DOCTORAL THESIS

Decentralized Location-aware Orchestration of Containerized Microservice Applications

Enabling Distributed Intelligence at the Edge

Lara Lorna Jiménez

Pervasive and Mobile Computing

LULEÅ UNIVERSITY OF TECHNOLOGY

LULEÅ
UNIVERSITY
OF TECHNOLOGY

Decentralized Location-aware Orchestration of Containerized Microservice Applications

Enabling Distributed Intelligence at the Edge

Lara Lorna Jiménez

Luleå University of Technology
Department of of Computer Science and Electrical Engineering
Division of Computer Science

Printed by Luleå University of Technology, Graphic Production 2020

ISSN 1402-1544
ISBN 978-91-7790-617-9 (print)
ISBN 978-91-7790-618-6 (pdf)

Luleå 2020

www.ltu.se

Decentralized Location-aware Orchestration of Containerized Microservice Applications

Enabling Distributed Intelligence at the Edge

Lara Lorna Jiménez

Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Sweden

Supervisors:

Olov Schelén and Kåre Synnes

To my parents

Abstract

Services that operate on public, private, or hybrid clouds, should always be available and reachable to their end-users or clients. However, a shift in the demand for current and future services has led to new requirements on network infrastructure, service orchestration, and Quality-of-Service (QoS). Services related to, for example, online-gaming, video-streaming, smart cities, smart homes, connected cars, or other Internet-of-Things (IoT) powered use cases are data-intensive and often have real-time and locality requirements. These have pushed for a new computing paradigm, *Edge computing*, based on moving some intelligence from the cloud to the edge of the network to minimize latency and data transfer. This situation has set new challenges for cloud providers, telecommunications operators, and content providers.

This thesis addresses two issues in this problem area that call for distinct approaches and solutions. Both issues share the common objectives of improving energy-efficiency and mitigating network congestion by minimizing data transfer to boost service performance, particularly concerning latency, a prevalent QoS metric.

The first issue is related to the demand for a highly scalable orchestrator that can manage a geographically distributed infrastructure to deploy services efficiently at clouds, edges, or a combination of these. We present an orchestrator using process containers as the virtualization technology for efficient infrastructure deployment in the cloud and at the edge. The work focuses on a Proof-of-Concept design and analysis of a scalable and resilient decentralized orchestrator for containerized applications, and a scalable monitoring solution for containerized processes. The proposed orchestrator deals with the complexity of managing a geographically dispersed and heterogeneous infrastructure to efficiently deploy and manage applications that operate across different geographical locations — thus facilitating the pursuit of bringing some of the intelligence from the cloud to the edge, in a way that is transparent to the applications. The results show this orchestrator's ability to scale to 20 000 nodes and to deploy 30 000 applications in parallel. The resource search algorithm employed and the impact of location awareness on the orchestrator's deployment capabilities were also analyzed and deemed favorable.

The second issue is related to enabling fast real-time predictions and minimizing data transfer for data-intensive scenarios by deploying machine learning models at devices to decrease the need for the processing of data by upper tiers and to decrease prediction latency. Many IoT or edge devices are typically resource-scarce, such as FPGAs, ASICs, or low-level microcontrollers. Limited devices make running well-known machine learning algorithms that are either too complex or too resource-consuming unfeasible. Consequently, we explore developing innovative supervised machine learning algorithms

to efficiently run in settings demanding low power and resource consumption, and real-time responses. The classifiers proposed are computationally inexpensive, suitable for parallel processing, and have a small memory footprint. Therefore, they are a viable choice for pervasive systems with one or a combination of these limitations, as they facilitate increasing battery life and achieving reduced predictive latency. An implementation of one of the developed classifiers deployed to an off-the-shelf FPGA resulted in a predictive throughput of 57.1 million classifications per second, or one classification every 17.485 ns.

Contents

Acknowledgments

I want to express my gratitude to my current and former supervisors, particularly Olov Schélen, Kåre Synnes, and Johan Kristiansson. I appreciate the great technical discussions, as well as your support and guidance throughout these years. You have helped me to broaden my horizons and inspired me to deepen my understanding within our field.

This thesis has granted me the opportunity to work alongside some of my colleagues, particularly Miguel Gómez Simón and Niklas Karvonen. It was a great experience, educational and, at times, downright hilarious; "I really really like Laban Movement Analysis" or the automation fever — there is no such thing as too many scripts 😊.

During my time at LTU, I have had the pleasure of meeting great people who have been both colleagues and friends. I have had a wonderful experience of getting to know all of you. Some of you, I have talked to sporadically about the rigors of the Ph.D. or life in general. With others, I have spent countless hours merely having fun and enjoying each other's company, sometimes involving far-out, crazy discussions. I will not name all of you; you know who you are. However, I would like to mention some of you in particular: Fredrik Bengtsson, Julia Fischer, Jaime García, Josef Hallberg, Fredrik Häggstrőm, Simon Jonsson, Niklas Karvonen, Basel Kikhia, Denis Kleyko, Sergio Martín del Campo Barraza, Arash Mousavi, Joakim Nilsson, Cristina Paniagua, and Pablo Puñal.

A very special thank you to Miguel, he has been my anchor in a stormy sea. Your support through life, sorrows, frustrations, and very long work hours has helped me to finish this voyage, and so, bring this chapter of my life to a close. I would also like to thank your parents, Carmen and Jesús; they took me in during very trying times.

A much-deserved thank you goes to my family. During this time, my sisters, Ruth and Daphne, and my nephews, Álvaro and Raúl, have provided their support and companionship in a multitude of ways through the good and the bad times. Finally, this thesis has been written in loving memory of my parents, Marisa and Jesús, whom I lost during the Ph.D. Their drive for knowledge and the opportunities they afforded me made me the person I am today. I will carry you with me always.

Madrid, July 2020
Lara Lorna Jiménez

Part I

Chapter 1

Introduction

"You never change things by fighting the existing reality. To change something, build a new model that makes the existing model obsolete."

R. Buckminster Fuller

1.1 Motivation and Problem Formulation

Consider, for example, an organization Ω, that has moved into the area of smart environments. Its main business is on connected cars. These have a multitude of onboard IoT devices that collect large amounts of data.

To be cost-effective and energy-efficient, they want to execute their smart car application, not only in the cloud but closer to the cars. For example, at micro data centers, at base stations, or potentially even directly on the vehicles, following the premise of the recent computing paradigm, *Edge computing*. This organization must now manage a geographically distributed resource infrastructure at scale to deploy and manage their service efficiently, instead of deploying an application at a few data centers.

These types of services can depend on numerous devices, diverse locations, and abundant resources. Thus forcing organizations, cloud providers, and telecommunications operators to pursue new ways to deploy and manage services within the brand-new context of edge computing. It is in this area of research, where this work provides the most significant contribution.

In the context of cloud and edge computing, to enable the distributed cloud, can at scale and efficient service deployment, management, and monitoring be provided such that service requirements to minimize data transfer and improve response times are met? Would edge computing solve the demands of all these services?

This question brought my work down two different routes in the same problem area. The primary path is related to efficient and scalable service deployment and monitoring.

The secondary path is about facilitating computational intelligence for certain services at device-level, for those services that may not fully benefit from edge computing.

In this thesis, for the primary path, I investigate energy-efficiency in data centers through efficient SW processes. To more easily develop and deploy SW processes, these can be sand-boxed using containers. Containerization, a form of OS-level virtualization, provides better resource utilization of the underlying HW. I worked on designing and developing a monitoring solution for containers.

Moreover, to successfully exploit the benefits brought forth by edge computing while leveraging those of cloud computing, we must establish the global manageability of these infrastructures. Consequently, the fundamental aspiration of this research is to explore an orchestrator design built on concepts of decentralization to ensure a high degree of scalability and the ability to manage a geographically dispersed and heterogeneous infrastructure. The application design and underlying infrastructure are based on containerization to provide resource-efficiency and deployment ease. Thus, the following research questions drive my work on this path:

Q1 *How can a monitoring solution be built to track resource consumption and limitations of containers, as well as container-related metadata, for an at-scale pool of container-enabled hosts?*

Q2 *How can an orchestrator that can manage a large pool of resources, surpassing the capabilities of current container orchestration solutions, which is resistant to churn, be designed? Can such an orchestrator provide global manageability across dispersed geographical boundaries to deploy and manage location-aware distributed applications?*

From the general research question posed, I established that edge computing may improve performance but may not entirely solve all applications' needs. This conclusion is particularly accurate for machine learning applications, one of the most wide-spread applications in the Internet-of-Things (IoT) and edge computing ecosystems.

Running machine learning models at the network edge instead of in the cloud improves predictive response times. Nevertheless, in many cases, this is insufficient as it does not result in the desired predictive throughput or latency. Additionally, leveraging the cloud, results in unsustainable data transfer volumes for the network, while relying on the edge cloud may — depending on the number of connected devices — also prove problematic. Moreover, in some cases, being contingent on an external system to execute the ML model is not possible or desirable. This context is where my research in this area takes place.

According to Saniye Alaybeyi, Senior Director Analyst at Gartner, "Most of the current economic value gained from machine learning is based on supervised learning use cases". This observation motivates my focus on supervised learning in this context. Consequently, this research path addresses the design of supervised machine learning algorithms for resource-scarce IoT or edge devices, often FPGAs or ASICs.

Many common machine learning algorithms require too complex or too resource-consuming implementations for the previously described context. They are unsuitable

for these devices as their designs do not fit the device architecture or do not adequately utilize it, making the algorithm inefficient. This gap drives the secondary path of my research, which deals with:

Q3 *How can purpose-built supervised machine learning algorithms be designed to work on resource-constrained devices to enable fast real-time prediction and minimize resource and energy consumption? How well do these perform in terms of predictive accuracy and latency?*

1.2 Methodology

The research presented in this thesis goes down two tracks, the inner and outer track, within the topic of cloud and edge computing, as can be seen in Figure 1.1. In the inner track, Papers A, B, and C tackle problems affecting both the centralized cloud and the distributed edge clouds. Thus, motivating why this track is positioned halfway between those two areas. Paper A deals with monitoring a containerized infrastructure, which may exist in both of these areas. The work done on Paper A led me to identify the need for a different method of providing orchestration for containerized applications on a geographically distributed infrastructure. Addressing this necessity is the research issue addressed in Papers B and C, which is a critical aspect of enabling the distributed cloud (i.e., the cloud to edge extension) and, hence, also falls across both fields.

Parallelly, the outer research track works on empowering IoT devices or edge devices, that are typically resource-scarce, to successfully run supervised machine learning algorithms locally. Papers D, E, and F undertake this problem area, operating in-between the space of devices and edge computing, though with a higher affinity to the field of devices.

The work carried out on these research topics operates on real-world problems, and produces real solutions. In this thesis, either Proof-of-Concept (PoC) or working prototypes — depending on the breadth and depth of the issues addressed — have been designed, implemented, and tested. The idea was to output a tangible result from the research, rather than exclusively running simulations. Additionally, working with the artifacts derived from these PoCs or prototypes yields a greater understanding of the problem under study.

An iterative research process was employed to bring the work to fruition. This process involves identifying a research issue, formulating a hypothesis (i.e., which leads to a series of research questions), designing and implementing a solution, carrying out relevant experiments, and analyzing the results. This process is continuously re-evaluated within each problem area until the proper research questions are adequately answered. Moreover, it also prompts new issues for future research.

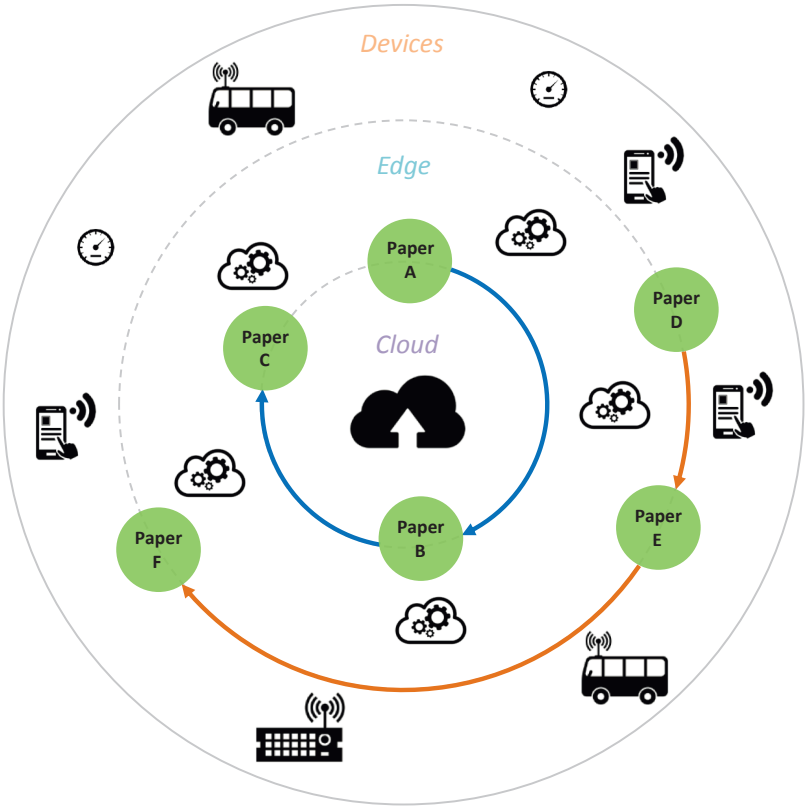

Figure 1.1: The contextualized research paper track of this thesis.

1.3 Thesis Outline

This compilation thesis is comprised of two parts. Part I offers a contextualized summary of the research carried out in this thesis. Part II presents the publications that have resulted from the research.

Part I is broken down into several chapters. Chapter 1 provides a top-level overview of the thesis research topic, presenting the research questions that motivate this work. Chapter 2 explains the research context under which this research has taken place. Chapter 3 presents a contextualized view of the work done on scalable monitoring of containers and scalable, decentralized orchestration of containers to enable the cloud-to-edge continuum. Chapter 4 addresses the issues related to enabling efficient supervised machine

learning on resource-constrained devices, such as IoT or edge devices. Chapter 5 offers a summary of the publications attached to this thesis and my contribution to these. Finally, Chapter 6 concludes Part I by answering the research questions, outlining the results obtained, and posing future research issues on the thesis topic.

Part II compiles the reformatted publications to fit the thesis layout. The attached publications include four published peer-reviewed conference manuscripts, a published journal manuscript, and another manuscript submitted to a journal for publication.

Chapter 2

Research Context

This chapter presents the research context that frames the work carried out in this thesis. My research leverages opportunities provided by this context and simultaneously addresses issues that arise within it. Consequently, this section presents an overall understanding of virtualization, cloud computing, edge computing, and the Microservice Architecture Pattern.

2.1 Virtualization

Virtualization enables more efficient use of resources through an abstraction of the physical infrastructure. This method transformed the industry, particularly data centers, concerning servers, storage, and networks. Server virtualization took its first steps in the early 1960s, when IBM launched the first commercial mainframe that supported virtualization, CP-67.

Since then, hypervisor-enabled virtualization has become widely adopted by industry. A hypervisor virtualizes the server hardware to operate virtual machines (VMs), thus supporting server multi-tenancy. Multi-tenancy is the ability to execute multiple workloads on the same hardware as if it were running on separate servers. This was a momentous shift in how software was deployed, which enabled the rise of cloud computing many years later.

There are two types of hypervisor virtualization, type I and II. The former is the one relevant in production-grade servers. There are several flavors of type I virtualization, each with different trade-offs. In type I virtualization, each VM runs a host operating system (OS), the application workloads, and the required libraries and binaries over the virtualized resources. Commonly, VM images have a size of tens of GBs. This form of virtualization offers the flexibility of running completely different OSs on the same server. Nowadays, cloud providers and organizations operate VMs through the major hypervisor vendors, such as Citrix XenServer, Microsoft Hyper-V, and VMware vSphere to ensure resource efficiency, flexibility, and security.

Another form of virtualization is containerization, an OS-level virtualization, that vir-

tualizes the OS kernel rather than the hardware. It leverages primitives like namespaces and control groups to containerize and manage processes, so these run isolated from other such processes and the underlying OS. The technology to sand-box processes has been around since the 1970s. However, its widespread use did not begin until 2013, when the Docker Engine project was released. Docker is a container platform, the first to provide a standardized packaging format based on containerization for application development, shipment, and deployment. It abstracted the application from the infrastructure, allowing uniform execution of containerized application workloads regardless of differences in the underlying infrastructure.

This is chiefly enabled through containers and their images. A container image is a standalone executable unit of software that packages an application and its dependencies. These are committed to image registries to ease reusability and deployment. Container images become containers at runtime. A container's workload can run, reliably and consistently, on any compatible container computing environment. A single server can efficiently host multiple containers, making this technology a powerful method of enabling multi-tenancy. Containers on the same host OS, share the kernel, while the containers each run isolated in user space. This can make containers very lightweight, typically tens of MB in size.

All major cloud providers and data center vendors, in their cloud-native Infrastructure-as-a-Service (IaaS) offerings, have containerization solutions. Docker is still the de facto industry solution. In 2018 83% [14] of containers in production were Docker containers. However, various Docker competitors have appeared over time: CoreOS's rkt (12%), Mesos containerizer (4%), LXC Linux Containers (1%), and OpenVZ. The appearance of different container runtimes led to the Open Container Initiative (OCI), which seeks to standardize container runtime and image specifications (i.e., runc), further improving and maturing the container ecosystem.

Containers and virtual machines are two different virtualization technologies that are not necessarily exclusive but can be complementary. They both pursue the common goals of resource-efficiency and separation of concerns, which in turn minimize costs [20]. Currently, when comparing these two technologies, we find, to some extent, a trade-off between resource-efficiency and degree of isolation.

Presently, the selling point of VMs when compared to containers is maturity and the high level of isolation these provide. These are the primary reasons why cloud providers are still using VMs. VMs are a tried-and-true, mature technology. Furthermore, the security of isolation provided by VMs is higher than that of containers, as the former do not share a kernel, which reduces the attack surface. Thus, application workloads running within a VM are independent of other VMs running on that hypervisor, and from the hypervisor itself. However, while containers are isolated from each other, they do share a kernel. The workloads within a container are visible from the host OS, and there is a small risk of an ill-intentioned containerized process breaking its containment and accessing the host OS. We may conclude that VMs are arguably more secure overall.

Containerization is a clear-cut winner when it comes to providing resource-efficiency, minimizing cost, and automating DevOps workflows. The size of a container image is

much smaller than that of a VM image, so containers take up fewer resources associated with virtualizing workloads. This means the potential of application deployment density on a given server is higher with containers than with VMs. Likewise, given their lightweight nature, containers are much quicker to spin up and take down than VMs. Container startup times typically are in the order of seconds, whereas VMs take minutes. Also, containers offer greater ease of portability than VMs, given their respective image sizes.

In the case of containers, in at scale scenarios, managing the underlying infrastructure and automating the containers' deployment becomes crucial. This is done by employing an orchestrator. There are various container orchestration solutions, such as Kubernetes, Apache Mesos, and Docker Swarm. Kubernetes is the current industry standard for container orchestration. These are all centralized solutions that have been designed to manage a cluster at a data center.

2.2 Recent Computing Paradigms

Cloud Computing emerged in the mid-2000s [57] to efficiently deliver services for users and enterprises through the Internet, bringing about a technological revolution in the Information and Communication Technology (ICT) sector. Cloud computing is the key enabler of the trend of Everything-as-a-Service (e.g., IaaS), through public, private, or hybrid deployment models. Its prevalence is primarily due to the successful exploitation of centralized resources via economies of scale, as provided by hyper-scale data centers present across the globe [57]. Data centers attempt to maximize resource-, energy-, and cost-efficiency while maintaining Service Level Agreements (SLAs). Cloud computing offers on-demand, flexible, and reliable compute, memory, storage, and network resources [5, 6].

This model's strengths lie in having services or applications operate at these centralized locations, with elastic provisioning, high availability, and resiliency, reachable from across the world. All these characteristics, opened the door to *Mobile Cloud Computing* (MCC) [21], where mobile devices could leverage the cloud infrastructure for computation offloading to accelerate application execution and reduce energy consumption [54, 69].

Cloud computing is a consolidated model, and it is likely to persist in the services' ecosystem for the foreseeable future. However, emerging businesses and applications are imposing requirements that are challenging to fit within the paradigm of cloud computing exclusively. These new services are generally, data-intensive, often with real-time constraints, and commonly, relying on large volumes of connected 'things' or users. These frame the core hurdles that cloud computing is not equipped to handle on its own adequately.

These stumbling blocks encountered by cloud computing have been prompted by the disruption of IoT and its related applications [8, 74], as well as, other *data-intensive* applications. IoT is a ubiquitous technology that extends the traditional Internet to connect it to embedded devices, often having sensors, actuators, and RFID tags [61]. These collect and act on data from their environment, setting requirements for processing, scalability,

communication, and storage [6]. The advancement of mobile, pervasive devices has facilitated the rise of IoT while simultaneously provoking a fundamental problem for the execution of their associated applications [67].

These networked IoT devices share the limitations inflicted by mobility: limitations on on-board resources, such as computational power, storage, and energy. Consequently, for resource-rich applications, these devices find themselves having to leverage the cloud infrastructure, much like in MCC [58]. Currently, the problem with that scenario lies in the vast number of IoT devices, and the high volume of data streams these produce [31]. Thus, these devices generate an ingress bandwidth to cloud infrastructures that telecommunications networks do not support, and which is also energy-inefficient, causing a larger carbon footprint. Also, the growing number of mobile devices leads to *scalability* issues for cloud computing infrastructures.

The near future will only further exacerbate this situation. According to Cisco Annual Internet Report [63], globally, there will be 29.3 billion networked devices by 2023, of which 13.1 billion will be mobile, and there will be 14.7 billion machine-to-machine (M2M) connections. Statistica establishes, as per recent studies, that the data volume of connected IoT devices worldwide is forecast to reach 79.4 zettabytes (ZBs) by 2025 [62].

The network infrastructure is not dimensioned to sustain the data volumes associated with these applications and devices. This has been recently validated by the increased demand exerted on the telecommunications networks because of governments' confinement measures across the world due to the SARS-CoV-2 pandemic. The network operators have had difficulties managing this increase in network traffic [13], to the point where large content providers, such as Youtube, Amazon Prime Video, Netflix, or Facebook, had to reduce their bit rate to aid in minimizing network congestion. Therefore, having these mobile devices put such stress on the network infrastructure will undoubtedly lead to network congestion and poor performance of the applications.

A study on public cloud providers [41] demonstrated that the average round-trip-time (RTT) between 260 global points and their corresponding optimal Amazon Elastic Compute Cloud (EC2) instance was 76 ms. Therefore, reliance on cloud DCs to alleviate the resource poverty of IoT and other mobile devices is not adequate for *latency-sensitive* applications, which require strict bounds on their latency. That is the case of, for example, augmented reality (AR), which establishes a maximal latency on the lower end of tens of milliseconds [58]. Another popular application example is machine learning, particularly when coupled with automation, where predictive latency becomes a decisive factor to performance. In this context, latency remains an obstacle for many real-time and immersive applications, even when good bandwidth is available [58].

The cited setbacks encountered by cloud computing have encouraged a dispersion of the infrastructure to provide decentralized processing [67], and are the main drivers behind the notion of the distributed cloud powered by *Edge Computing* [5,6,57,61]. This paradigm has emerged in recent years and has gathered momentum as research interest, and industry investment has increased [57].

The fundamental idea behind edge computing is to have a resource-rich infrastructure with reliable connectivity, likely with cloud computing characteristics, at the edge of the

network. This edge infrastructure allows the execution of services or applications in proximity to the service-consuming end-users or data-generating devices. These services are packaged within VMs, containers, or both, to maximize resource utilization, improve isolation to facilitate security, resource management, and metering, much like in cloud computing. Furthermore, the edge provides context awareness, flexibility, scalability, and agility, facilitating the increase of Quality-of-Service (QoS) and Quality-of-Experience (QoE), while decreasing costs, optimizing resources, and generating new avenues for revenue.

The definition of 'edge' varies widely [27, 48]. It may refer to the LAN network, the cellular network operators' base stations, the cellular network operators' core network, the end-devices, or others [31,67]. This highlights the reason behind a diverse terminology for this dispersed infrastructure — edge nodes, fog nodes, mist nodes, cloudlets, MEC, or micro data centers — which depend on the context that initially defined them.

Edge computing pursues solutions to the drawbacks of leveraging cloud computing for IoT [27, 57, 74] and mobile device [48, 58] applications. First, it provides proximity of the services to the edge, reducing end-to-end latency, lowering jitter, and increasing bandwidth, which translates to highly responsive services. Second, it lowers bandwidth utilization from the edge to the cloud and back, by performing edge analytics. This action increases scalability by intelligently handling more devices and their data streams, at the edge. Third, data management at the edge supports policies for data security, privacy, and integrity. Fourth, services operating at the edge can mask temporary failures of the distant cloud, since the edge has a higher likelihood of communication survivability.

Industry has tackled edge computing through different approaches. These efforts have resulted in Fog computing [6, 7, 12, 17, 18, 27], Cloudlets [6, 24, 57, 58, 67], MEC [25, 55, 67], and micro datacenters [4, 6]. In this thesis, these are considered to stand under the umbrella term of edge computing. These technologies originally targeted slightly different objectives, though some of their differences have blurred and much overlap may be found.

In some cases, even edge computing has its limitations, such as in the case of some machine learning services, where running computational intelligence at the network edge is not enough or not an option. Accordingly, in such cases, we must resort to deploying the machine learning models directly at the end-devices, or edge devices if applicable. Many of these devices rely on FPGAs, ASICs, or microcontrollers. These are resource- and energy-efficient, can be size-friendly at the expense of being resource-constrained, and can efficiently target the needs of the specific application. Additionally, in the case of FPGAs, these rely on Reconfigurable computing, making this hardware particularly attractive for IoT devices. Nevertheless, this imposes further constraints on the machine learning algorithms themselves, which drives the research issue of adapting or creating new algorithms to fit these conditions.

2.3 Enabling Applications in the Cloud-Edge Continuum

Another aspect to consider in the previously presented context is the application architecture of the cloud and edge computing applications, and those that wish to leverage both complementing paradigms.

To fully exploit the benefits brought on by the cloud computing model, *Cloud-native* [22] was born. It affects the way an application is designed and deployed. Cloud-native is based on three principles: containerization, microservices, and dynamic orchestration, allowing applications to be infrastructure-agnostic and horizontally scalable. This implies that cloud-native applications are broken down into a series of loosely coupled services, following a Microservice Architecture Pattern [45], which eases scalability and development. Each of these services is containerized to facilitate portability, reproducibility, transparency, and resource isolation. Finally, an orchestration platform manages these containerized services to elastically schedule them, providing resiliency and availability, service discovery, and resource utilization optimization.

Building a microservice application can be beneficial, but it is not without its difficulties. It removes complexity from the individual services but creates a distributed system, which means greater complexity at the system level. One of the chief complications is a high communication complexity between these fine-grained services. Therefore, it should not be lightly adopted for all applications, as it may be less complex and more efficient to design them as monoliths. As such, as a rule-of-thumb, the Microservice Architecture Pattern makes sense only for complex applications that require scalability.

While "Cloud is about how you do computing, not where you do computing," according to VMware CEO Paul Maritz, the same is not valid for edge computing where context is very relevant. This led to the *Edge-native* [59] initiative, which establishes that applications should be designed to move processing (i.e., code) to the edge, so data is treated there, rather than shipping all data to a central cloud. Consequently, an application that wishes to leverage both cloud and edge should employ different geo-distributed components, where some are dynamically executed at various edge clouds while others are run in central clouds, likely enabled through an orchestrator. This is a good match for the Microservice Architecture Pattern as well. Accordingly, these applications operate across various tiers depending on the available infrastructure, the application design, and the high-level application requirements provided to an infrastructure-aware global orchestrator.

