landing pad is responsible for maintaining a NAP state object that stores information about mobile agents hosting execution or serving a rear guard. The landing pad thread is responsible for informing the failure detection thread which landing pad that needs to be monitored. NAP uses a linear broadcast strategy that refines the strategy implemented by Schneider, Gries & Schlichting [107]. NAP also expands upon the existing TACOMA [108] and is able to provide low-cost fault tolerance to its users.

Leners et. al has implemented a system for failure detection and recovery in distributed systems through a network of spies [109]. Falcon Spy employs a network of spies in the application, operating system, virtual machine and network switch layers on the system being monitored. Spies are deployed at the different layers to hinder disruption. The motivation behind Falcon Spy is to enable effective failure detection and improve the previous method (end to end timeouts).

2.7.3 Dryad and Cogset

Dryad is a general-purpose distributed execution engine developed by Microsoft, used to execute coarse-grained data-parallel applications [91]. One of Dryads key features is to allow the user to construct an execution DAG through a custom graph description language. The Dryad run-time maps the graph onto physical resources. The graph vertices allow an arbitrary number of inputs and outputs, unlike MapReduce [19], which only supports single inputs and outputs. A job manager contains the application-specific code used to construct the communication graph. It also schedules work across available physical resources, which are maintained by a name server. Dryad supports three different channel protocols:

1. File: This is the default protocol. A producer writes information to disk in a temporary file and a consumer reads from this file.

2. TCP Pipe: Information is continually passed via Transmission Control

Protocol (TCP). It requires no disk access, but limits the vertices to schedule at the same time.

3. Shared-Memory FIFO: Information is passed through First In First Out (FIFO) queues within shared memory. This makes the communication overhead extremely low, but it requires end-point vertices to run within the same process.

The fault tolerance policy of Dryad assumes that the computation performed is deterministic. The job manager is informed through a daemon that monitors the vertex whenever an execution fails. If failure occurs due to a read error on an input channel, the default policy marks the execution record as failed, which terminates the process if it is running.

Cogset has a different design compared to Dryad. It is a high-performance engine that builds on the principles of MapReduce [19], but uses static routing (contrary to MapReduce, which uses dynamic routing) while preserving non-function properties [20]. These characteristics are associated with traditional engines. Cogset provides a few fundamental mechanisms for reliable and distributed storage of data and parallel processing of statically partitioned data at its core. The use of static routing ends up reducing the need to store temporary copies of the intermediate data. On the other hand, it also requires tighter coupling between components for processing and storage within the system. Note that Cogset outperforms Hadoop, even when having an extra Hadoop layer for compatibility.

2.7.4 SEDA and Vortex

SEDA (staged event-driven architecture) is a highly concurrent Internet service that simplifies the construction of well-conditioned services [110]. SEDA applications are constructed as a network of stages. A stage is defined as an application component that consists of three sub-components:

1. An event queue that handles incoming events.

2. An event handler.

3. A thread pool.

SEDA applications are composed of a network of such event-driven stages. They are connected with queues. This architecture is best suited for well-conditioned events, meaning that they behave like simple pipelines. Stages are scheduled by a set of dynamic controllers. They also manage resource usage. Isolating

threads to single stages makes thread synchronization and race conditions easier to handle.

Vortex is a fully event-driven multiprocessor operating system that supports performance isolation [24]. The kernel and applications in Vortex are structured as stages, like the previously discussed SEDA [110] and they communicate through event passing. Each available CPU manages a separate event scheduling tree. They are tree structures that consist of event queues. The nodes in the tree run separate event schedulers that moves events towards the root. Events are enqueued at the leaf nodes. The Vortex EST mechanism also has the ability to install different schedulers in the trees nodes.

2.8 Summary

The chapter presents relevant background and related work for the thesis. The history and areas of AI, starting from the Turing paper in 1950, to expert systems in 1980, an the recent rise in neural networks and deep learning is explored. The chapter also covers fundamentals of edge computing and how AI and edge computing can be merged together. We have explored different distributed software architectures that are relevant for out system, in particular the controller/agent architecture, but also the pipeline architecture. We have looked into the different failure models that can appear in distributed systems and how different types of fault tolerance can be achieved. Related work shows that there is a high amount of research that has been done within these areas. This applies both for achieving fault tolerance and for supporting a distributed DAG computation model.

The next chapter provides an analyze what requirements are necessary to build our system with regards to the problem definition and the application domain, using our acquired knowledge from this chapter as well.

3

Requirement Analysis

This chapter discusses the Dutkat-project and use it to outline the requirement specification for Áika. The requirement specification is divided into Functional Requirements and Non-Functional Requirements.

Fish is today considered as one of the most important food sources in the world. It is estimated that fish takes up around 17% of the current production of edible protein on a global scale, but this number is expected to increase with population growth [111]. At the same time, the fishing industry has fallen victim to crime in the form of illegal fishing and over-exploitation. The United Nations war on crime has estimated that billions of US dollars are lost each year due to fish crime [112]. These problems affect the sustainability of fish stocks and it is important to combat these forms of criminal activities in order to enable a sustainable fishing as the demand for fish increases.

It has been argued that installing equipment for 24/7 monitoring and surveillance on fishing vessels can aid in the combat against fish crime. This is currently being done in Denmark and the Norwegian government is currently investigating similar solutions [113]. The deployment of such systems does, however come with some serious disadvantages as well. Without proper automation, monitoring of the data would have to be done manually, which will require a tremendous amount of resources when the system is deployed aboard many vessels. Automatic analysis of sensor and surveillance data combined is in itself a challenge that must be thoroughly researched. Another problem is privacy, as surveillance in such a high degree may be considered personally

invasive.

Dutkat [31] is a multimedia system proposed for catching illegal catchers and being privacy-preserving at the same time. This is enforced by storing data within a secure storage, which may only be accessed if a true positive flag of illegal activities has been raised. We aim to use Dutkat as a motivation for the work on Áika. The system is implemented with Dutkat in mind, specifically. Despite this, one aim is to make Áika generalizable for other cases as well.

3.1 Low-Bandwidth Edge Environments

Fishing vessels operating in the Arctic sea may have trouble connecting to high-bandwidth networks. This makes it necessary for Dutkat to rely on satellite broadband. This introduces four new challenges for the system:

- **Cost:** Communicating over satellite broadband requires additional physical equipment to function properly, but also a subscription to a satellite broadband. Depending on the use case, both of these can be very expensive.

- **Connectivity:** Digital communication between the fishing vessels and the mainland operational centers is often limited, because the communication is primarily facilitated by satellites.

- **Bandwidth:** The bandwidth of satellite broadband is very limited, which raises issues when there is a need for transporting voluminous amounts of data that is generated continuously on the fishing vessel. A 2017 report published by the Arctic Council Secretariat [114] states that maritime users at 72 degrees north suffer from lack of bandwidth and unstable communications at sea. This is the case, even for narrow-band communications. They also state that an increased reporting of catch from fisheries will require more capacity for sustainable management. Furthermore, research shows that generated data cannot be transported across satellite broadband without compression [32], which can lead to data being corrupted by noise and thus affect the machine learning inference accuracy.

- **Partitioning:** The issue with low bandwidth can further lead to problems with partitioning. If data is generated in large volumes, it becomes infeasible to partition the data over several locations, which makes it necessary to store most of the data on the vessel.

These challenges reveal that data transportation across a satellite broadband can cause a severe bottleneck in the system. This can be solved by moving parts or the entire computational process to each individual fishing vessel. By stationing a cluster of computers aboard the fishing vessel, the data can be processed directly from the source. This was, for instance done with StormCast. The result and the data itself can then be stored securely on board the vessel. The system can thus be programmed to raise a flag in the occurrence of abnormal events and only then make contact with authorities on mainland. Placing a central station on the mainland that collect such anomalies, can be used as a tool for inspection authorities for performing inspection of vessels.

Stationing all hardware aboard the fishing vessels may not be ideal due to the equipment being vulnerable, which can make it necessary to utilize a hybrid solution, where additional hardware is stationed on the mainland in addition to the stationed hardware on each fishing vessel. That way, lightweight data streams can be transported to the mainland via satellite broadband to be processed there. Larger volumes of data may be processed on the fishing vessel. Raw data can also be stored on-board as well.

3.2 Fault Tolerance and Security

As the title of the Dutkat paper states, the aim of the system is to catch illegal catchers. The disadvantage of moving computation to the edge is that it requires physical equipment to be stationed directly on the fishing vessel. In the case where a crew aboard a fishing vessel has illegal intends, the equipment is at risk of being damaged physically or compromised in some form.

Because of the underlying threats that physical equipment stationed on fishing vessels is exposed to, it is necessary to develop a system that is able to withstand failures.

On the other hand, there is nothing that can stop the crew from tampering with the equipment directly, or in-directly. TThe crew can simply unplug the power-source of the entire system, or throw all of the equipment overboard. Because of this, it is necessary to not only have a system that can withstand failures, but also is able to detect them and report them to authorities. This can be done by stationing a monitoring node on the mainland that communicate with the system over the satellite broadband. By continually messaging the system with regular time intervals, it can know if the system is currently functioning or not. If the node is unable to make contact with the system, the node can notify authorities that it has discovered an anomaly and recommend inspection. In the event of partial failure within the system, the system itself would have to

detect these failures, log them and inform the node.

The system is not only vulnerable to physical damage. The potential for digital system invasion also needs to be considered. This can be done by using strict authorization and encryption when passing messages between nodes. It is important to ensure that not only the data, but also weights from machine learning models. Machine learning models are vulnerable against adversarial attacks and it has been proven that only a slightly amount of added noise can completely change the classification of a feature vector [115].

3.3 Privacy-Preserving Data

One advantage of moving parts of the computation process to the fishing vessel is that data can be filtered at the vessel directly before sending it (or parts of it) to the mainland, which can increase privacy. Moving the computational process closer to voluminous data was one of the main motivations behind TACOMA in 1993 [108]. By storing private data within secure storage units directly on the fishing vessel, we can ensure that the sensitive data remains private.

There is currently ongoing research investigating file system support in Dutkat that can enable privacy-preserving analysis [32]. Dorvu is a novel DFS implemented as a Filesystem in Userspace (FUSE) application. Storage is implemented by mirroring the contents of a directory on a local file system. When Dorvu is mounted to a local folder, the mirrored folder will be populated with internal files and encrypted data files. These files can only be read through Dorvu, or by performing encryption manually.

Dorvu handles three file types internally. The file content is stored in data files. When a new file is created, a configuration file is automatically created with it. Each data file must have a configuration file in order to be visible in the directory listing. They are used for reference in configuration files and contain a JSON listing of group definitions, composed of a name and a list of public key SHA-256 signatures.

The experiments in Dorvu show that a FUSE based implementation does not significantly affect the performance for the use-case of optimizing in low-bandwidth transfer. The biggest performance bottlenecks in the file system is file versioning and metadata operations. The experiments also show that reading large files of 64-128 MB makes encryption the largest overhead.

By mounting the Dorvu file system to Áika, we can ensure that sensitive data is

stored in a privacy-preserving manner. By keeping the sizes of the checkpoints small, we can ensure that encryption overhead does not become a big problem. Since Dorvu is a DFS, it also comes with the advantage that checkpoints still can be retrieved, despite device failure. This makes it simpler to resume the computation process on a replica device.

The Dorvu file system is currently under development at The Corpore Sano Centre and it is currently not a finished system. The file systems requirements have been collected through dialogue with the developers. Dorvu should satisfy the following requirements:

- **Encryption:** Transparent encryption upon read/writes.

- **Access Control:** Users can provide public keys to determine accessible files. Access Control is enforced through encryption and inaccessible files are hidden from the directory listings.

- **Policies (Meta Code):** Policies can be pre-applied to files to guarantee policy compliance while readable. They can consist of arbitrary code that modify data before writing, or create new files.

- **Versioned File View:** Files can be created in multiple versions to comply with different policies and be accessible for different users. This should be transparent for the user.

- **POSIX compliance:** Dorvu should support automatic data production from arbitrary sources. This compliance involves supporting a regular file system interface and let users utilize Dorvus functionality purely through creation and modification of files and directories.

- **Data Distribution:** All files will be distributed among several nodes. Materialization at each node may be restricted by bandwidth. Dorvu therefore needs to be able to identify files of the highest semantic significance to expend bandwith efficiently. This should be applied to data distribution as well. Users should be able to express data distribution and bandwidth prioritization manually for their own files.

3.4 Laws, Regulations and GDPR

Dutkat is a surveillance system that analyzes data from multiple sensors. This may also include video surveillance data from cameras aboard fishing vessels and data generated by these cameras will be considered sensitive as long as

people are identifiable on the recording. Because of this, it is important to consider the legal aspect and take laws and regulations on surveillance and privacy into account when constructing the system.

Privacy

The right to privacy is considered a qualified, fundamental human right that can be found in many national legislations. Over 130 countries have constitutional statements regarding the protection of privacy. These countries are scattered across every region in the world. The Norwegian constitution states that:

Everyone has the right to the respect of their privacy and family life, their home and their communication.[1]

The statement has been a part of the Norwegian Constitution since it initially was formed in 1814. Furthermore, the United Nations Universal Declaration of Human Rights states that:

No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.[2]

The right to privacy is further enforced by EU through General Data Protection Regulation (GDPR). If it where to become mandatory to deploy Dutkat on fishing vessels in Norway, the system must ensure that the privacy of actors aboard each fishing vessel is rightfully protected.

GDPR is considered the most strict data privacy and security law in the world. It is governed by the European Union (EU). The Law enforcement directive (LED) is a special directive that is directed towards law enforcement. GDPR states, under article 35 that A *Data Protection Impact Assessment (DPIA)* has to be made as a part of *protection by design* principle [116]. A DPIA is a process for building and demonstrating compliance. The regulation of DPIA is necessary to consider in the Dutkat-project, as the system will likely process sensitive/personal data and non-compliance of DPIA can lead to fines from supervisory authorities.

In addition to GDPR, the European Commission presented the AI Act on 21 April 2021 as a novel regulatory framework for AI [117]. The act is currently a draft that seeks to codify a standard for AI in EU, requiring it to be legal and

1. Norwegian Constitution, §102, first sentence. Translated by Stortinget, 2018.
2. United Nations Universal Declaration of Human Rights, §12.

ethically and technically robust. Due to AI being used in Dutkat for analyzing surveillance data, the act needs to be taken into consideration, given that it may become a standard for EU countries in the future.

Pseudonymization vs. Anonymization

An important distinction to make when handling sensitive data is the difference between pseudonymization and anonymization. According to the Oxford Learners Dictionary, the term "anonymization" has the following meaning, with regards to computing:

the process of removing from internet data any information that identifies which particular computer the data originally came from [118].

The term "Pseudonymization", according to the Cambridge Dictionary, is defined as:

The process in which information that relates to a particular person, for example, a name or email address, is changed to a number or name that has no meaning so that it is impossible to see who the information relates to [119].

In GDPR, data that is pseudonymized will still be classified as sensitive and therefore also falls under GDPR laws. The issue lies in how the data is handled. If blurring faces in the images makes it impossible to identify a person today, but we do, however not know if technology in the future will be able to reverse this process. Anonymized data does, however not fall under GDPR, as anonymized data ensures that privacy is protected. One way to avoid GDPR is therefore to anonymize the the sensitive data. The challenge lies in determining if data is pseudonymized or anonymized.

Responsibility and Accountability

Another thing to consider is the responsibility and accountability in terms of the data and who owns it. Laws define that the people owning the equipment generating the data is responsible for it. Determining the owner of the data is an important decision that needs to be made when utilizing Dutkat in practice. If the equipment is ruled as government owned, the government is accountable for it.

If the system is enforced by the government, it needs to be ensured that vessels are treated equally, such that enforcement of using the system does not lead to discrimination.