Chapter 3

Scalable Orchestration and Monitoring of Containers

This chapter covers two research areas within this thesis related to research questions **Q1** and **Q2**. First, this section provides a contextualized explanation of the proposed resilient orchestration solution based on decentralization, particularly for dispersed infrastructures and applications. Second, it also presents a summarized view of the research done on a distributed and scalable monitoring system for containers.

3.1 A Decentralized Container Orchestration Platform

The requirements demanded by some existing services (e.g., online-gaming) and the emergence of new services (i.e., IoT-related services) have brought about a paradigm change from relatively more simple cloud-based applications running at data centers, to more complex applications that must operate across different geographical locations. This is brought forth by the need to extend the cloud, from centralized data centers, to the network edge.

As has been shown in Section 2.2, cloud and edge computing are two complementary paradigms. While an application may operate uniquely at the edge or solely within centralized clouds, for IoT and other data-intensive or latency-sensitive applications, leveraging both scenarios is the winning strategy. The research focus in this area is on enabling infrastructure manageability in the cloud-edge scenario [49]. This is an identified open issue in related literature.

For example, M. Satyanarayanan et al. coined the concept of Cloudlets, and stated that "there are also substantial hurdles in managing dispersed cloudlet infrastructure" [57], identifying it as a critical technical problem for enabling cloudlets, and therefore, more generally, edge computing. Further, they consider that "one of cloud computing's driving forces is the lower management cost of centralized infrastructure. The dispersion

inherent in edge computing raises the complexity of management considerably.". Therefore, proposing innovative technical solutions to resolve this issue is a research priority for edge computing.

Current container orchestration solutions have a limited ability to scale and are designed to orchestrate at cluster-level, and are, therefore, ill-equipped to orchestrator geo-spatially spread infrastructure [52]. For example, Kubernetes, the industry standard for orchestration of containers, is a centralized solution that can only scale up to 5000 nodes [38]. Since these orchestrators do not meet the mentioned requirements, new orchestrator solutions are necessary. An effort in this direction is currently being made by Google's KubeFed [39], which is a federated Kubernetes, meaning it coordinates multiple Kubernetes clusters through federation. This project is currently in its alpha phase. Moreover, another recently started project, KubeEdge [73], is underway to attempt to meet the requirements imposed by infrastructure at an edge for the orchestration of containers, such as intermittent connectivity.

A new orchestration solution in this new paradigm must provide global manageability over a large, geographically spread out number of resources or nodes [30]. These nodes may be physical or virtual entities with the ability to host containerized processes. These nodes are likely heterogeneous, some of which may, potentially, be mobile. Nodes may join, leave, or intermittently connect to the orchestrator network. Moreover, the orchestrator must robustly automate deployment in this new context by supporting a more seamless orchestration of cloud-native/edge-native applications in the form of geo-spatially spread containerized microservice applications.

This thesis proposes an innovative Proof-of-Concept orchestrator, HYDRA, that seeks to meet these objectives. The design of HYDRA focuses, from its inception, on providing scalability via principles of decentralization supported by a peer-to-peer (P2P) distributed hash table (DHT) overlay network. The functionalities that enable orchestration are designed on top of this overlay, pursuing high availability, dynamicity, resiliency, and partition tolerance.

The orchestrator and the applications, or other orchestrator deployment objects, it deploys are two distinct areas of concern. The orchestrator acts as a middleware to make the underlying infrastructure and its complexities transparent to the deployment objects. Some of the previously listed characteristics, such as high availability (HA), scalability, and resiliency, must be supported by the orchestrator. Nevertheless, others must also be simultaneously, innately enabled by the deployment objects themselves.

For example, HA of the orchestrator does not imply HA of the running applications. The former is provided by the very nature of this orchestrator's design, by automatically managing replicas according to submitted deployment object specifications. In the case of the high orchestrator scalability, it refers to the volume of nodes that can be part of the orchestrator. The structured P2P overlay network builds orchestrator scalability likely in the hundreds of thousands. In contrast, application scalability is the ability to scale to meet the demand the application is experiencing, which is granted by the orchestrator that scales it according to the specifications provided. Nevertheless, how energy-efficient or resource-conscientious, and how cost-effective the application is when meeting the

demand, depends on its architectural design, which is outside the orchestrator's purview.

Orchestrators seek to automate application deployment and infrastructure management. The objective is for the orchestrator to do all the heavy-lifting, so, in a sense, application developers may plug-and-play their applications. This transparency is not without its drawbacks [66]. Orchestrators are complex systems. Inevitably, so is the proposed orchestrator, perhaps more so, in light of its decentralized architecture.

In traditional, centralized orchestration solutions, one or a small group of nodes act as master nodes, whereas the rest operate as worker nodes. The master provides the intelligence, the control plane. The workers acquiesce to the control demands; they are the data plane of the orchestrator. In contrast, the proposed orchestrator builds a flat architecture of nodes, where all nodes are independent and operate autonomously. Each node runs both the control and data plane.

Consequently, a node may both host deployment objects and manage remote deployment objects. The communication of these nodes builds the orchestrator pool and brings to bear the orchestrator's functionalities. Essentially, a partitioning scheme is adopted, such that autonomous nodes provide scalability, while control of each deployment object is independent and self-organizing, providing system robustness [30].

This orchestrator is structured through IDs, meaning nodes can leverage the orchestrator structure to search across the orchestrator network for a given ID. Accordingly, the orchestrator resources, such as deployment objects, some deployment object-related resources, nodes, and other orchestrator resources that need to be discoverable, have an associated ID to be seekable by the orchestrator nodes.

To design an orchestrator that aligns with the cited context requirements, I have identified some key system functionalities during my research on this topic. Many of these are common to container orchestration systems. However, the solutions to enabling these functionalities differ significantly from that of traditional orchestrators, since this is a decentralized orchestrator.

Papers B and C cover some of these functionalities, at least in part, while others have been left out-of-scope. The following subsections provide a short, contextualized account of the work published in these areas, without repeating their content. Some of these subsections also present explanations on the necessary considerations for those out-of-scope functionalities. Additionally, some of the research carried out is not included in the papers compiled in this thesis. I also provide an insight into that work.

The core requirements or functionalities that I have identified and designed to enable the proposed decentralized orchestration system are: *context awareness*, *node discovery*, *management of deployment objects*, *replication and scaling*, *service discovery and load balancing*, *scheduling and searching*, *multi-tenancy*, *distributed consensus*, *platform-independence*, *networking*, *security*, and *monitoring and logging*.

3.1.1 Context Awareness: Location and Latency

The proposed orchestrator must successfully manage an at-scale infrastructure spread out across various geographical locations. Simultaneously, it must manage deployment

objects, such as applications whose services may have to operate at different locations according to user-specified constraints. This requires a location-aware mechanism that fits the orchestrator design. Moreover, latency is another useful context variable that each orchestrator node may apply to improve orchestrator performance.

A location-aware design based on IDs is presented in Paper C, enabling it to operate as either a location-agnostic or location-aware orchestrator. In contrast, Paper B's orchestrator is location-agnostic. Paper B was a first step towards the desired orchestrator, and its focus was on decentralization to achieve scalability rather than on location awareness.

The proposed location-aware design entails mapping the ID-based scheme the orchestrator employs, to location, similar to the network IP addressing scheme of CIDR blocks. These IDs are employed for node discovery, deployment object discovery, application management, scheduling, and other orchestrator functionalities. In the current orchestrator design, the location-aware ID-based scheme must be predefined with some forethought to adapt to changes in the infrastructure.

An external system to the orchestrator performs the initial assignment of the location node IDs, reassigns location node IDs for mobile nodes, and provides the location ID design to the nodes.

3.1.2 Node Identifiers and Discovery

The proposed orchestrator is designed on principles of P2P to ensure high scalability through decentralization. Nodes that join or are part of the orchestrator must be able to find other orchestrator nodes without a centralized system enabling this. Thus, nodes self-register to the orchestrator network employing an ID-based identifier scheme coupled to a node discovery strategy based on Kademlia's Peer-to-Peer (P2P) distributed hash table (DHT) and node lookup algorithm [43]. In Paper B, node IDs are location-agnostic. In contrast, in Paper C, each node carries two identifiers, a location-related node ID and a location-agnostic node ID, enabling a mechanism of dual node discovery.

3.1.3 Management of Deployment Objects

In the orchestrator, various deployment abstractions may be employed: *Job*, *StatelessSet*, *Service*, and *Application*. These objects describe the deployment types that have been currently considered.

An Application is composed of one or more Services, each wrapping a StatelessSet object, containing one or more co-located Capsules, where a Capsule is formed by one or more tightly coupled, co-located containers. A Capsule resource is equivalent to a Kubernetes Pod [11].

Papers B and C employ a simplified version of an application, by defining an application as a set of services, each composed of a single container, where these services may intercommunicate. The application abstraction exists with the intent of better supporting the Microservice Architecture Pattern [2]. Nevertheless, the orchestrator's deployment objects support any application architecture, since distinct use cases might have different

needs to support their end-devices functionality [27]. The focus of the orchestrator research presented in this thesis has been on microservices. These closely match the needs of applications wanting to leverage cloud and edge computing [55], as has been explained in Section 2.3.

A deployment resource, such as an application has a unique identifier, an ID in the same address space as the node IDs. This ID plays a vital role in application management and discovery. There is no central control in this geographically distributed and decentralized orchestrator. Therefore, this orchestrator offers a self-administered, adaptable set of nodes to manage each application in pursuit of its services' availability and resiliency. Specific roles –presented in Papers B and C– enable this behavior. One or more nodes dynamically adopt these roles on the self-organized orchestrator network. This design favors the orchestrator's adaptability to changes in the overlay network.

The node or nodes providing management of a deployment object acquire the controller role. The controller strives to meet the desired state of the deployment object. The tasks carried out by this management entity depend on the deployment object specifications provided. Some of these tasks may include creating, modifying, deleting the deployment object and its associated containers, searching for resources and criteria to deploy the corresponding containers, tracking the state of hosting nodes, self-healing of capsules, maintaining replicas, autoscaling replicas, carrying out health checks for the service endpoints, carrying out service rollouts or rollbacks, configuring service reachability, ensuring discovery between application services, and others.

The orchestrator's management strategy for all deployment types is similar, except for when the application resource defines its services with location affinity. In Paper B, the deployment of an application is unrelated to the service location. Thus, the application ID determines the nodes that manage the application. In contrast, Paper C details a location-aware application deployment where the application ID, and service IDs derived from the application ID, facilitate application and related service management. This location-aware management ensures proximity to the managed services to guarantee orchestration and application performance.

3.1.4 Replication and Scaling

To provide resiliency and enable high availability of a service, the orchestrator creates and maintains healthy replicas of the service. Additionally, a feedback-driven orchestration of the deployment objects is necessary to detect changes in infrastructure performance and QoS metrics [68]. Thus, containers are ephemeral and may be dynamically substituted by new instances.

This replication functionality is offered by, for example, the StatelessSet deployment type. StatelessSet provides a mechanism to replicate the capsule or capsules on different hosting nodes in the orchestrator. Paper C presents a replication strategy for stateless services in the proposed orchestrator. This strategy involves the replication of capsules as passive or active replicas. The number of replicas that must be maintained may be fixed, or an autoscaling function may be attached to the deployment type. Automatic

horizontal scaling sets an upper and lower limit on the replica size. The replica set is automatically scaled within those limits according to some orchestrator metrics, such as, the average CPU utilization of the running replicas, which is a workload-agnostic metric. However, it would be beneficial to perform scaling according to other custom metrics that are workload-specific.

A controller of the deployment object enforces the control, self-healing, and scaling of replicas. The current orchestrator design does not consider stateful replicas since that entails persisting container state to volumes, which requires storage orchestration that the design does not contemplate.

3.1.5 Service Discovery and Load Balancing

Service discovery and load balancing are necessary aspects of an orchestrator. However, the orchestrator-related publications in this thesis do not cover these specific areas. Nevertheless, during the research on this topic, these have been considered.

An orchestrator may scale a service to satisfy demand. When a service is scaled, more or fewer instances, or merely different instances, of that service become available. Load balancing allows these replicas to serve clients transparently and distribute load among the replicas according to a load balancing algorithm. For its part, service discovery facilitates that a client looking to communicate with a service will successfully reach one of those service replicas wherever they may be.

A Service deployment object makes the replicas discoverable to other SW components from inside or outside the orchestrator, so that the service replicas may be ephemeral and dynamically discoverable, allowing a service to operate reliably.

The focus has been on, in-orchestrator rather than on external, service discovery. However, several solutions could be adopted for the latter feature, such as proxying to the service before binding of service port to node port, setting up an externally reachable load balancer, or through an ingress controller. Internal orchestrator service discovery may provide native and non-native service discovery. A client (e.g., a front-end service instance) running within the orchestrator may choose to employ native service discovery by requesting the service endpoints (e.g., of a back-end service) also deployed in the orchestrator, through the API server locally present at all orchestrator nodes. Then, the API server retrieves from the orchestrator network the requested resource.

In this orchestrator, non-native service discovery relies on proxying to forward inbound traffic to the required service, following a server-side discovery pattern with optional load balancing. With that purpose in mind, service controllers generate a service Virtual IP (VIP). This VIP is static for the service and associated with the variable service endpoints, which are the service replicas. This association resides within the corresponding service controllers and the service registry, so other orchestrator nodes are unaware of it. The main issue in this situation is that there is no centralization, and distributing this information to all orchestrator nodes would not be practical or even workable. Therefore, when a service client wants to communicate with the service, the node hosting the would-be client, retrieves the necessary service information from the orchestrator network via

ID. It registers to the corresponding controllers to obtain the updated service endpoints. It also sets up the necessary routing rules on the node to route packets to the service endpoints. It proxies the client requests to one of the service replicas with the possibility of load balancing over these replicas.

The idea behind an Application deployment abstraction is to enable the interaction between multiple Services seamlessly. As such, it would make sense to expand on the previous solution to build a service mesh [26, 42] for application inter-service communication. This mesh could be employed to facilitate, configure, and monitor routing among services in the manner of the Microservice Architecture Pattern [34]. Various service mesh solutions exist, such as Istio [60] or linkerd [29].

Whether the service mesh should be built at node-level or run at capsule-level as a sidecar, should be studied. In any case, a typical service mesh operates through a centralized control plane that pushes rules to its agents, which run at the service hosting nodes. This approach fits the proposed orchestrator's application management strategy when the application is location-agnostic. However, when it is location-aware, it does not, as the application control architecture is separated into various service controllers (i.e., the leaf controllers presented in Paper C). Thus, the service mesh control plane, for a location-aware application, must be built through coordination of the different service controllers of an application.

3.1.6 Scheduling

Given the decentralized nature of the proposed orchestrator, a node may find itself in charge of deploying one or more capsules, a StatelessSet, or a Service. However, that node is unaware of the nodes on the orchestrator that might fit the deployment object's specifications. In a centralized orchestration solution, a central location holds the information about all nodes. Finding a node becomes a matter of selecting from the entire pool of available nodes, the node or nodes that are best suited to host the containers. However, scheduling in a decentralized orchestrator requires first finding nodes and then establishing the viability of the nodes found. *Node viability* is the capacity a node has of hosting a deployment object, given its requirements.

This research established two approaches to tackle this problem. To either decouple the search of nodes from the scheduling of deployment objects to nodes entirely or not to. Fully decoupling these two processes would more closely match the scheduling approach taken in a centralized orchestrator.

This approach could, potentially, ease design and implementation complexity, but at the expense of effectiveness and efficiency. Therefore, the option chosen was that of not fully decoupling scheduling from searching. Thus, in this orchestrator, node viability is not decided by the node attempting to deploy an object, but locally by each node that obtains a viability query.

This orchestrator builds a structured P2P overlay network organized through IDs. When the orchestrator operates as a location-aware orchestrator (i.e., in Paper C), this structure provides an affinity to location. However, it bears no relation to other node

characteristics that may be relevant when looking for viable nodes.

Filters

Filters describe the requirements of a deployment object. These filters may refer to resource availability, location, node labels, specific nodes, image availability, affinity and anti-affinity constraints, storage, hardware, software, or other policy requirements. The values of a set of these filters are employed to match, for example, a capsule wishing deployment, to the corresponding node characteristics. The location filter of a deployment object determines where to search for viable nodes on the orchestrator network. The other filters, permit a node to determine its hosting viability for a given deployment object. Paper C employed the location and resource availability filters, whereas Paper B only used the latter.

Search for resources

To provide nodes with the corresponding filters so they may decide whether they are viable nodes for that deployment object, we must first find nodes on the orchestrator network. Thus, a key concern becomes successfully searching on the orchestrator network for nodes, while leveraging the orchestrator properties, attempting to distribute the load evenly across the orchestrator network, maximizing the probability of reaching new nodes during a search, and if possible, maximizing the likelihood of finding a viable node, if it exists. To that effect, this research considered two options.

The **first option** deals with devising search algorithms to adequately search across the ID-space, despite there being no correlation between IDs and what we are searching for. This research presented two search algorithms in this category, partially in Paper B, and more fully in Paper C. These are the *Random ID* search algorithm and the *Maximized XOR Distance between Queried IDs* search algorithm. The former establishes a baseline search algorithm for this orchestrator. Paper C presents the quantitative results for this baseline. For the latter search algorithm, the research papers do not present a quantitative analysis. However, preliminary results suggest a 20% improvement in the number of messages needed to successfully deploy an application (i.e., with the specifications defined in Paper B), compared to the Random ID search algorithm.

Both of these search algorithms assume the node processing deployment holds no prior knowledge about the characteristics of other nodes on the orchestrator. However, a design variant would have nodes provide the relevant filter-related information to nodes on the orchestrator network that they come into contact with. This strategy could positively impact searches, though at the expense of exchanging more data in each query sent.

The presented first approach provides search algorithms resulting in best-effort solutions while benefiting from the existing orchestrator structure. However, this approach does not guarantee to find a viable node on the orchestrator, if it exists. This led to the **second option**, which builds on the orchestrator's ID design to create a self-organizing structure of nodes, targeting the node characteristics that filters represent. This second approach, which has been designed but has not been the focus of any of the research pa-

pers in this thesis, intends to ease filter-driven searches in this decentralized orchestrator and increase the likelihood of finding nodes that can host the deployment objects.

Scheduling Profile

The current scheduling profile in this orchestrator deals with a pre-processing policy, followed by a filter-enabled search, and finally, a scoring policy. Pre-processing determines which information to include when requesting a node to confirm or deny its viability and how viability should be confirmed. In short, it dictates what the viability query to a node will contain. The pre-processing carried out in Papers B and C were scope and ordering. The scope established the containers for which the node receiving the request would determine viability. Ordering sorted the containers by the only filter (i.e., resource availability filter) employed in descending order for the single resource under consideration, memory (i.e., RAM). Thus, during the filter-driven search, via one of the search algorithms (i.e., the Random ID search algorithm), a node receiving the viability request would respond for the first, and therefore the largest service that it could host.

As nodes confirm viability, the scoring policy comes into play. Scoring establishes fitness. Therefore, it determines the fittest node to host a service, out of the viable nodes found, or which service –out of the ones that we are attempting to deploy– would best fit a given node. In Paper C, this entailed that as a node confirmed viability for a service, the service would be bound to that node for deployment unless that service had already been bound, in which case it would bind the next service, which would be smaller. Thus the node would also be able to host it. This particular policy follows a first-fit decreasing (FFD) algorithm [19], but it would not be feasible when using more resources or filters unless a strict priority was given.

3.1.7 Multi-tenancy

Multi-tenancy is an important feature that allows orchestrators to have different actors using the underlying infrastructure, where limits can be set for each actor. This feature is necessary for organizations with different workloads, costumers, or teams sharing the same managed infrastructure, such as for development, testing, and production use cases, or solely for multi-application deployments. Like Kubernetes, centralized orchestrators enable orchestrator-wide namespaces. These set a scope for naming and limiting part of the cluster for some users. In contrast, as the proposed decentralized orchestrator is in its early stages, only workload multi-tenancy is considered in its current design. A tenant is simply a deployment object (e.g., an application). As such, the limitations are set on a per-object basis.

3.1.8 Distributed Consensus

When the proposed orchestrator deploys an object, such as a service, the node on the orchestrator network whose ID is closest to the service ID acquires the controller role

for that service, which is to say the management role. This node is in charge of various control functions. It deploys the corresponding capsules, performs health checks of hosting nodes and capsules, enforces automatic horizontal scaling or maintains a specific replica set, redeploys failed capsules, makes the service discoverable, or other functions. The node with the controller role becomes the anchor to this service within the decentralized orchestrator. Moreover, a node may act as an anchor to multiple deployment objects. However, nodes are prone to involuntary or voluntary disruptions, thus becoming unavailable. If that were to occur, the deployment object, in this case, the service and its associated containers, would become orphaned. To mitigate the possibility of this occurring, multiple nodes (i.e., typically 3 or 5) could acquire the management of an orchestrator object. This functionality requires a distributed consensus.

Distributed consensus ensures various agents, or nodes, reach state agreement, such as on a value committed to a data store cluster, or on which node is the cluster leader. This field is a complex research area in itself. This thesis does not seek to provide a final solution to this particular problem in the orchestrator. However, it does consider some design options.

The nodes that take on the controller role (i.e., N nodes), form the logical controller cluster for that service. The controller role is composed of control functions and data. The control functions enable the previously listed features (e.g., deployment of capsules). The control data is necessary to maintain the desired and track the current state of the service. Therefore, the controller cluster for a service can be separated into a control plane and a data plane.

Given the two planes that the logical controller cluster may be broken down into, this research proposes two topological designs. These enable the logical controller cluster with strong consistency and partition tolerance:

External: The initial service controller sets up a distributed data store cluster as the data plane for the service, separate from the service controllers, which only host the control plane. The data store cluster may use the Raft consensus algorithm [47], like an etcd [40] cluster does, to perform leader election and data replication in its cluster. Further, the data store registers that initial service controller as a master and determines it to be the leader, not of the data store cluster but of the N nodes.

The data store also finds, via an orchestrator client, the N nodes closest to the corresponding service ID. The data cluster registers those nodes as masters and elects a leader, the one whose ID is closest to the service ID. Further, the data store also ensures that the list of N nodes registered reflects the nodes on the orchestrator closest to the service ID. It registers, de-registers, and elects the leader among them as needed. The leader is only re-elected when the previous leader has been de-registered. The N nodes' leader is the one to perform changes to the service and write them to the data store. Further, the N nodes' leader ensures that the distributed data store cluster always has the required number of healthy nodes.

Attached: This topology implies that both the control and data plane run on the

service controllers, that is, the data store cluster and the N nodes are merged. Therefore, each service controller locally stores the service data. These data must be consistently replicated among the service controllers. This topology may employ Raft, but the elected leader among the service controllers must ensure that the service controller cluster is composed of the nodes whose IDs are closest to the service ID. Meaning membership to the service controller cluster must be modified to reach that desired state, which should be done in a rolling update fashion.

3.1.9 Other functionalities

Platform-independent orchestrator

The nodes of the orchestrator may be heterogeneous on multiple fronts, not only in terms of hardware. This heterogeneity may be due, but not limited to, the OS, the onboard container runtime, or underlying cloud or edge provider infrastructure. Therefore, the orchestrator should provide multi-platform and multi-provider compatibility. This strategy abstracts the orchestrator functionality from the underlying infrastructure, thus pursuing the separation of concerns.

Networking

Networking is an essential aspect of an orchestrator. It enables routing packets between nodes on the orchestrator network (i.e., node-to-node (N2N) routing), between a host and the containers locally deployed, between containers on the same node, and between containers on different nodes (i.e., container-to-container (C2C) routing).

The in-host networking functions are, for the most part, readily provided by the container platforms. In the proposed orchestrator, nodes must expose their IPs during node discovery, such that other orchestrator nodes can route to the node using that IP. In other words, N2N network reachability is a prerequisite of the current orchestrator design. Further, to enable C2C networking, the corresponding unique CIDR blocks for the container runtimes must also be facilitated. The current orchestrator design only lightly touches upon this area. However, multiple solutions work on this topic, such as Flannel, Weave, or Calico [1].

Security

Security is a key feature needed by distributed systems. Paper B lightly touches upon this aspect of the orchestrator. There, public-key cryptography restricts access to application owners to obtain information or modify an application. However, there are many other security considerations across the stack [51]. These relate to both the applications and the orchestrator, which should be addressed. While these considerations are out-of-scope for this thesis, they should be researched so that the security solutions fit, as much as possible, the requirements imposed by the orchestrator's decentralized architecture.

Monitoring and Logging

Another two features essential to an orchestrator, which this thesis does not actively address, are monitoring and logging. While both are necessary for the applications themselves, the orchestrator must also have these functionalities to detect and diagnose possible problems in the system. This is particularly true for the decentralized orchestrator proposed. While it provides greater scalability than a centralized orchestrator, it is also more complex and, therefore, more difficult to troubleshoot.

3.2 Scalable Monitoring of Container-based Infrastructures

In light of the importance of containers for cloud and edge computing, particularly their potential for scale, which is far greater than that of VMs, part of the research is on the field of scalable monitoring of containerized infrastructures, specifically of the Docker container platform.

Monitoring is essential to ensure successful application deployment and management, though it is often undervalued. Monitoring entails obtaining telemetry to detect and diagnose issues, prevent failures, and troubleshoot problems, thus allowing teams to identify and resolve these quickly.

Monitoring is necessary at all levels of the stack. However, the research related to monitoring in this thesis refers to monitoring the container-based infrastructures, with at scale monitoring considerations. Since containers operate across the DevOps workflow (i.e., development, staging, and production), efficient monitoring is particularly important. Therefore, this research focuses on monitoring the containers themselves, not the containerized application workloads (i.e., application-level metrics).

Paper A addresses this particular line of research by proposing a scalable container monitoring system. The solution was carried out on Docker Engine v0.92 through v1.3.1 when the transformation of application development and deployment through containerization had just begun. At the time of writing this thesis, the Docker Engine is on v19.3.

When this research took place, many metrics that are now integrated into the Docker platform or available through other tools, were not present. The Docker platform would provide only a couple of top-level metrics about the containers. External tools like Google's cAdvisor existed, but it too provided only what Docker itself provided, with a few more metrics. Moreover, it did not visualize historical data as it only allowed viewing those metrics in near real-time, and no architecture to facilitate at scale monitoring was proposed. Paper A attempted to fill this gap by providing a monitoring agent to collect a large number of container resource usage and limitation metrics and container platform-related information. This agent was complemented with an existing host monitoring agent (i.e., Host sFlow). Additionally, a collection architecture to monitor container-enabled hosts at scale was proposed, which included visualization of the collected information.

Given Docker's relevance, even in its early stages, the industry was working towards similar goals. Since the release of Paper A, many industry solutions, both proprietary and open-source, have become available or improved their solutions: cAdvisor, Datadog, Prometheus, ELK (Elasticsearch, Logstash, Kibana), Sysdig, Zabbix, Nagios, among others. Further, the Docker platform itself has evolved to natively provide many of the metrics supported in Paper A.

Monitoring of at scale container-based infrastructures is not only about the measurements obtained but also about how telemetry is collected and managed to support scalability. Scalability is an essential requirement of a monitoring system for one of the research interests presented in this thesis, the design of a highly scalable, decentralized orchestrator of containers.

Monitoring of orchestrators is necessary to oversee the orchestrator itself, and also the container-based infrastructure. Besides, obtaining telemetry in orchestrators is about identifying and tracking issues in the infrastructure, and enabling certain orchestration functionalities (e.g., service scaling). In centralized orchestrators, this boils down to cluster telemetry. However, given the inherent complexity of decentralized systems, architecting a monitoring solution for the decentralized orchestrator proposed in this thesis would be more challenging.

Proposing a monitoring solution for the proposed orchestrator design is out-of-scope in this thesis. However, it is worth noting that Paper A achieved a scalable monitoring setup through federation, but it would not be the best fit for the proposed orchestrator. The reason for this is that it would be a monitoring system entirely independent from the orchestrator. Thus, it would not benefit from the orchestrator's architecture and vice-versa. Paper A's design would align better with KubeFed's architecture. The monitoring system design that would best suit this thesis' proposed orchestrator would leverage the orchestrator ID-based design, likely tied to the succinctly mentioned self-organizing system designed around scheduling filters (Section 3.1.6).

3.3 Use cases

The edge computing paradigm has attracted the attention of various industries. These are attempting to enable edge computing from different perspectives. The leading edge computing technologies are Cloudlets, Fog computing, MEC, and Micro datacenters, as has been mentioned in Section 2.2. Each of these is pursued by different actors, each proposing their own infrastructure at the network edge, according to their specific edge definition. These edge computing proposals can be complementary, and, at times, their differences may be somewhat ambiguous. However, they do have a certain degree of bias towards the specific industry market that furthers each initiative.

For example, Cisco promoted fog computing, composed of fog nodes distributed in a multi-level hierarchy spanning the network space between the cloud and the edge [12]. These nodes may be either physical or virtual switches, routers, servers, access points, or any network device that presents some virtualization hosting capabilities. Cisco is a preeminent network HW vendor to telecommunications companies, which motivates

their interest in moving into this space. Many companies have joined this enterprise, broadening the scope of fog within the Open Fog Consortium [10].

On the other hand, MEC is sponsored by telecommunications operators. MEC initially stood for Mobile Edge Computing and was later renamed to Multi-Access Edge Computing. Network operators provide the infrastructure that cloud and content providers leverage to provide their services. However, network providers' value has been diminished in the chain, putting network operators at a distinct disadvantage. In bringing the cloud closer to the edge, MEC could provide network operators new revenue streams by catering their cloud-enabled edge infrastructures at the Radio Access Networks (RANs) to authorized third-parties, as a service. Thus, network operators would act like edge cloud providers close to mobile subscribers. Consequently, network operators could provide an ideal environment for content providers to reliably and flexibly deploy their applications or services, at a very lucrative location, the edge of the network, so these can meet their business-critical objectives [67]. The proposed environment should be application-agnostic, where applications could leverage real-time radio network information to provide ultra-low latency and high bandwidth [25].

The concept is that infrastructure will, potentially, be found on various tiers across the cloud-to-edge continuum, and at different geographical locations. The actors involved determine where this infrastructure is and who makes use of it. It may be at base stations, micro data centers in cities, edge servers, Road Side Units (RSU), connected vehicles, edge routers, etc. It is yet unclear how these proposals will finally be commercialized. However, their current objectives suggest that they shall provide cloud computing capabilities, with IaaS as one of their possible offerings. Figure 3.1, shows infrastructure spread across the network space between the cloud and different edge definitions, encompassing some of the edge infrastructures proposed by these edge computing technologies.

Figure 3.1: Visualization of various edge computing infrastructures, as well as cloud infrastructures.

Thus, for example, an organization may have considerable infrastructure (i.e., VMs) at various locations, perhaps from different cloud/edge providers, or also from privately owned infrastructure. This consumer of cloud and edge computing would apply HYDRA to build a self-organizing orchestration solution to maintain global manageability over that decentralized infrastructure, easing application or service deployment by abstracting the complexity of the underlying infrastructure.

The specific use cases that would coax an organization into obtaining this at scale, dispersed infrastructure, would be the need to support any number of edge computing use cases. Some use cases are the ones that support applications for the Internet-of-Everything (IoE), such as, those related to connected video cameras (e.g., face recognition), modern sensor-rich aircraft, enabling or complementing smart transportation (e.g., autonomous cars or vehicle-to-everything – V2X), connected homes, smart cities, smart healthcare, finance, disaster management, smart energy (i.e., power grids), Industrial IoT – IIoT (e.g., smart manufacturing), speech recognition, online-gaming, and augmented reality (AR), etc.

Figure 3.2 exemplifies the simplified scenario of an extensive, dispersed cloud/edge infrastructure managed by the proposed orchestrator, HYDRA. The orchestrator serves the at-scale use cases of connected cars and smart homes by deploying their geo-spatially spread applications. Each application is composed of several services. These services may run on HYDRA-managed infrastructures, such as, at base stations, micro data centers, or directly at homes or cars. This last case refers to a HYDRA node operating in a car to deploy the appropriate car application-related service or at a smart home to do the same for the smart home application.

Another use case would be for network providers that have moved into the edge provider business by setting up infrastructure at the edge of their cellular network. However, it might include other network edges. As discussed, this infrastructure should provide cloud-like capabilities. Thus, this edge cloud provider could benefit from the proposed orchestrator to manage its infrastructure, producing a fully managed container orchestration service offering to third-party clients. Clients would simply submit containerized application specifications to the edge cloud provider, who would manage the deployment complexities of both infrastructure and orchestration on their behalf, as a service. Additionally, telecommunications operators could, potentially, also make use of this orchestrator to manage their infrastructure to deploy their virtual network functions (VNFs) in containers.

The orchestrator presented in this thesis has been described as obtaining an application deployment request from, for example, an application developer, that the orchestrator then deploys and manages according to the specifications provided. Another situation arises in the scenario presented by, for example, Cloudlets, where mobile subscribers such as cell phones or other mobile devices, could be the ones to request a service at the edge they are connected to. These mobility scenarios could also be considered.

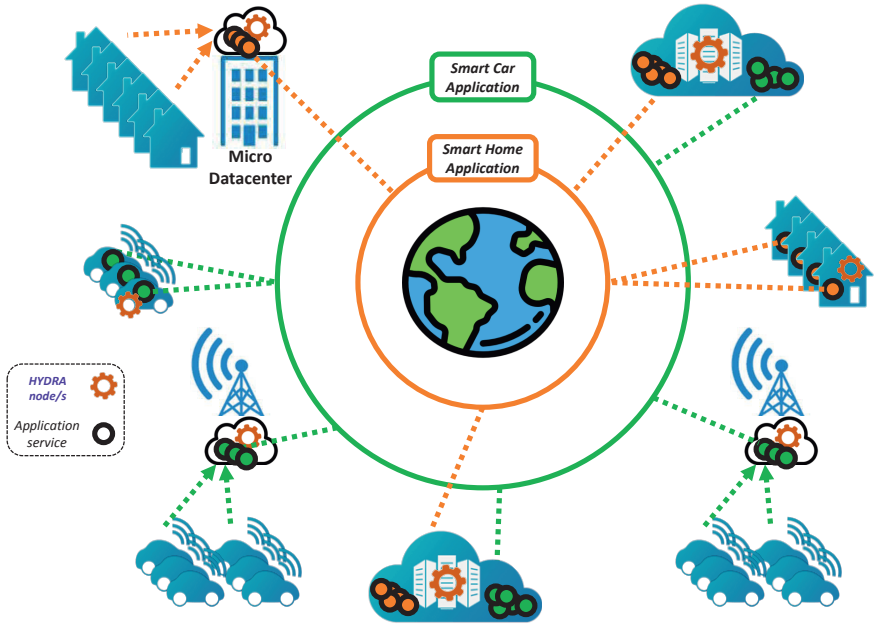

Figure 3.2: An at scale, geographically distributed infrastructure managed by HYDRA to orchestrate two application use cases, connected cars and smart homes. Each application is composed of various services that run at the different cloud or edge locations.

Chapter 4

Classifiers for resource-constrained devices

This chapter presents a contextualized account of the research carried out on designing machine learning algorithms for resource-constrained devices at the edge of the network. This research area addresses the research question **Q3**.

4.1 Computationally Inexpensive Classifiers for edge and IoT devices

Wearable and IoT devices have become pervasive in society. These devices have resulted in an exponential increase in available data. To extract knowledge from these data, AI has been widely adopted. One of the research interests presented in this thesis is on ML solutions through supervised learning techniques. This is a common field for AI services, where a classifier is run directly at end-devices.

There is a tradeoff between a ML model's predictive accuracy and its predictive latency. This tradeoff relates to model complexity and hardware characteristics. However, it is also affected by location. Generally, these ML models run in cloud DCs, resulting in prediction latency due to data transfer between the device and the cloud.

The communication latency between device and cloud may be ~ 100 ms. This latency could go down to $\sim 5 - 10$ ms if the model is brought closer, as per edge computing. Nevertheless, relocating the ML model to operate directly at the device may result in the most notable latency reduction [9]. Simultaneously, this results in less data being transferred and processed by the upper tiers [70]. This strategy mitigates pressure on edge clouds if these are employed, therefore favoring scalability at the network edge. Additionally, the devices do not have to stream data to an upper layer through radio access technologies, which can be energy-consuming [57].

As a rule-of-thumb, moving computational intelligence out from the cloud results in a lower resource and power budget, the farther we move out of the cloud, and the

closer we move to the devices. We conclude that in most cases, employing the processing capabilities of these IoT devices to run the ML algorithms results in resource and energy constraints, as these devices employ FPGAs, ASICs, or microcontrollers, generally with limited capabilities.

The research presented on this topic explores the considerations of executing ML algorithms under these limitations, focusing on minimizing latency prediction, conserving resources, and pursuing low energy consumption. These are tackled from the general perspective of ML algorithm optimization, and of ASIC or FPGA's architectural capabilities.

Research efforts to adapt existing machine learning algorithms to solve classification problems on resource-constrained devices exist, often with the intent of HW accelerating the most resource-consuming aspects of common ML algorithms [16, 32, 46, 50, 53, 56, 65]. However, the work presented in this thesis designs algorithms purposefully built to leverage the innate characteristics of these devices. Therefore, Papers D, E, and F propose ML classifiers designed with those considerations in mind.

Floating-point arithmetic is common to the implementation of AI applications [3, 71]. However, this type of logic is resource-intensive, relying on a few standard integer quantization techniques. There are efforts towards improving floating-point operation efficiency for the deployment of AI on these devices [23, 32]. Traditionally, fixed-point arithmetic is used to deploy floating-point algorithms to FPGA or ASIC hardware as it minimizes resource utilization and speeds up the calculation. However, it hurts accuracy, still consumes considerable resources, and complicates implementation. Moreover, on FPGAs, these arithmetics require the use of digital signal processing (DSP) slices, which some lower-end FPGA devices lack [37]. Therefore, the proposed algorithms in this thesis rely on bitwise operations, to reach our objectives of fast computation, and resource and energy conservation.

Other architectural considerations of these HW are parallelism and pipelining. This research pursues ML algorithms' design to exploit and, preferably, ease the associated implementation complexity. When the ML algorithm is designed to benefit from these capabilities, it can result in high classification throughput when the maximum clock speed is employed, or in energy savings when that frequency is slowed down. Therefore, this research considers and analyzes the algorithms' structure in terms of their capability to be parallelizable and pipelined.

The approach taken to supervised learning in this research has been the design of algorithms inspired by artificial neural networks (ANNs), Elementary Cellular Automata (ECA), and in Paper F, hyperdimensional computing. ANNs are a popular ML technique with the ability to map complex, non-linear input-output relationships when multiple layers are involved. Moreover, they can be executed in parallel, and they can solve both regression and classification problems, though the proposed algorithms in this thesis have focused on classification [28]. ECA has been shown to work at the edge of chaos, leading to the emergence of complex patterns from simple rules. It has low computational complexity and is parallelizable, as it does not require a global state [15]. Hyperdimensional computing relies on basic boolean algebra, making it suitable for implementation on low-

level devices. Likewise, it employs binary representation that is flexible in the number of bits employed, and its distributed representation can provide tolerance to noise [33].

Another consideration is that of efficiently employing the scarce resources of these devices. This entails minimizing computational complexity and memory consumption of the proposed algorithms while enabling predictive effectiveness. Implicitly, this also reduces energy expenditure. As such, the proposed algorithms in Papers D, E, and F, strike a balance between some of the following considerations: the number of input features, the number and nature of the connections between neurons, the number of neurons per layer, the number of hidden layers, and encoding of input data.

Moreover, as previously mentioned, all the algorithm proposals exclusively employ binary values, to steer clear of floating-point arithmetic. Therefore, the NN-inspired algorithms build networks based on binary data, meaning weights become unnecessary (i.e., more precisely, all connections between neurons have unit weight). This design choice also reduces the algorithm's memory footprint, as weights need not be stored in memory. Thus, it is the connections themselves that become important, not the weights. Furthermore, a sparsely connected network is pursued to reduce computational complexity. The activation function of a NN is often referred to as the NN powerhouse. Unlike in a traditional ANN, the activation function must be able to extract knowledge from binary data, and its computational complexity must also be addressed. This led to ECA's use as a non-linear activation function, which required simple bitwise operations while providing emergence-based characteristics.

Paper D presented CAMP as the first step towards a supervised ML algorithm with these characteristics. The network designed employs binary data input without encoding, input limited to three features, three equal-sized layers (input, hidden, and output), ECA rule 110, and with certain GA training parameters [36]. However, Paper E proposed CORPSE, which expands on the previous design, by identifying, varying, and analyzing a series of tunable hyperparameters: number of layers, whether to allow for spatiotemporal cell neighborhoods for ECA, generation method of class bit patterns, similarity measure, and the ECA rule used as the activation function. Further, this paper carries out an experimental analysis to optimize the GA to train the proposed algorithm [35].

Finally, Paper F differs from the two previous designs in that the input data is encoded (i.e., scatter code), ECA rule 90 is employed, and the network does not employ any hidden layers but instead pursues the mapping of the input space to a high-dimensional space through hyperdimensional computing. Robustness to post-training noise is also tested. Moreover, we implement the algorithm on a readily-available, off-the-shelf, low-end FPGA. The FPGA implementation could perform a classification every 17.485 ns at its maximum frequency [37].

4.2 Use Cases

The characteristics of the proposed algorithms imply that the use cases that would benefit from these algorithms are those requiring supervised learning, particularly when addressing classification problems, as this research has not been tried on regression problems. Moreover, the use cases should need to meet one or a combination of these conditions: resource-constrained environments, limited power consumption, or fast real-time requirements.

The following are some scenarios that align with these conditions. Context-aware services tend to require unobtrusive devices, meaning these must be small, and therefore, resource and energy-constrained. Additionally, some of these have real-time demands driving the on-board execution of a classifier. Intermittent, non-existent, or too energy-draining connectivity of these resource-limited IoT devices to the network edge or the cloud, also motivates the use of the proposed algorithms. In such situations, intelligent resource usage and conservative energy considerations can be critical, particularly in battery-powered, long-running devices at isolated locations or with limited network coverage. Moreover, in some cases, fast, actionable knowledge is critical, such that system response time is a QoS KPI, more so than accuracy [70].

These conditions can be found across various fields, such as space, military, aviation, scientific research, medical, mining, petroleum, robotics, autonomous driving, predictive maintenance, networking, and anomaly detection. The problem areas for which these algorithms have been tested, and are more likely to be successfully applied are pattern and activity recognition. These are broad domains that include, for example, signal classification (e.g., of car sensors [64], biomedical equipment, military sensors, space equipment [72]), image classification (e.g., text recognition for postal services [28], IoT cameras), or others (e.g., behavioral user patterns, network packet classification [44], context services for elderly care).

Chapter 5

Contributions

The following section describes the six publications included in this composition. Publications A, B, and C refer to the area of research covering containerization, concerning monitoring and orchestration of microservice applications. Publications D, E, and F relate to developing and optimizing supervised machine algorithms for resource-constrained devices.

5.1 Paper A

Title: CoMA: Resource Monitoring of Docker Containers

Authors: Lara Lorna Jiménez, Miguel Gómez Simón, Olov Schelén, Johan Kristiansson, Kåre Synnes, and Christer Åhlund.

Status: Published in the 5th International Conference on Cloud Computing and Services Science, 2015, Lisbon, Portugal.

Summary: This paper presents a monitoring solution for containerization platforms. This publication proposes a new Container Monitoring Agent complemented by an existing host monitoring agent and a collector architecture that monitors hosts and container platforms in federated clusters. The Container Monitoring Agent obtains information about resource usage and limitations of the containers in terms of CPU, memory, and disk I/O, from the Linux kernel. Further, the agent also supplies information about the containers and images obtained from the container platform API. This paper performs a validation assessment of the accuracy of the resource usage data reported by the Container Monitoring Agent.

Personal Contribution: I did the design, implemented the agent and collector system, carried out the experiments, and performed the analysis conjointly with the second author. I wrote the paper with feedback from co-authors.

Relevance: This paper addresses research question one (Q1).

5.2 Paper B

Title: DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications
Authors: Lara Lorna Jiménez, and Olov Schelén.
Status: Published in the 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications (IEEE Cloud Summit), 2019, Washington, D.C., USA.

Summary: This publication presents the first step towards a highly scalable orchestrator that can operate at the cloud and all the way to the edge, in the scenario brought forth by the new computing paradigm, edge computing. It focuses on the deployment of containerized microservice applications.

The orchestrator must manage a geographically spread out infrastructure, and the applications deployed might also require location awareness. However, DOCMA is a first approach towards the desired orchestration solution, so one of its limitations is that it is location-agnostic. The orchestrator builds on peer-to-peer principles to enable vast scalability and resiliency, which is the focus of this publication. It also provides an initial role-based scheme to enable orchestration of microservice applications over the managed infrastructure. These roles are dynamically adopted by nodes on the orchestrator network according to an ID.

This publication carries out a series of statistical experiments to quantitatively demonstrate DOCMA's capacity to scale. The experiments were set up through the simulation of a homogeneous network of DOCMA nodes, with varying network size, and average pre-existing workload. DOCMA was proven to scale to, at least, 7000 nodes.

Personal Contribution: I designed the orchestrator, implemented DOCMA as a Proof-of-Concept and the Experimenter to carry out the experiments, designed and carried out the experiments, and analyzed the experimental results. I wrote the paper with feedback from the co-author.
Relevance: This publication partly addresses research question two (Q2).

5.3 Paper C

Title: HYDRA: Decentralized Location-aware Orchestration of Containerized Applications
Authors: Lara Lorna Jiménez, and Olov Schelén.
Status: Submitted to the IEEE Journal of Transactions on Cloud Computing.

Summary: HYDRA is the container orchestration system proposed in this paper. DOCMA, in Paper B, is an orchestrator built for vast scalability but without location awareness. In contrast, HYDRA focuses on producing an orchestrator that can also manage a geographically dispersed infrastructure and deploy distributed, location-aware applications. Thus, HYDRA is designed to operate as both a location-agnostic

and location-aware orchestrator. If it is the latter, then the deployment objects may also be deployed with affinity to location.

Location awareness significantly modifies the orchestrator design. This publication provides a design to map location to the IDs employed across the orchestrator and defines control networks to exclude nodes from hosting control functionalities, if desired. Each node can be identified and searched for through either a node ID, which is static, or a location node ID, which depends on the node's location. This enables a mechanism of dual node discovery. This orchestrator is based on a series of roles that have associated behaviors to enable the orchestrator functionalities. One of those roles deals with application management. To provide fast, reliable control over location-aware applications, this role is taken on by different nodes according to the location of the services of each application. Additionally, two search algorithms to search for nodes across the orchestrator network to host services are presented, as well as a scheduling algorithm to host services at nodes that are found to be viable.

The publication experimentally demonstrates the proposed orchestrator's ability to scale to, at least, 20 000 nodes. Moreover, the orchestrator was shown to process the simultaneous deployment of 30 000 applications on a 15 000 node network, with a successful outcome in almost all cases. The scheduling and search algorithms employed proved to be a useful baseline for comparing other such solutions for this orchestrator. Additionally, the orchestrator may operate with or without location awareness with a minimal statistical impact on its performance. Furthermore, when it operates with location awareness, a network partition of one of the orchestrator regions proved to have a minimal impact on successful application deployment.

Personal Contribution: I designed the orchestrator, implemented it, designed and carried out the experiments, and analyzed the experimental results. I wrote the paper with feedback from the co-author.
Relevance: This manuscript addresses research question two (Q2).

5.4 Paper D

Title: A Computationally Inexpensive Classifier Merging Cellular Automata and MCP-Neurons
Authors: Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg.
Status: Published in the 10th International Conference on Ubiquitous Computing and Ambient Intelligence, 2016, Gran Canaria, Spain.

Summary: Context-aware services often employ machine learning techniques to process data from mobile and wearable devices. Some context-aware services may enforce the processing to occur on these devices directly, motivated by performance requirements (e.g., response time). These devices are generally resource-constrained, such as ASICs, FPGAs, or microcontrollers. Often, machine learning algorithms are too resource-consuming or complex for this hardware. Therefore, it is necessary to design algorithms that fit these

devices' implementation requirements and resource limitations. Hence, this publication presents CAMP, a computationally inexpensive classifier with a small memory footprint combining Binary Cellular Automata and an MCP-network (i.e., McCulloch-Pitts neuron model network). It builds a sparse binary neural network with the 110 cellular automata rule as the activation function. The network is trained through a genetic algorithm to obtain the best network connections for a given dataset.

CAMP was tested on five classification datasets, and its performance on these was compared against that of six other standard machine learning algorithms. The results show that CAMP performs with comparable accuracy to other more advanced machine learning algorithms for datasets with few classes. However, it performs considerably worse for datasets with a higher number of classes. We conclude that considering its simplistic design, CAMP can be employed to solve certain classification problems in severely resource-constrained environments.

Personal Contribution: I aided in the design, state of the art, evaluation, and writing of the paper.
Relevance: This paper targets research question three (Q3).

5.5 Paper E

Title: Classifier Optimized for Resource-constrained Pervasive Systems and Energy-efficiency
Authors: Niklas Karvonen, Lara Lorna Jiménez, Miguel Gómez Simón, Joakim Nilsson, Basel Kikhia, and Josef Hallberg.
Status: Published in the International Journal of Computational Intelligence Systems, 2017.

Summary: This manuscript presents CORPSE, an algorithm built on principles of elementary cellular automata and sparse neural networks. This work is an extension of the work done in Paper D, and as such, this algorithm also targets design and implementation for resource-constrained devices. In this paper, we identify some hyperparameters of the proposed algorithm, determine the effect of these on algorithm performance, and determine genetic algorithm settings to improve the classifier's training outcome. The identified hyperparameters are the depth of the network, whether a bias layer is employed, whether to allow neurons to connect to any previous layer, how the bit patterns of the classes are selected, the similarity measure employed, and the cellular automata rule used.

The performance of CORPSE was tested on eight binary class datasets of varying types. The results show an improved performance of CORPSE. Despite its simple design, it has comparable performance to some more sophisticated machine learning algorithms.

Personal Contribution: I collaborated in identifying the hyperparameters of CORPSE and the settings of the genetic algorithm to train it, designing the experiments for these, carrying out these experiments, analyzing them, and writing the paper.

Relevance: This publication addresses research question three (Q3).

5.6 Paper F

Title: Low-Power Classification using FPGA – An Approach based on Cellular Automata, Neural Networks, and Hyperdimensional Computing
Authors: Niklas Karvonen, Joakim Nilsson, Denis Kleyko, and Lara Lorna Jiménez.
Status: Published in the 18th IEEE International Conference On Machine Learning And Applications, 2019, Florida, USA.

Summary: This publication presents a classifier targeting resource-constrained devices (e.g., FPGAs), based on Elementary Cellular Automata (ECA), Artificial Neural Networks, and Hyperdimensional computing. These low-level devices have desirable characteristics for wearable and IoT devices. The proposed HyperCorpse algorithm's structure fits FPGAs' requirements to use its architecture and resources efficiently. The input data for this classifier is scatter coded and processed to a high dimensional output layer through an ECA rule. Its training is performed through a genetic algorithm.

The evaluation of the classifier was performed on two datasets and compared against a baseline classifier. The results show that the algorithm performs on par with Naive Bayes and is more robust to post-training noise. Moreover, the proposed classifier was implemented on a commercially available, off-the-shelf FPGA and executed for one of the datasets. This implementation produced 57.1 million classifications per second, at its maximum frequency. We conclude that HyperCorpse is a viable classifier for devices where low power-consumption, efficient use of resources, fast real-time responses, or robustness to post-training noise are needed.

Personal Contribution: I aided in the design and analysis carried out, as well as in writing part of the paper.
Relevance: This publication addresses research question three (Q3).

Chapter 6

Conclusions and Future Work

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein

This chapter brings this thesis' comprehensive summary to a close. Here, I present my observations and the results obtained to answer the research questions posed in Section 1.1. Additionally, I identify several avenues of research within the discussed thesis topics that would merit further research.

6.1 Conclusions

Q1 *How can a monitoring solution be built to track resource consumption and limitations of containers, as well as container-related metadata, for an at-scale pool of container-enabled hosts?*

Containerization isolates and limits processes within the OS through the Linux kernel's features, namespaces and control groups (a.k.a., cgroups). Therefore, to accurately monitor the containers deployed at a host and the container platform itself, we must understand how containerization is enabled at the kernel level. Retrieving these data provides the means to understand the resource utilization and limitations of containers, which is key to providing efficient services.

When this research took place, and even today, Docker was the *de facto* containerization industry solution. Docker did not provide these data in its early versions, except for the two most basic metrics. Therefore, in Paper A, we designed and implemented an agent to sample this information from the kernel and the container platform, at a configurable sampling rate. We also complemented it with an existing host monitoring agent. This setup provides a holistic picture of a node to facilitate adequate monitoring.

Further, as the infrastructure involved in deploying applications may, in many cases, be composed of many hosts, potentially spread across, for example, various data centers, we propose a scalable monitoring solution for the collection of these data through cluster

federation. This collector also provided the means to visualize the collected data. Further, Paper A quantitatively verified the accuracy of the collected data, and the lack of overhead between running processes containerized versus natively was also shown.

Q2 *How can an orchestrator that can manage a large pool of resources, surpassing the capabilities of current container orchestration solutions, which is resistant to churn, be designed? Can such an orchestrator provide global manageability across dispersed geographical boundaries to deploy and manage location-aware distributed applications?*

To ensure vast scalability, availability, and resiliency the orchestrators proposed in Papers B and C, are based on principles of decentralization, built over a structured P2P DHT overlay network that provides resistance to churn. In this design, each node runs the orchestrator system and can discover other nodes on the orchestrator network relying on an ID-based distributed data structure.

Nodes in the proposed orchestrator are independent from each other, meaning there is no central control system and nodes act autonomously – every node is the orchestrator. This principle is the basis to attain the orchestrator's high scalability. The research carried out on this topic demonstrates the ability of the proposed container orchestrator to scale past the capabilities of current container orchestration solutions. Paper C shows the orchestrator scaling to at least 20 000 nodes. Moreover, scalability in terms of simultaneous processing of application deployment requests was also tested with 30 000 applications on a 15 000 node network.

The orchestrator functionalities required to reach global manageability of the underlying infrastructure and automate application management, deployment, and scaling are implemented on top of this P2P overlay. These functionalities require a unique design to work in harmony on top of the overlay network.

Various containerized deployment objects may be deployed to the orchestrator. However, the focus has been on facilitating microservices, as these fit the requirements of cloud-native and edge-native applications. In part, a series of roles that the appropriate nodes dynamically adopt for each deployment object, enable the orchestrator. This adaptability to changes in the orchestrator network is an essential strength of the orchestrator. Deployment objects trigger various control behaviors that facilitate the automation of deployment object management. For example, some of these deal with ensuring the correct number of replicas are functioning, scaling the replica size, scheduling deployment objects, or maintaining service discoverability.

Paper B delivers a location-agnostic orchestrator as a stepping stone towards the intended solution, whereas the orchestrator presented in Paper C can be both, location-agnostic and location-aware. Therefore, the latter can manage a geographically dispersed infrastructure to deploy distributed applications that operate across geographical locations. This research has shown that the proposed orchestrator can successfully deploy applications over a simulated, geographically-dispersed infrastructure when operating with location awareness.