There is currently a similar project regarding surveillance of fishing vessels is currently ongoing in Denmark, where fishing vessels have surveillance equipment of CCTV installed on-board fishing vessels in exchange for a larger fishing quota. This is, however a pure surveillance system and does therefore not perform any form of AI to analyze the data. Instead, the surveillance footage is analyzed by human personnel. For this case, the owners of the fishing vessels have been made the owners of the data. The project has, however been criticized for creating unnecessary suspicion of fishing and for violating the rights of fishermen by the Danish Fisheries Association [120].

There is also current ongoing research within the Corpore Sano Center on how laws of privacy should be dealt with in the Dutkat-project.

3.5 Incentive for Fisheries

Another important challenge to consider for surveillance systems like Dutkat, is that they will be met with resistance unless an incentive for using the system is provided to the users. It is also important that this incentive is sufficiently beneficial for the users. As discussed in the previous section, the Danish Fiseries Association has critiqued the surveillance system in Denmark, despite the project (at least for now) is voluntary and provides the vessels with larger quotas.

During the Fall of 2021, researchers from the Corpore Sano Center visited Hermes[3], a Norwegian fishing trawler. We found that employees of Hermes seemed positive to the project, although they highlighted the importance of not labeling anyone as criminals and they should therefore not be treated otherwise. They did, however consider the system as a potential for gaining increased revenue, as Dutkat could potentially be used for labeling fish as *sustainable* for better sales. Hermes already has camera equipment installed on-board and has used it for streaming to showcase their work[4]. Some employees, however, expressed concern regarding the cameras and that they would become uncomfortable if they knew that they where being monitored.

3. https://www.hermesas.no/
4. https://www.youtube.com/watch?v=e_2_f8vHaB0

3.6 Requirement Specification of Áika

Requirements engineering is a process in which required services for a system and the constraints which the system operates under are specified [121, p. 54-55]. Requirements are descriptions that state the system service and/or constraints generated during the requirement engineering process [121, p. 102-105]. After conducting an analysis of the Dutkat-project as a use case for Áika, we will now use it and the problem definition to state requirements for the system. The requirements are divided into two main categories: Functional and non-functional requirements.

3.6.1 Functional Requirements

Functional requirements are defined as statements of services that the system should provide, how the system should react to particular inputs, how the system should behave in particular situations or what the system should not do [121, p. 105-107]. In this section, we outline the functional requirements for the system.

- **DAG computation model:** The system must support general distributed DAG computation model composed of up to multiple pipelines that run coherently. The system must use some form of configuration format, such that the users of the system can specify the graph themselves. The distributed DAG model must support machine learning inference tasks to run within the DAG through pre-trained machine learning models. The system must also support that machine learning inference may be executed over multiple nodes.

- **Load Balancing:** The system must be able to support load balancing scheme as a part of the DAG model. The system must support load balancing specification from the configuration of any sub-graph within the main DAG.

- **Logging:** Any form of failure that is detected within the system must be logged, if possible. Ideally, the failures detected should also be classified at different levels such that authorities can easily make conclusions on whether a vessel should be investigated further or not.

- **Communication from external sources:** The system must support communication from external resources, such that the system can acknowledge that it is still up and running. In addition, the system must be able to provide its current state to an external source. The system should also be able to provide a history of registered failures as well.

- **Failure Detection:** The system must have a way of detecting failures that occur internally, in case an adversary agent tries to temper with the system software, or the hardware which the system runs on.

- **Failure Recovery:** The system must have a recovery scheme that enables it to recover from failures to the best extent possible. This includes software failures that can happen internally in the system and hardware failures (for example if a computer is unplugged from the rack).

3.6.2 Non-Functional Requirements

Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and constraints imposed by standards. They are often applied to the system as a whole rather than individual system features and services [121, p. 107-111]. In this section, we will outline the non-functional requirements related to the system.

Fault Tolerance

The system will operate in an untrusted edge environment, where it is difficult to reach the system if it fails partly, or as a whole. It is therefore important that the system operates with a fault tolerance factor that is as high as reasonably possible. In the case of failure, the system needs to recover as quickly as possible to the previous stored state in order to resume the computation process. The fault tolerance requirement is further specialized into reliability and availability:

- **Reliability:** Since monitoring of fishing vessels is used as a case for this project, the system may operate in an environment with low bandwidth, making it difficult to maintain reliable communication with external sources. The system should still be able to provide a response as quick as possible, given that either an external source tries to reach the system, or an internal system component attempts to reach another component within the system.

- **Availability:** It is crucial for the system to be available, as it may be used to both process voluminous amounts of data at the edge and communicate with authorities at the same time. Because of this, the system availability must be maximized to be as high as possible, in case an external source attempts to reach the system and to enable the system to process all the data. If the DAG computation fails to deliver analytical results due to

failure, some part of the system should remain intact and responsive to external sources.

Resilience

Loading machine learning models into memory can be time consuming, which might affect the recovery time negatively as well. The system should therefore support resilient schemes by redundancy where replicated components process the same data, simultaneously. If one component fails, the other component is still processing the data and thus ensures that the throughput, at least to some extent remains stable, despite failure. The system should support configuration of resilient algorithms like triple-modular redundancy, given that a user wants to add them to the system.

Confidentially

In the context of Dutkat, the system will operate as a distributed system that enables data to be processed directly aboard the fishing vessel itself. Since video surveillance data is a potential source for the system to process, the system must keep this data confidential by making sure that it remains on the vessel. In the case where an external source queries the system for data, it should only return meta-data, such as failure logs, the systems state, or anonymous data in order to maintain confidentiality of the actors aboard the fishing vessel. This is also the case if any of the machine learning algorithms detect suspicious behavior aboard the vessel as well.

Integrity

With regards to Dutkat, one of the main goals of the system is to process large amounts of data to detect suspicious behavior. It is vital that the data that flows through the system remains consistent and that failures do not affect this. The system should therefore be equipped with data integrity constraints, such that the data can remain consistent throughout its life cycle.

3.7 Summary

All aspects presented in this chapter are all important when considering the design and implementation of our system. When considering the case of the Dutkat-project, where fishing vessels are being monitored directly, it is impor-

tant to consider the actors right to confidentiality and therefore provide a system that does not leak sensitive data. In addition, the system cannot trust the actors either and therefore relies on fault tolerant schemes, such that the system can run continuously while the vessel is operating off-shore at sea. We suggest using a DAG computation model, so the system may support multiple types of computation graphs.

The next chapter outlines the design of Áika as a whole and of each component within the system.

/4 Design

This chapter provides the design and the overall architecture Áika. Section 4.1 provides an overview of the system as a whole, section 4.3 describes the design of the controllers, section 4.4 outlines the design of the different agents, while section 4.5 provides a description of an optional remote monitor, that can be used to monitor Áika from a trusted location.

4.1 System Overview

The system is designed to execute multiple distributed machine learning pipelines in a DAG computation format on edge clusters. It is particularly intended to be utilized in untrusted edge environments, where actors in the environment cannot be trusted and access to high-speed Internet may be limited. Limited Internet connections can lead to data being generated at a higher rate than the connection can transport. The solution is to move the analytical process to the edge, which is where the data is generated by sensors.

The systems requirements can be compressed into two main objectives. The system must:

1. Be fault tolerant in case of failure.

2. Support a DAG computation model.

The purpose of the DAG computation model is to enable complex distributed analytics to be done on the edge, while fault tolerance is necessary to ensure that the system does not fail at run-time.

To support the fault tolerance requirement, we design Áika as a hierarchical system, where a controllers is responsible for monitoring the remaining part of the system and execute recovery if a component fails.

As Figure 4.1 shows, the overall system is composed of multiple processes that communicate in a cluster. Each process has a specific role and it is not changed during run-time. The processes are as following:

- **Agent:** The agent is responsible for processing data. The agent can either fetch this data from disk itself, or receive data from other agents.
- **Local Controller:** The local controller is responsible for monitoring the agents that reside on the same physical node as the local controller itself. Each physical device has at least one local controller running. A local controller without any agent to monitor is referred to as a *replica*. This type of local controllers can be used for recovery if an entire physical node fails.
- **Cluster Controller:** The cluster controller is responsible for managing the entire cluster. It communicates with the local controllers to ensure that each physical node is running. If a physical node shuts down, the cluster controller is responsible for initiating recovery, either directly on the node where the failure occurred, or on a replica.
- **Monitor:** This is an additional process that is meant to be physically located on a trusted location, unlike the system itself. The monitor is responsible for communicating with the system to ensure that it is up and running. In the case of failure, the monitor may notify personnel, or authorities about this.

The different roles will be covered more in depth in Section 4.3, Section 4.4 and Section 4.5.

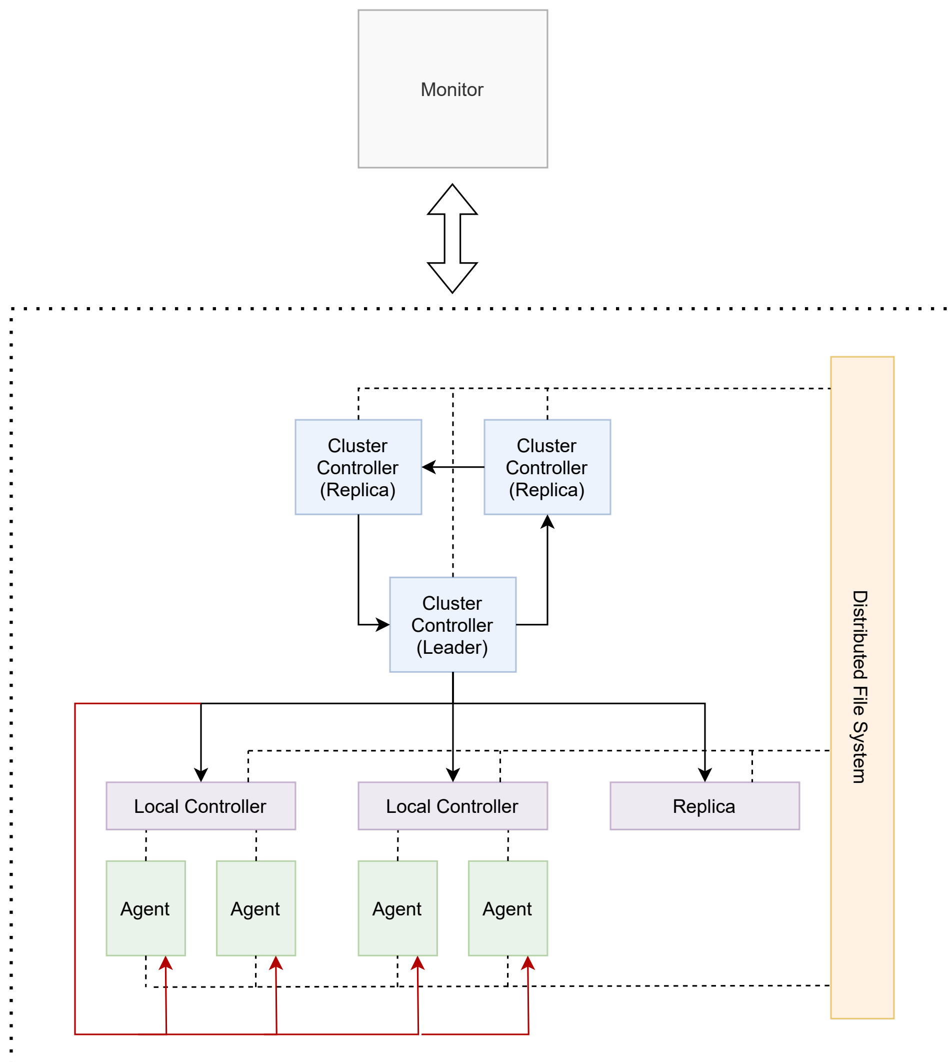

Figure 4.1: Áikas architecture. Arrows represents client/server communication. Red arrows represents communication that only may occur during recovery. The figure does not include communication between agents. A cluster controller communicates with local controllers residing within each physical cluster node. The local controller ensures that each agent runs. It may also report abnormal behavior. A monitor may also be mounted to the system from a trusted location, which the cluster controller may report any suspect behavior to. In addition to this, all nodes in the cluster are connected to a DFS that enables file sharing across the nodes. This is practical for recovery.

4.2 System Components Structure

The system itself is designed with a hierarchical structure. This also applies to each individual component in Áika as well. In Áika, each individual component is designed as a multi-threaded process. A main thread is responsible for initializing and monitoring child threads, where the child threads execute some type of behavior in the system. This can for example be to initialize a multi-threaded server, or communicate with another component in the system through a client, or execute some form of custom work. This varies from component to component. If any of the child threads fails, they will be restarted by the main thread.

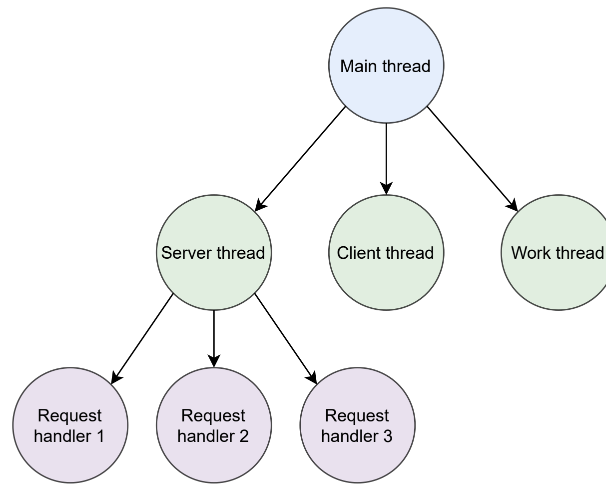

Figure 4.2: The general process structure of Áikas components. Each process is organized in a hierarchy of threads, where a main/parent thread starts the child threads. The child threads are restarted by the parent thread if they fail. Servers spawn multiple request handler threads to enable requests from multiple sources to be handled concurrently.

Each child thread is also a daemon, meaning that if the main thread shuts down, the entire process will fail. We chose to use daemons to ensure that servers are shut down if the main thread fails. It enables the recovery to be executed cleaner, as servers occupying ports no longer pose a risk during the recovery procedure. Figure 4.2 illustrates how processes can be organized into a hierarchy of threads.

4.3 Controllers

The system is implemented with a hierarchical multi-layered Controller/Agent design, where agents are managed by local controllers, residing on each physical

device. The local controllers are further managed by a cluster controller, which is responsible for managing the remaining components in the system.

4.3.1 Local Controller

The local controller is responsible for monitoring the agents residing on the same physical nodes. It ensures that each agent is running and in the case of crash failure, restarts the agent that crashed. It also logs the crash and the time of detection. In the event of a physical device crashing, a replica local controller will be responsible for restarting and recovering the local agents that crashed. This type of recovery process is initiated by a cluster controller.

4.3.2 Cluster Controller

The cluster controller is responsible for monitoring the entire cluster of computers that runs the system. The cluster controller has a controller/agent-relationship with the local controller, where the local controller functions as the agent. Whenever a remote monitor attempts to connect to the cluster controller, it must provide a response to it to ensure the monitor that the system is running.

If a local controller fails, the cluster controller will attempt to recover it. The cluster controller initiates node failure recovery if it fails to recover the failed local controller. This means that the configuration of the failed local controller is forwarded to a replica local controller.

The cluster controller is duplicated to avoid complete system failure in the case where it crashes. The cluster controllers are organized into a chain (see Figure 4.3) where each cluster controller responds to ping requests from their predecessor while pinging their successor. Each cluster controller has the full system configuration and therefore knows about all components. If the cluster controller in the chain fails completely and cannot be recovered, the predecessor will remove the cluster controller from its configuration and move to the next cluster controller in the chain.

If the main cluster controller fails, the duplicated controller that monitors the leader will attempt to recover it. If it fails the attempted recovery, it will instead become the new leader.

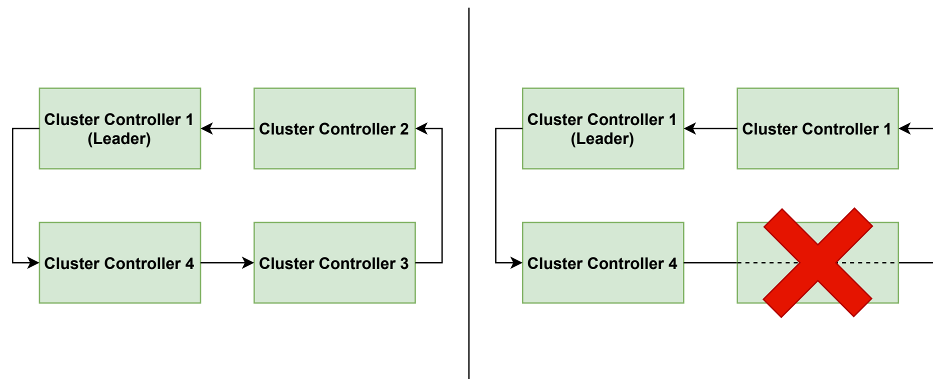

Figure 4.3: The cluster controller is replicated and connected in a chain.

4.4 Agents

The agents are responsible for working with and processing the data and they make up the core building blocks of the DAG computation model. The general task of each agent is thus to receive or fetch work (either from another agent or from file), then process data based on the work received before passing the results further ahead in the graph. Work items are transferred over client/server connections.

Figure 4.4 shows the general structure of the agents. A client or server thread is used to request or receive data from the previous agent. The thread continuously puts work items received on an input queue, which a work thread retrieves items from, before doing some type of work on the item. The result is put on an output queue. Another client or server thread is then used to forward the result to the next agent in the graph.

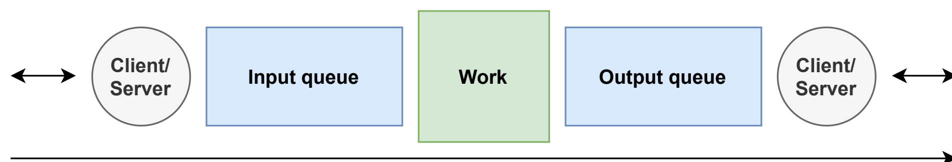

Figure 4.4: Shows the general structure of the agents.

Preserving data integrity despite failure is done through the use of persistent queues, as they continuously write items to file as they are being inserted into the queue. In the case of failure, an agent will always be able to resume from the previous checkpoint upon recovery, as long as it is connected to the DFS. By writing the persistent queues to file in the DFS, a replica local controller will also be able to resume the work if the physical node shuts down, since it also will be connected to the same file system. A mechanism in the queue enables

items to only be removed from file after work on the item has been completed and the result has been forwarded to the next queue. This mechanism is used both during work and during communication between agents to ensure that items are not lost.

The agents constitute the building blocks for a DAG, which is configured by the user. The DAG can be configured to be complex and, for instance consist of nodes that receive data from multiple sources, or passes data forward to multiple sources. A set of base agents has therefore been created, which uses different combinations of client and server threads at each end. Each agent has particular use cases where they are useful. Note that the figures of each individual agent has abstracted the persistent queues between client and work away for simplicity.

Left Worker Agent

The left worker agent is composed of a server thread on the left side, where items are received and a client on the right side, where items are forwarded. This is illustrated in Figure 4.5, where other agents can put items on the left agents input queue by making a request the left agents server. The left agent is responsible for forwarding the item to another agent itself by performing its own remote enqueue call to the agent.

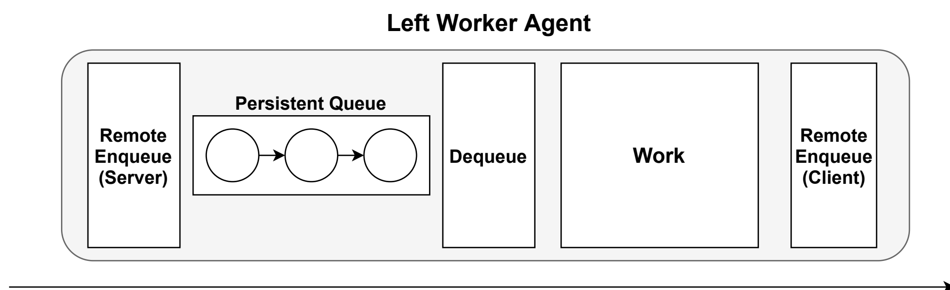

Figure 4.5: Left worker agent. A worker agent that dequeues messages from a local queue before the analysis process, but enqueues the result remotely.

This type of agent is useful in cases where data is received from multiple sources. The client on the right side enables the agent to forward the same item to multiple sources. An example use case for the left agent could be to use it as a voter agent for implementing N-modular redundancy.

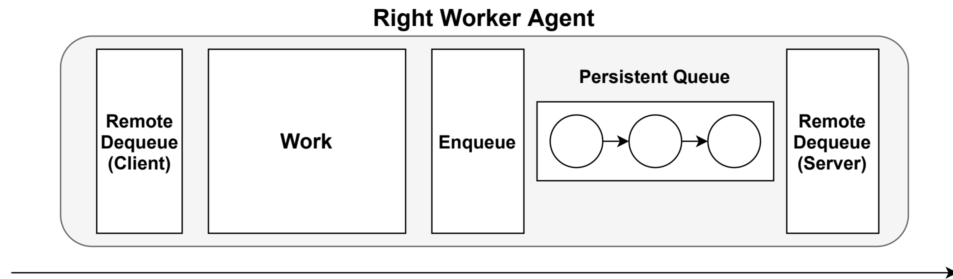

Figure 4.6: Right worker agent. A worker agent agent that dequeues messages from a remote queue before the analysis process, but enqueues the result on a local queue.

Right Worker Agent

The right worker agent is composed of a client on the left side and server on the right side (see Figure 4.6), This means that the agent fetches items itself from a single agent through a remote dequeue call, while items are forwarded when other agents requests them.

The server on the right side enables the agent to scatter items to different agents. This is useful in situations where load balancing due to upcoming computation heavy work. The consequence of using the right worker agent is that the client enables it to fetch data from a single source only.

Double Worker Agent

This type of agent contains servers both before and after the processing the item (see Figure 4.7). This makes the agent completely passive, as messages are only received and forwarded through requests from other agents.

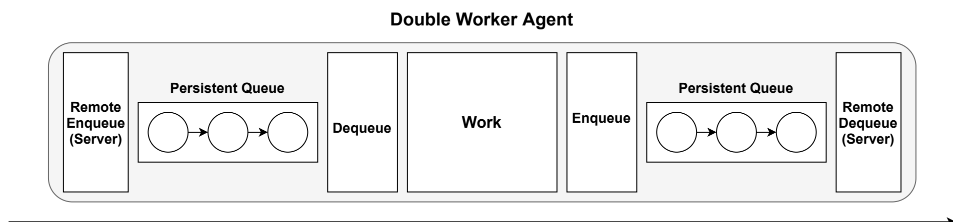

Figure 4.7: Double worker agent. A worker agent containing servers both before and after processing the item.

This type of agent can be useful in cases where it receives messages from and scatters messages to multiple sources.

Initial Worker Agents

The purpose of the initial agent is to fetch data from some location in a custom manner (implemented by the application developer), before forwarding it to the next pipeline stage. The data flow is illustrated in Figure 4.8. It is meant to be used as the first stage in the pipeline. Initial agents can either use a client or a server to forward items further into a pipeline. In Section 6.4 we demonstrate how the initial server agent can be used for load balancing for counting words in a textual document. The experiment is further detailed in Appendix C.

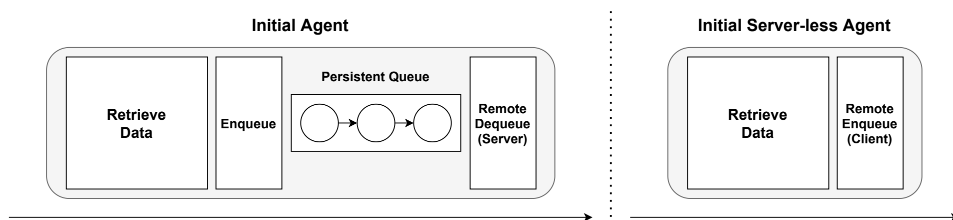

Figure 4.8: Initial worker agents. This type of agent is used to initiate one or several pipelines. This is done by having the agent continuously retrieve data from a source and then forward it to either a local (see left figure), or a remote (see right figure) queue.

Final Worker Agents

The purpose of the final worker agent is to do the final work on an item at the end of a pipeline within the DAG. Because of this, it does not have any output queue, or client/server thread after work. The final agent can utilize a client for fetching items, or a server for receiving them. These two types of final worker agents are illustrated in Figure 4.9.

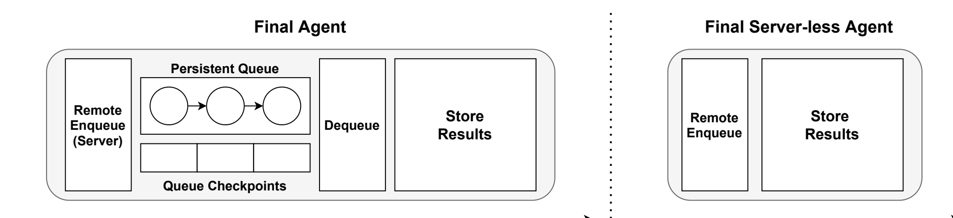

Figure 4.9: Final worker agents. This type of agent is used to finalize one or several pipelines. The agent retrieves the end results from either a local or a remote queue, then handles the result in some custom manner.

Queue Agent and Server-less Agent

The queue agent is only composed of one scheduling queue, which leaves the responsibility of enqueueing and dequeueing messages entirely up to other agents. It is passive, like the double queue agent and it is also useful in similar cases where work is not required to be done on the item in between. It can be used as a collection point for data from multiple sources that is afterwards scattered.

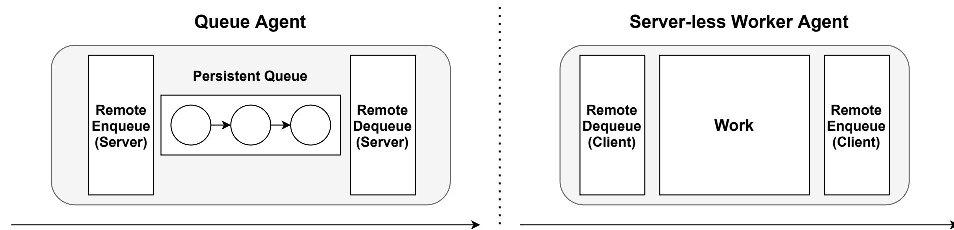

Figure 4.10: Queue agent and server-less worker agent. The agent to the left contains a single queue without any analysis. The agent to the right does not have any servers and receives items by making requests through clients on both sides.

The server-less agent contains only clients and is therefore responsible for both fetching items from another agent and forwarding items after processing (see Figure 4.10). In Section 6.5, we demonstrate how a server-less agent can be used to retrieve items from a load balancing queue on another agent and forward the item to feature extraction models on multiple agents to increase performance. The DAG design is also elaborated in Appendix D.

4.5 Monitor

The remote monitor resides at a physically remote location compared to the system and is responsible for communicating with the system, potentially over low bandwidth. The reason for using a remote monitor is that it is not possible to fully control the physical equipment completely in an untrusted edge. We cannot assume that the equipment is held entirely safe, as people may want to harm the physical equipment, or interfere with the signals. The system may also be difficult to reach, physically.

The remote monitor is used as a safety device that resides within a safe location that can monitor that the system is running. By continuously communicating with a cluster controller in the system, the monitor can receive information about the system, that is: which physical devices that are running, if any known

crashes has occurred, or additional information regarding the system. If the remote monitor fails to reach the system, it may classify this as an abnormal event and report it to authorities. The authorities can then themselves consider if they want to investigate the issue further and potentially inspect the location where the system resides.

4.6 Summary

This chapter has presented the design of Áika. The system is designed with a hierarchical infrastructure of controller/agent relationships, where a cluster controller monitors and manages the system as a whole and local controllers monitor and restart agents running on the same physical node. Each component is multi-threaded with a hierarchical structure as well. The cluster controller is replicated in a chain to avoid single point of failure. The agents are designed to use different combinations of clients and servers to receive and send items. A monitor that is physically located on a safe place is used to verify that the system remains operable and running.

The next chapter builds on this chapter and outlines the implementation of Áika.

5

Implementation

This chapter presents implementation details of Áika. Section 5.1 presents specific details for the system as a whole, such as programming language, libraries, etc. Section 5.2 outlines testing of Áika. Section 5.3 covers the implementation of the cluster controller. Section 5.4 covers the implementation local controller. Section 5.5 covers the implementation of the base agents. Section 5.6 provides the implementation of the monitor. Section 5.7 describes the implementation of crash reporting, before covering the implementation of the killer in Section 5.8.

5.1 Implementation Specific Details

Áika and all its components is written in Python 3. Python is a high-level scripting language, which makes it excellent for building prototypes. In addition, it has an active and open community that provides Python developers with specialized libraries (like the persistent queue library that we use). As of December 2021, Python remains at the very top of the list of popular programming languages, with a share of 30.21%, according to PYPL [122]. Python is particularly relevant for Áika, due to the language support for scientific programming [123] and powerful deep learning libraries.

All internal and external client/server communication is carried out via TCP

Socket Servers. This includes communication between agents, cluster controllers, local controllers and the monitor. TCP is used as a protocol due to its reliable transmission, which makes it a better option for transporting important data compared to User Datagram Protocol (UDP). The state of the service is check pointed regularly through the use of persistent queues.

The following Python libraries have been used to implement the core of our system:

- **abc:** For implementing abstract base classes.
- **json:** For parsing JSON configuration files.
- **pickle:** For serialization of data.
- **socket:** For creating TCP clients.
- **paramiko:** For setting up SSH clients to connect to cluster nodes.
- **unittest:** For testing the functionality of units.
- **threading:** For enabling multi-threading.
- **socketserver:** For implementation of TCP servers.
- **persistqueue:** For persistent queues between agents.
- **setproctitle:** For setting and getting the process title of each agent manually.

5.2 Testing

Verifying that the underlying functionality of the agents works, is done with unit testing as a method for validation. The unit tests are implemented through Pythons unittest library. Each agent is implemented as a package and each package contains a test file, which contains relevant tests for the specific agent. This also includes mock servers and clients for scheduling data items for processing. An overlying script is used to execute these tests through Pytest. Since the implemented agents themselves and the tests utilize multi-threading, each test is executed in complete isolation from the others to avoid unexpected behavior.

5.3 Cluster Controller

The cluster controller is implemented as a class composed of multiple threads. Startup and recovery of both local controllers and cluster controllers is implemented similarly. The cluster controller distinguishes between three types of recovery:

- **Recovery by SSH:** If the local controller or cluster controller is located in a remote cluster node, the cluster controller will establish connection to the node through an SSH client. The controller is then started by executing the startup script on the node.

- **Direct recovery:** This routine is performed if the local controller or cluster controller resides on the same physical node. In this case, the startup script is executed directly.

- **Recovery by replica:** If the cluster controller discovers that a physical cluster node has failed, it will execute recovery by replica, given that a replica is available. It does this by forwarding the configuration of the failed local controller to the first replica in the list of available replicas through a TCP request. The replica will then start the agents that failed. Since the persistent queues are stored in the DFS, the agents can resume from the previous checkpoint. The cluster controller will also notify agents with agent that communicated with the failed agents about their new location, so they can re-establish communication. If there is no replicas available the system goes into system failure mode.