Q3 *How can purpose-built supervised machine learning algorithms be designed to work on resource-constrained devices to enable fast real-time prediction and minimize resource and energy consumption? How well do these perform in terms of predictive accuracy and latency?*

The work carried out in this area determines that for many applications, the predictive response times offered by running the ML models in the cloud or even at the network edge do not result in the desired QoS latency. Additionally, relying on the cloud results in unsustainable data transfer volumes for the network, while depending on the edge cloud may also be problematic for its scalability. Moreover, in some cases, being contingent on an external system to execute the ML model is not possible. Consequently, this research tackles the design of supervised machine learning algorithms at resource-scarce IoT devices that are often FPGAs or ASICs.

To tackle the concerns of minimizing the resource and energy expenditure of these devices and enabling fast real-time predictions, we determine that the ML algorithms must be parallelizable to employ this hardware efficiently. Thus, we identify three key characteristics to design ANN-inspired supervised ML algorithms with reduced computational complexity for these devices. These core attributes are, to employ only binary values, operate solely with boolean algebra, and have the network be sparsely connected with few layers. This led to the use of Elementary Cellular Automata as an activation function and employing hyperdimensional computing.

Three different binary-valued classifiers based on these principles were designed and compared against conventional supervised learning classifiers, with various datasets. We concluded that the ANN-inspired ECA-activated classifiers of Papers D and E performed better for binary classification problems, and had similar accuracy to other common ML algorithms. Additionally, Paper F also employed hyperdimensional computing to build a classifier for multi-class problems. This resulted in an algorithm with comparable accuracy to the baseline classifier (i.e., Naive Bayes), and surpassed it in the presence of post-training noise.

This last classifier was implemented for a signal processing, 3-class problem on a commercially available, off-the-shelf, low-end FPGA. This produced 57.1 million classifications per second, or one classification every 17.485 ns, at an accuracy of 78.88%, when operating at its maximum frequency. It is worth noting that reducing this frequency would lower energy consumption at the expense of increasing the classifier's predictive latency.

6.2 Future Work

The research presented in this thesis has addressed the questions posed in Section 1.1. However, various open issues within the research topic of this thesis would warrant future work.

Q1 *Monitoring of containers*

Containers are ephemeral, and many now build distributed, inter-communicating applications. These distributed systems are comprised of many moving parts, which makes identifying problems difficult. Further, the complexity of handling resiliency, deployment, and availability, at a growing scale, are pushing for orchestration solutions. Consequently, individual container monitoring has become a stale approach to the problem. Monitoring must be made across the stack of servers, VMs, containers, services of containers, applications of containers, application-centric information, and the orchestration system.

Monitoring at numerous layers generates an explosion of metrics and logs, much of which goes unused, causing a problem of performance, cost, implementation, and efficiency. Therefore, contextualizing key metrics to pre-process at the source and identifying new units of measurement is necessary to garner more sparse and relevant data, that can be actionable and easily tracked. The volume of monitoring data marks it as a 'big data' problem, which should benefit from artificial intelligence to provide understandable insight and, given that monitoring is still reactive for the most part, it should work towards recommending or automating actions, perhaps through the orchestration system. These aspects of containerized environments call for new monitoring practices. While monitoring solutions are moving in this direction, much work is still left to be done.

Q2 *Container orchestration*

Orchestration systems are large and complex. The container orchestration system proposed in this thesis is a PoC of a new kind of container orchestrator. Nevertheless, during my research, I have identified many research options, some of which remain open.

The current container orchestration solution for location awareness is flexible in that it maps location to IDs. This mapping can be extended to include new mappings (i.e., new locations) if adequately designed. However, it is not adaptable to radical re-defining changes to already existing mappings. Therefore, a possible research avenue would be to determine whether there is a method to establish location awareness in this orchestrator that would perform as well or better while providing greater flexibility.

One of the identified functionalities to enabling this orchestrator is the consistent replication of a deployment object and its associated functions. In Section 3.1.8 two distributed consensus topological solutions were presented to this effect. A quantitative study of their performance and impact is another interesting area for investigation.

Some orchestrator KPIs have been identified and analyzed in the presented research. These could be expanded on and analyzed, such as the average time to deploy an object (i.e., with the relevant parameters), or the mean-time-to-resolve (MTTR) the incident of

a capsule, service, application, or node that goes down or becomes unhealthy, and deploy a new replica or any of the corresponding actions to those incidents.

The impact of the replica types (i.e., passive or active) defined in this orchestrator, would warrant an analysis to determine the respective overheads and performance drawbacks for different application types. Further, the orchestrator could easily store container images as a distributed image registry. This could be studied and, potentially, employed as a means of optimizing container deployment.

In the case of the scheduling of a deployment object, the tests carried out used a FFD algorithm, for a single filter (i.e., two if we consider location) with a single resource (i.e., memory), for an application composed of 5 services, and with the Random ID search algorithm. The other search algorithm, for which preliminary results have been offered in this thesis, should be studied to fully describe its performance. Another area of interest is investigating other search algorithms and more significant variability of the characterized applications employed during analysis.

Moreover, multi-filter search analysis would yield more accurate information on which to assess search or scheduling algorithms in this orchestrator. Additionally, an experimental analysis to determine how the propagation of nodes' specifications across the network would improve scheduling performance concerning the data overhead incurred would provide intriguing insights into how best to define this aspect of the orchestrator. However, perhaps the most compelling research path within this matter would be the analysis of the self-organizing hybrid network centered around node specifications proposed in Section 3.1.6. This solution has been designed to improve the likelihood of finding a viable node on the network, if it exists, matching the filter-based requirements of one or more capsules to deploy them successfully.

Another aspect of the proposed orchestrator that should be quantified is the resources each orchestrator node consumes, in terms of CPU, memory, disk, network, and power under different orchestrator network scenarios.

Finally, there is a trade-off between scalability and resource usage overhead. Though very challenging and costly to carry out, an interesting research issue would be to compare the performance of the proposed orchestrator, HYDRA, to that of Kubernetes or, if possible, KubeFed concerning their characteristics.

Q3 *Computationally inexpensive classifiers*

In Papers D and E, the proposed classifiers perform better for binary-class than for multi-class classification problems. This behavior should be analyzed to improve the algorithm design, which would improve the overall accuracy of the classifiers when solving any classification problem. We could address this issue in a variety of ways. First, the single multi-class problem could be computed as a series of binary classification problems. Second, the dimensionality of the output layer could be increased for more straightforward discrimination between classes. Third, the hidden layers' optimal dimensionality could be studied to imbue the network with a higher capacity for complex input-output mappings. However, these approaches would increase implementation and computational complexity, each to a different degree.

To a certain extent, through the use of hyperdimensional computing, Paper F implements the second approach. However, of the two multi-class datasets tested, there is an issue with one of the classes, likely due to an error in the encoding process or in training, which should be addressed. This algorithm would also benefit from analyzing the impact of including a single hidden layer and its dimensionality.

The supervised machine learning algorithms presented in this thesis have been tested to solve classification problems, but their capacity to solve regression problems has not been tried. This would be an interesting area for future research. Moreover, the fitness function employed during the training of these algorithms may be too coarse-grained, leading to local optima. Therefore, other fitness functions should be considered. Additionally, other research paths would be exploring other training methods, trying non-uniform cellular automata for neuron activation, or investigating other types of activation functions.

Bibliography

[1] Docker overlay networks: Performance analysis in high-latency environments. [Online; accessed 2020-06-15. https://delaat.net/rp/2015-2016/p50/report.pdf].

[2] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973, 2018.

[3] J. Amrutha and A. S. Remya Ajai. Performance analysis of backpropagation algorithm of artificial neural networks in verilog. In *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 1547–1550, 2018.

[4] Victor Bahl. Cloud 2020: The emergence of micro datacenter for mobile computing. *Microsoft, Redmond, WA, USA*, 2015.

[5] K. Bilal, S. U. R. Malik, S. U. Khan, and A. Y. Zomaya. Trends and challenges in cloud datacenters. *IEEE Cloud Computing*, 1(1):10–20, 2014.

[6] Kashif Bilal, Osman Khalid, Aiman Erbad, and Samee Khan. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks*, 130, 10 2017.

[7] Flavio Bonomi and Rodolfo Milito. Fog computing and its role in the internet of things. *Proceedings of the MCC workshop on Mobile Cloud Computing*, 08 2012.

[8] A. Botta, W. de Donato, V. Persico, and A. Pescapé. On the integration of cloud computing and internet of things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 23–30, 2014.

[9] Sergio Branco, Andre Ferreira, and Jorge Cabral. Machine learning in resource-scarce embedded systems, fpgas, and end-devices: A survey. *Electronics*, 8, 11 2019.

[10] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience*, 50(5):719–740, 2020.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[12] Charles C. Byers and Patrick Wetterwald. Fog computing distributing data and intelligence for resiliency and scale necessary for iot: The internet of things (ubiquity symposium). *Ubiquity*, November 2015.

[13] Massimo Candela, Valerio Luconi, and Alessio Vecchio. Impact of the covid-19 pandemic on the internet latency: a large-scale study, 2020.

[14] Eric Carter. Docker usage report.— sysdig, 2018. [Online; accessed 25-July-2020. https://sysdig.com/blog/2018-docker-usage-report/].

[15] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15, 01 2004.

[16] Nicholas Cotton, Bogdan Wilamowski, and G. Dundar. A neural network implementation on an inexpensive eight bit microcontroller. In *the 12th International Conference on Intelligent Engineering Systems*, pages 109 – 114, 03 2008.

[17] A. V. Dastjerdi and R. Buyya. Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116, 2016.

[18] K. Dolui and S. K. Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6, 2017.

[19] Győrgy Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i) $\leq$ 11/9 opt(i) + 6/9. In *Lecture Notes in Computer Science*, volume 4614, pages 1–11, 01 2007.

[20] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[21] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1):84–106, 2013.

[22] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[23] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, June 2017.

[24] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 166–176, 2013.

[25] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing – a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[26] Kasun Indrasiri and Prabath Siriwardena. Service mesh. In *Microservices for the Enterprise: Designing, Developing, and Deploying*, pages 263–292, Berkeley, CA, 2018. Apress.

[27] M Iorgam, L Feldman, R Barton, MJ Martin, N Goren, and C Mahmoudi. Fog computing conceptual model, recommendations of the national institute of standards and technology. In *NIST Special publication 500-325*. 2018.

[28] K. T. Islam, G. Mujtaba, R. G. Raj, and H. F. Nweke. Handwritten digits recognition with artificial neural network. In *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*, pages 1–4, 2017.

[29] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[30] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23, 03 2014.

[31] Lara Lorna Jiménez and Olov Schelén. Docma: A decentralized orchestrator for containerized microservice applications. In *2019 IEEE Cloud Summit*, pages 45–51. IEEE, 2019.

[32] Jeff Johnson. Rethinking floating point for deep learning, 2018.

[33] Kanerva and Pentti. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1, 06 2009.

[34] M. Kang, J. Shin, and J. Kim. Protected coordination of service mesh for container-based 3-tier service traffic. In *2019 International Conference on Information Networking (ICOIN)*, pages 427–429, 2019.

[35] Niklas Karvonen, Lara Lorna Jiménez, Miguel Gomez, Joakim Nilsson, Basel Kikhia, and Josef Hallberg. Classifier optimized for resource-constrained pervasive systems and energy-efficiency. *International Journal of Computational Intelligence Systems*, 10:1272, 01 2017.

[36] Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg. A computationally inexpensive classifier merging cellular automata and mcp-neurons. In *Ubiquitous Computing and Ambient Intelligence*, pages 368–379, Cham, 2016. Springer International Publishing.

[37] Niklas Karvonen, Joakim Nilsson, Denis Kleyko, and Lara Jiménez. Low-power classification using fpga–an approach based on cellular automata, neural networks, and hyperdimensional computing. In *The 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 370–375, 12 2019.

[38] Kubernetes. Scalability updates in kubernetes, March 2017. [https://kubernetes.io/blog/2017/03/scalability-updates-in-kubernetes-1-6/].

[39] Kubernetes. Kubernetes cluster federation v2 – kubefed, July 2020. [https://github.com/kubernetes-sigs/kubefed].

[40] Lars Larsson, William Tarneberg, Cristian Klein, Erik Elmroth, and Maria Kihl. Impact of etcd deployment on kubernetes, istio, and application performance. *ArXiv*, abs/2004.00372, 2020.

[41] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. Association for Computing Machinery.

[42] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[43] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[44] Oriol Mula-Valls. Master's thesis. A practical retraining mechanism for network traffic classification in operational environments, June 2011. Polytechnic University of Cataluña, Spain.

[45] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[46] Jason Oberg, Ken Eguro, Ray Bittner, and Alessandro Forin. Random decision tree body part recognition using fpgas. *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 330–337, August 2012.

[47] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *USENIX*, pages 305–320, 01 2014.

[48] G. Orsini, D. Bade, and W. Lamersdorf. Computing at the mobile edge: Designing elastic android applications for computation offloading. In *2015 8th IFIP Wireless and Mobile Networking Conference (WMNC)*, pages 112–119, 2015.

[49] Claus Pahl, Pooyan Jamshidi, and Danny Weyns. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *Journal of Software: Evolution and Process*, 29:e1849, 02 2017.

[50] M. Papadonikolakis and C. Bouganis. Novel cascade fpga accelerator for support vector machines classification. *IEEE Transactions on Neural Networks and Learning Systems*, 23(7):1040–1052, 2012.

[51] Riccardo Pecori. S-kademlia: A trust and reputation method to mitigate a sybil attack in kademlia. *Computer Networks*, 94:205 – 218, 2016.

[52] Platform9. Container Management: Kubernetes vs Docker Swarm, Mesos+Marathon and Amazon ECS. [Online; accessed 05-February-2020; https://platform9.com/wp-content/uploads/2018/08/kubernetes-comparison-ebook.pdf].

[53] Y. Pu, J. Peng, L. Huang, and J. Chen. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 167–170, 2015.

[54] Ruay-Shiung-Chang, J. Gao, V. Gruhn, J. He, G. Roussos, and W. Tsai. Mobile cloud computing research - issues, challenges and needs. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 442–453, 2013.

[55] Dario Sabella, Vadim Sukhomlinov, Linh Trang, Sami Kekki, Pietro Paglierani, Ralf Rossbach, Xinhui Li, Yonggang Fang, Dan Druta, Fabio Giust, Luca Cominardi, Walter Featherstone, Bob Pike, Shlomi Hadad, Linh Sony, Vmware Fang, and Bob Acs. Developing software for multi-access edge computing. 02 2019.

[56] Fareena Saqib, Aindrik Dutta, J. Plusquellic, Philip Ortiz, and Marios Pattichis. Pipelined decision tree classification accelerator implementation in fpga (dt-caif). *Computers, IEEE Transactions on*, 64:280–285, 01 2015.

[57] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[58] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[59] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante. The seminal role of edge-native applications. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 33–40, 2019.

[60] Rahul Sharma and Avinash Singh. Istio virtual service. In *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes*, pages 137–168, Berkeley, CA, 2020. Apress.

[61] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[62] Statisca. Data volume of iot connections worlwide in 2018 and 2025, in zettabytes, February 2020. https://www.statista.com/statistics/1017863/worldwide-iot-connected-devices-data-size/.

[63] Cisco Systems. Cisco Annual Internet Report, 2018-2023, June 2020. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[64] Wu Tianshu, Zhang Zhijia, Liu Zhanning, Liu Yunpeng, and Wang Shixian. Detection and implementation of driver's seatbelt based on fpga. *Journal of Physics: Conference Series*, 1229:012075, 05 2019.

[65] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: multi-core, gp-gpu, or fpga? In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012*, pages 232–239, 04 2012.

[66] Maarten Van Steen and Andrew S. Tanenbaum. *Distributed Systems 3rd ed.* distributed-systems.net, 2017.

[67] Fernando Vargas Vargas. Master's Thesis. Cloudlet for Internet-of-Things, 2016. [KTH University, Sweden].

[68] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[69] X. Wang and Z. Jin. An overview of mobile cloud computing for pervasive healthcare. *IEEE Access*, 7:66774–66791, 2019.

[70] Maciej Wielgosz and Michał Karwatowski. Mapping neural networks to fpga-based iot devices for ultra-low latency processing. *Sensors*, 19:2981, 07 2019.

[71] Z. Xie. A non-linear approximation of the sigmoid function based on fpga. In *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*, pages 221–223, 2012.

[72] Xilinx. Xilinx launches "industry's first" 20 nanometer space-grade FPGA chip. [Online; accessed 21-July-2020].

[73] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[74] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The cloud is not enough: Saving iot from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

Part II

Paper A

CoMA: Resource Monitoring of Docker Containers

Authors:
Lara Lorna Jiménez, Miguel Gómez Simón, Olov Schelén, Johan Kristiansson, Kåre Synnes, and Christer Åhlund

Reformatted version of paper originally published in:
The 5th International Conference on Cloud Computing and Services Science, 2015, Lisbon, Portugal.

CoMA: Resource Monitoring of Docker Containers

Lara Lorna Jiménez, Miguel Gómez Simón, Olov Schelén, Johan Kristiansson, Kåre Synnes, and Christer Åhlund

Abstract

This research paper presents CoMA, a Container Monitoring Agent, that oversees resource consumption of operating system level virtualization platforms, primarily targeting container-based platforms such as Docker. The core contribution is CoMA, together with a quantitative evaluation verifying the validity of the measurements reported by the agent for three metrics: CPU, memory and block I/O. The proof-of-concept is implemented for Docker-based systems and consists of CoMA, the Ganglia Monitoring System and the Host sFlow agent. This research is in line with the rising trend of container adoption which is due to the resource efficiency and ease of deployment. These characteristics have set containers in a position to topple virtual machines as the reigning virtualization technology in data centers.

1 Introduction

Traditionally, virtual machines (VMs) have been the underlying infrastructure for cloud computing services [23]. Virtualization techniques spawned from the need to use resources more efficiently and allow for rapid provisioning. Native virtualization (Type I) [20] [19] is the standard type of virtualization behind cloud services. There are several established platforms that offer this type of virtualization such as, Xen hypervisor [1], Linux Kernel Virtual Machine (KVM) [18] and VMware [19]. Full-virtualization, Para-virtualization and Hardware-assisted virtualization are different techniques that attempt to enhance the effectiveness of VMs, with varying degrees of success and certain tradeoffs. However, none of these techniques are on par with today's expectations in the cloud computing industry. It has been demonstrated that VMs introduce a significant overhead that does not allow for an optimized use of resources [22]. The unfulfilled potential for improvement of VMs is where OS-level virtualization comes in.

OS-level virtualization has become popular in recent years by virtue of its resource efficiency. This light-weight type of virtualization executes processes quasi-natively [10], [21]. On top of a shared Linux kernel, several of what are generally referred to as "containers" run a series of processes in different user spaces [9]. In layman's terms, OS-level virtualization generates virtualized instances of kernel resources, whereas hypervisors virtualize the hardware. Moreover, containers run directly on top of an operating system, whereas VMs need to run their OS on top of a hypervisor which creates a performance overhead [22]. The downside of containers is that they must execute a similar OS to the

one that is hosting the containers. There are various implementations of OS-level virtualization, with differences in isolation, security, flexibility, structure and implemented functionalities. Each one of these solutions is oriented towards different use cases. For example, *chroot() jails* [9] are used to sandbox applications and Linux containers (LXC) [1] are used to create application containers.

In March 2013, the Docker platform was released as an open-source project based on LXC and a year later, the environment was moved from LXC to libcontainer [2]. Docker is based on the principles of containerization, allowing for an easy deployment of applications within software containers as a result of its innovative and unique architecture [10]. Docker implements certain features that were missing from OS-level virtualization. It bundles the application and all its dependencies into a single object, which can then be executed in another Docker-enabled machine. This assures an identical execution environment regardless of the underlying hardware or OS. The creation of applications in Docker is firmly rooted in the concept of versioning [6]. Modifications of an application are committed as deltas [7], which allows roll backs to be supported and the differences to previous application versions to be inspected. This is an exceptional method of providing a reliable environment for developers. Furthermore, Docker promotes the concept of reusability, since any object that is developed can be re-used and serve as a "base image" to create some other component. Another essential aspect of Docker is that it provides developers with a tool to automatically build a container from their source code.

The main difference between a Docker container and a VM is that while each VM has its own OS, dependencies and applications running within it, a Docker container can share an OS image across multiple containers. In essence, a container only holds the dependencies and applications that have to be run within them. For example, assuming a group of containers were making use of the same OS image, the OS would be common to all containers and not be duplicated contrary to the case of a VM topology.

Docker has become the flagship in the containerization technology arena since its release [10] [4]. This open-source project has gained much notoriety in the field of cloud computing, where major cloud platforms and companies (e.g. Google, IBM, Microsoft, AWS, Rackspace, Red Hat, VMware) are backing it up. These companies are integrating Docker into their own infrastructures and they are collaborating in Docker's development. Recently, a few alternatives to Docker have cropped up, such as Rocket [3], Flockport [4] and Spoonium [5].

An adequate monitoring of the pool of resources is an essential aspect of a cloud computing infrastructure. The monitoring of resources leads to improved scalability, better placement of resources, failure detection and prevention, and maintenance of architectural consistency, among others. This is relevant for VMs, and it is just as applicable to OS-level virtualization. Out of this need to monitor containers, within the paradigm of OS-level virtualization platforms, the following research questions have been addressed

[1] https://linuxcontainers.org/
[2] http://www.infoq.com/news/2013/03/Docker
[3] https://coreos.com/blog/rocket/
[4] http://www.flockport.com/start/
[5] https://spoon.net/docs

in this paper:

- How could an OS-level virtualization platform be monitored to obtain relevant information concerning images and containers?

  This paper presents an investigation of this issue as well as an implementation of a Container Monitoring Agent.

- Is the resource usage information about the running containers reported by our Container Monitoring Agent valid?

  This paper details a quantitative evaluation of the validity of the measurements collected by the proposed Container Monitoring Agent.

The rest of the paper is organized as follows. Section 2 presents state-of-the-art research related to the monitoring of containers and virtual machines. Section 3 explains the different components of the monitoring system architecture. Section 4 presents the results obtained. Section 5 discusses the research questions. Finally, Section 6 presents the conclusions of the paper and discusses the possibilities for future work.

2 Related Work

There is an active interest in industry and in research to build monitoring solutions [17]. In [13] the term monalytics is coined. Monalytics refers to a deployment of a dynamically configurable monitoring and analytics tool for large scale data centers, targeting the XEN hypervisor. One of the monalytics' topologies defined, matches the architecture chosen for the monitoring solution provided in this paper. The research of [15] is centered on providing a state monitoring framework that analyzes and mitigates the impact of messaging dynamics. This technique ensures the trustworthiness of the measurements collected by a distributed large-scale monitoring tool on XEN-based virtual machines. Most existing monitoring research for data center management target virtual machines. To the best of our knowledge, at the time of writing this paper, there were no research papers on the topic of monitoring the Docker platform.

When this monitoring solution for Docker was developed, there were a lack of open-source implementations to monitor Docker. However, recently, several other monitoring systems for Docker have appeared.

Datadog agent [3] is an open-source tool developed by *Datadog Ink* which monitors Docker containers by installing an agent within the host where the Docker platform is running. It is an agent-based system which requires metrics to be pushed to the *Datadog cloud* thereby making the task of monitoring entirely dependent on *Datadog's cloud*. Unlike the *Datadog agent*, the monitoring agent for the Docker platform presented in this paper is not only open-source, but also independent of any particular collector. It can be integrated within different monitoring architectures after the proper configuration is done.

cAdvisor is an open-source project created by Google Inc [2] to monitor their own *lmctfy* containers [11]. Support to monitor the Docker platform was later added to it. Therefore, this monitoring tool provides Docker metrics, which are shown in real time but are not stored for more than one minute. This feature may be useful to test container performance but, due to the small data time frame displayed, it is not possible to get a historical of the metrics collected.

The Host sFlow agent [12] is an open-source tool to monitor the resource consumption of a host. It has recently incorporated the option to monitor the Docker platform, making it a viable open-source monitoring solution for Docker. However, this agent adheres to the sFlow standard [6], which enforces constraints on the information it is able to send as there is no dedicated sFlow structure for Docker. By contrast, the solution provided in this paper does not have limitations on the metrics that can be obtained. The monitoring agent presented here can be modified to select which metrics, out of all the available, to monitor.

As shown above, only the solution proposed in this paper manages to provide a monitoring module for Docker that is open-source, does not adhere to a particular collector or monitoring framework provided some configuration is done, and allows for the selection of a certain subset of metrics from all the available ones.

3 System Architecture

To monitor Docker containers, three separate modules could be employed: our Container Monitoring Agent (CoMA), a metrics' collector and a host monitoring agent. The solution proposed in this paper, to create a distributed Docker monitoring system consists of: CoMA, the Ganglia Monitoring System and the Host sFlow agent.

In Figure 1, a possible layout of the proposed monitoring architecture is shown. This figure represents three clusters. Each one of the monitored hosts represented in Figure 1 have the structure presented in Figure 2.

3.1 CoMA: Container Monitoring Agent

We have developed CoMA, the agent that monitors containers of OS-level virtualization platforms such as Docker. CoMA retrieves information about the containers and images in the Docker platform. It also tracks the CPU, memory and block I/O resources being consumed by the running containers. CoMA can be accessed as an open-source project at *https://github.com/laraljj/CoMA*.

The collection of Docker metrics in CoMA is accomplished via two modules. One module makes requests to the platform's Remote API [5] to collect data about the containers and images. These include: the number of Go routines being executed by the Docker platform; the images that have been created in each host and information about these (e.g. size and virtual size); the number of containers that have been created and

[6] http://www.sflow.org/developers/specifications.php

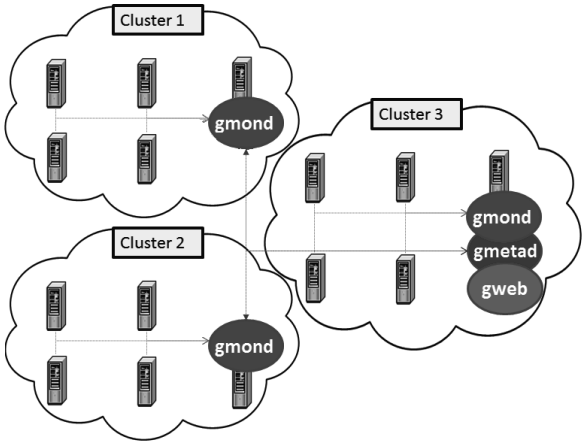

Figure 1: Cloud topology.

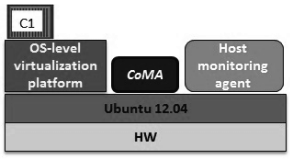

Figure 2: General overview of the host.

from which image they have been built; the status of all the containers in the host (i.e. whether they are running or stopped).

The second module obtains information about resource usage and resource limitations of the running containers. These metrics are obtained by accessing the control groups (cgroups) feature of the Linux kernel [16], which accounts and sets limits for system resources within different subsystems. The container resources monitored by this module include CPU, memory and block I/O. The number of measurements recorded by CoMA vary according to the number of containers that have been deployed. For the Docker platform as a whole, there are 18 CPU related metrics. Per container, 18 CPU related metrics, 24 memory related metrics, and 100 block I/O related metrics are measured. This means that when a single container is running, there are a total of 160 measurements available, 142 of these are container specific and 18 of these are related to the Docker platform itself. Therefore, when two containers are running there will be 302 measurements, 142 measurements for one container, 142 measurements for the other container and 18 measurements for the Docker platform. The metrics that are being reported by CoMA can be selected according to the needs of each specific deployment, so that only the values of those metrics are dispatched to the collector, instead of all of

them.