Recovery is carried out if the controller fails to respond to the cluster controller for a configured amount of attempts.

5.4 Local Controller

The local controller is also implemented as a class composed of multiple threads. It contains a server that is used to respond to requests from the cluster controller. Upon startup the local controller will always check if the agents are running before starting them. The reason for this is that, in the case of recovery, the local controller may have crashed without the agents crashing. Attempt to recover the agents would in this case lead to agents being duplicated.

Recovery of agents is executed as direct recovery in the cluster controller. The local controller does this because it only monitors agents that reside on the same node as itself. The local controller monitors the agents by assigning each

agent a unique ID (process name) upon startup. This ID can be specified in the configuration. By checking if the unique ID is running, the local controller can know if the agent is running. This approach ensures that failure is detected quicker, because there is no direct communication between the local controller itself and the agent. One disadvantage with the solution is security risks, as someone with access to the physical node could start a process with the same name as one of the agents before killing it. This could trick the local controller to believe that the agent is still running.

5.5 Agents

Each base agent is implemented with an object-oriented approach. The application developer creates an agent by creating a new class that inherits one of the base agents described in Chapter 4 (see Section 4.4). When an agent receives an item from another agent in the DAG, it passes through the *process_message* method, which has not been implemented by the base agents. It is up to the application developer to specify what this method does. An example can be seen in Listing 5.1, where the *process_message* method increments the item before passing it on.

Listing 5.1: A simple, customized agent implementation. The agent receives a configuration string in JSON format when it is initialized. For this example, the *item* argument from the process_message method comes in the form of an integer, and the method simply increments it before returning the result.

```
1 class CustomAgent(LeftAgent):
2     def __init__(self, configuration):
3         super().__init__(configuration)
4
5     def process_message(self, item):
6         result = item + 1
7         return result
```

The advantage of this is that the entire communication between the agents are completely abstracted away from the application developer. Another advantage is that the application developer may use any supported Python library to process or analyze the item, for instance Tensorflow [94], Pytorch [124], OpenCV [125], Pandas[126], or SciKit-Learn[127]. This enables the application developer to mix libraries if the circumstances requires it.

Each item that is passed forward in an agents pipeline contains a list of pipeline IDs. The reason for this is that each item can belong to multiple pipelines that initially start out with a common path before branching out. The agent uses

a simple algorithm that detects whether the pipeline IDs are heading in the same direction through the agents own lookup table. It enables the application developer to specify branches in the DAG through the configuration file.

If the algorithm detects a branch, the same item is forwarded to more than one different agent, with modified pipeline IDs. The aim of the algorithm is to avoid item duplication in early stages of the pipeline where the computation result will be the same for all pipelines that contain the item.

Áika uses a static configuration format in order to construct the agents forming one or multiple DAGs, as well as replication nodes and cluster controllers with replication. The configuration format is explained in more detail in Appendix A. An example of how a complex DAG can be constructed with the configuration format can be found in Appendix B.

5.6 Monitor

The monitor is implemented as a simple program with a single TCP client that is used to connect to the system. This program sends a single request to a cluster controller, given that the host and port provided are correct. The monitor can send four different types of requests, which are shown in Table 5.1.

Request Type (ID)	Description
1	Simple ping request to acknowledge that the system is running.
2	Get the current state of the system. Retrieve the current configuration.
3	Get recent log records. Use record type values and timestamp to filter out records.
4	Shut down the system with immediate effect. The leader cluster controller forwards the signal to the other nodes.

Table 5.1: Monitor Request Table.

When the monitor queries the system for any information, only the request ID is sent along with meta data that further specifies the request. Since the system may operate in an environment with low bandwidth, it is necessary to keep messages as small as possible when sending information to external sources, like the monitor.

5.7 Logging

The systems ability to operate in an environment that cannot be trusted makes it necessary to log any suspicious behavior in order to verify that the system remains operable and trustworthy. It is also necessary to maintain a history of failures or other types of abnormal behavior within the system for investigation purposes. If the monitor queries any of the leaders for the log data, they can retrieve them. It is then up to the monitor to decide if the behavior occurring can be characterized as suspicious enough so that authorities should be contacted.

Log Record Type (ID)	Description
1	A thread crashed unexpectedly and had to be restarted.
2	A server failed to respond to a request from a client.
3	An agent unexpectedly shut down and had to be restarted.
4	A local controller failed and had to be restarted.
5	A cluster controller failed and had to be restarted.
6	An entire physical node has failed. Recover to another node.
7	A suspicious request was received from some source.
8	System Failure. The system either shut down unexpectedly, or failed to respond to the monitor.

Table 5.2: Failure Log Record Table.

To distinguish between different types of abnormal events within the system, we split log records into different types, where each type has a unique identifier. A description of each identifier can be found in Table 5.2. As the table shows, Áika generally distinguishes between eight categories of log records.

Since more or less every component in Áika is multi-threaded, there is a consistent risk that one of the threads might fail without shutting down the process itself. If one of the child threads fails, the main thread will restart it and log the behavior. If the main thread fails, the entire process will shut down, since all child threads are daemons of the main thread.

If an agent process fails, it is the responsibility of the physical nodes local controller to restart the agent. The local controller also logs the failure, if the agent had to be restarted. If a local controller, or an entire physical node fails, the cluster controller is responsible for recovering the local controller and writing the log to file.

The system will also log any communication failure that is detected and any suspicious requests (for instance requests in unexpected formats) received. The entire system may also shut down, due to physical nodes failing, leading to the system running out of replica components to use. This behavior will also be logged, given that any of the cluster controllers detects it before being shut down. The monitor may also consider the system as failed if it fails to get a response from the system within a given time frame.

The log records are stored within a single table in an SQLite3 [128] database that resides in the DFS. SQLite3 is a relational database management system (RDBMS) with a dynamic and weakly typed syntax. This combined with its lightweight features makes SQLite3 a good choice for prototype systems like Áika.

5.8 Killer

The purpose of the *killer* is to kill cluster controllers, local controllers and/or agents, such that Áikas ability to recover processes can be investigated. The killer is a process that forcefully kills selected processes in the system during a run-time at a pre-configured rate. It does this by selecting a random component from a pre-determined list of components at every cycle. If the component is located on the same host as the killer, the killer simply invokes a pkill command based on the unique process name of the component. If the component is located on a different host than the killer, the killer instantiates an SSH-client that is used to reach the host. The pkill command is then invoked through the SSH client.

Figure 5.1 illustrates the killer as a separate process that has knowledge about all the components in the system. The killer gains knowledge about the system by parsing the configuration file of the system upon initialization. Because of this, the user that invokes the killer can select custom components from the system that the killer knows about. This is done by removing specific components from the configuration before invoking the killer with said configuration. The killer can also be configured to only kill specific types of components, like agents, local controllers, or cluster controllers.

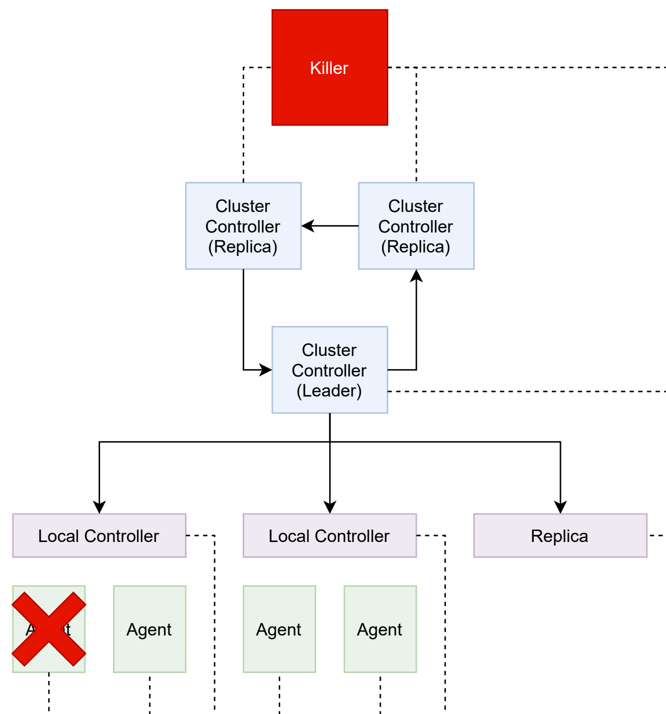

Figure 5.1: Illustration of the Process Killer. This continuous running process reads the system configuration and then kills a randomly selected system component every N seconds, where N is a configurable number. The killer can also be configured to kill only a certain type of system component.

5.9 Summary

This chapter outlines implementation details of Áika. The system is written in Python 3, due to the language high-level syntax combined with its support for executing machine learning tasks through powerful libraries. Áika is implemented as a distributed system composed of multi-threaded components. The components uses TCP to communicate with each other. Áikas agents are implemented with an object-oriented approach that enables the user to use inheritance to implement custom agents that perform customized work. These components have been tested with unit tests to ensure that their program flow works as expected. The system has support for responding to external sources, like a monitor and provides meta-data to it regarding system state and the history of detected failures. The system has implemented logging with a simple scheme to SQLite databases, where logs are inserted with an ID that describes the type of failure that has been detected.

The next chapter evaluates Áika, and reviews the non-functional requirements of the system based on the experimental results.

6 Evaluation

This chapter outlines an evaluation of Áika. An overall experimental setup is presented in Section 6.1. Section 6.2 outlines measurements on minor, but important components of Áika. Section 6.3 evaluates the system through end-to-end measurements. Section 6.4 evaluates the systems performance through a MapReduce inspired distributed word counter. Section 6.5 evaluates the systems performance through a distributed deep feature extractor. The non-functional requirements are then reviewed based on the experiment results and the discussion made from them in Section 6.6.

6.1 Overall Experimental Setup

The experiments are carried out at the UV-cluster, a computer cluster owned by the Department of Computer Science at UiT The Arctic University of Norway. The UV-cluster uses the Rock (version 7.0) operating system, which is based on CentOS 7. Each computer node runs the Ext4 file system locally and the home catalog is shared from the UV cluster server through the Network File System (NFS). The expected ping is estimated to be at approximately 1 millisecond and the transfer rate is approximately 100 MByte/s, which is normally what the Gigabit cards are able to handle.

The experiments are performed on the compute-6 nodes, exclusively. The

compute-6 nodes are a set of homogeneous nodes on the cluster. The compute-6 nodes consist of 55 Lenovo P310 computers with one Intel Core(TM) i7-6700 @ 3.40GHz with 4 cores, 32GB (4 x 8GB) RAM and a Nvidia GM107GL [Quadro K620] GPU, each. The Intel Core(TM) i7-6700 CPUs has a base clock frequency of 3.40GHz, but also supports turbo boost functionality, where the clock frequency can be as high as 4 GHz. Due to the potential dynamic change in frequency, we carry out the experiments multiple times and compute the average result along with the standard deviation of the results.

6.2 Micro-Benchmarks

Gaining insight into the smaller components of the system is important. We have therefore made a few micro-benchmarks within Áika to get some insight on how the time spent doing different tasks affects the performance.

Micro-benchmark	Time (seconds)
Local Controller Initial Startup	0.3 seconds (first local controller)
Local Controller Further Startup	0.15 seconds (per local controller after the first)
Agent Startup	0.02 - 0.04 seconds
Pass integer item from agent to agent	0.007 - 0.009 seconds

Table 6.1: Results from micro-benchmarks.

Table 6.1 contains a few micro benchmarks within Áika. We experienced that it takes the cluster controller approximately 0.3 seconds to start a single local controller on a separate node with an SSH client. After setting up one local controller, the time increases with approximately 0.15 seconds per additional local controller. The local controller, however spends approximately between 0.02 and 0.04 seconds to start an agent. This demonstrates that the local controller not only can be used to off-load the work load of the cluster controller, but also can manage to recover agents in a shorter amount of time, leading to more efficient recovery procedure. This is especially the case if agents shut down often.

Passing integer items from one agent to the next agent in a pipeline, takes approximately 0.008 seconds from the previous agent to the next. During this time, the item moves through two persistent queues and one TCP stream. One interesting finding is that the time spent passing an item from agent to agent

seemed to increase as the item moved further into the pipeline, but this could also be purely coincidental.

6.3 End-to-End Evaluation

Insight into how information flows through the system is necessary in order to understand the systems drawbacks and how failure affects the system. This can be done by measuring the throughput of the system within regular time frames at the end of a pipeline.

Setup

Since the system uses persistent queues to store information between computation steps, it is necessary to measure how the persistent queues affect the throughput of the system. It could also be interesting to observe how crashes in the system affects the performance. Both of these aspects can be demonstrated by constructing a system run-time composed of agents that are lined up in a pipeline.

The agents in the system can be split into three types:

- **Initial Agent:** Responsible for initializing data. This is done by forwarding integers into the pipeline.
- **Worker Agent:** Responsible for retrieving integers, increment them and forward them further into the pipeline. The worker agent may also sleep after the integer has been incremented to simulate work time.
- **Final Agent:** This is the final destination of the data in the pipeline. The agent is responsible for receiving the data and increment a counter of the number of requests received. A separate thread measures the number of requests received every fifth second and write the number to file.

The system is configured as a single pipeline, composed of one initial agent, one final agent and five worker agents.

The results are obtained by running the system run-time with the pipeline configuration over the course of 1,200 seconds (20 minutes). The number of requests received is counted at the end of the pipeline. The counter is reset every 5 seconds. The time frame of 5 seconds is used to hinder the thread that writes the result to disk from affecting the performance of the system. In

addition to this, the moving average of series with highly varying values is also computed. For all cases where this is done, we use a sliding window with a size n=5.

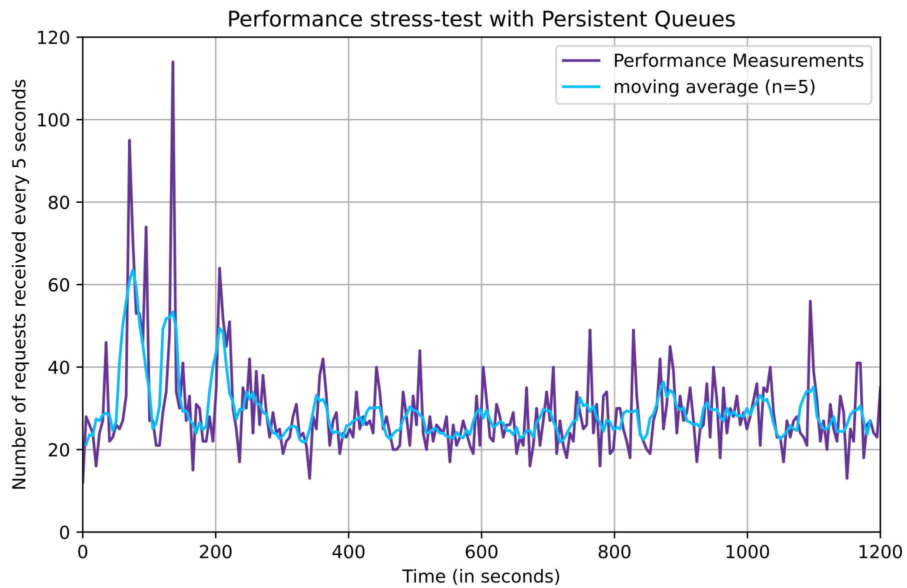

Figure 6.1: The obtained results from stress-testing the system with persistent queues over the course of 1,200 seconds. Number of requests are measured and reset every 5 seconds. The moving average is computed with a window size n=5.

Results

Figure 6.1 illustrates the number of requests received at the end of the pipeline every 5 seconds over the course of 1,200 seconds (20 minutes). The plot demonstrates the systems maximal performance with the use of persistent queues, where each worker in the pipeline receives an integer that is simply incremented before it is passed on without sleeping. From the plot, we can also observe that the system has a high throughput in the initial stages of the process, before it becomes relatively stable at around 250 seconds, where the number of requests processed ranges between 20 and 40. This demonstrates the systems maximum throughput without considering the workload.

Figure 6.2 illustrates the same measurements as Figure 6.1 with the same setup, but utilizes in-memory queues in the agents instead of persistent queues. The in-memory queue is configured to hold maximum 100 items. Despite having an identical configuration as the system from Figure 6.1 (aside from the use of

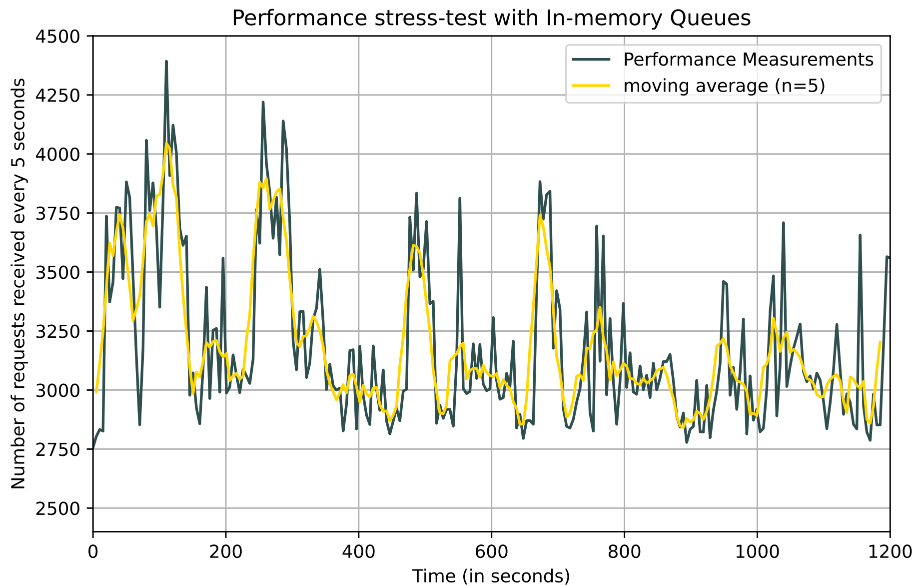

Figure 6.2: The obtained results from stress-testing the system with in-memory queues over the course of 1,200 seconds. Number of requests are measured and reset every 5 seconds. The moving average is computed with a window size n=5.

in-memory queues), the system clearly has a higher performance when using in-memory queues. This is not unexpected, as any form of computation on a data item should be performed faster when the item is fetched volatile memory, instead of disk. The plot does however illustrate that the performance with in-memory queues can be up to 100 times better compared to the system with persistent queues.

Figure 6.3 illustrates the results obtained when using a system configuration similar to the previous ones, but each worker agent now sleeps for 0.9-1.1 second in addition to incrementing the integers before propagating them forward in the pipeline. The purpose of sleeping for 0.9-1.1 seconds is to simulate work. The plot contains results from both using persistent queues (top) and in-memory queues(bottom). Contrary to previous measurements, the persistent queue configuration performs better when more time consuming jobs are performed, compared to the configuration with in memory queues. The performance also seems to be more stable, ranging mostly between 3 and 5 requests per 5 seconds for persistent queues, but between 2 and 5 requests per 5 seconds for in-memory queues.

The results illustrate that the overhead created when writing the queues to

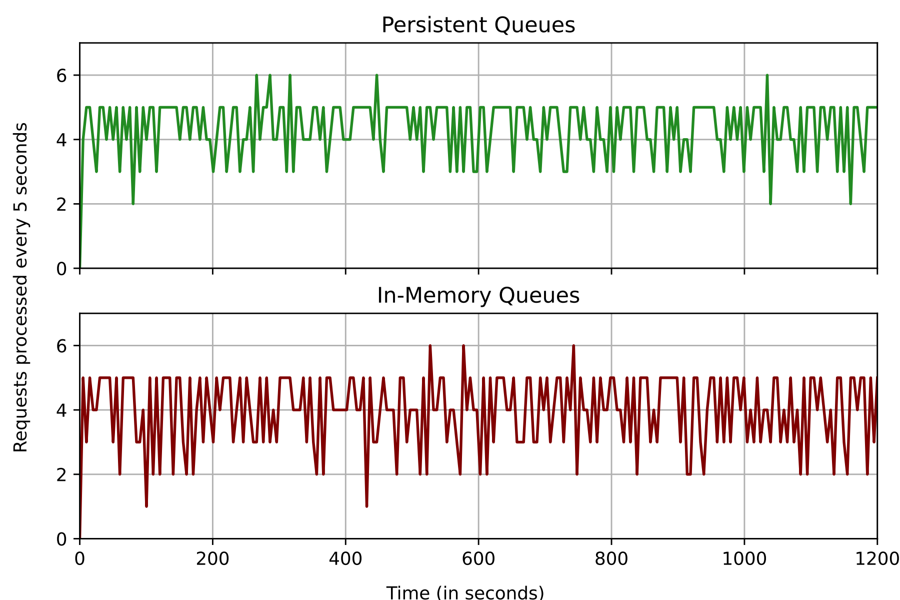

Figure 6.3: Results obtained when measuring the number of requests received where worker agents in the pipeline sleep for one second after processing an item. Measurements are done with persistent queues (top) and in-memory queues(bottom). The number of requests received are measured and reset every 5 seconds.

file continuously becomes negligible with regards to the overall performance. Despite this, the queues for these measurements only contain integers. With such small values the time spent writing to file will be a lot less compared to e.g. images. It is therefore important to consider the size of the data when working out a system configuration with optimal performance for carrying out the desired task.

Figure 6.4 illustrates the different performance results obtained when stress-testing the system with and without the killer deployed. The killer is configured to kill a random worker agent or initial agent every 15 seconds during the entire run-time in the run where it is deployed. Both measurements are similar both in terms of the series shape and the performance. When the killer is deployed, the measurements does, however become slightly worse. Despite the agents being killed every 15 seconds, the performance still remains relatively similar in terms of stability. Agents being killed every 15 seconds and the number of requests received are measured every 5 seconds means that the performance should decrease in every 3rd measurement. This could explain the lack of visible dips in performance throughout the run-time, overall.

Figure 6.4: The results obtained when measuring the number of requests received during stress-testing over the course 1,200 seconds without (top) and with (bottom) the killer deployed. The number of requests received is measured and reset every 5 seconds.

Queue type used	Min Value	Mean value	Standard deviation	Median Value	Max Value
Persistent Queue (No sleep)	12	29.09	11.47	27	114
In-memory Queue (No sleep)	2757	3184.43	343.37	3061.5	4393
Persistent Queue (Kill every 15 sec.)	9	25.42	12.9	23	89
Persistent Queue (Sleep 0.9-1.1 sec.)	0	4.38	0.88	5	6
In-memory Queue (Sleep 0.9-1.1 sec.)	0	4.01	1.11	4	6

Table 6.2: A summary derived from the continuous performance experiments explained previously. The table contains the minimum, maximum, mean, median and standard deviation values for each measurement performed throughout their entire run-time of 1,200 seconds.

Table 6.2 contains a summary of values related to the experiments on continuous performance results. The measurements in the table confirms the overhead created when using persistent queues becomes negligible when the workload is increased. Not only are the results more stable for persistent queues (standard deviation 0.88 vs 1.11), it also performs better overall with an average of 4.38 requests processed per 5 seconds compared to 4.01 seconds for the in-memory queue. The performance does increase slightly, from 29.09 during the stress-test, to 25.42 during the same test with the Killer component deployed. The stability of the system also seem to decrease slightly, as the standard deviation goes from 11.47 to 12.9 when the killer component is deployed.

Discussion

It is important to note that despite the performance gain from using in-memory queues, the potential consequence of this is that the system remains unable to recover properly from faults if any component were to shut down during the run-time. A local controller may resume an agent that crashes, enabling it to continue working. When using in-memory queues the data stored will, however be lost if the agent itself were to crash. This would require a complex recovery routine, where the agent initializing the pipeline would have to re-retrieve the specific items that has been lost during the crash and propagate them through the entire pipeline, all over again. In the case where the system operates in a trusted environment where the run-time does not last long or is communication-intensive, it could benefit the user to use in-memory queues in favor of persistent queues.

The results from Figure 6.4 illustrate how the system manages to remain stable in terms of performance, despite worker agents being regularly killed. The local controllers are constantly monitoring the agents and could therefore explain why the system manages to remain stable. Despite this, the performance of the system is still affected by this and that having processes killed between small intervals could be fatal for the systems performance. This is one of the reasons for monitoring and reporting every crash that is detected in the system.

6.4 Distributed Word Counter

To evaluate if Áika can handle simple analytical tasks, that does not necessarily involve highly complex computations, we deploy a distributed application for counting words from a text file.

Setup

The distributed word counting application follows a MapReduce inspired design. It counts the words provided in a single, given text file. The data set used is a generated data set based on the 1,000 most common words, according to Education First [129]. The data set is generated by loading the 1,000 most common used English words into a list and randomly select words from the list until the size of the data set is large enough.

The agents in the system can be split into three types:

- **Split worker:** Is responsible for dividing the data set into parts that each individual mapper can work on.
- **Map worker:** Tokenize the text it has been given, iterate over and count every word. The counted values are stored in a simple Python dictionary.
- **Reduce worker:** Combine all dictionaries given from the mappers into one, total dictionary.

The system is configured to use a single splitter to initialize the word counting, between 1 and 16 number of mappers and a single reducer that combines the results from each mapper. The system is configured with static load balancing, where each map worker fetches one job from the split workers queue, respectively. This also means that each worker perform their work in one single iteration. The experiment is repeated 15 times so that the stability of the system can be measured as well. See Appendix C for a more in-depth explanation of the word counter setup.

Results

The results obtained from running word counting on a 100 MB and a 300 MB data set can be found in Figure 6.5. The figure indicates that the system is able to scale well in terms of handling embarrassingly parallel algorithms, due to the slope having an expected concave shape. Despite this, the slope starts to flatten at around 8-9 seconds. One reason for this is that creating more map workers is more time consuming for the system, as the workers are instantiated local controllers, which are further instantiated from a single cluster controller. In addition to this, the use of a single reducer could lead to a minor bottleneck, since it would become responsible for combining all word counting maps, alone.

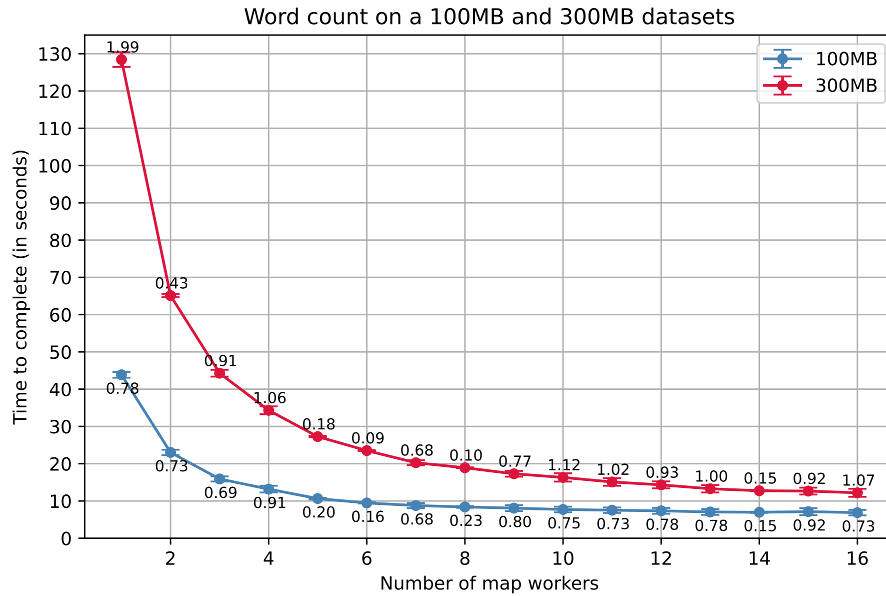

Figure 6.5: The obtained results from counting the words in a 100 MB data set. The error bars displays the standard deviation and the value is displayed above each data point.

Discussion

As the micro-benchmarks shows, the cluster controller spends approximately 0.30 seconds starting up a single local controller. Afterwards, it spends approximately 0.15 seconds extra for each additional local controller. The local controller, on the other hand, spends approximately between 0.02 and 0.04 seconds to start a single agent. This in total makes up over 1 second of overhead when the number of map workers i 6 or more, which partly explains the overhead seen in the graph. Furthermore, it is necessary to take communication overhead, file reading and writing (due to the persistent queues) and the startup wait time that each agent has into consideration. A part of the overhead could also be explained by the fact that each mapper loads their entire partition into memory before mapping. When the size of the data set increases the overhead may therefore also increase.

6.5 Distributed Deep Feature Extraction

The purpose of the Distributed Deep Feature Extraction experiment is to evaluate if Áika is able to perform machine learning tasks with different approaches.

Another purpose for the experiment is to investigate the systems ability to scale with these different approaches.

Setup

We use the STL-10 image data set to perform the feature extractions [130]. The STL-10 data set is inspired by the CIFAR-10 data set [131], although there are some differences between them, such as the images having a higher resolution (96x96 instead of 28x28). The feature extractions are performed on the entire data set of 113,000 images in batches, with 500 images per batch. The use of batches could give an indication on how the system performs with larger scaled images.

The system extracts features from the data with the use of three pre-trained Keras [132] models. The system is configured and tested with two different approaches:

1. All three models are loaded into N workers, which perform feature extraction on all three models, sequentially.

2. The three models are distributed among 3 workers, such that the feature extraction process can be done in parallel.

The setup is described in more detail in Appendix D[1].

Results

It takes approximately 3,336 seconds (almost 56 minutes) to perform feature extractions on all 113,000 images when the feature extractions on a single machine. The measurement on a single machine has been repeated three times.

The results obtained when performing feature extraction on 113,000 96x96x3 images in batches of 500 images per batch, using the 2 approaches described previously can be found in Figure 6.6. Both approaches seem to scale at a similar rate in terms of number of sub-graphs. The distributed feature extraction approach is clearly more efficient, but also requires more workers.

One interesting finding is that the sequential feature extraction approach

1. Due to issues with compatibility between software versioning on the cluster, we had to perform the experiments using the CPUs of the computers instead of the GPUs.

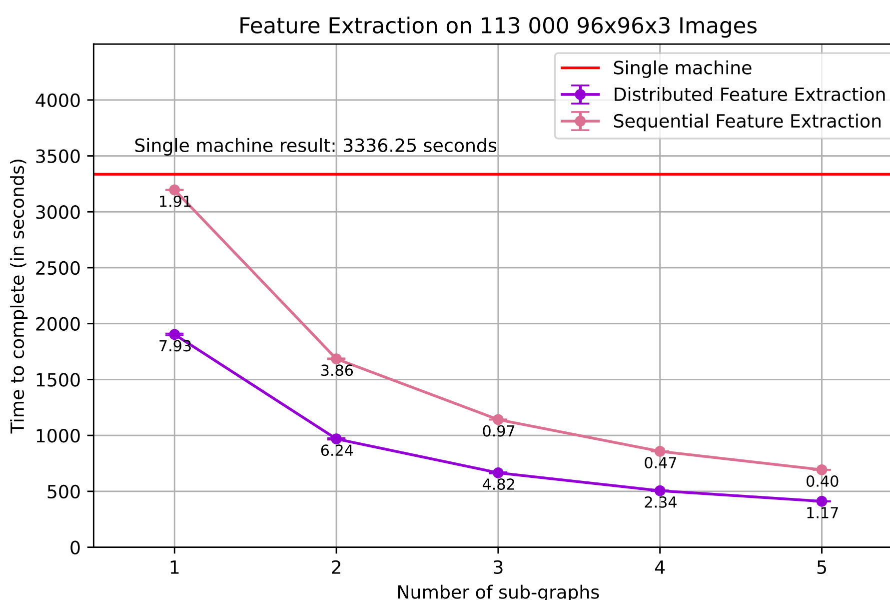

Figure 6.6: The obtained results from doing feature extraction with pre-trained VGG-16, DenseNet-121 and ResNet-50 models, where the models are either distributed among three workers (Distributed Feature Extraction), or put in a sequence on a single worker (Sequential Feature Extraction). The images have been processed with a batch size of 500.

performs better compared to the single machine benchmark, even when using a single worker. This is surprising, because of the additional overhead that the sequential feature extraction approach receives due to latency and such. The reason could also, however be purely coincidental and due to varying processing frequency or memory management in Python. Despite this, it is still an interesting result.