3.2 Complementary Components

The Ganglia Monitoring System

The Ganglia Monitoring System [14] is an open-source distributed monitoring platform to monitor near real-time performance metrics of computer networks. Its design is aimed at monitoring federations of clusters. The Ganglia Monitoring System was selected as collector due to its capacity to scale, its distributed architecture and because it supports the data collection of the Host sFlow agent. However, in the interest of fitting the requirements of a different system, a stand-alone collector could be used instead.

The system is comprised of three different units: *gmond*, *gmetad* and *gweb*. These daemons are self-contained. Each one is able to run without the intervention of the other two daemons. However, architecturally they are built to cooperate with each other.

Gmond is a daemon that collects and sends metrics from the host where it is running to *gmetad*. This is not a traditional monitoring agent, as it does not sit passively waiting for a poller to give the order to retrieve metrics. *Gmond* is able to collect metric values on its own, but *gmond*'s built-in metrics' collection may be replaced by the Host sFlow agent. This setup implies that, for the architecture chosen, *gmond* acts as in-between software layer for the Host sFlow agent and *gmetad*.

Gmetad is a daemon running a simplified version of a poller, since all the intelligence of metric retrieval lays, in our case, with the Host sFlow agent and *gmond*. *Gmetad* must be made aware of from which *gmonds* to poll the metrics. *Gmetad* obtains the whole metric dump from each *gmond*, at its own time interval, and stores this information using the RRDtool (i.e. in "round robin" databases).

Gweb is Ganglia's visualization UI. It allows for an easy and powerful visualization of the measurements collected, mostly in the form of graphs. New graphs combining any number of metrics can be generated, allowing the visualization of metrics to be customized depending on individual needs.

The Host sFlow Agent

The Host sFlow agent was selected as the host monitoring agent to track the resources consumed by the OS in the host running the OS-level virtualization platform. Monitoring resources both at the host level and at the virtualization platform level makes it possible to compare the values of the metrics for soundness checks, tracking problems at both levels.

The Host sFlow agent may retrieve information from within an OS running on bare metal or from within the hypervisor if the aim is to monitor virtual machines. This agent can be run in multiple OSs and hypervisors. The agent itself obtains the same metrics as *gmond*'s built-in collector does. The difference between these two solutions is that the sFlow standard, used by the Host sFlow agent to relay metrics, is considerably more efficient than *gmond*. This is because each sFlow datagram carries multiple metric

values, which reduces the number of datagrams that need to be sent over the network. For example, monitoring 1,000 servers with *gmond* would create the same network overhead as 30,000 servers with the sFlow protocol [14]. The sFlow protocol's efficiency justifies the usage of the Host sFlow agent in this monitoring system.

4 Evaluation

The primary evaluation objective is to assess the validity of the values of the metrics collected with CoMA. Validity in this context means to establish, for the metrics reported by CoMA, whether the measured values reflect the real values. Given the numerous metrics reported about CPU, memory and block I/O, a small subset of these metrics has been selected for the evaluation. This validity assessment is carried out on user CPU utilization, system CPU utilization, memory usage and number of bytes written to disk. User CPU utilization and system CPU utilization refer to the percentage of CPU that is employed to execute code in user space and in kernel space, respectively. It should be noted that for all tests, Ubuntu 12.04 LTS and the Docker platform 1.3.1 have been run on the same Dell T7570 computer. This computer runs an Intel Pentium D 2.80 GHz (800 MHz) and 2 GB of RAM at 533 MHz.

The evaluation presented collects for each host: host-specific metrics (reported by the Host sFlow agent) and the metrics from the Docker platform (reported by CoMA). The purpose of collecting the values for both sets of metrics was to compare and contrast the host's resource consumption against the resource consumption of the Docker platform, that is, against the resource consumption of the collection of containers and images within the host. The objective of this comparison is to offer a reliable overview of the system from a resource usage perspective. Comparing both sets of metrics, a system administrator can pinpoint the origin of a particular misbehavior by determining if the issue is due to the Docker platform (i.e. a specific container) or due to some other problem within the host but independent of the containers.

4.1 Validity of CPU and Memory Measurements

Three different scenarios have been set up to assess the collected values of the memory and CPU-utilization metrics. For all scenarios, 6 rounds, each one of 30 minutes have been run. Scenario 1 (Figure 3) presents a baseline of the CPU and memory usage while the OS is executing Docker, which runs a single unlimited container executing */bin/bash*, the Host sFlow Agent and CoMA. The container was not actively used during this scenario. Scenario 2 (Figure 4) is set up like Scenario 1, except for the fact that a workload generator was executed natively in the OS. This means that the process did not run containerized. *Stress-ng* [7] has been used to generate load on both CPU and memory. In order to create load on the CPU, two workers were launched so that there would be a worker per core. Each CPU worker executed *sqtr(rand())* to generate load. To stress the

[7] http://kernel.ubuntu.com/ cking/stress-ng/

memory, five workers were started on anonymous mmap, each one of these workers was set to 420MB. Scenario 3 (Figure 5) has been laid out exactly like Scenario 2, the only difference being that the *stress-ng* processes were executed containerized.

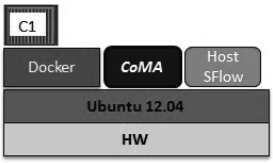

Figure 3: Scenario 1, no workload (baseline).

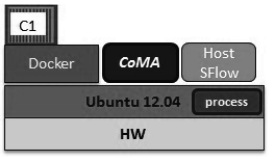

Figure 4: Scenario 2, workload on host OS.

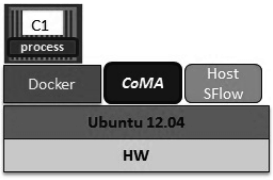

Figure 5: Scenario 3, workload in container.

CPU: A Single Container

The data obtained from each scenario were processed. Figure 6 shows user CPU, system CPU and total CPU utilization reported at the host level for each scenario. Figure 7 displays CPU utilization pertaining to the process or processes running within that single container deployed in the three scenarios.

In order to determine that the measurements of the CPU metrics collected with CoMA are valid, the data obtained from Scenario 3, which can be visualized in Figure 6 and 7,

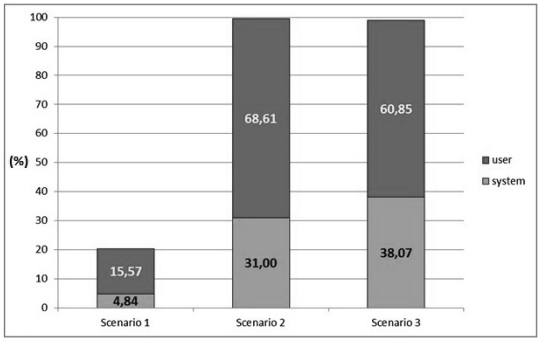

Figure 6: Host CPU utilization reported by the Host sFlow agent.

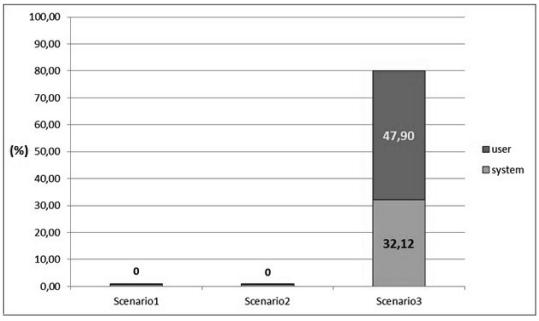

Figure 7: Container CPU utilization reported by CoMA.

has been compared. The total CPU of Scenario 1 (Table 1), aggregated to the total CPU of the container reported by CoMA in Scenario 3 (Table 2), should resemble the total CPU reported by the host in Scenario 3 (Table 1). The aggregation of those first two values (20.42% and 78.41%) results in a total CPU of 98.83% and a standard deviation of 6.5. The total CPU utilization of the whole host in Scenario 3 is 98.92% with a standard deviation of 15.29. These results verify that the values of the CPU metrics gathered by

Table 1: Host CPU utilization reported per scenario. Total CPU is the aggregation of user CPU and system CPU. Standard Deviation (SD).

	CPU system (%)	SD CPU system (%)	CPU user (%)	SD CPU user (%)	Total CPU (%)	SD Total CPU (%)
Scenario 1	4.84	1.79	15.57	5.59	20.42	5.87
Scenario 2	31.00	14.92	68.61	14.91	99.61	20.60
Scenario 3	38.07	11.20	60.85	11.16	98.92	15.29

Table 2: Container CPU utilization reported by CoMA per scenario. Note that there are no processes running in Scenario 1 and Scenario 2.

	CPU system (%)	SD CPU system (%)	CPU user (%)	SD CPU user (%)	Total CPU (%)	SD Total CPU (%)
Scenario 1	-	-	-	-	-	-
Scenario 2	-	-	-	-	-	-
Scenario 3	32.12	9.44	47.90	9.77	78.41	2.78

CoMA are valid.

The CPU utilization data retrieved from this evaluation allows for other noteworthy observations to be made. In Figure 6 and Table 1 a small difference of 0.69% can be ascertained, between running the *stress-ng* processes natively (Scenario 2) or containerized (Scenario 3). The disparity that exists between these two scenarios is due to several reasons. First, the intrinsic variable nature of the data collected has a direct impact on the results attained. However, its irregularity is acceptable as the standard deviations calculated demonstrate, since these are reasonable and valid for these data. Second, the *stress-ng* processes themselves may be accountable for a certain variation.

It can also be noticed that there seems to be a tendency in the way these *stress-ng* processes are executed. When *stress-ng* was run within a container more system CPU utilization was accounted for compared to when *stress-ng* was run natively. The effect is the exact opposite when it comes to user CPU utilization, as can be visualized in Figure 6. This last observation has been verified by computing the correlation coefficient between system CPU utilization and user CPU utilization. A nearly perfect negative correlation of -0.99 was obtained for Scenario 2 and -0.98 for Scenario 3.

CPU: Multiple Containers

These scenarios prove that the CPU utilization retrieved by CoMA, of one container, is valid. However, whether the agent is able to properly report the CPU metrics for multiple simultaneously running containers should also be demonstrated. For this purpose, two tests were carried out. For the first test, ten containers ran the exact same *stress-ng* process to generate load on the CPU with two workers, one per core. In accordance with the default process scheduler in Linux, the Completely Fair Scheduler (CFS) [8], the expected outcome of this test is for each container to employ a similar portion of the total CPU. As it can be observed in Figure 8 and Table 3, each container is granted an average of around 8% of the CPU. The aggregation of each container's total CPU utilization adds up to 80.07% which almost matches the total CPU utilization of the whole Docker platform (80.13%) as reported by CoMA.

This test shows that each container reports its own set of CPU measurements independently and is able to do so effectively. However, a different test was carried out to verify this by running asymmetric processes in two containers. Each container ran

[8] http://lwn.net/Articles/230501/

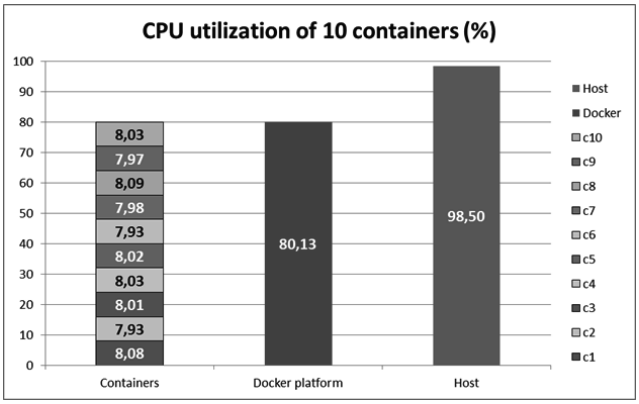

Figure 8: Total CPU utilization of 10 containers, where each container runs the same process generating a symmetric CPU load across all containers. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 3: Total CPU utilization and standard deviations (SD) for Figure 8.

Total CPU utilization (%)						
Host		Docker platform		Containers		
Total CPU	SD Total CPU	Total CPU	SD Total CPU	Name of container	Total CPU	SD Total CPU
98.50	3.30	80.13	1.10	c1	8.08	0.23
				c2	7.93	0.18
				c3	8.01	0.26
				c4	8.03	0.22
				c5	8.02	0.38
				c6	7.93	0.30
				c7	7.98	0.24
				c8	8.09	0.28
				c9	7.97	0.20
				c10	8.03	0.28

the *stress-ng* process with different settings so as to generate an uneven load across the two containers. As represented by Figure 9 and Table 4, CoMA reported just that. A container used 48.61% of the total CPU whilst the other container employed 18.57% of the total CPU. Both containers together used 67.18%, which resembles the value (67.20 %) reported by CoMA of the total CPU utilization of the Docker platform.

Memory: A Single Container

The memory data captured for all scenarios is displayed in Figure 10. It should be mentioned that there is a greater fluctuation in the memory-reported values than in the CPU values. This phenomenon is due to the manner in which Linux manages its memory.

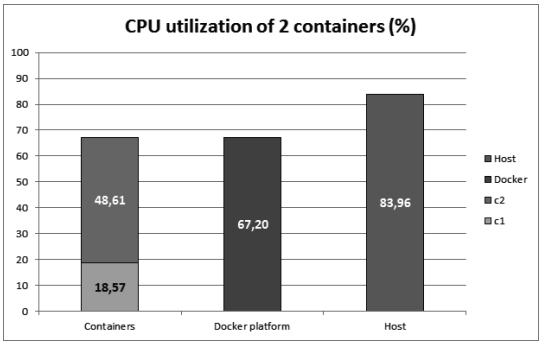

Figure 9: Total CPU utilization of 2 containers with asymmetric processes running in each container. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 4: Total CPU utilization and standard deviations (SD) for Figure 9.

Total CPU utilization (%)						
Host		Docker platform		Containers		
Total CPU	SD Total CPU	Total CPU	SD Total CPU	Name of container	Total CPU	SD Total CPU
83.96	1.46	67.20	0.65	c1	18.62	0.28
				c2	48.61	0.59

The memory management methodology applied by Linux varies according to the needs of the OS at any given time. This adds another layer of complexity when analyzing the metrics collected. The host's memory usage in Scenario 1 (360.73MB) aggregated to the container's memory usage reported by CoMA in Scenario 3 (1349.09MB), should be somewhat similar to the host's memory consumption in Scenario 3 (1544.37MB). In this case there is a difference of around 165MB. As it has been explained before, this discrepancy is caused by the memory management enforced by Linux, as well as by the error introduced in the averaging process of the results.

Table 5: Memory usage and standard deviations (SD) for Figure 10.

	Host-reported memory		Container-reported memory	
	Memory usage (MB)	SD Memory usage (MB)	Memory usage (MB)	SD Memory usage (MB)
Scenario 1	360.73	11.35	4.84	0.02
Scenario 2	1591.14	224.43	0.14	0.17
Scenario 3	1544.37	209.47	1349.09	184.57

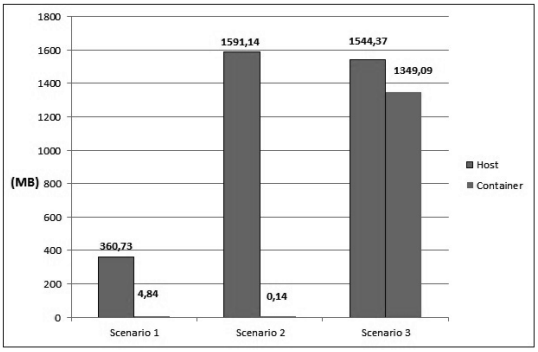

Figure 10: A comparison of the memory usage reported by the host and the container per scenario.

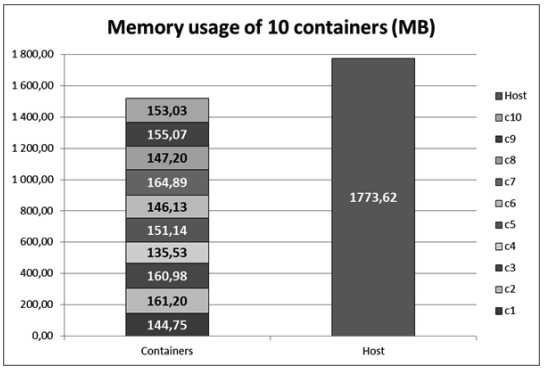

Figure 11: Memory usage of 10 containers, where each container runs the same process. This generates a symmetric memory use across all containers. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Memory: Multiple Containers

Much like it happens with CPU, the previous scenarios establish that the memory metrics provided by CoMA are valid for a single container. The two CPU tests performed with multiple simultaneously running containers, were also carried out for memory. As it can be observed in Figure 11 and Table 6, when the same process is running in each container, the memory usage value presented by CoMA per container has a greater variability than that observed in the same test for CPU utilization. As it has been previously explained, these fluctuations are due to the changeable nature of how memory is managed by the OS. However, each container's memory usage is close to 152MB. The aggregated memory usage of all 10 containers adds up to 1519.92MB. The Host sFlow agent reports a memory

Table 6: Memory usage and standard deviations (SD) for Figure 11.

Memory (MB)				
Host		Containers		
Memory usage	SD Memory usage	Name of container	Memory usage	SD Memory usage
1773.62	59.57	c1	144.75	20.60
		c2	161.20	20.87
		c3	160.98	11.56
		c4	135.53	15.96
		c5	151.14	11.86
		c6	146.13	14.54
		c7	164.89	14.50
		c8	147.20	12.54
		c9	155.07	20.74
		c10	153.03	27.69

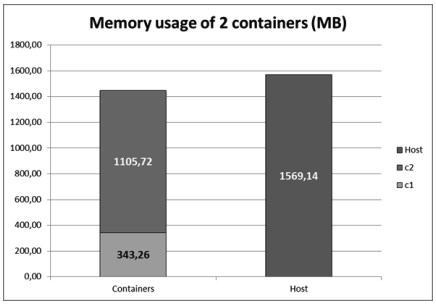

Figure 12: Memory usage of 2 containers with asymmetric processes running in each container. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

usage of 1773.62MB for the whole host during this test. The difference of 253.70MB between these last two values, represents the memory being employed by the OS to run non-containerized processes. A second test, where two containers were configured to make a disparate use of memory was also carried out. Figure 12 and Table 7 reflect the results obtained, which are consistent with the values gathered when running a symmetric memory load on 10 containers.

Table 7: Memory usage and standard deviations (SD) for Figure 12.

Memory (MB)				
Host		Containers		
Memory usage	SD Memory usage	Name of container	Memory usage	SD Memory usage
1569.14	194.24	c1	343.26	65.01
		c2	1105.72	130.16

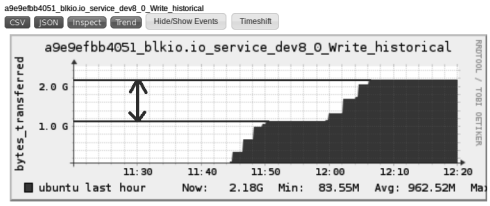

Figure 13: Bytes written to disk by container.

4.2 Validity of Block I/O Measurements

To evaluate whether the block I/O measurements gathered by CoMA were solid, the I/O tool *fio* [9] was used to write 1000MB directly to the ext4 filesystem mounted by the host by making use of the -v flag [8] on the container. In order to achieve this, *fio* was configured to initiate five workers, each worker performing random write operations of 200 MB in the shared folder between the host and the container.

The test of writing 1000MB to disk was executed at 12:00 and it finished by 12:07. As Figure 13 shows, exactly 1000MB were reported to have been written during that time.

A separate test was created, following the same principle previously explained, to write to disk from three simultaneously running containers. *Fio* was configured for each container with a disparate number of workers and file sizes. The first container spawned two workers, each of which had to write 300MB to the shared folder. The second container initiated three workers, each with a file size of 250MB. The third container started five workers, where each worker had to write 200MB to disk. For each container, the number of bytes that CoMA reported were written to disk was exactly right, down to the last byte. The first container took 20 minutes to write the 600MB to disk. The second and third container took around 16 minutes to write 750MB and 1000MB to disk, respectively. The time taken for each container to complete the task of writing these files to memory is closely linked to the number of workers running and the number of containers writing to disk.

5 Discussion

This section discusses CoMA as well as the evaluation results obtained in terms of the research questions proposed.

How could an OS-level virtualization platform be monitored to obtain relevant information concerning images and containers?

CoMA retrieves information about the usage of CPU, memory and block I/O of running containers from the Linux kernel's control groups. It also fetches the data concerning images and containers using Docker's Remote API.

[9] http://freecode.com/projects/fio

Is the resource usage information about the running containers reported by our Container Monitoring Agent valid?

The evaluation provides validation across the following three blocks of metrics: CPU, memory and block I/O. The most complex metric to validate was the memory usage of a container. This is due to the way memory is managed in an OS, which causes the memory usage baseline in Scenario 1 to account a slightly overestimated usage. The authenticity of all the measurements that can be collected with CoMA could not be tested because of the number of metrics that CoMA is able to gather. Nevertheless, since the values are being reported by the Linux kernel, assessing at least one metric from each group of metrics is sufficient to establish the validity of CoMA. It should be mentioned that CoMA can be modified to only dispatch a subset of desired metrics.

CoMA's CPU utilization is dependent on the test-bed that has been set up. This means that CoMA's resource usage is contingent on the hardware that has been employed, the number of containers that had been deployed, the number of metrics being sent and the sampling rate set for CoMA. This last value can be configured to obtain measurements closer or further away from real-time monitoring, depending on the requirements. There are certain tradeoffs in the selection of the sampling rate. A higher sampling rate would mean obtaining more accurate measurements in terms of time, but more resources would be used in order to monitor the platform. It is worth mentioning that CoMA itself consumes around 15.25% of CPU with a standard deviation of 5.87 for the specific test-bed presented in the evaluation section. This number may seem high, but it is relative to the hardware being employed. An Intel Pentium D 2.8GHz and 2GB RAM at 533MHz was used in this case. Had conventional cloud computing hardware been used, this percentage would be much lower. Moreover, in this test-bed all available metrics are collected, if fewer of them were collected the percentage of CPU used would decrease. It should also be mentioned that the monitoring solution itself shall be optimized so as to minimize its impact.

It has been previously mentioned that CoMA could be employed to monitor similar OS-level virtualization platforms. For this to happen, said OS-level virtualization platform would have to account resource usage in a similar fashion to Docker, i.e. using the Linux kernel's control groups. However, the information pertaining to the containers and images that is collected through Docker's Remote API, is specific to the Docker platform itself.

6 Conclusion and Future Work

Monitoring the resource consumption of OS-level virtualization platforms such as Docker, is important to prevent system failures or to identify application misbehavior. CoMA, the Container Monitoring Agent presented in this paper, reports valid measurements as shown by our evaluation. It currently tracks CPU utilization, memory usage and block I/O of running containers. CoMA could be configured to gather a subset of the available metrics to suit the monitoring needs of a particular system or application. This paper

has presented a possible implementation solution of CoMA to build a distributed and scalable monitoring framework, using the open-source projects Ganglia and the Host sFlow agent.

It would be positive to monitor the network usage of the containers, since this feature has not yet been implemented in CoMA. Moreover, establishing thresholds on certain metrics collected by CoMA to trigger alarms or actions would be beneficial. Also, assessing CoMA's behavior when numerous containers are deployed on commonly used hardware in data centers is required. This would be a proper test-bed to gauge CoMA's performance in a realistic cloud computing scenario.

Another area for further research would be to employ machine learning techniques on the values collected, to maximize resource usage by modifying each container's resource constraints based on the needs of the running containers. There is also the possibility of applying data analytics on the information captured by CoMA to build an autonomous system for container placement within a cloud or across clouds.

There are new and upcoming OS-level virtualizations platforms that could rival Docker, such as Rocket. CoMA could be also employed and evaluated with these recent virtualization platforms.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[2] Bryan Lee. cAdvisor monitoring tool , Accessed: 2014.

[3] Datadog Inc. Docker-ize Datadog with agent containers , Accessed: 2014.

[4] Docker Inc. What is Docker technology ?

[5] Docker Inc. Docker remote API, Accessed: 2014.

[6] Docker Inc. Docker working with LXC, Accessed: 2014.

[7] Docker Inc. File sytem architecture of the Docker platform , Accessed: 2014.

[8] Docker Inc. Volume system with Docker, Accessed 2014.

[9] Thomas Nyman N. Asokan Elena Reshetova, Janne Karhunen. Security of os-level virtualization technologies. 2014.

[10] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[11] Google Inc. lmctfy: Let Me Contain That For You , Accessed: 2014.

[12] InMon Inc. HostsFlow monitoring tool , Accessed: 2014.

[13] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: Online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[14] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock. *Monitoring with Ganglia*. O'Reilly Media, Inc., 1st edition, 2012.

[15] Shicong Meng, Arun K. Iyengar, Isabelle M. Rouvellou, Ling Liu, Kisung Lee, Balaji Palanisamy, and Yuzhe Tang. Reliable state monitoring in cloud datacenters. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 951–958, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Paul Menage. Control Groups (cgroups) Documentation , Accessed: 2014. Available since: 2004.

[17] Buyya R. Leitner P. Haller A. Ranjan, R. and S Tai. A note on software tools and techniques for monitoring and prediction of cloud services softw: Pract. exper., 44: 771-775. 2014.

[18] I. Tafa, E. Beqiri, H. Paci, E. Kajo, and A. Xhuvani. The evaluation of transfer time, cpu consumption and memory utilization in xen-pv, xen-hvm, openvz, kvm-fv and kvm-pv hypervisors using ftp and http approaches. In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 502–507, Nov 2011.

[19] VMware Inc. Understanding Full Virtualization, Paravirtualization and Hardware Assist, 2007. Accessed: 2014 (white paper).

[20] VMware Inc. Virtualization Overview, 2007. Accessed: 2014 (white paper).

[21] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240, Feb 2013.

[22] Fei Xu, Fangming Liu, Hai Jin, and A.V. Vasilakos. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, Jan 2014.

[23] Kejiang Ye, Dawei Huang, Xiaohong Jiang, Huajun Chen, and Shuang Wu. Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective. *IEEE-ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing*, 0:171–178, 2010.

Paper B

DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications

Authors:
Lara Lorna Jiménez and Olov Schelén

Reformatted version of paper originally published in:
The 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications (IEEE Cloud Summit), 2019, Washington, D.C., USA.

DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications

Lara Lorna Jiménez and Olov Schelén

Abstract

The advent of the Internet-of-Things and its associated applications are key business and technological drivers in industry. These pose challenges that modify the playing field for Internet and cloud service providers who must enable this new context. Applications and services must now be deployed not only to clusters in data centers but also across data centers and all the way to the edge. Thus, a more dynamic and scalable approach toward the deployment of applications in the edge computing paradigm is necessary. We propose DOCMA, a fully distributed and decentralized orchestrator for containerized microservice applications built on peer-to-peer principles to enable vast scalability and resiliency. Secure ownership and control of each application are provided that do not require any designated orchestration nodes in the system as it is automatic and self-healing. Experimental results of DOCMA's performance are presented to highlight its ability to scale.

1 Introduction

Digital transformation is currently driven by the Internet-of-Things (IoT). This constantly increasing market means that billions of devices will need to be connected to the network and have their data processed. Gartner's [4] assessment is that connected devices will number at least 20 billion by 2020. They expect these devices to collect more than 1.6 trillion GB of data. Many industries make use of IoT, such as the automotive industry, smart homes and cities, manufacturing, healthcare, insurance, logistics, and agriculture. Each of these requires diverse applications for different types of devices to process their data and obtain value. These applications have different requirements, many of which establish low-latency constraints. However, it is clear that they all receive considerable data streams and generate substantial data to perform their intended functionality. The Cisco Global Cloud Index [2] estimates that IoT applications will generate approximately 847 zettabytes of data by 2021. This new context poses technological difficulties that must be addressed.