Discussion

The results demonstrates that distributing the feature extraction models across several workers is beneficial for the systems performance. It is, however important to note that the distribution sub-graph requires three workers instead of a single one. The results from running the experiment with three single workers instead of a sub-graph of three workers proves to be more beneficial in terms of pure performance. The reason for this is that the VGG-16 model spends more time extracting features compared to the other two models, which makes the system scale less evenly. However, if the system were to further utilize a classifier that blocks until all features extracted for the batch of data has been

received, the distributed approach may prove to be more beneficial for rapidly classifying features.

6.6 Review of Non-Functional Requirements

This section presents a review of the non-functional requirements outlined in Chapter 3, based on the resulting system and the results from the experiments discussed earlier in this chapter.

Fault Tolerance

The hierarchical design of Áika provides Áika with fault tolerance. The use of local controllers on each physical node ensures that failing agents can be restarted in an effective manner. Using a cluster controller to recover from physical node failure also ensures that the system can remain operable.

- **Reliability:** The system implements reliability through the hierarchical threading structure. Because of this, processes can restart failing threads to make sure without needing to restart the entire process. Consider that the work thread of an agent fails due to a corrupt item. Instead of having to restart the entire process (thereby closing servers that are important for other processes), the main thread can instead simply restart the work thread as soon as it notices that it is no longer running.

- **Availability:** The systems fault tolerance policy enforces that the system remains available for as long as possible. The use of local controllers to recover failing agents ensures that recovery time is short, compared to using the cluster controller for recovery, which aids in making the system as available as possible.

Confidentially and Integrity

Confidentiality of sensitive data can be achieved by utilizing the Dorvu file system, which ensures that data is stored confidentially through encryption. Since the file system is distributed and we assume that all nodes have access to this system, which makes it possible to store confidential data in the file system instead of passing it directly over TCP.

Data integrity is provided through the utilization of persistent queues that are

used between computation steps. The persistent queues does not remove any items from file before work on the item is done and the result has been put on the next queue. Because of this, no items should be lost in the event of failure. Despite this, the system may still be vulnerable

Resilience

Resilience by redundancy is not implemented in Áika by default, but the use of the DAG computation model enables application developers to construct N-model redundancy as a part of the configuration.

6.7 Discussion

There is no doubt that the use of persistent queues can lead to a bottleneck within the system, given that the data is large enough. Moreover, utilizing persistent queues on every agent before the analytical procedure can lead to an unnecessary overhead in the systems performance, as data would need to be sent to every worker, then written to file.

6.8 Summary

This chapter provides an evaluation of Áika based on different experimental results, organized into micro-benchmarks, end-to-end benchmarks and benchmark of specific cases. The results demonstrate that the use of persistent queues can become a bottleneck in the system compared to in-memory queues, but does not affect the performance when the time spent working on each item is larger than the time spent moving the items themselves between two agents. Based on the case experiments with the distributed word counter and the distributed feature extractor, we can conclude that the system scales well and provide stable results, but that a slight overhead makes the performance increase stagnate earlier than expected.

The next chapter makes concluding remarks and provides some future work.

Conclusion and Future Work

This chapter outlines concluding remarks with regards to the thesis based on our findings. Some potential future work is also provided.

7.1 Conclusion

This thesis has presented Áika, a prototype (POC) for a system created for executing distributed ai applications in untrusted edge environments. As a part of the thesis, we have designed, implemented and evaluated Áika. We wanted to investigate how a system supporting machine learning inference in untrusted edge environments could be built to support a wide range of computational graphs through a DAG computation model, while making the system tolerant to failures.

Through a hierarchical design, we utilize local controllers monitoring agents on physical nodes to perform quick recovery when failure occurs. A cluster controller is used to further invoke node recovery, where agents from a failed physical node is moved to a replica. The cluster controller is replicated in a chain to avoid single point of failure. We enable application developers utilizing

the system to construct complex distributed DAGs composed of agents through a relatively simple JSON configuration format.

Based on the results, the system manages to have a stable throughput despite agents crashing every 15 seconds. When measuring with data-intensive tasks, we discovered that the use of persistent queues can be even more beneficial and stable compared to in-memory queues. This applies in cases where the workload on an item exceeds the time spent transporting the item. In Section 6.4 and Section 6.5 we illustrate how Áika can be used to create DAGs of varying complexity that use load balancing and divide work among agents to optimize performance. The results from these experiments demonstrate that Áika is a scalable and supports different DAG designs, which further allows construction of distributed machine learning inference pipelines.

7.2 Future Work

Áika is currently a simplified prototype and there are many potential directions of future work.

Proof of Applicability (POA)

The system has not been tested in a proper untrusted edge environment and should therefore be further developed as a POA, for instance with regards to the Dutkat-project. For this case, Áika should be tested on a computer cluster that is small enough to be deployed on fishing vessels. In the Dutkat-paper, the authors discuss the potential of using Nvidia Jetson NX computers [31].

The system should also be implemented in a more secure and stable programming language, that is better suited for large applications. We recommend to implement a complete version of the system in Rust [133], due to its static type system providing strong guarantees for isolation, concurrency and memory safety. Rust also enables fine-grained control over memory representations. Alternatively, the system could be implemented in the Go Programming Language[134], due to its focus on performance through Go-routines, static typing and implementation of memory safety.

Dynamic Load Balancing

In the current implementation of Áika, load balancing has to be specified through the configuration format before starting the system. Implementing a dynamic load balancing scheme that can be utilized automatically at run-time in areas of the DAG with low throughput could both simplify the application developers job and lead to better performance results.

Add/Remove Resources

The system configuration is currently pre-determined before the run-time is initialized. It is therefore not possible to add or remove resources at run-time without interrupting the run-time itself. The system has to be re-initialized if new resources are to be added. Implementing support for adding and removing resources is another potential feature for future work.

Security

The communication in the system is currently done over pure TCP sockets and are not secured in any way due to the scope of this thesis. If the system is going to be deployed in a real setting, it is necessary to implement overall security policies throughout the system, to ensure that sensitive data does not leak out to unauthorized and malicious actors. Implementing the system with end-to-end encryption when communicating could hinder man-in-the-middle attacks and a proper authorization scheme could hinder outside sources that should not have access from interfering with the systems servers. This is a crucial requirement for the system to run in a realistic setting.

Trusted Execution Environments

Another potential direction for the system is to look into how Trusted Execution Environments, like Intel SGX or Arm TrustZone can be used to protect the systems integrity during run-time. The utilization of trusted execution environments can for example be used to hinder data poisoning and similar attacks. This has been done on a smaller scale with implementation of neural networks that act in trusted execution environments, like DarkneTZ [135] and HybridTEE [136].

Improved Configuration

The configuration format is currently relatively simple, but constructing large and complex graphs can be difficult, as it gets harder to keep track of the different ports and hosts that are used by each agent. Although the startup script simplifies the process by automatically adding some configuration parameters, the configuration can still be altered to a simpler format.

Identify Stragglers

Since the system is designed to continuously run over a longer period of time, stragglers that slows down processes may become a problem. A potential way to do this is to expand upon the killer component that we have used for benchmarks, such that it can detect straggling processes and then kill them. The local controller will then restart them. Another potential solution is to let the local controller itself identify if the working agent is a straggler.

Debugger

The lack of a debugger currently makes it difficult for application developers to implement distributed machine learning solutions with the system. If the system is going to be applied in real cases, it is necessary to have a proper debugger that can detect and forward errors to the developer during the development stage of an application.

Data Splitting

The system does not currently support data being split and distributed to multiple agents from anything else than initial agents. It could be useful to implement steps similar to the Reduce step in MapReduce and would add another way of implementing a distributed DAG.

Configuration Format

The system uses a static configuration format written in JSON. The configuration format determines how data flows between each node within the DAG, and the number of replicas that will be utilized during the run-time.

To make the process of writing configurations simpler, we have implemented a startup script that can be used to both verify and make corrections to the configuration format before starting a new run-time. The startup script is responsible for parsing the configuration file and find any missing keys and add an appropriate values for them. In the case where finding a value for the missing key is not intuitive, the script raises an exception instead, so that the application developer is informed about the mistake.

Listing A.1 illustrates the general configuration format for the system. The system has a single log file, which each component may insert records into when abnormal events have been detected. The configuration is mostly composed of ports for serving other system components, unique identifiers, checkpoint files and the host which each component runs on. The list of replica nodes contains replicated local controllers. The list of crashed nodes contains all local controllers that have crashed. We have excluded agent specific configuration values from this configuration, as they are covered more in-depth in other parts of the thesis, for example in Appendix B. Agent specific configurations may also vary highly, as the application developer may be inclined to implement custom agents with custom configuration values.

Listing A.1: The general configuration format for the system. The values shown are mandatory for each component type, and needs to be added manually, or automatically through the startup script (startup.py). The agents contain additional configuration values (such as input port, lookup tables, etc.) that depend on the type of agent used.

```
1  {
2      "log_file" : "<string>",
3      "cluster_controllers" : [
4          {
5              "ID" : "<string>",
6              "host" : "<string>",
7              "monitor_port" : <int>,
8              "controller_port" : <int>,
9              "recovery_port" : <int>,
10             "execution_file" : "<string>.py",
11             "log_file" : "<string>.db"
12         } ...
13     ],
14     "local_controllers" : [
15         {
16             "ID" : "<string>",
17             "host" : "<string>",
18             "ping_port" : <int>,
19             "recovery_port" : <int>,
20             "execution_file" : "<string>.py",
21             "log_file" : "<string>.db",
22             "agents" : [
23                 {
24                     "ID" : "<string>",
25                     "host" : "<string>",
26                     "execution_file" : "<string>.py",
27                     "queue_log1" : "<string>",
28                     "queue_log2" : "<string>",
29                     "cluster_controller_port": <int>,
30                     "log_file" : "<string>.db"
31                     "<agent specific configurations ..>"
32                 } ...
33             ]
34         }
35     ],
36     "replica_nodes" : [],
37     "crashed_nodes" : [],
38 }
```

/B

Local Run-time Example

This appendix demonstrates the correspondence between the configuration format and the DAG data flow format.

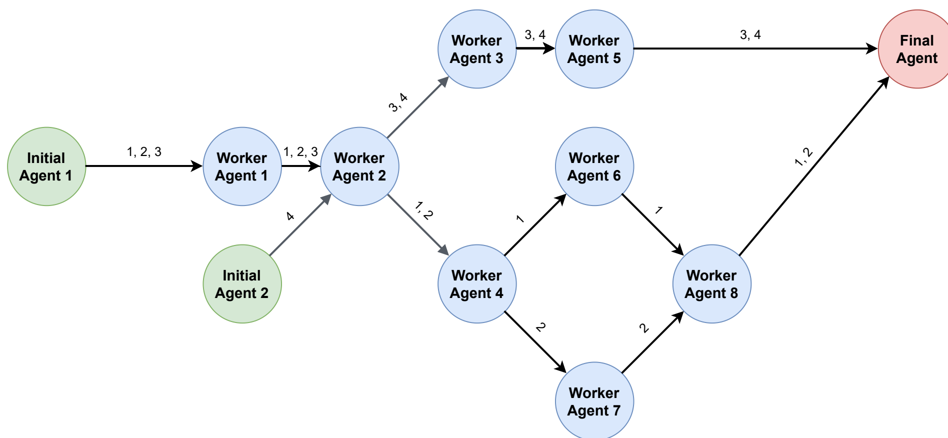

Figure B.1: An example DAG constructed from the configuration shown in Listing B.1.

Figure B.1 illustrates the DAG that is given from the agent configuration given in Listing B.1.

Listing B.1: DAG configuration. The listing illustrates a complete configuration of agents that is used to create the DAG. For this example, The DAG is composed of left worker agents, as they enable data to be passed to multiple other agents.

```
1  "agents" : [
2      {
3          "host" : "localhost",
4          "lookup_table" : {
5              "1" : { "host" : "localhost", "port" : 7001 },
6              "2" : { "host" : "localhost", "port" : 7001 },
7              "3" : { "host" : "localhost", "port" : 7001 }
8          }
9      },
10     {
11         "host" : "localhost",
12         "input_port" : 7001,
13         "lookup_table" : {
14             "1" : { "host" : "localhost", "port" : 7002 },
15             "2" : { "host" : "localhost", "port" : 7002 },
16             "3" : { "host" : "localhost", "port" : 7002 }
17         }
18     },
19     {
20         "host" : "localhost",
21         "lookup_table" : {
22             "4" : { "host" : "localhost", "port" : 7002 }
23         }
24     },
25     {
26         "host" : "localhost",
27         "input_port" : 7002,
28         "lookup_table" : {
29             "1" : { "host" : "localhost", "port" : 7004 },
30             "2" : { "host" : "localhost", "port" : 7004 },
31             "3" : { "host" : "localhost", "port" : 7003 },
32             "4" : { "host" : "localhost", "port" : 7003 }
33         }
34     },
35     {
36         "host" : "localhost",
37         "input_port" : 7003,
38         "lookup_table" : {
39             "3" : { "host" : "localhost","port" : 7005 },
40             "4" : { "host" : "localhost","port" : 7005 }
41         }
42     },
43     {
```

```
44          "host" : "localhost",
45          "input_port" : 7004,
46          "lookup_table" : {
47              "1" : { "host" : "localhost", "port" : 7006 },
48              "2" : { "host" : "localhost", "port" : 7007 }
49          }
50      },
51      {
52          "host" : "localhost",
53          "input_port" : 7005,
54          "lookup_table" : {
55              "3" : { "host" : "localhost", "port" : 7011 },
56              "4" : { "host" : "localhost", "port" : 7011 }
57          }
58      },
59      {
60          "host" : "localhost",
61          "input_port" : 7006,
62          "lookup_table" : {
63              "1" : { "host" : "localhost", "port" : 7008 }
64          }
65      },
66      {
67          "host" : "localhost",
68          "input_port" : 7007,
69          "lookup_table" : {
70              "2" : { "host" : "localhost", "port" : 7008 }
71          }
72      },
73      {
74          "host" : "localhost",
75          "input_port" : 7008,
76          "lookup_table" : {
77              "1" : { "host" : "localhost", "port" : 7011 },
78              "2" : { "host" : "localhost", "port" : 7011 }
79          }
80      },
81      {
82          "host" : "localhost",
83          "input_port" : 7011,
84      }
85  ]
```

C Distributed Word Counter

This appendix explains the Distributed Word Counter used for the text processing experiment covered in Chapter 6 (Section 6.4).