The primary industries that must enable this new paradigm are the telecommunications operators and the cloud service providers. In terms of network infrastructure, a new standard for mobile networks, 5G, is already being developed. 5G provides a considerably more extensive data capacity than its predecessor, as well as higher speed and low-latency. In terms of application deployment, cloud computing (CC) has been

and will continue to be the leading solution. CC operates by centralizing computing, memory and storage resources in data centers (DCs), which are strategically placed near the energy sources needed to power them. CC is cost-efficient, but most importantly, it has worked towards maximizing resource utilization and optimizing its infrastructure. However, CC cannot overcome the hurdles imposed by IoT, mainly longer latencies and network saturation. This led to the emergence of fog or edge computing (EC) [9], which is based on the idea of extending the cloud by placing resources on which to execute applications or services closer to the edge of the network. The understanding of "edge" varies. It can be understood as being anywhere in between small hubs of servers in a city to micro-DCs at a base station or even the end-devices. EC is a complementary solution to CC and pursues three main objectives: minimize latency, reduce core network traffic, and improve data security and privacy through proximity to the data source.

To deploy applications in this archetype of the extended cloud, an orchestrator is needed that can scale to a vast pool of geographically distributed and, likely, heterogeneous nodes. We must also consider how those applications should be designed, and thus, defined in this orchestrator. It is imperative that applications are deployed in a resource efficient manner. Virtualization serves to maximize the usage of the underlying hardware while isolating processes. Virtual machines (VMs) have been employed for this purpose in CC for many years. However, lightweight virtualization through process containerization is popular, given that it provides better performance than VMs in the form of lower overhead and faster deployment speed [3]. With the debut of the containerization platform, Docker, containerized processes offer portability and operational simplicity while promoting modularity and an effective development pipeline. Therefore, our extended cloud orchestrator will deploy containerized applications. For the architectural design of the applications themselves, we are moving away from the monolith to a microservice-style architecture (MSA) [7], which consists of breaking down an application into loosely coupled services that are independent and that intercommunicate through, preferably, application-agnostic protocols. Thus, services become reusable components that are independently developed. Given MSA's modularity, services can also be scaled separately depending on the needs of each service instead of having to scale the whole application. MSA and containers share many core traits, reinforcing one another harmoniously.

In conclusion, the required orchestrator is one that can deploy containerized MSA applications in the extended cloud scenario. Orchestrators for containerized MSA applications exist. However, these orchestrators do not scale to the number of nodes that will be required in this new paradigm, nor are they designed to work in the EC scenario. To approach this complex problem, we explore the following question: "How can an orchestrator for containerized MSA applications be designed that scales to a large pool of nodes beyond the current scale provided by existing orchestrator solutions while being adaptable to the EC scenario?"

Thus, we created DOCMA, an orchestrator for MSA applications that contributes by the following novel features compared to the current state-of-the-art orchestrators:

- DOCMA can scale to a large pool of geographically distributed nodes, satisfying the requirements of edge computing.

- DOCMA is distributed and decentralized: nodes make decisions locally and are independent of one another.
- All nodes are orchestrators: each node can deploy applications, host services, and monitor applications.
- DOCMA is self-healing and resilient, making it highly available. A network partition may occur, or nodes may join or leave the network. The orchestrator ensures that the applications are running when events such as these take place. Self-healing is performed by the orchestrator through its innate characteristics, ensuring that other nodes naturally take up the functions that have been dropped by nodes that are no longer available.

This paper focuses on presenting the overall system design for which a PoC implementation has been developed.

2 DOCMA: an overview

DOCMA is a scalable orchestrator for containerized MSA applications. Orchestration in the proposed solution is fully automatic. The only human input needed is that of the application configuration data; thereafter, the application runs autonomously for the duration of its life-cycle. The orchestrator creates a vast overlay network of computers on which to deploy these applications. These computers may be virtual (i.e., virtual machine) or physical entities. DOCMA is designed with a flat rather than hierarchical architecture, meaning it is distributed and fully decentralized. All nodes in the orchestrator are equal. Any node can be an application orchestrator node for zero or more applications. All nodes are resource-provider nodes; that is, they may all host services of different applications. DOCMA builds this overlay network of nodes based on Kademlia's [6] routing strategy. Nodes in DOCMA are a heterogeneous or a homogeneous set of networked computers with the ability to host containerized processes. DOCMA's functionality focuses on searching across the network for the resources required to deploy the applications, keeping those applications running, performing self-healing when nodes fail or become unreachable, and ensuring that all applications' services are discoverable.

2.1 Applications

Applications in DOCMA are characterized as a collection of services that intercommunicate and are defined by the application owner via the *application configuration data*. These data include affinity or anti-affinity constraints between services. They also outline each service by providing image and repository information, resource usage requirements, and other data (e.g., ports, number of replicas, replica types, whether it is a task or a long-lived service). Once deployed, *application deployment data* and *service liveness data* for these services are automatically kept by the orchestrator. Application configuration, deployment and liveness data hereafter are referred to as *application metadata*.

2.2 Routing

When joining the network, nodes must either generate a random ID or obtain it from an external system. For the orchestrator to be used in the EC setting, the latter situation is mandatory. That external system links the geographical location to an ID. However, this feature is assumed in this paper. The node ID will uniquely identify each node in the network. DOCMA employs a 256-bit address space, meaning a node ID is represented by 256 bits. Each node maintains a routing table to be able to communicate with other network nodes. The address space defined can be seen as a binary tree, where each node is a leaf. The routing table can store a small subset of all leaves or nodes compared to all the nodes that may exist in the network. The routing table is composed of 256 buckets. There is one bucket per bit of the ID. A bucket can store up to k node entries. As the node learns of other nodes on the network (i.e., <ID, IP, port>), these are stored in the appropriate k-bucket. Thus, bucket 0 can hold nodes from half of the whole network address space. However, the bucket will only hold up to k of those nodes. Bucket 1 will store up to k nodes from a quarter of the remaining address space, and so on.

These buckets are constantly updated as each node receives requests or responses from other nodes. This ensures that every node has an updated but limited view of the network, which enables the system to scale successfully to a large number of nodes. However, being able to contact a bounded number of nodes on the network is only one part of the routing equation. Another important capability is for any node on the network to be able to find any other node through its ID, given its limited knowledge of the network, which is facilitated via Kademlia's node lookup iterative algorithm, which is an efficient search algorithm that returns the k closest nodes to a given ID. This search is $O(log_b N)$. Both updating the routing table and the node lookup algorithm are based on the distance between IDs. Kademlia's distance function is calculated by performing an exclusive OR (XOR) between IDs. XOR is a logical distance function that is not related to geographical closeness. It is a simple distance function with great properties for building an overlay network. To gain a better understanding of Kademlia's peer-to-peer (P2P) distributed hash table (DHT) routing and its properties, we recommend reading [6].

2.3 Roles

DOCMA defines four roles per application: entry, controller, leader and service hosting. A node that is involved in orchestrating a particular application will generally take on one of these roles. To deploy an application, all roles must be taken up by nodes on the network. A node may adopt the same or different roles for different applications. For example, a node may be the leader for one application and the service hosting node for a service replica of another application. Ideally, and most likely, for a given application, all of its roles are acquired by separate nodes.

Entry

The entry role is adopted by a node when it is the first node of the network to receive a request related to a given application. It may be a deployment, information, a deletion or a modification request for an application on the network. This node searches for the nodes that should assume or that have already adopted the controller role for this application and submit the request to them. The entry role is ephemeral, existing only while the request and subsequent answer take place. It acts as a relay between the issuer of the request and the orchestration nodes (i.e., those nodes acting as controllers and leader) of that application.

Controller

This is the passive orchestrator role of an application. There are β controller nodes per application. Controllers are the β nodes on the network whose IDs are closest to the application ID. A controller holds the application's metadata. Controllers are in charge of ensuring that there is always an active leader for the application. For that purpose, controllers perform leader election among themselves when necessary. This is done through DOCMA's consensus algorithm.

Leader

This is the active orchestrator role of an application. The role of the leader for a given application is taken up by a single node on the network. A node takes up this role either as determined by the first-time deployment of the application or through consensus among controllers during a leader election process. A leader is responsible for deploying, removing or modifying the application or its services. It tracks service liveness and performs fail-over and service recovery for the application's services when necessary. It also ensures that there are β active controller nodes.

Service hosting

This role applies to any node that is hosting a service replica of a given application. A service hosting node is initially a node on the network that is queried by a leader about whether it can run a service replica given the resources it requires. The node locally determines whether it can run the service replica. If it can, it reserves those resources expecting a deployment command from the leader. If that command arrives before a given time, it deploys the service replica. If it times out instead, the reserved resources are freed. The service hosting node periodically sends heartbeats to the leader for the purpose of maintaining service liveness awareness.

2.4 The DOCMA protocol

The DOCMA protocol employs eleven remote procedure calls (RPCs) to maintain communication between nodes across the network and execute the orchestrator logic. Two

RPCs are part of the Kademlia protocol: *Ping* and *Find node*. These are employed for routing purposes. The other RPCs are as follows: 1) *DeployApp* 2) *Status* 3) *Remove* 4) *Update* 5) *Controller* 6) *Resources* 7) *Deploy* 8) *Data* 9) *Vote*. These RPCs are used to build DOCMA as a fully decentralized orchestrator for MSA applications.

3 Deploying an Application

A DOCMA client may submit an application deployment request to the DOCMA network through any node via a *DeployApp* RPC. The client only needs to know and be able to communicate with at least one node on the network. The node that receives a *DeployApp* RPC becomes an *entry node* for that deployment request. When sending a deployment request, the client locally generates a public-private key pair and an application ID for the submitted application. The 256-bit application ID is created by hashing the application configuration data provided by the application owner and the locally generated private key. This is the application ID in DOCMA throughout the entire life-cycle of the application. The private key and application ID are only accessible to the application owner at the client that locally generated them both. Once generated, the client sends the public key and the deployment request to the entry node (Figure 1, #1). Public-key cryptography is employed as a means of securing deployed applications on DOCMA. Then, any requests sent to the orchestrator network related to an application are signed by the application owner, and any answers sent back by DOCMA are encrypted, making them only readable by a user possessing the appropriate private key (i.e., the application owner).

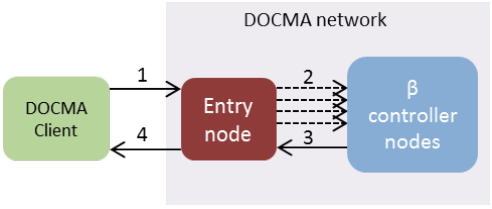

Figure 1: A DOCMA client submits requests regarding an application to any node on the network, which becomes the entry node. Depending on the request, this node establishes or contacts the controllers for that application ID.

To enable deployment of the application, the entry node sends asynchronous *Controller* RPCs to the β closest nodes to the application ID on the network (Figure 1, #2). Upon receiving the RPC, that group of nodes become the β *controller nodes*. These β nodes are found through the node lookup algorithm mentioned in Section 2. The first deployment of an application establishes the initial *leader node*. This leader is the node whose ID is closest in terms of XOR distance to the application ID out of the controllers

identified by the entry node. Upon receiving the RPC, the leader attempts deployment of the application. This process is further explained in Section 4. However, if the leader is unable to find viable nodes to host the application's services, then deployment fails. Eventually, the entry node receives a deployment confirmation or failure along with application configuration data if it is successful (Figure 1, #3).

A user may also request information about a deployed application employing a *Status* RPC, modify an application through an *Update* RPC or remove an existing service or application from the network via a *Remove* RPC. The requests traveling across the DOCMA network are always signed. Therefore, the controller nodes and the leader node for a given application only accept and carry out those requests that are signed by the application owner. The response sent back (Figure 1, #4), if any, is encrypted with the application's public key.

4 Orchestrating an Application

Every application in the orchestrator is associated with a set of orchestrator nodes, which are those nodes with the roles of leader and controllers for that application. These roles are not permanently set to specific nodes. Instead, they are dynamically adopted as circumstances in the network change, such as when nodes leave or new nodes join the network. Controllers together with the leader act as anchors for a given application running on the DOCMA network.

A *leader node* carries out the deployment of all the services defined by the application configuration data. This node must use a search algorithm that employs *Resources* RPCs to search the network for a set of nodes that can host the required number of replicas for each particular service given the resources necessary for each service to operate. Once found, those nodes receive deployment requests via *Deploy* RPCs. Those nodes will deploy the designated services and thus become *service hosting nodes*. We designed various search algorithms to search for resources across this overlay network. However, these are not presented in this paper.

For a deployed application, the leader maintains the application configuration, deployment and liveness data. The latter two are received from the service hosting nodes when a deployment change occurs or through heartbeat messages. If a service replica fails or becomes unreachable, the leader node is in charge of redeploying it and distributing the new deployment data through *Data* RPCs. Thus, application services are kept running and discoverable to the concerned parties. However, any node on the DOCMA network may leave, fail or become network partitioned at any given time. Therefore, a redundancy mechanism in the orchestration of an application is essential. Thus, a leader republishes the application metadata through *Controller* RPCs to the β closest nodes to the application ID at a randomized interval determined by the republishing parameter, which activates or maintains those nodes as controllers for that application with the latest application metadata. As nodes leave and join the network, the β controller nodes may change. A new node (Figure 2, N41) may be closer to a given application ID and take another node's (Figure 2, N50) position as controller when republication next

occurs. The now-former controller (Figure 2, N50) eventually sheds that role.

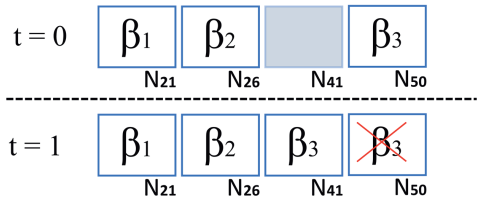

Figure 2: An application with ID 39 has 3 controllers: N21, N26 and N50, as they are the closest to the application ID out of all the nodes on the network. If a new node (N41) joins the network when republication from the leader to the controllers occurs, N41 will become a controller to that application and N50 will cease to hold that role after a given time.

A *controller node* is a β redundancy node for the orchestration of a given application. In simplistic terms, controllers act as possible backup leader nodes. Any of them is ready to become a leader if required. Their core objective is to ensure that there is always one active leader for the application, guaranteeing minimum interruption in active application orchestration, which is ensured via a consensus algorithm to perform leader election among nodes to elect a new leader when the previous leader has been deemed unreachable. We designed a consensus algorithm for this particular purpose. In essence, the consensus algorithm acts in a two-step process. First, a decision about whether the leader is unreachable is obtained by using *Controller* RPCs. Second, if the leader is unreachable, a new leader is elected out of the participating nodes by way of a leader election through *Vote* RPCs. All participating nodes in the election work towards the same goal. They vote for the node whose ID is closest to the application ID. Once a consensus is reached, the new leader node is elected.

The search algorithms and consensus algorithm designed, the process of republication and replication of services, and the failure recovery mechanism of DOCMA will be presented at greater length in an upcoming publication.

5 Related Work

Orchestrator solutions for microservice applications are not new. There are multiple commercial container orchestration solutions, such as Kubernetes, Mesos, and Docker Swarm. However, these orchestrator solutions are centralized. They control a cluster comprised of a limited number of nodes to which they deploy containers. Options to orchestrate multiple clusters exist, such as Kubernetes' Federation, which is performed through a layer on top of the cluster orchestrator, and therefore, still centralized. These characteristics are not favorable in terms of scalability or response time, both of which are important for EC. We propose a different architectural approach to an orchestra-

tor solution by building it as a scalable system that can operate in the context of edge computing via decentralization and location awareness. A comparison of container orchestration solutions can be found at [8].

A method for orchestrating microservices for IoT using Docker and EC is presented in [1]. However, the objective of this paper is to provide an architectural blueprint of an IoT MSA application that efficiently leverages EC by employing Docker Swarm. In contrast, our proposed orchestrator works towards deploying a large number of MSA applications to a far-reaching pool of distributed nodes. Therefore, it is an orchestrator for cloud service providers, not for the deployment of a single application. The architectural design of the application is a different matter that is outside the scope of the proposed orchestrator.

DOCMA's routing is based on Kademlia, a P2P DHT. There are other such DHT network solutions, such as Chord, Pastry, CAN, and Tapestry. However, Kademlia has the same or better routing performance, is more resistant to churn and is the fastest at bootstrapping on a large scale. Many systems employ Kademlia, or variations of it, for routing: the Kad network, some blockchain designs (e.g., Holochain, Ethereum), and interplanetary file system (IPFS).

6 Results and Discussion

This publication presents a fully decentralized orchestrator for containerized MSA applications. This orchestrator sets the foundation on which to build an orchestrator for MSA applications for the upcoming edge computing paradigm. Our focus when designing this orchestrator is on creating a scalable solution that will allow applications services to be deployed across geographically distributed clusters according to service-specific requirements, i.e., a given service might be deployed closer to a particular edge DC to minimize latency to users there. This paper presents a quantitative analysis of DOCMA.

DOCMA's design is that of a fully decentralized distributed system, meaning its redundancy is equal to the number of nodes that make up DOCMA. DOCMA is tolerant to node churn and network partitioning. Any request that is served by the orchestrator can be served by any node on the network, e.g., application deployment request. This characteristic makes DOCMA particularly resilient, while its routing and discovery mechanisms fashion a largely scalable system. Given DOCMA's design, it could potentially scale to 100,000+ nodes, much like Kademlia has been proven to scale to. Nevertheless, while decentralization provides some distinct advantages, it incurs implementation and optimization complexity and troubleshooting difficulty.

A PoC implementation of DOCMA is employed to assess the performance of the orchestrator. In this paper, DOCMA is analyzed in terms of scalability and performance, which is done by performing experiments where an application is deployed to the DOCMA network. Each experiment has an established network size and an overall network workload percent when the experiment begins. Given those two parameter inputs, the outcome is analyzed in terms of the number of messages required to successfully deploy an application in each scenario.

An external system, which will be referred to as the Experimenter, was developed to carry out the experiments. Values for the parameters that determine each experimental setup are provided to the Experimenter. This system executes each experiment, which entails deploying the number of DOCMA nodes defined for that experiment and providing the configuration for the experiment to those nodes. Once all nodes have registered and obtained their configuration information, the Experimenter requests deployment of an application to the network. The outcome of that deployment and its metrics are provided by the leader for that application to the Experimenter. The Experimenter then brings down the DOCMA network of nodes and proceeds to execute the next experiment.

All the experiments presented here are performed by simulating both the DOCMA nodes and the application deployment. As such, each DOCMA node obtains as part of its configuration, a random ID, its HW capacity and available HW resources. This information is used by each node to simulate its resources, which determines how the node answers requests to host services. Moreover, when a node deploys a service, it is a simulated deployment, meaning the resources needed to execute that service are used up by no longer being available node resources, but the service image is not pulled from a registry into the node, and the service container is not run. These experiments were run on a VM of 32 vCPUs and 32 GB of RAM, where each node was run as a process.

For all the experiments presented in this paper, the same HW is provided to all nodes, meaning it is a homogeneous network setup. For these experiments, the HW is defined as a single resource, specifically 64 GB of RAM. The nodes are also provided available HW (i.e., memory) for each experiment. This value of available HW is different for every node on the network. If, for a given experiment, the network of nodes is determined to be at a 90% load on average, then the Experimenter dynamically generates the available HW value for each node employing a Gaussian distribution while ensuring that this value is always smaller than the node's assigned HW and that the generated overall average load across the whole network of nodes is kept at 90%. Therefore, no two iterations of the same experiment are the same. Likewise, when an application is deployed during an experiment run, its services, specifically all service resource requirements, are generated using a Gaussian distribution. Applications in all the presented experiments are defined as having five services, and on average, services of a generated application require 2% of the average node HW on the network, which means that no two generated applications have the same combination of service resource requirements; that is, all generated applications are different.

For this paper, a total of fourteen experiments were run. Figure 3 presents the results of these experiments as a graph of the mean and standard deviation of the total number of RPC messages that were sent to deploy an application. The mean and standard deviation of each experiment was computed from 100 iterations of that experiment. These experiments were run by varying two parameters: the average pre-existing workload on the network of DOCMA nodes and the network size, that is, the number of DOCMA nodes. The first parameter was set at two values for these experiments: 30% (i.e., Figure 3 - lighter) and 90% (i.e., Figure 3 - darker). The second parameter was varied at seven different levels from 1,000 nodes to 7,000 nodes, with a step of 1,000 nodes, as presented

in Figure 3's *x-axis*.

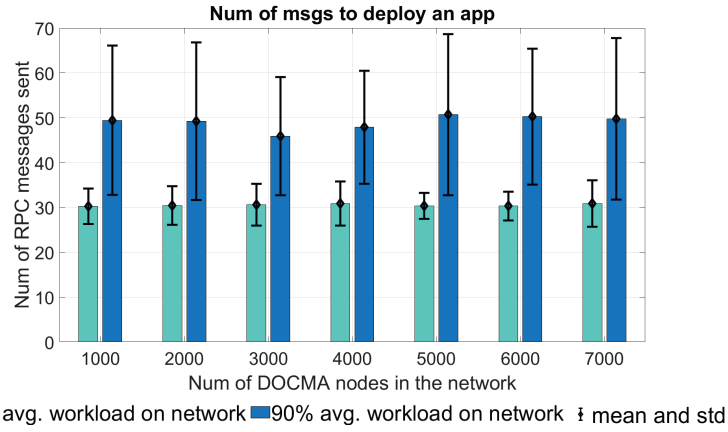

Figure 3: The number of RPC messages sent to deploy an application using DOCMA for 14 experiments is shown. Each bar presents the mean and standard deviation resulting from 100 iterations of the same experiment.

The results were computed as the number of RPC messages sent instead of as a measurement of time because the latter depends largely on the experimental setup. If nodes are simulated to be geographically distributed, the latency set between nodes greatly affects the outcome. However, the number of RPC messages is a more unbiased assessment in determining what is needed to deploy an application during a given experiment in DOCMA. The number of bytes sent was not employed as the outcome variable either because, depending on the serialization used for the RPCs, the output values would be noticeably different, whereas the number of RPC messages sent is unrelated.

As observed in Figure 3, the results obtained seem to imply that the number of RPC messages sent to deploy an application when the pre-existing network load is at 30% does not noticeably vary independently of the number of nodes on the network. The same applies to the 90% workload scenario. To establish whether these variations are statistically significant, three one-way ANOVA tests were run. The first ANOVA determined whether the difference in the average number of RPC messages sent to deploy an application in the 30% average workload scenario was statistically significantly dependent on the number of nodes on the network. The second test was performed for the same purpose but for the 90% average pre-existing workload scenario. Finally, the third test established whether there was statistical significance in the variations between the number of RPC messages sent in the 30% average workload scenario versus the 90% scenario, independent of the size of the network. For these ANOVA tests, the significance level set was 5%, and the null hypothesis was that all means were equal.

The first ANOVA test results can be seen in Figure 4 (i.e., left F-statistic). In this

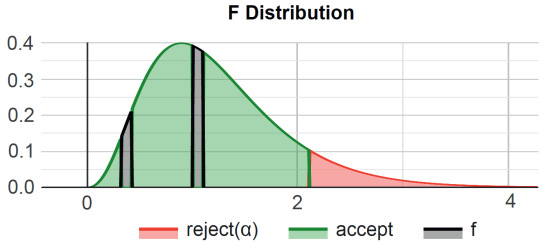

Figure 4: The left F-statistic is for the 30% workload ANOVA, and the right statistic is for the 90% test, which determines whether the null hypothesis for the one-way ANOVA tests is accepted.

test, the null hypothesis is accepted, meaning all averages are statistically equal with P-value = 0.899 and F-statistic = 0.369. The second test also accepts the null hypothesis with P-value = 0.39 and F-statistic = 1.052, as shown in the Figure 4 (i.e., right F-statistic). However, the third test determines that the difference between the averages is statistically significant with P-value $\simeq 0$ and F-statistic = 68.33.

Given the results of the first two one-way ANOVA tests, we may conclude that the number of nodes in the network does not affect how many messages are sent to deploy an application on the network, which validates the scalability of DOCMA. However, this conclusion can only go as far as the network size for which experiments have been made. The third ANOVA test indicates that when resources are scarcer on the network nodes, more messages are sent to deploy an application.

7 Conclusions and Future work

This paper investigated the design of a decentralized orchestrator for containerized MSA applications, called DOCMA, whose design principles targeted high scalability and resiliency of the orchestrator. The proposed system provides resistance to node churn, self-healing of the services deployed, and network partition tolerance.

Experimental results and analysis of the scalability of DOCMA are presented. The experimental analysis revealed that for the network sizes analyzed, the number of messages sent to deploy an application did not vary, which proves DOCMA's ability to scale, at least to 7,000 nodes. Further experiments should be performed to quantitatively analyze the scalability of DOCMA past this number.

The experiments performed also showed that when resources are sparser (i.e., overall network workload at 90%), more messages are used to deploy an application. However, a more detailed study of this aspect would be beneficial for discovering the limitations of DOCMA in regard to finding resources, which is closely related to the search algorithm used to find those resources and the scheduling algorithm to make use of the resources once located. Therefore, considerations of various search and scheduling algorithms should be made. Moreover, in this paper, no attempt to optimize the search

algorithm used and other relevant parameters has been made, meaning optimization of DOCMA is another prospect for future work.

Finally, a PoC implementation of the outlined design of DOCMA was performed. The orchestrator presented would benefit from including a monitoring solution that fits the system architecture. Thus, a variation of CoMA, the monitoring system proposed in [5], could eventually be employed.

References

[1] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, Sep. 2018.

[2] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2016-2021, 2016. [Online; accessed 05-February-2019].

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[4] Gartner. Leading the IoT, 2017. [Online; accessed 05-February-2019].

[5] Lara Jiménez, Miguel Gomez, Olov Schelén, Johan Kristiansson, Kåre Synnes, and Christer Åhlund. Coma: Resource monitoring of docker containers. In *Proceedings of the 5th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pages 145–154. INSTICC, 2015.

[6] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[7] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[8] Platform9. Container Management: Kubernetes vs Docker Swarm, Mesos+Marathon and Amazon ECS. [Online; accessed 05-February-2019].

[9] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

Paper C

HYDRA: Decentralized Location-aware Orchestration of Containerized Applications

Authors:
Lara Lorna Jiménez and Olov Schelén

Reformatted version of paper submitted to:
IEEE Journal of Transactions on Cloud Computing, 2020.

HYDRA: Decentralized Location-aware Orchestration of Containerized Applications

Lara Lorna Jiménez and Olov Schelén

Abstract

The edge computing paradigm, spurred by the Internet-of-Things, poses new requirements and challenges for distributed application deployment. There is a need for an orchestrator design that leverages characteristics that enable this new paradigm.

We present HYDRA, a decentralized and distributed orchestrator for containerized microservice applications. This orchestrator focuses on scalability and resiliency to enable the global manageability of cloud and edge environments. It can manage heterogeneous resources across geographical locations and provide robust application control. Further, HYDRA enables the location-aware deployment of microservice applications via containerization such that different services of an application may be deployed to different computing locations according to the needs common of edge applications.

In this paper, the experiments run show the orchestrator scaling to 20 000 nodes and simultaneously deploying 30 000 applications. Further, empirical results show location-aware application deployment does not hinder performance of HYDRA, and the random resource search algorithm currently being employed, may be used as a baseline algorithm to find resources in this fully decentralized orchestrator. Therefore, we conclude that HYDRA is a viable orchestrator design for the new computing paradigm.

1 Introduction

The explosion of the connected "things" brought forth by the Internet-of-Things (IoT) and cyber-physical systems (CPS) is changing the way we live and work. In 2017, there were 18 billion networked devices and connections. According to Cisco Visual Networking Index [6] this figure shall grow to 28.5 billion by 2022. This means that, worldwide, there shall be 3.6 networked devices and connections per capita. Just over half of those devices and connections are estimated to be Machine-to-Machine (M2M).