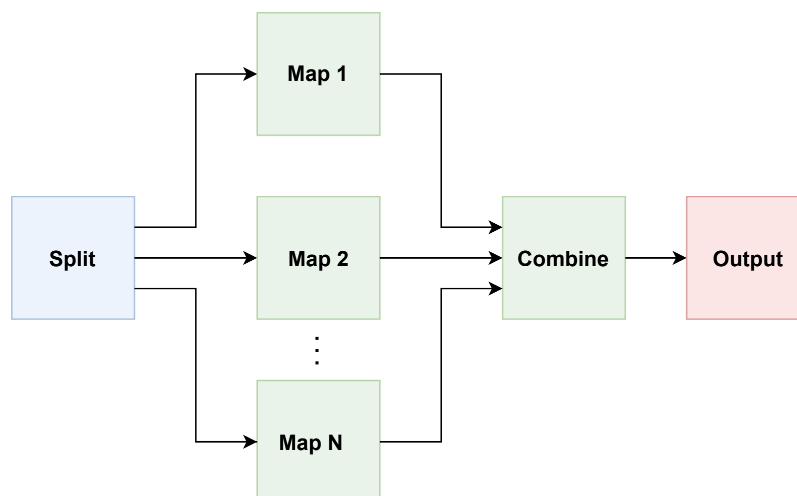

Figure C.1: Illustration of DAG for distributed word counter.

The distributed word counter program uses a similar approach as MapReduce. The process is initialized by an initial agent that splits the workload into chunks that are equally shared among the working agents. The working agents fetch one workload each from the initial agent, loads the data set partition assigned into memory and counts word by word through iteration of the text. The results

are stored in a map that contains words as keys and number of occurrences as values. Each map agent forwards the resulting map to one reducer agent which combines the maps together to gain the total results from the entire text.

Figure C.1 illustrates the DAG for the distributed word counter program. The program can be scaled by adding additional map agents to the configuration. Unfortunately, Áika does currently not support splitting data and sending the splits to different sources, like in MapReduce, and the design therefore relies on using a single reducer agent to gather the results. The consequence of this is that the reduce/combine agent may become a bottleneck if the document contains a high number of unique words. It may also receive requests from many mappers at the same time.

The data set used is generated with a script that builds a textfile by randomly selecting words from a list of words. This process is done iteratively until the the document has reached the pre-determined size. Since the byte size of a word depends on the string, the pre-determines size is only an approximation, and may therefore be off by a few bytes. For the experiment, text files of 100MB and 300MB is used.

Distributed Deep Feature Extractor

This appendix explains the Distributed Deep Feature Extractor used for the image experiments covered in Chapter 6 (Section 6.5).

The distributed deep feature extractor program uses a similar design as the word counter program from Appendix C, but is slightly more complex. The aim of the program is to evaluate the performance of the system when feature extraction is performed on multiple machine learning models that are either distributed across several nodes, or performed in sequence on a single node. We use three different deep learning models as example models in order to perform feature extraction on images: VGG-16, DenseNet-121 and ResNet-50.

These models are either distributed across three nodes, such that the feature extraction can be done simultaneously, or put on a single node, such that the feature extractions will be performed in sequential order on the three models. The features are only extracted from the images before the program moves on, and they are thus not written to disk or stored anywhere.

The system is configured with a similar design as the word counter program. An initial agent initializes the pipeline by splitting the workload and inserting it on a local queue, which worker agents can fetch from. It is also responsible for starting the timer for performance measurement. A final agent is placed

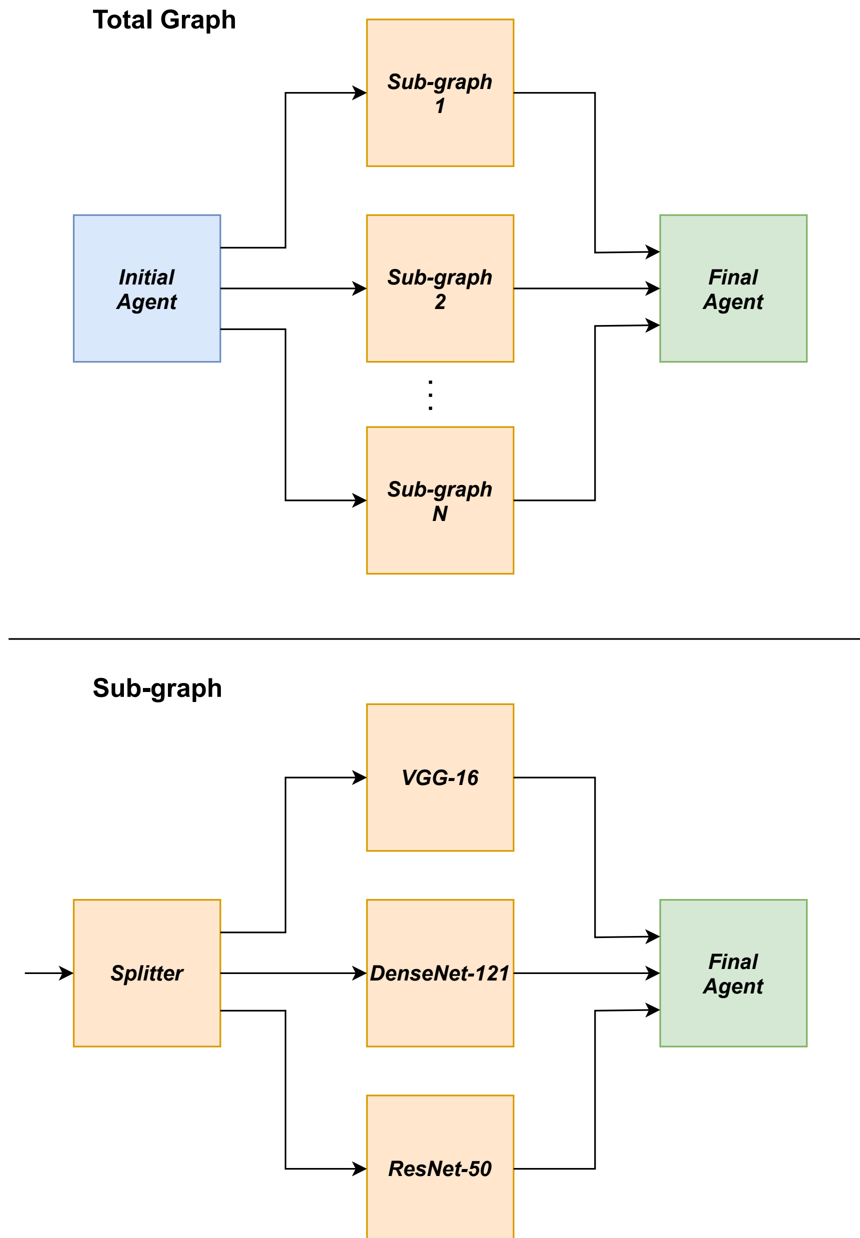

Figure D.1: The DAG for the distributed deep feature extractor with distributed machine learning models.

at the end of the pipeline. It is responsible to count the number of features extracted, and then stop a timer when the correct number of features has been extracted.

The worker agents can be composed of a single agent that fetches workloads from the initial agents queue and perform feature extraction with all three models sequentially. They can, alternatively be built as sub-graphs, where a splitter agent fetches workload items from the initial agent, then forwards the workload to three workers. Each worker contains a single model that is used to perform feature extractions, and they can therefore perform feature extractions simultaneously. This solution is illustrated in Figure D.1. This design enables the system to be configured with a combination of load balancing, and divided work. That is, the design supports multiple sub-graphs to be used for load balancing (seen at the top figure), while at the same time distributing the machine learning models (seen in the bottom figure).

Bibliography

[1] Mahadev Satyanarayanan. "The Emergence of Edge Computing." In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[2] Wazir Zada Khan et al. "Edge computing: A survey." In: *Future Generation Computer Systems* 97 (2019), pp. 219–235. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.050. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18319903.

[3] Phil Marshall et al. *State of the Edge: A market and Ecosystem Report for Edge Computing*. Tech. rep. The Linux Foundation, 2021. URL: https://project.linuxfoundation.org/hubfs/LF%20Edge/StateoftheEdgeReport_2021.pdf.

[4] Kun-Hsing Yu, Andrew L Beam, and Isaac S Kohane. "Artificial intelligence in healthcare." eng. In: *Nature biomedical engineering* 2.10 (2018), pp. 719–731. ISSN: 2157-846X.

[5] Inga Strumke et al. "Artificial Intelligence in Gastroenterology." In: (2022).

[6] MA Riegler et al. "Artificial intelligence in the fertility clinic: status, pitfalls and possibilities." In: *Human Reproduction* 36.9 (2021), pp. 2429–2442.

[7] Andrea M Storås et al. "Artificial intelligence in dry eye disease." In: *The Ocular Surface* (2021).

[8] Lex Fridman et al. "MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of Driver Behavior and Interaction With Automation." In: *IEEE Access* 7 (2019), 102021–102038. ISSN: 2169-3536. DOI: 10.1109/access.2019.2926040. URL: http://dx.doi.org/10.1109/ACCESS.2019.2926040.

[9] Hsu-kuang Chiu et al. "Probabilistic 3D multi-modal, multi-object tracking for autonomous driving." In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 14227–14233.

[10] Scott A. Wright and Ainslie E. Schultz. "The rising tide of artificial intelligence and business automation: Developing an ethical framework." In: *Business Horizons* 61.6 (2018). ETHICS, CULTURE, AND PEDAGOGICAL PRACTICES IN THE GLOBAL CONTEXT, pp. 823–832. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2018.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0007681318301046.

[11] Ion Stoica et al. *A Berkeley View of Systems Challenges for AI*. Tech. rep. UCB/EECS-2017-159. EECS Department, University of California, Berkeley, 2017. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-159.html.

[12] Fabrizio Carcillo et al. "Combining unsupervised and supervised learning in credit card fraud detection." In: *Information sciences* 557 (2021), pp. 317–331.

[13] Shuiguang Deng et al. "Edge Intelligence: the Confluence of Edge Computing and Artificial Intelligence." In: *CoRR* abs/1909.00560 (2019). arXiv: 1909.00560. URL: http://arxiv.org/abs/1909.00560.

[14] Junyu Liu et al. "Network Densification in 5G: From the Short-Range Communications Perspective." In: *IEEE Communications Magazine* 55 (Dec. 2017), pp. 96–102. DOI: 10.1109/MCOM.2017.1700487.

[15] D. E. Comer et al. "Computing as a Discipline." In: *Commun. ACM* 32.1 (Jan. 1989), 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[16] G Hartvigsen and D Johansen. "Stormcast — A Distributed Artificial Intelligence Application for Severe Storm Forecasting." eng. In: *IFAC Proceedings Volumes* 21.12 (1988), pp. 99–102. ISSN: 1474-6670.

[17] D. Johansen, R. van Renesse, and F.B. Schneider. "Operating system support for mobile agents." In: *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. 1995, pp. 42–45. DOI: 10.1109/HOTOS.1995.513452.

[18] Dag Johansen. "Mobile agent applicability." eng. In: *Personal and ubiquitous computing* 2.2 (1998), pp. 57–67. ISSN: 0949-2054.

[19] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[20] Steffen Viken Valvåg. "Cogset : A High-Performance MapReduce Engine." In: (Jan. 2012).

[21] Håvard Johansen, André Allavena, and Robbert van Renesse. "Fireflies: scalable support for intrusion-tolerant network overlays." eng. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on computer systems 2006*. EuroSys '06. ACM, 2006, pp. 3–13. ISBN: 9781595933225.

[22] Håvard D. Johansen et al. "Fireflies: A Secure and Scalable Membership and Gossip Service." In: *ACM Trans. Comput. Syst.* 33.2 (2015). ISSN: 0734-2071. DOI: 10.1145/2701418. URL: https://doi.org/10.1145/2701418.

[23] Audun Nordal et al. "Balava: Federating Private and Public Clouds." In: *2011 IEEE World Congress on Services*. 2011, pp. 569–577. DOI: 10.1109/SERVICES.2011.21.

[24] Robbert van Renesse et al. *Vortex. An event-driven multiprocessor operating system supporting performance isolation*. eng. Universitetet i Tromsø, 2003.

[25] Anders T Gjerdrum et al. "Performance Principles for Trusted Computing with Intel SGX." eng. In: *Cloud Computing and Service Science*. Communications in Computer and Information Science. Cham: Springer International Publishing, 2018, pp. 1–18. ISBN: 9783319949581.

[26] Robert Pettersen, Håvard Johansen, and Dag Johansen. "Secure Edge Computing with ARM TrustZone." In: Jan. 2017, pp. 102–109. DOI: 10.5220/0006308601020109.

[27] Anders Tungeland Gjerdrum. *Diggi : a distributed serverless runtime for developing trusted cloud services*. eng. Tromsø, 2020.

[28] Olav A. Norgård Rongved et al. "Real-Time Detection of Events in Soccer Videos using 3D Convolutional Neural Networks." In: *2020 IEEE International Symposium on Multimedia (ISM)*. 2020, pp. 135–144. DOI: 10.1109/ISM.2020.00030.

[29] Debesh Jha et al. "ResUNet++: An Advanced Architecture for Medical Image Segmentation." In: *2019 IEEE International Symposium on Multimedia (ISM)*. 2019, pp. 225–2255. DOI: 10.1109/ISM46123.2019.00049.

[30] Debesh Jha et al. *NanoNet: Real-Time Polyp Segmentation in Video Capsule Endoscopy and Colonoscopy*. 2021. arXiv: 2104.11138 [eess.IV].

[31] Tor-Arne S. Nordmo et al. "Dutkat: A Multimedia System for Catching Illegal Catchers in a Privacy-Preserving Manner." In: *Proceedings of the 2021 Workshop on Intelligent Cross-Data Analysis and Retrieval*. ICDAR '21. Taipei, Taiwan: Association for Computing Machinery, 2021, 57–61.

ISBN: 9781450385299. DOI: 10.1145/3463944.3469102. URL: https://doi.org/10.1145/3463944.3469102.

[32] Aril Bernhard Ovesen et al. "File System Support for Privacy-Preserving Analysis and Forensics in Low-Bandwidth Edge Environments." In: *Information* 12.10 (2021). ISSN: 2078-2489. DOI: 10.3390/info12100430. URL: https://www.mdpi.com/2078-2489/12/10/430.

[33] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE." In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[34] Robert Garner. *Early Popular Computers, 1950 - 1970*. http://ethw.org/Early_Popular_Computers,_1950_-_1970. [Online; accessed 29-November-2021]. 2015.

[35] John McCarthy. "What is artificial intelligence?" In: (2007).

[36] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[37] Anne Håkansson and Ronald Lee Hartung. *Artificial Intelligence : concepts, areas, techniques and applications*. eng. Lund, 2020.

[38] Bruce G Buchanan and Edward A Feigenbaum. "Dendral and meta-dendral: Their applications dimension." eng. In: *Artificial intelligence* 11.1 (1978), pp. 5–24. ISSN: 0004-3702.

[39] Edward Shortliffe. "Mycin: A Knowledge-Based Computer Program Applied to Infectious Diseases*." In: *Proceedings / the ... Annual Symposium on Computer Application [sic] in Medical Care. Symposium on Computer Applications in Medical Care* (Oct. 1977).

[40] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. "A Survey of the Usages of Deep Learning for Natural Language Processing." In: *IEEE Transactions on Neural Networks and Learning Systems* 32.2 (2021), pp. 604–624. DOI: 10.1109/TNNLS.2020.2979670.

[41] Sean Eom and E Kim. "A survey of decision support system applications (1995–2001)." In: *Journal of the Operational Research Society* 57.11 (2006), pp. 1264–1278.

[42] Sean Eom and E Kim. "A survey of decision support system applications (1995–2001)." In: *Journal of the Operational Research Society* 57.11 (2006), pp. 1264–1278.

[43] Weiming Shen and Douglas H Norrie. "Agent-based systems for intelligent manufacturing: a state-of-the-art survey." In: *Knowledge and information systems* 1.2 (1999), pp. 129–156.