These devices are used in connected industries such as connected cars, cities, energy, health, homes, work, supply chains, and retail. They generate significant amounts of data. By 2022 it is estimated that they will be responsible for 25 exabytes of data per month [6].

These data are fed to applications that process them to provide some intelligent response or analysis. These applications have been running at cloud-enabled data centers

(DC). While cloud computing (CC) has established benefits by being successful at running services while minimizing costs and maximizing the utilization of resources [3], some issues have been encountered when it comes to the world of connected devices. If all the data these devices generate is sent across the network, this could produce unnecessary congestion [5]. This would result in a larger carbon footprint as data transfer volumes increase. Therefore, we should also attempt to mitigate unnecessary data transfer with energy-efficiency in mind. Further, some use cases have real-time constraints that the cloud cannot meet given the physical distance between the DCs and the devices. In other words, there is too much round-trip delay.

This has spearheaded a paradigm that has now taken root - edge computing (EC) or fog computing [2,10,27,28]. EC is complementary to CC, its core mission is to move some intelligence to the edge of the network [17] and keep data close to its source [8]. There is no single definition of "edge". It is dependant on terminology and also on the application or even on a given application function. For example, the edge could be a mobile [12] or stationary device, a local gateway, or even a micro-DC at a base station [27].

This means that applications that wish to successfully leverage the cloud-edge paradigm must evolve to be geographically distributed. More importantly, for us in this paper, this paradigm exposes the necessity of a flexible way of managing a geographically dispersed resource infrastructure on which to deploy services [22], such that global manageability of cloud environments may be resolved [12]. These prerequisites have driven us to propose an innovative orchestrator, ***HYDRA***, which is a fully decentralized and distributed orchestrator for the deployment of containerized microservice applications.

Application development has experienced a push towards continuous integration and continuous delivery or deployment (CI/CD). This has led to the emergence of the Microservice Architecture pattern [20], which designs an application as a group of loosely coupled distributed services [11]. Applications that leverage the cloud-edge computing paradigm must also operate across geographical locations, which fits well with the Microservice Architecture pattern.

Virtualization is one of the cornerstones of cloud computing as it is an essential aspect of maximizing resource efficiency. The defacto virtualization is virtual machines (VMs), though there has been a shift towards containerization [21]. This is a lightweight virtualization that has opened up a more flexible and resource efficient way of maximizing resource utilization, while simultaneously easing deployment complexity. Process containerization is a way of sand-boxing one or more processes and its dependencies. Multiple sand-boxes or containers, may operate on the same HW. They all share the same underlying OS, but remain isolated from other on-board containers through namespaces and cgroups [4]. Containerization platforms, such as Docker, allow these to be packaged as images which can be deployed as containers to execute the sand-boxed processes on any HW where the platform is present [14].

Containerization matches the shipping demands of microservice applications. Each containerized service is packaged as a self-contained image, allowing for independent development, testing and deployment of the services of an application. This eases and expedites application deployment and development. Given their characteristics, microservices

and containers work harmoniously and add value to one another.

In the cloud-edge paradigm, resources are distributed not only across DCs but also at the edge [10]. Therefore, there are large amounts of geo-distributed and likely heterogeneous resources. There is a need for an orchestrator that can efficiently operate under the conditions of the resources (i.e., large-scale, geo-dispersion, and heterogeneity) and meet the deployment and control demands (i.e., geo-distribution, load-scaling, geo-scaling, replication, and robustness) of applications for cloud-edge scenarios, which we consider to be containerized microservice applications for the previously detailed reasons. In essence, we adopt a dynamic partitioning scheme, where nodes operate independently to provide scalability, while control on a per application basis self-organizes a structure to enable robustness of the system [12].

The orchestrator presented in this paper is suitable in various contexts where an infrastructure needs to be managed at scale to deploy containerized applications, which could benefit different industries and verticals. For example, a cloud provider could use HYDRA within the context of a Platform-as-a-Service to deploy and manage their infrastructure and clients' containerized applications, either in a traditional cloud scenario (i.e., DCs) or the future cloud-edge setting (i.e., DCs, edge clouds and even edge devices). Much like cloud providers, telecommunications operators moving into the Multi-Access Edge Cloud (MEC) [25] could employ this orchestrator to manage their cloud-edge infrastructure and deploy applications across edge clouds. Moreover, industries that have resources across multiple clouds (i.e., public or private, at DCs or EDCs, on premises or not) could employ HYDRA to orchestrate applications over their whole available infrastructure.

In this paper, we present a Proof-of-Concept (PoC) orchestrator that has been devised to fulfill the listed concerns. HYDRA is a decentralized orchestrator for containerized microservice applications that can be location-aware. This orchestrator builds a peer-to-peer (P2P) overlay network of nodes, where each node is both orchestrator and resource. A node may find any node on the network based on its identifier, which may or may not be related to its location, dependant upon configuration. Furthermore, any node may act as the orchestrator, so any node is able to deploy an application, meaning the node has the ability to find nodes on the network with the resources and conditions necessary to deploy that application's services. Management of an application deployed in this decentralized orchestrator is enabled via roles, which determines the node's behavior in respect to that application. These roles are taken up by a dynamic set of nodes on the network as determined by HYDRA's design, which favors HYDRA's adaptability to changes in the overlay network. This supports the deployment of containerized applications to this decentralized orchestrator according to the requested needs of the application.

This design enables HYDRA to tackle its primary concerns which are scalability, availability, resource efficiency, capacity of geo-distribution, and resiliency. Thus, we posed the following research questions to analyze some of the aspects of HYDRA's design:

- **Orchestrator scalability:** We would like to explore HYDRA's ability to scale. Can HYDRA scale up to 20 000 nodes?

- **Application deployment scalability:** Does HYDRA manage to deploy a large number of applications in parallel? Is there a statistical significance in HYDRA's capacity to simultaneously deploy applications depending on the scale of the HYDRA network?
- **Search for resources:** Is the random search algorithm based on IDs proposed in this paper suitable as a baseline resource search algorithm for HYDRA? How successful is this search algorithm at finding available resources on the network to deploy applications?
- **Performance of location awareness:** How does HYDRA perform when operating as a location-agnostic versus a location-aware orchestrator?
- **Orchestrator resiliency to a network partition:** How is HYDRA's performance affected by a network partition when there is location awareness?

2 HYDRA System Design

This section presents a bird's eye view of HYDRA to provide a general understanding of some of the functions designed into this fully decentralized orchestrator. Some of these are covered in depth in this paper while others are succinctly presented here.

As an application orchestrator, HYDRA manages resources and applications. The orchestrator may be resource location-aware or location-agnostic, which in turn determines whether applications may be deployed with location awareness or not. This is enabled by the following functionality:

2.1 ID-based Identifier Design

The nodes and applications the orchestrator manages are identified by IDs. An ID is a number of bits. In some cases these IDs must allow for location awareness. In those cases, an ID-based design allowing the successful mapping of IDs to geographical locations is needed to deploy location-aware applications. This is further covered in Section 3.

2.2 Node Discovery

HYDRA is fully decentralized so that any node can find any other node on the network without any central system providing that information. Therefore, all nodes work autonomously. The node discovery mechanism employed by the presented orchestrator is based on Kademlia's [19] distributed hash table and node lookup algorithm.

This translates into each node having its own routing table, which it fills and updates as it comes across other nodes on the network. A routing table is separated into buckets, one bucket per ID bit. This determines in which buckets nodes get stored. The bucket size is configurable (e.g., 20 nodes) which determines the maximum number of nodes

each bucket can hold. This allows every node to have a partial and updated view of the HYDRA network.

Kademlia's node lookup algorithm allows a node to discover the logically closest nodes to a given ID in terms of XOR distance. This is done through an iterative process that queries other nodes for the nodes they know of that are closest to the target ID. This is done until the closest nodes to the target ID are found.

Node discovery enables any ID, which may refer to a node or an application, on the network to be found.

2.3 Application Management

Application definition: Certain application information is necessary to deploy an applications' services, which is provided by the application owner upon deployment. This information is explained in Section 4.

Orchestration roles: A series of roles designed for HYDRA are key to the deployment and control of applications in this decentralized orchestrator. For each application, these roles are dynamically adopted by the corresponding nodes on the orchestrator network as explained in Section 5.

Application control types: To facilitate application control in a location-aware or location-agnostic orchestrator, different application control types are necessary. One of the possible designs for application control is covered in Section 6.

Service Scheduling: A scheduling algorithm, which is mentioned in Section 8, determines how the services a node is attempting to deploy are placed according to the resources found by the resource search algorithm.

Service recovery and replication: The roles managing an application, supervise the application's services, performing service failure prevention and recovery through service replication. These concepts and their design in HYDRA are further explained in Sections 9 and 10.

Service discovery: An orchestrator does not necessarily provide a mechanism for service discovery, thereby leaving it up to the application developer to execute its own. However, it is preferable to natively include this functionality in the orchestrator given that it already monitors the deployed services. This native service discovery design is not the focus of this paper. The initial design follows the patterns of third-party registration and server-side discovery.

2.4 Search for Resources

To deploy the services of an application, the deploying node must be able to find the resources those services require. This is performed via a resource search algorithm. In this paper, we present two search algorithms in Section 8 that allow the orchestrator to maintain full decentralization.

2.5 Distributed Consensus

The roles controlling a particular application operate as a group. That is, multiple nodes take on a particular role to provide redundancy, thus ensuring control of that application. To maintain the group, a distributed consensus algorithm is necessary to replicate data across these nodes and perform leader election when necessary. This aspect of the orchestrator is not crucial to the performance evaluation of HYDRA addressed in this paper. Therefore, this topic is out-of-scope for this paper.

3 Location-aware Nodes

HYDRA builds a flat P2P network for the orchestration of containerized microservice applications. The orchestrator is composed of all the nodes on its network and there is no centralized control. Each node acts independently, where each one may simultaneously manage resources (i.e., nodes) and applications. Moreover, network reachibility is a prerequisite among the participating nodes in HYDRA.

A *node* is defined as being HW, either virtualized or not, with the ability to execute containerized services. Further, a resource is identified by two IDs: the node's identity and the node's location. Both IDs are on the same address space, defined by a number of bits (e.g., 256 bits). The *node ID* (***NID***), or identity ID, is a location-agnostic ID. The *node location ID* (***LID***), is location-aware, thus it is linked to the physical location of the resource.

Nodes can find each other by means of either the location-agnostic IDs or the location-aware IDs and a search algorithm. This means, a node can always find another node on the network based on who they are and based on where they are. This is a method of dual discovery, which is useful in a series of scenarios, such as in targeting specific resources for application deployment or in allowing the existence of mobile resources on the network. The NID assigned to a node is unique and immutable. However, the LID may be updated throughout the life of the resource on the network if its geographical location changes.

When HYDRA is deployed as a location-aware orchestrator, both of these IDs are necessary. This allows applications that are launched to the orchestrator to run as location-aware or location-agnostic applications. However, if HYDRA is deployed as a location-agnostic orchestrator, only the NID is employed. In this case, the services of the deployed applications may run anywhere on the network. No control may be enforced to deploy the application's services to specific areas of the network. This scenario may be useful when the use case is that of a cluster on which services are deployed or when service location is irrelevant.

When a HYDRA network is deployed, the nodes may either randomly generate their own NID or the NIDs may be provided by an external system. To randomly create a NID, the address space must be large enough so that the probability of collision becomes insignificant, as NIDs must be unique in the network. The LIDs must be provided by a system mapping geographical location to an ID design strategy.

In this paper, we present a location awareness scheme that links LIDs to their physical location on the network. Thus, when a HYDRA node looks for resources to deploy services it may do so according to where that service should run at via the LIDs, while keeping the system fully decentralized. This is particularly relevant for edge computing applications, as location-aware service deployment is necessary.

The LIDs are generated following a hierarchical pattern somewhat similar to the IP-addressing design, where a number of bits identify different top-layer (e.g., layer-1) areas of a physical network. Each of those top-layer areas may be further broken down into smaller network areas via a number of addressing bits following the layer-1 bits. Each new layer may be separated into more layers. The number of bits left when the lowest layer is reached are the number of bits that can address the nodes in that quadrant of the defined network. The limitation is set by the size of the ID address space, together with the number of layers that we wish to define and the number of nodes that must be addressed.

An example of the location-aware ID design may be one where 6 layer-1 network areas are defined. We will refer to these as *regions*. Thus, layer-1 is identified by 3 bits. These regions could be representing nodes present in North America, South America, Europe, Asia, Africa, and Australia. Assuming region 1 (i.e., North America) is separated into 3 areas (e.g., US, Mexico, Canada), layer-1_1's layer-2 can be addressed by 2 bits. Layer-$1_1.2_1$ (i.e., North America-US) contains 5 data centers and 5000 edge data centers, which can be addressed by 13 bits. Each central data center and edge data center could further address smaller areas within each of their networks (e.g., data halls), though that is not done in this example. Thus, if the ID address space is defined by 160 bits, for layer-$1_1.2_1.3_1$ (i.e., North America-US-DC1) a maximum of approximately 5.575 x 10^{42} nodes may be addressed.

Moreover, in the location-aware ID design phase, the control networks must be established. We define a *control network* to be the highest layer in the design which is a low-latency network. Thus, in the location-aware ID design, the layers addressing a low-latency network (e.g., a DC) should be tagged. This is required knowledge for an effective application deployment. In the previous example, layer-$1_1.2_1$ would be tagged with 5005 control networks, the 5 DCs and 5000 edge DCs, represented by their respective mascaras.

This schema allows the design of the network to be coarse- or fine-grained depending on the awareness that we wish to infuse to the HYDRA network. All nodes on the HYDRA network know of this overall ID scheme, as it is used to correctly deploy applications. The design should be with forethought so that it encompasses future changes to the orchestrator network. If the ID-based location design needs redesigning after HYDRA is operating, that would potentially require a translator mechanism from the old design to the new one. However, this is not the focus of this paper.

4 Applications

In HYDRA an application is defined as a series of loosely coupled services, whose deployment may be location-aware or location-agnostic. An application is uniquely identified by its location-agnostic root ID, which is on the same address space as the NIDs. The *application root ID* is created by hashing the application deployment data and a generated private key, when an application is first submitted for deployment to the orchestrator. This ID is employed to deploy the application, modify or remove it, maintain its service replicas running and retrieve information about the application's state.

The *application deployment data* defines an application. That is, it establishes the number of services it consists of, the images and repositories for those services, the resource usage requirements, the number of replicas per service and the type of replicas, the information about which services need to communicate, and the affinity and anti-affinity constraints between the services. Further, there is also *application configuration data* which lists how the application services have been deployed, and service liveness data that details the state of the application services. All this aggregated information is the *application metadata*.

5 Roles for Decentralized Orchestration

Each node on the HYDRA network can take on any of the four roles defined, for any number of applications. These roles enable the orchestration of applications:

5.1 The Entry Role

Given that HYDRA builds a fully decentralized orchestrator, any node on the network is able to take requests to deploy, remove or modify applications. When one of those requests is received by a node on the network, the node takes the *entry* role for that application request. This role is dropped once the request is served. When a node becomes an entry node for an application, it looks for and relays the request to the corresponding nodes that are in charge of that application within the network.

5.2 The Controller Roles: Root and Leaf

There are two types of *controller* roles: root and leaf. A node may become a controller for an entire application (i.e., *root controller*) or for a set of services of an application (i.e., *leaf controller*). The function of the controllers is to supervise an application, be it in part or as a whole.

β and θ are replication factors that determine the root and leaf controller set size, respectively. These, along with the number of leaf controller sets, are determined on a per-application basis. Moreover, an application might not have leaf controllers depending on the application control type employed. Leaf controllers are for location-aware control, while root controllers are location-agnostic.

The *root IDs* and *leaf IDs* of an application determine, at deployment, which nodes on the network acquire the role of root or leaf controller, respectively. The β nodes, whose NIDs are closest to the application root ID adopt the root controller role for the application. This is an application-wide role, so there are only β nodes on the HYDRA network with this role for a given application. Moreover, leaf controllers are the θ nodes whose LIDs are closest to a leaf ID. However, there may be multiple leaf controller sets for an application. This is further explained in Section 6.

A node that adopts a controller role, has that role in one of two states: *active leader* state or *passive follower* state. In a controller set, all nodes are passive controllers except one, which is the leader.

The leader controller commits the relevant controller set data to the follower controllers to maintain the data replicates consistent across the controller set. Moreover, the leader maintains the controller set size constant by substituting unavailable controller nodes with new ones, thus preserving the controller set as nodes come and go from the orchestrator network. Further, the passive controllers ensure the existence of one leader within the controller set. This is done through a leader election via a consensus algorithm. The node in the controller set whose ID is closest to the corresponding root or leaf ID, is the one that acquires the leader state when a leader election takes place.

5.3 The Service Host Role

Any node can host services for different applications. Each node answers queries about the local resource availability to other orchestration nodes that are attempting to deploy an application's services. It also locally deploys and brings down the services that it has agreed to host. Once, one or more services of an application are deployed at a node, the node has acquired the *service host* role for those applications' services. This node will monitor and sends heartbeats to the corresponding leader controller for each of those services, providing service liveness. If a given leader is not reachable, it obtains the new leader from that controller set.

5.4 The Roles in Concert

When an application is deployed, different nodes will take on these roles for that application. The entry role at a node will exist temporarily, while the deployment request is being performed. The controller roles will be taken up by different nodes on the network. Control of an application is dynamic as there is no fixed set of nodes to manage the application, but rather a fluid group of nodes, dynamically determined by the corresponding control ID. Moreover, depending on the number of services, their replicas, scale-out demands, and other considerations, the number of nodes that will take the service host role for an application may fluctuate. Therefore, throughout the life-cycle of the application, which nodes adopt these roles may vary as nodes join or leave the network, or as application requirements change.

As mentioned previously, a node can assume any of these roles for multiple applications. For example, a node may be the leader root controller for an application, a passive

leaf controller for another application and the service host for four service replicas, each of a different application.

6 Location-aware Application Management

As the orchestrator presented in this paper is decentralized, a method that allows the management and deployment of containerized applications to the orchestrator is vital. In this section, HYDRA's design for location-aware and location-agnostic application management is presented. However, the focus of the design presented in this paper is on location-aware microservice applications.

There are various strategies to manage location-aware application deployment in the orchestrator. The advantages and disadvantages of each depend on the use case. Here, we showcase the decentralized application management strategy. It is the most scalable and one of the more complex strategies. Further, it follows similar principles to HYDRA's resource management and discovery strategies.

In the decentralized application management strategy, there is no centralized point from which all applications are deployed and managed. Rather, each application is supervised independently. This provides scalability and dynamicity to the orchestrator.

We define two types of application control within the presented management strategy. These are graphically explained in Figure 1, where the application example is composed of 9 services.

6.1 Type 1 Application Control: Flat

Type 1 application control is best suited for a location-agnostic HYDRA orchestrator because resource location is unknown and therefore, an application's services are also deployed without relation to location.

As presented in Figure 1, on the left, there is a single management level for each application, represented by a set of root controller nodes. All services of the application are managed directly by them. Root controllers deploy, monitor, and modify the services of an application. These ensure failure recovery, auto-scaling, and service discovery for all the application's services. Further, the controller set performs replication of data and leader election when necessary, to ensure constant management of the application.

6.2 Type 2 Application Control: Layered

This type of application control is suitable for a location-aware HYDRA where an application's location-aware services are expected to run across various areas of the network.

There are two control levels: root and leaf. Root controllers serve each application as global control, but with limited responsibilities. Leaf controllers provide autonomous supervision of the set of services of the application that fall within their purview.

Figure 1 shows, on the right, an example of *type 2 application control*. In this example, three regions (i.e., layer-1's) are visible, though there may be more defined in the

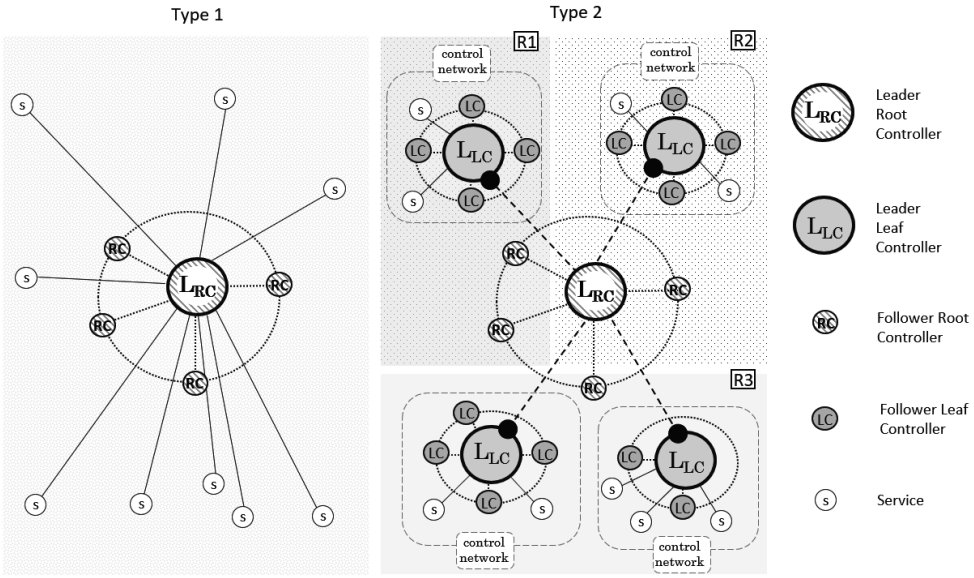

Figure 1: This shows an example of the control of an application with each of the presented application control types. The application is composed of 9 services. In type 1 (on the left), the HYDRA orchestrator is not location-aware and therefore, neither is the placement of services. There is a single management layer, performed by the root controllers. In type 2 (on the right), HYDRA is location-aware and so is the application deployment. Global management is carried out by the root controllers and local management is performed by the leaf controller sets.

ID-based location-aware HYDRA design. However, the figure presents the ones relevant to this particular application. This applies to the control networks as well. The application deployment presented is location-aware, and performed as per this application's definition. The application is composed of 9 services. Two services fall within a control network in Region 1 (R1), another two in a control network within Region 2 (R2), and within Region 3 (R3), two services are within one control network and another three in another control network. There is a leaf controller set per control network where there are services of the application and a global root controller set.

The logic of root controllers varies according to the application control type. In type 2 application control, the root controllers' objective is to maintain consistency of the application definition among leaf controller sets, and activate or remove leaf controller sets if the application is modified or its requirements vary. Therefore, the initial application deployment, removing, or including new service definitions is carried out by the root controllers of that application. Moreover, requests about the application state and its services can be made to these controllers, as they can easily aggregate these data. Further, root controllers may also monitor and provide failure recovery for cases of consensus

failure for leaf controllers (i.e., if it were to happen that too many nodes of the set fail simultaneously and consensus cannot be reached.).

There can be one or more sets of leaf controllers for an application. Leaf controllers track the application's services which operate within their network domain. Therefore, for the application services they handle, they provision resources, deploy and remove services, maintain service replicas, track service liveness and perform auto-scaling when necessary. Moreover, leaf controllers implement service discovery for their local services and coordinate with relevant leaf controllers to provide service discovery of remote application services. Each leaf controller set operates independently from other leaf controller sets and the root controllers.

Employing leaf controllers at each control network where there are services of the application, allows quick control of the services for auto-scaling, redeployment, or service discovery as the controllers and the controlled services are on the same low-latency network.

6.3 Application Root ID and Leaf IDs

The application root ID defines which nodes acquire the root controller role for an application. This ID is static during the application life-cycle. This grants the root ID the ability to be the application anchor, which allows that application to be found independently of where its services are running. Further, the nodes that take the role of root controller for an application are the nodes whose NIDs (i.e., identity and not location IDs) are closest to the root ID. This allows the nodes that take the root controller role for an application to be on different network areas, as the NID is location-agnostic.

The leaf ID determines which nodes adopt the leaf controller role for that leaf controller set. The number of leaf controller sets of an application may vary, some may become disposable or new ones may become necessarily. This is determined by where application services are set to execute. Thus, application control adapts to the changes in service location autonomously. Therefore, leaf controllers are directly linked to location.

Each leaf ID is generated by appending to the prefix of the corresponding control network, the missing bits taken from the application ID. Hence ensuring the nodes that hold the leaf controller role, in that control network, for that application, are always within the same low-latency network as the nodes that host the supervised services.

The information needed to obtain the leaf IDs of an application is readily available to those nodes that may require it, such as the application service host nodes, the application root controllers, or other application leaf controllers. Moreover, it is worth noting that while only a dynamic subset of nodes on the orchestrator are the ones managing and hosting each application, any node on the HYDRA network can find the management nodes for said application, provided they have the information required to do so.

7 Application Deployment

A comprehensive sequence diagram depicting the process of a successful location-aware application deployment is presented in Figure 2.

A *HYDRA client* is external to the orchestrator network. This client performs requests to any node on the HYDRA network to deploy, remove, modify or obtain the status of an application.

The user wanting to deploy an application, provides the application definition to a HYDRA client. These data determine how the application services are deployed on the orchestrator network, and how the application is managed.

A HYDRA client deploys an application by submitting a request to the orchestrator, which is achieved by submitting it to any node on the network. The node that receives this request becomes an entry node for that deployment request. When sending a deployment request, the client locally generates a public-private key pair and an application ID for the submitted application request. This will be the application ID in HYDRA throughout the whole life-cycle of the application. The private key is only accessible to the application owner at the client that locally generated it. The HYDRA client forwards the deployment request along with the public key and application ID to the entry node. Public-key cryptography is employed so hence forth, requests relating to an application are signed by the application owner and the response is encrypted, making it only readable by a user possessing the appropriate private key.

The entry node performs a search for the β nodes whose NID is closest to the root ID of that application. These β nodes are found through the node lookup algorithm mentioned in Section 2 and explained in [19]. The entry node sets up the root controller role at each of these nodes. The leader among them is the node whose NID is closest in terms of XOR distance to the root ID. The leader root controller generates the leaf IDs pertinent to the application as per the application definition. It then performs a search for the θ nodes whose LIDs are closest to each leaf ID and activates the leaf controller role at each of those nodes for the respective leaf ID. The leader of each leaf controller group is in charge of deploying the services of the application that fall within its control network.

A leader leaf controller, to deploy the services it supervises, first looks throughout its managed network for viable nodes that could host those application-defined service replicas, via scheduling and resource search algorithms. This process is explained in Section 8.

Each leaf controller set notifies the root controller set of their success, or lack thereof, in deploying the services they regulate. Thus, failure to deploy part of the application may occur. Eventually, the entry node receives a deployment confirmation or failure for the application along with deployment information if it was successful. If a partial deployment failure occurred, the services successfully deployed and the controller roles assigned are brought down.

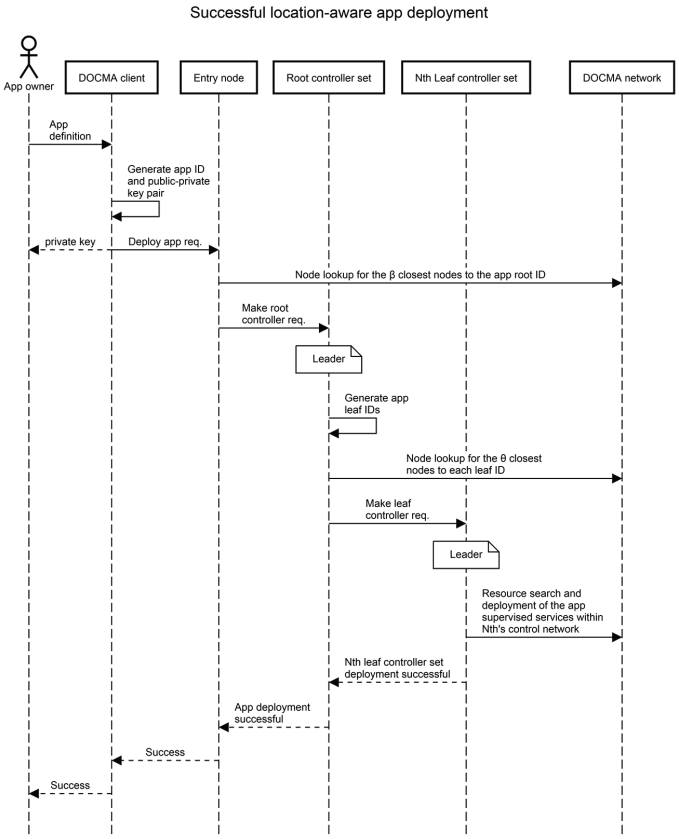

Figure 2: This sequence diagram illustrates the process of successfully carrying out a location-aware application deployment in HYDRA. Thus, type 2 application control is employed. All responses in this process are not shown for brevity. It is understood that an application has 1 or more leaf controller sets. This figure only shows the Nth leaf controller set to represent all the leaf controller sets running that same sequence in parallel.