[44] Peter Stone and Manuela Veloso. "Multiagent systems: A survey from a machine learning perspective." In: *Autonomous Robots* 8.3 (2000), pp. 345–383.

[45] Craig W. Reynolds. "Flocks, Herds and Schools: A Distributed Behavioral Model." In: *SIGGRAPH Comput. Graph.* 21.4 (1987), 25–34. ISSN: 0097-8930. DOI: 10.1145/37402.37406. URL: https://doi.org/10.1145/37402.37406.

[46] Lars Kunze et al. "Artificial Intelligence for Long-Term Robot Autonomy: A Survey." In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 4023–4030. DOI: 10.1109/LRA.2018.2860628.

[47] Kwang-Kyo Oh and Hyo-Sung Ahn. "A survey of formation of mobile agents." In: *2010 IEEE International Symposium on Intelligent Control*. 2010, pp. 1470–1475. DOI: 10.1109/ISIC.2010.5612920.

[48] S Binitha, S Siva Sathya, et al. "A survey of bio inspired optimization algorithms." In: *International journal of soft computing and engineering* 2.2 (2012), pp. 137–151.

[49] Xin Feng et al. "Computer vision algorithms and hardware implementations: A survey." In: *Integration* 69 (2019), pp. 309–320. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.07.005. URL: https://www.sciencedirect.com/science/article/pii/S0167926019301762.

[50] Peijun Ye, Tao Wang, and Fei-Yue Wang. "A Survey of Cognitive Architectures in the Past 20 Years." In: *IEEE Transactions on Cybernetics* 48.12 (2018), pp. 3280–3290. DOI: 10.1109/TCYB.2018.2857704.

[51] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. "Multimodal Machine Learning: A Survey and Taxonomy." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.2 (2019), pp. 423–443. DOI: 10.1109/TPAMI.2018.2798607.

[52] Samira Pouyanfar et al. "A Survey on Deep Learning: Algorithms, Techniques, and Applications." In: *ACM Comput. Surv.* 51.5 (2018).

ISSN: 0360-0300. DOI: 10.1145/3234150. URL: https://doi.org/10.1145/3234150.

[53] Geert Litjens et al. “A survey on deep learning in medical image analysis.” In: *Medical image analysis* 42 (2017), pp. 60–88.

[54] Ethem Alpaydin. *Introduction to Machine Learning (3rd Edition)*. eng. Cambridge, MA, USA, 2014.

[55] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey.” In: *CoRR* cs.AI/9605103 (1996). URL: https://arxiv.org/abs/cs/9605103.

[56] Qiong Liu and Ying Wu. “Supervised Learning.” In: (Jan. 2012). DOI: 10.1007/978-1-4419-1428-6_451.

[57] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. DOI: 10.1017/CBO9780511801389.

[58] William A Belson. “Matching and prediction on the principle of biological classification.” In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 8.2 (1959), pp. 65–75.

[59] Arti Patle and Deepak Singh Chouhan. “SVM kernel functions for classification.” In: *2013 International Conference on Advances in Technology and Engineering (ICATE)*. 2013, pp. 1–9. DOI: 10.1109/ICAdTE.2013.6524743.

[60] R. Sathya and Annamma Abraham. “Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification.” In: *International Journal of Advanced Research in Artificial Intelligence* 2.2 (2013). DOI: 10.14569/IJARAI.2013.020206. URL: http://dx.doi.org/10.14569/IJARAI.2013.020206.

[61] J. Macqueen. “Some methods for classification and analysis of multivariate observations.” In: *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*. 1967, pp. 281–297.

[62] Harold Hotelling. “Analysis of a complex of statistical variables into principal components.” In: *Journal of educational psychology* 24.6 (1933), p. 417.

[63] Karl Pearson. “LIII. On lines and planes of closest fit to systems of points in space.” In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.

[64] Ian T Jolliffe. *Principal component analysis*. eng. New York, 2002.

[65] *Semi-supervised learning*. eng. Cambridge, Massachusetts, 2010.

[66] H. Scudder. “Probability of error of some adaptive pattern-recognition machines.” In: *IEEE Transactions on Information Theory* 11.3 (1965), pp. 363–371. DOI: 10.1109/TIT.1965.1053799.

[67] C.J Merz, D.C St. Clair, and W.E Bond. “SeMi-supervised adaptive resonance theory (SMART2).” eng. In: *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. Vol. 3. IEEE, 1992, 851–856 vol.3. ISBN: 0780305590.

[68] Jesper E Van Engelen and Holger H Hoos. “A survey on semi-supervised learning.” In: *Machine Learning* 109.2 (2020), pp. 373–440.

[69] Popescu Marius et al. “Multilayer perceptron and neural networks.” In: *WSEAS Transactions on Circuits and Systems* 8 (July 2009).

[70] Manjunath Jogin et al. “Feature Extraction using Convolution Neural Networks (CNN) and Deep Learning.” In: *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2018, pp. 2319–2323. DOI: 10.1109/RTEICT42901.2018.9012507.

[71] Ian Goodfellow. *Deep learning*. eng. Cambridge, Massachusetts, 2016.

[72] Fuzhen Zhuang et al. *A Comprehensive Survey on Transfer Learning*. 2020. arXiv: 1911.02685 [cs.LG].

[73] Y. Lecun et al. “Gradient-based learning applied to document recognition.” In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[74] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[75] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory.” In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[76] Dor Bank, Noam Koenigstein, and Raja Giryes. “Autoencoders.” In: *CoRR* abs/2003.05991 (2020). arXiv: 2003.05991. URL: https://arxiv.org/abs/2003.05991.

[77] Blesson Varghese et al. “Challenges and Opportunities in Edge Computing.” In: Nov. 2016. DOI: 10.1109/SmartCloud.2016.18.

[78] Alin-Gabriel Gheorghe et al. "Decentralized Storage System for Edge Computing." In: *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. 2019, pp. 41–49. DOI: 10.1109/ISPDC.2019.00009.

[79] M. van Steen and A.S. Tanenbaum. *Distributed systems, 3rd ed., distributed-systems.net*. eng. The Netherlands? 2017.

[80] *Tech Confronts Its Use of the Labels 'Master' and 'Slave'*. https://www.wired.com/story/tech-confronts-use-labels-master-slave/. [Online; accessed 16-September-2021]. 2020.

[81] *There's an industry that talks daily about 'masters' and 'slaves.' It needs to stop*. https://www.washingtonpost.com/opinions/2020/06/12/tech-industry-has-an-ugly-master-slave-problem/. [Online; accessed 16-September-2021]. 2020.

[82] *Master,' 'Slave' and the Fight Over Offensive Terms in Computing*. https://www.nytimes.com/2021/04/13/technology/racist-computer-engineering-terms-ietf.html. [Online; accessed 16-September-2021]. 2021.

[83] Dag Johansen et al. "USING SOFTWARE DESIGN PATTERNS TO BUILD DISTRIBUTED ENVIRONMENTAL MONITORING APPLICATIONS." In: (1997).

[84] Nikolay Baychenko. "Implementing a master/slave architecture for a data synchronization service." In: 2018.

[85] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System." In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), 29–43. ISSN: 0163-5980. DOI: 10.1145/1165389.945450. URL: https://doi.org/10.1145/1165389.945450.

[86] C. V. Ramamoorthy and H. F. Li. "Pipeline Architecture." In: *ACM Comput. Surv.* 9.1 (Mar. 1977), 61–102. ISSN: 0360-0300. DOI: 10.1145/356683.356687. URL: https://doi.org/10.1145/356683.356687.

[87] John Hennessy et al. "MIPS: A microprocessor architecture." In: *ACM SIGMICRO Newsletter* 13 (Dec. 1982), pp. 17–22. DOI: 10.1145/1014194.800930.

[88] Maria Aspri, Grigorios Tsagkatakis, and Panagiotis Tsakalides. "Distributed Training and Inference of Deep Learning Models for Multi-Modal Land Cover Classification." In: *Remote Sensing* 12.17 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12172670. URL: https://www.mdpi.com/2072-4292/12/17/2670.

[89] C. A. R. Hoare. "Communicating Sequential Processes." In: *Commun. ACM* 21.8 (1978), 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[90] Thomas H. Cormen. "Directed Acyclic Graphs." In: *Algorithms Unlocked*. 2013, pp. 71–89.

[91] Michael Isard et al. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks." In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), 59–72. ISSN: 0163-5980. DOI: 10.1145/1272998.1273005. URL: https://doi.org/10.1145/1272998.1273005.

[92] Jeffrey Dean et al. "Large Scale Distributed Deep Networks." In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[93] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding." In: *CoRR* abs/1408.5093 (2014). arXiv: 1408.5093. URL: http://arxiv.org/abs/1408.5093.

[94] Martín Abadi et al. "Tensorflow: A system for large-scale machine learning." In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[95] Hermann Kopetz and Paulo Veríssimo. "Real Time and Dependability Concepts." In: *Distributed Systems (2nd Ed.)* USA: ACM Press/Addison-Wesley Publishing Co., 1993, 411–446. ISBN: 0201624273.

[96] M. Farrukh Khan and Raymond A. Paul. "Chapter 4 - Pragmatic Directions in Engineering Secure Dependable Systems." In: *Dependable and Secure Systems Engineering*. Ed. by Ali Hurson and Sahra Sedigh. Vol. 84. Advances in Computers. Elsevier, 2012, pp. 141–167. DOI: https://doi.org/10.1016/B978-0-12-396525-7.00005-8. URL: https://www.sciencedirect.com/science/article/pii/B9780123965257000058.

[97] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." In: *J. ACM* 32.2 (1985), 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[98] J. Aspnes. "Time- and Space-Efficient Randomized Consensus." In: *Journal of Algorithms* 14.3 (1993), pp. 414–431. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1993.1022. URL: https://www.sciencedirect.com/science/article/pii/S0196677483710229.

[99] A. Fox and E.A. Brewer. "Harvest, yield, and scalable tolerant systems." In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 1999, pp. 174–178. DOI: 10.1109/HOTOS.1999.798396.

[100] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." In: 33.2 (June 2002), 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[101] Seth Gilbert and Nancy Lynch. "Perspectives on the CAP Theorem." In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389.

[102] Jan Himmelspach and Adelinde M. Uhrmacher. "The Event Queue Problem and PDevs." In: *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*. SpringSim '07. Norfolk, Virginia: Society for Computer Simulation International, 2007, 257–264. ISBN: 1565553136.

[103] Chao Wang, Christopher Gill, and Chenyang Lu. "FRAME: Fault Tolerant and Real-Time Messaging for Edge Computing." In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 976–985. DOI: 10.1109/ICDCS.2019.00101.

[104] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. "The Design and Performance of a Real-Time CORBA Event Service." In: *SIGPLAN Not.* 32.10 (Oct. 1997), 184–200. ISSN: 0362-1340. DOI: 10.1145/263700.263734. URL: https://doi.org/10.1145/263700.263734.

[105] Yotam Harchol et al. "CESSNA: Resilient Edge-Computing." In: *Proceedings of the 2018 Workshop on Mobile Edge Communications*. MECOMM'18. Budapest, Hungary: Association for Computing Machinery, 2018, 1–6. ISBN: 9781450359061. DOI: 10.1145/3229556.3229558. URL: https://doi.org/10.1145/3229556.3229558.

[106] D. Johansen et al. "NAP: practical fault-tolerance for itinerant computations." In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. 1999, pp. 180–189. DOI: 10.1109/ICDCS.1999.776519.

[107] Fred B Schneider, David Gries, and Richard D Schlichting. "Fault-tolerant broadcasts." eng. In: *Science of computer programming* 4.1 (1984), pp. 1–15. ISSN: 0167-6423.

[108] R. V. Renesse, D. Johansen, and F. Schneider. "An introduction to the TACOMA distributed system. Version 1.0." In: 1995.

[109] Joshua Leners et al. "Detecting failures in distributed systems with the Falcon spy network." eng. In: *Proceedings of the Twenty-Third ACM*

Symposium on operating systems principles. SOSP '11. ACM, 2011, pp. 279–294. ISBN: 9781450309776.

[110] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), 230–243. ISSN: 0163-5980. DOI: 10.1145/502059.502057. URL: https://doi.org/10.1145/502059.502057.

[111] Christopher Costello et al. "The future of food from the sea." eng. In: *Nature (London)* 588.7836 (2020), p. 95. ISSN: 0028-0836.

[112] United Nations Office on Drugs and Crime. *Fisheries Crime*. 2016.

[113] Industry Ministry of Trade and Fisheries. "Framtidens Fiskerikontroll." In: *NOU 21:19* (2019).

[114] Arctic Council Task Force on Telecommunications Infrastructure in the Arctic. *Telecommunications infrastructure in the Arctic: a circumpolar assessment*. Arctic Council Task Force on Telecommunications Infrastructure in the Arctic (TFTIA). 90 pp. 2017.

[115] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. "Adversarial examples in the physical world." In: *CoRR* abs/1607.02533 (2016). arXiv: 1607.02533. URL: http://arxiv.org/abs/1607.02533.

[116] *Guidelines on Data Protection Impact Assessment (DPIA) and determining whether processing is "likely to result in a high risk" for the purposes of Regulation 2016/679*. https://ec.europa.eu/newsroom/just/document.cfm?doc_id=47711. [Online; accessed 2-December-2021]. 2017.

[117] Mauritz Kop. "EU Artificial Intelligence Act: The European Approach to AI." In: *Transatlantic Antitrust and IPR Developments* (2021). [Online; accessed 2-December-2021].

[118] *Anonymization (computing)*. *Oxford learner's dictionaries*. eng. Oxford.

[119] *pseudonymization*. *Cambridge Dictionary*. eng. Cambridge.

[120] *CCTV cameras onboard fishing vessels is going beyond all limits*. https://thefishingdaily.com/denmark-fishing-industry-blog/cctv-cameras-onboard-fishing-vessels-is-is-going-beyond-all-limits/. [Online; accessed 2-December-2021]. 2021.

[121] Ian Sommerville. *Software engineering*. eng. Boston Mass., 2016.

[122] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. https://pypl.github.io/PYPL.html. [Online; accessed 09-December-2021]. 2021.

[123] Joakim Sundnes. *Introduction to Scientific Programming with Python*. eng. Simula SpringerBriefs on Computing. Cham: Springer International Publishing AG, 2020. ISBN: 9783030503550.

[124] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[125] G. Bradski. "The OpenCV Library." In: *Dr. Dobb's Journal of Software Tools* (2000).

[126] Wes McKinney et al. "Data structures for statistical computing in python." In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[127] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python." In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[128] Richard D Hipp. *SQLite*. Version 3.31.1. 2020. URL: https://www.sqlite.org/index.html.

[129] *1000 most common words in English*. https://www.ef.com/wwen/english-resources/english-vocabulary/top-1000-words/. [Online; accessed 11-November-2021]. 2021.

[130] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning." In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 215–223.

[131] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images." In: (2009).

[132] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[133] Nicholas D. Matsakis and Felix S. Klock. "The Rust Language." In: *Ada Lett.* 34.3 (Oct. 2014), 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: https://doi.org/10.1145/2692956.2663188.

[134] Robert Griesemer et al. "Hey! Ho! Let's Go!" In: Google Open Source, 2009.

[135] Fan Mo et al. "DarkneTZ: Towards Model Privacy at the Edge Using Trusted Execution Environments." In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. MobiSys

'20. Toronto, Ontario, Canada: Association for Computing Machinery, 2020, 161–174. ISBN: 9781450379540. DOI: 10.1145/3386901.3388946. URL: https://doi.org/10.1145/3386901.3388946.

[136] Akshay Gangal, Mengmei Ye, and Sheng Wei. “HybridTEE: Secure Mobile DNN Execution Using Hybrid Trusted Execution Environment.” In: *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 2020, pp. 1–6. DOI: 10.1109/AsianHOST51057.2020.9358260.