8 Location-aware Search for Resources

The node in charge of deploying one or more services of an application, does so by means of a resource search algorithm and a scheduling algorithm. In this section, we focus on the resource search algorithm. We do not explore scheduling algorithms in this paper. The resource search algorithm looks for viable node candidates throughout the network. The scheduling algorithm determines how services are placed once those viable nodes are found. Viability is established in terms of resource availability on a node in comparison to the resources required by the application's service or services.

The proposed system is fully decentralized, therefore, there is no global replication of where resources are available, and no optimal scheduling can be pursued if scalability is to be maintained. Moreover, availability of resources can be highly volatile which further encumbers finding a solution. The sought after resources are represented by a resource vector (e.g., RAM, storage, network and CPU), depicting the required resources to execute the containerized service. Searching across the network for a vector of resources is achievable in HYDRA as the aim is to present an approximate solution with good performance, given that it is an NP-complete problem.

While node discovery is done in HYDRA through Kademlia's node lookup algorithm based on a structured P2P network of IDs, searching for available resources across the whole network is more challenging. This search can be likened to seeking resources on an unstructured P2P network, since our P2P network is structured around NIDs and LIDs, which bear no relation to availability of resources. However, the structured P2P network used for node discovery, together with HYDRA's resource search algorithm could potentially be used to help ensure a search for resources across the whole network.

In this paper we explore a baseline resource search algorithm, where the prerequisite is that in terms of discovering resource availability, all nodes are independent. Thus, we define three properties which we require of a resource search algorithm. The first property is that it must maximize the probability that the nodes queried about their resources during a given search are all different nodes. This provides a greater likelihood of finding a node that can host the service for which the search is taking place. The second property is to ensure load is evenly spread across the HYDRA network. This requires a search algorithm that does not consistently query an area of the network over the rest. The third property is that the algorithm should leverage the existing system properties, meaning we aim to make use of the ID structure present in HYDRA to search for resources.

We propose employing one of the following methods as an algorithm to search for resources across the network that meet the previously stated requirements: 1) Random ID 2) Maximized XOR distance between queried IDs. Both algorithms share two variables: γ and δ. γ is the number of parallel queries that are sent out during a search round. δ represents the search depth, that is, how many search rounds are done before the algorithm quits searching if resources have not been found by then.

This paper focuses on location-aware application deployment, therefore, when mentioning the node that is carrying out the deployment, we are referring to a leader leaf controller. This node would look for the resources to deploy one or more services according to the type of scheduling used, and the affinity and anti-affinity constraints of those services. Further, reservation of resources during a search, may or not be carried out. For the purpose of simplifying the explanation of the resource search process, we assume that the node is looking to deploy a single location-aware service replica and no resource reservation is done.

8.1 Random ID

The first algorithm begins by having the node performing the search generate a random ID within the target network area where we are looking to deploy the service. Then, the node performs a node lookup for the γ nodes whose LID is closest to the generated random ID. These nodes are asynchronously queried about the resources needed by the service. The node that is able to host the service responds accordingly. In this case, since there is a single service to deploy, the deploying node issues a service deployment request to the first node to respond positively to the resource query. The process of generating a new random ID and querying the γ closest nodes for resources would be repeated until a candidate node successfully hosted the service or until δ subsequent rounds of resource queries were not to yield viable candidates, in which case deployment for that service replica would fail.

8.2 Maximized XOR Distance between Queried IDs

This algorithm focuses on maximizing the probability that new nodes are found in each round of node discovery during the resource search process. Unlike the first algorithm which generates a random ID within the corresponding network location for each round of resource queries, this algorithm generates a random ID only for the first round. Assuming there is a need for subsequent rounds, the IDs employed to lookup nodes on the network, are not randomly generated but maximally distant in terms of XOR distance to the previously generated IDs in this resource search process, though always within the service's designated network location. For example, in a 4-bit address space, given a service that should run within the 1XXX network area, the first round could randomly generate an ID e.g., 1000, and query about resource availability to the nodes closest to that ID. The second round would employ the ID: 1111 to do the lookup for nodes to query about resources. The third and fourth rounds would yield ID:1100 and ID:1011.

Each of these resource search algorithms has different advantages and disadvantages that affect system performance. In this paper, we analyze quantitatively the performance of the random ID resource search algorithm, to determine whether it is a viable baseline algorithm to search for resources while keeping this orchestrator fully decentralized.

9 Replication of Services

A mechanism for failure prevention and recovery of applications deployed to the HYDRA orchestrator, apart from the previously presented orchestrator characteristics, is provided through service replication.

Nodes on the network may crash or become unavailable for a number of reasons. However, in order to provide high availability of services, these must be replicated so there can be fail-over. This enables a given service to be available, with no downtime when load balancing is done over these replicas. In the current design of the orchestrator, replicas are of two types: *live replicas* and *passive replicas*.

Live replicas are useful for coherent deployment of solutions providing N-way active replication or N+M hot standby replication at application level. Notice that replication of running state between running replicas is not the responsibility of an orchestrator. When such replication is needed in stateful services, it is up to the application developer to define the data exchange across service replicas.

Passive replicas are cold standby replicas, meaning the replicas exist at a node when the service's image is stored there. This image is deployed and spun up into a live replica when one of the live replicas fails. A passive replica obviously has only configuration state, no running state. It is only once the passive replica turns into a live replica that running state could be provided to the replica. Again it is up to the application developer to enable such replication when needed. However, for a stateless service there is no need for that.

The benefits of supporting live and passive replicas as part of HYDRA is coherent application deployment of live replicas and that the orchestrator can keep the total number of replicas in an application quite constant by creating new ones on demand when some replicas crash. Note that passive replicas do not take up as many resources as live replicas, and containers are very quick to deploy. Depending on the use case, the service may be stateless, or it may be stateful with an expected response time that is not too demanding. In such cases, having passive replicas could be more resource efficient. Nevertheless, a service may be defined with both, a live and passive replication factor. Each of these replicas, be it live or passive, are stored on separate nodes.

It is HYDRA's responsibility to ensure that the desired number of replicas are up and discoverable on the network, with high availability. When a service replica is determined to be unreachable, then another replica must be set up to substitute it.

This is done through the process of failure recovery presented in Section 10. All service replicas are deployed as per the explanations in Section 7 and 8. Once deployed, the configuration information for the deployed service replica is provided to its corresponding controllers, which in turn make the information available to concerned parties via HYDRA's service discovery.

10 Failure Recovery

Nodes in the HYDRA network may crash or leave, or become unreachable due to network partitions or network delay. Given that high availability and resiliency are major concerns of this solution, measures need to be taken to ensure services continue operating in these eventualities. The system cannot differentiate between these problems so a mechanism that operates with the desired behavior independently of the reason for the problem is needed. There are different options to consider when establishing a method to recover from failure.

Firstly, failure should be prevented when a situation that leads to failure is known. A failure prevention mechanism for nodes to leave the network in a controlled fashion is necessary. The departing node deregisters each of its hosted services. Thus allowing the controllers to initiate failure recovery for each of those service replicas immediately.

Moreover, the departing node initiates a leader election for those applications for which it is a leader controller. This would limit the time needed for new leaders to take over, as well as prevent or minimize the time that those service replicas are down. Further, failure prevention is also done for the control of each application, through the root ID design. The root ID is location-agnostic, therefore, it is improbable that all the root controllers of a given application are within the same control network. Thus, the application root controllers are unlikely to simultaneously fail. This design consideration limits the possibility of leaving an application bereft of management.

Secondly, for those failure situations that can not be prevented, failure recovery mechanisms are required. These mechanisms are centered around service replicas and controller nodes.

10.1 Service Replicas

A service replica is registered by the service host node to the corresponding controller nodes. A heartbeat mechanism is put in place to maintain knowledge of the service replicas' state. Thus, a leader controller may determine a service replica to be unreachable, which results in the process explained in Section 8, to find a viable node to host a new instance of that service. If a passive replica fails, a new passive replica will be allocated. If a live replica fails, and there are passive replicas in place for this service, one of those is turned into a live replica, and the passive replica is then substituted. If the replica is a live replica but there are no passive replicas for this service, then a new replica is deployed to a node on the network. New service replicas become discoverable to other services via HYDRA's service discovery at the corresponding controller nodes.

10.2 Controllers

A controller, be it a root or leaf controller, may become unreachable. This is handled through the consensus algorithm that replicates the pertinent data among the controller set, maintains a constant number of controllers in the set, and ensures there is one controller in the leader state. Thus, a controller that fails or leaves the network is substituted by HYDRA, as a method of recovering from the failure of that node and of preventing failure for the controller set.

As all controllers of a leaf controller set operate under the same low-latency network. The whole leaf controller set may fail. In such a case, the root controller set of the application provides failure recovery by redeploying control and the corresponding services to another part of the network, if applicable by the application requirements.

11 Related Work

Kubernetes has become the primary container orchestration platform in industry for the deployment of cloud-native applications [9]. It is an open source project launched by Google which has built a large community following. Its main focus is on cluster

management for container deployment, addressing use cases of internet-scale companies as well as those of cloud-native developers of different scale requirements [9]. However, Kubernetes is a centralized orchestrator where a master keeps track of a cluster of nodes and deploys containerized services to the cluster. While Kubernetes' scalability in terms of node management capacity has evolved over time, it seems to top out at managing 5000 nodes [18] largely due to its centralized nature.

Many organizations have had to manage multiple Kubernetes clusters, be it for scalability reasons or due to requirements on geographical dispersion of resources. In an effort to provide a means of managing multi-cluster scenarios, Kubernetes Cluster Federation (KubeFed) V2 [16] is in the works. At the time of writing, KubeFed is in its alpha phase. KubeFed is also a centralized solution that designs a 2-tier cluster control. A control plane is set on top of the Kubernetes clusters, where one of the clusters acts as the uber-master of all the registered Kubernetes clusters. Applications are deployed to the managed clusters through this top-layer controller.

For edge computing scenarios, another Kubernetes project, KubeEdge [26] is being developed. This is also a centralized orchestration solution based on Kubernetes. The aim of this project is to enable a Kubernetes cluster that is capable of managing resources in the centralized cloud and at the edge, to deploy applications across both. KubeEdge's intent is to improve Kubernetes' ability to scale, though it is unclear to which degree. This project is also in the early stages and no benchmarks have been carried out. KubeEdge also addresses heavily demanded requirements of IoT applications: device management, device health checking, digital twin, message brokers for the common IoT protocols, etc. IoT platforms such as AWS Greengrass [1] and Azure IoT Edge have also provided solutions to these requirements. HYDRA's current design does not currently include these functionalities. However, for an edge computing scenario, it would be beneficial for HYDRA to ship these functionalities out-of-the-box, as it would greatly simplify the development of IoT applications deployed to it.

The focus of Kubernetes, KubeFed, and KubeEdge is on cluster orchestration, multi-cluster orchestration, and cluster with edge nodes orchestration, respectively. While all these approaches tackle different use cases, they are extensions of the popular centralized Kubernetes design. HYDRA aims to explore a design that would be applicable to any of these use cases, based on a self-organizing decentralized approach.

There is no comparison between HYDRA and the Kubertenes projects in terms of functionality, as the former is a PoC while the latter are industry solutions. Nevertheless, HYDRA's objective is to present and analyze an alternate orchestrator design, where the focus has been on geographical dispersion of resources, scalability, and resiliency requirements from the outset. This approach is opposed to that of an orchestrator for cluster management that is evolved to meet requirements of the distributed cloud and edge computing.

Much research focuses on the design of applications for edge computing, while minimizing the relevance of orchestration in such a scenario. The authors in [23] present an architecture for Industrial IoT applications based on fog computing and a containerized microservice-style architecture orchestrated through Docker. While the aim is that of

scalability and resiliency, the problem is solely tackled through a 3-tier application architecture design, where containerized services are executed at each layer: cloud (enterprise tier), gateway (mediation layer), and end-devices (sensing layer). Nevertheless, when analyzing deployment of such applications, particularly with scalability and resiliency in mind, two main considerations have to be made: orchestration and application design. The orchestration refers to how the infrastructure is managed and how applications are deployed and supervised. The application design is about how a particular application is architected. The latter concern is addressed in this paper, while the former is not.

A proper application architecture may allow scalability and resiliency of that application, but the underlying orchestrator must also enable it. However, in this solution the orchestration is centralized at the cloud layer where a Docker manager tracks resources across layers and deploys services. The centralized orchestration solution would not meet the requirements for edge applications on two fronts: scalability and management of edge nodes. If a single application is deployed to the proposed orchestrator solution and the number of resources, across all three layers, that need to be managed to meet the requirements of said application do not exceed around 2000 nodes, then scalability would not be a problem. If more resources need to be managed to run that particular application or more applications, then this solution would not scale. Typically, the expectation of IoT scenarios is that of thousands of end-devices, which would make this paper's orchestration setup unfeasible for those scenarios. Moreover, the proposed orchestration is centralized in the cloud, so resources and services would be controlled from there. In edge computing scenarios, nodes such as the end-devices, may have a limited bandwidth or an unreliable network, which would make orchestration more failure-prone, leading to less service availability, which is particularly critical in real-time applications. Therefore, resiliency would also be negatively affected.

When designing HYDRA, we have focused on the principle of separation of concerns. HYDRA expects applications to be submitted as a series of services, following the Microservice Architecture pattern. Nevertheless, application design is not HYDRA's concern. It is up to the application developers to decide how to architect their application to meet whichever requirements they deem relevant. HYDRA focuses on scalability and resiliency by decentralizing orchestration through a P2P overlay network. Further, on a per application basis, the orchestrator keeps control of the application services close to where those services run to minimize control latency. Moreover, HYDRA's design allows all the nodes to operate independently, meaning that if HYDRA is used in a use case that extends the cloud to the edge, edge nodes can operate even in an unreliable network.

12 Experimental Design

This section presents five experimental scenarios, examining five different variables: network size, number of simultaneously deployed applications, pre-existing network workload, location awareness, and loss. Each scenario generally examines two or three of these variables. Each experimental setup aims at showcasing HYDRA's behavior and performance under different situations. The results of these are presented and discussed

Table 1: Experimental Scenarios

	Network Sizes	# of Parallelly Deployed Apps	Preexisting Network Workload	Location Awareness	Loss (Netem)
A	5000 to 20 000; step of 5000	1	30% and 90%	No	No
B	5000 and 15 000	10, 30, 100, 300, 1000, 3000, 10 000, 30 000	70%	No	No
C	5000 to 20 000; step of 5000	Respective maximum	95%	No	No
D	15 000	30 000 (10 000 / region)	70%	Yes - 3 regions	No
E	15 000	30 000 (10 000 / region)	70%	Yes - 3 regions	1 region network partitioned

in Section 13. A summary of the experimental scenarios is represented in Table 1.

A *HYDRA node* runs as a group of processes, which can be seen, in a simplified way, as a server and a number of clients. A *host* is an Amazon EC2 instance which accommodates multiple HYDRA nodes. The EC2 instance type used varies, as does the number of HYDRA nodes an instance hosts. In a real-world use case of HYDRA, there would be one HYDRA node per host. However, in these experiments we are simulating a HYDRA network though only in terms of available resources (i.e., a HYDRA node is provided this information as a value in each experiment instead of taking it directly from the host it is running on). To carry out the experiments, the Experimenter is employed. The *Experimenter* is a program that has been implemented to set up an experimental scenario, run a series of experiments, and compute the results of those experiments. For a given experimental scenario there will be one or more hosts, each running multiple HYDRA nodes, as well as another VM instance hosting the Experimenter.

During a given experiment, the Experimenter deploys a number of HYDRA nodes to one or more hosts given the network size for that experiment. The nodes obtain their *configuration* from the Experimenter. For a given node A, this configuration includes: node A's ID, a bootstrap node, node A's HW (i.e., simulated instance type), node A's available resources at the experiment's outset, the convergence value, and a series of configuration information that will determine how the HYDRA node behaves when deploying applications.

When there is no location awareness, each node uses the NID. However, when there is location awareness, the LID is also employed. The NID is randomly generated. The LID is assigned based on the node's location and the ID-based location design, as explained in Section 3. In those experiments where there is location awareness, the location design is simple. Each host is awarded part of the network, meaning, we use one EC2 instance per region. Therefore, all the nodes running in the same host, are considered to be in the same network location. Thus, the LID of each node is obtained by randomly generating an ID as per the node's instance-defined location ID.

For the location-aware application deployment results carried out in this section, each application (i.e., all its services) were deployed to a single control network. Therefore, in the location-aware experiments, there is a single leaf ID per application. Further, we forwent the creation of the root controllers for all applications. The leader leaf controller, for the only leaf controller set of each application, was the only controller employed when deploying each application.

The *bootstrap node* is any node on the network. A node joining the orchestrator network must know of at least one other node on that network to be able to join. A new node on the network will bootstrap to the one node it knows of (i.e., the bootstrap node), which results in it learning of other nodes on the network and other nodes on the network learning of it. For the experiments presented in this paper, the mechanism for selecting the bootstrap node for each node has been through random selection. This means that when a node gets its configuration information from the Experimenter, the bootstrap node it obtains is randomly selected from the nodes that have already obtained their configuration.

Prior to beginning an experiment, the Experimenter generates: all the NIDs and LIDs for that particular experiment, the simulated instance type for each node, and how many of these resources it has at its disposal at the beginning of the experiment.

The simulated instance types defined during these experiments only take into account RAM memory as a resource. While this definition can be expanded upon to include more resources of relevance such as CPU, disk or network bandwidth, this has not been done for the experiments presented in this paper. However, all experiments have been defined on a heterogeneous network. Thus, two simulated HW definitions (i.e., simulated instance types) have been characterized resembling Amazon EC2's instance types: m5.4xlarge (64GiB of RAM) and m5.large (8GiB of RAM). For all the experiments run, 70% of the network is of type m5.4xlarge while 30% is of type m5.large. The simulated HW assigned to each HYDRA node is randomly appointed, even when location awareness in an experiment is active. These are general purpose instances, which is why they were chosen.

The amount of available resources at each node is generated by the Experimenter depending on the designated *preexisting network workload* for each experiment. For example, when the preexisting network workload variable is set to 80%, this means that at the beginning of that experiment, on average, the HYDRA network will have 20% of its resources available to deploy services. The available resources at each node are created through a normal distribution so that the simulated available resources for any node do not exceed the resources of that node's assigned instance type (e.g., m5.4xlarge) while maintaining the corresponding overall network workload average.

The *convergence* value is defined as the minimum number of entries on a HYDRA node's routing table that should enable the node to deploy applications. The behavior of having the nodes have a deep knowledge of the network prior to deploying the applications has not been studied. We have opted for a lenient requirement of 3 entries, for all experiments, when it comes to the convergence value.

Once all the nodes on the network have obtained their respective configurations, the network is left to its normal, decentralized behavior. Each node will bootstrap to their bootstrap node. The *bootstrap process* consists of the joining node performing a node lookup for its own NID and LID, where the bootstrap node is the first node that gets queried for the closest nodes it knows of to that ID. The bootstrap process is considered to be successful if the node lookup returns at least three other nodes on the network. A joining node will attempt to successfully bootstrap 100 times before quitting. However, all nodes should successfully bootstrap on their first attempt, unless the requests used during a bootstrap process are continuously timing out (e.g., extreme packet loss).

Concurrently with the bootstrapping process, each HYDRA node monitors their routing table size to determine when convergence has been reached. When this occurs, the Experimenter is notified. When all the nodes on the network have reached convergence, the Experimenter deploys the microservice applications to the network pertaining to that experiment and awaits the reception of all application deployment metrics. For the purpose of the experiments presented in this paper, an application deployment metric provides the information about whether or not the application could be successfully de-

ployed, the total number of messages issued in successfully deploying or in attempting to deploy the application, and this information is further broken down into three types: *Find Node*, *Resources* and *Deploy*.

The number of microservice applications that are deployed varies depending on the experiment being carried out. However, the definition of an application is maintained relatively constant. For all experiments, a microservice application is composed of five services. Further, the resources needed by these services are generated following a two-tailed normal distribution with mean 1024 and standard deviation (SD) 1000 (Figure 3) so that on average all services of an application are 2% of the weighted average of the simulated instance types on the network (i.e., 966 MB). Thus, all applications deployed have the same number of services but their size, in terms of resources, fluctuates around a certain value. This means that it is likely that no two applications generated by the Experimenter are the same, though it can happen. Thus, Figure 4 displays the actual distribution of services sizes for an experiment run. In much the same way, the preexisting workload on the network will yield a different distribution of available resources each time an experiment is run.

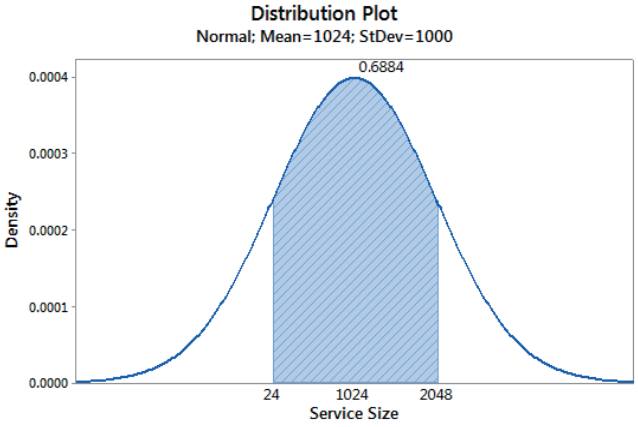

Figure 3: This distribution was employed to generate the service size for all the services generated during each experiment run.

Applications are deployed to the network depending on their ID. In these experiments, the application ID is randomly generated. For those experiments where there is no location awareness, the node that the Experimenter requests deploys each application will be the leader node for that application, which is the node whose ID is closest to the application's ID. However, when location awareness is enabled, the whole application (i.e., all its services) is defined to run at a given network area. Therefore, to deploy these

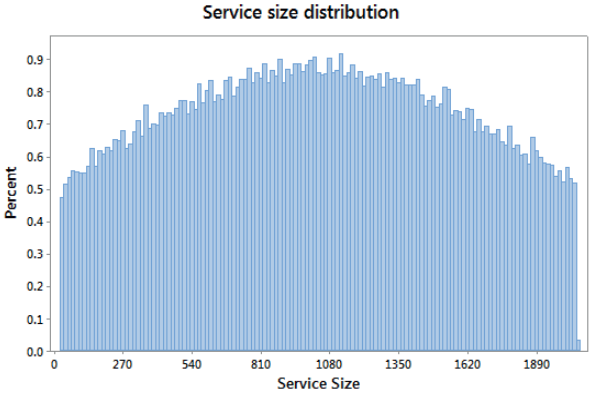

Figure 4: This histogram shows the service sizes that were generated for a given experiment run from making use of the distribution shown in Figure 3.

location-aware applications, the corresponding leaf ID is generated, and the closest node to that ID is the one the Experimenter requests performs deployment of the application's services.

The node that the Experimenter requests deploys an application, employs a search algorithm to attempt to find resources to deploy the services of that application. In this paper, the search algorithm used in all experiments is the random resource search algorithm explained in Section 8. The parameter values of this search algorithm are kept constant for all experiments:

- $\gamma = 5$
- $\delta = 10$
- Parallelism type: strict
- Reservation time: 0

Moreover, a scheduling algorithm is used in tandem with the resource search algorithm. The scheduling algorithm employed in these experiments is a first-fit decreasing (FFD) algorithm [7]. This means that the services a node attempts to deploy are ordered from largest to smallest, and as resources are found, the largest services that fit are deployed first.

The leader node makes use of the search algorithm to query nodes across the whole network when there is no location awareness and only those nodes that lay within the area of the network that the application belongs to, given its leaf ID, when there is location awareness. In the latter case, this means that a whole application (i.e., all its services)

is deployed to a particular location in the network. This is true for the implementation of HYDRA used to carry out the experiments in this paper. However, the design of HYDRA is such that any service of an application can be deployed to any location. Thus, a whole application need not be deployed to the same location. Nevertheless, the current implementation of HYDRA used to carry out the experiments listed in this paper, is limited in its functionality to deploying all the services of an application to the same location when there is location awareness.

Throughout these scenarios, different network sizes are simulated. For network sizes of 5000 or 10 000 nodes, the Amazon EC2 instance used to run the HYDRA nodes was a c5.9xlarge (36 vCPU, 72GiB RAM, and 10Gbps network bandwidth). However, for 15 000 and 20 000 node simulations, the instance type was a c5.18xlarge (72 vCPU, 144GiB RAM, and 25Gbps network bandwidth). These instance types are compute optimized, as we are executing many servers on a single host (e.g., with a 20 000 node network, the instance is simultaneously running 20 000 servers and many more clients) and the instance is also handling and processing a considerable volume of packets.

For all the scenarios delineated in this paper, the Experimenter runs on its own host. The c5.large (2 vCPU, 4GiB RAM, and up to 10Gbps network bandwidth) was employed as the Experimenter instance type.

In scenarios A, B, and C all the experiments run the HYDRA nodes on a single host. The instance type employed in each scenario depends on the network size of each particular experiment, as explained previously. In these scenarios, no packet delay or loss was explicitly configured, and given that all the HYDRA nodes are running on the same host, the delay between nodes is negligible. Further, location awareness was inactive in these scenarios, meaning all IDs were randomly assigned to both, the nodes and the applications deployed.

13 Experimental Results and Discussion

In this section the results of the experimental scenarios are presented, analyzed and discussed. In the explored scenarios, the response variable is either the number of applications that are successfully deployed to the orchestrator network or the average number of messages needed to deploy an application on the network.

13.1 Network Scalability

This scenario aims to analyze the ability of HYDRA to scale in terms of network size. In a previous publication [13] this was analyzed for network sizes from 1000 nodes to 7000 nodes with a step of a 1000.

In this paper, we set out to test HYDRA's scalability further. In this experimental scenario, 8 experiments were run, each with 16 replicates. We deployed HYDRA in a network size of 5000, 10 000, 15 000 and 20 000 nodes. The preexisting network workload was varied between 30% and 90% for all network sizes. Thus, the 8 experiments are characterized by these 8 different values. The outcome of these can be seen in Figure 5.

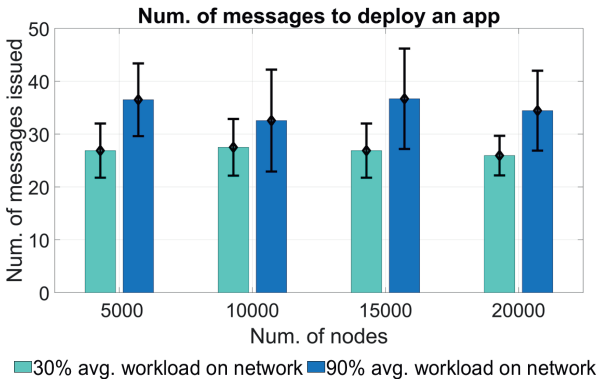

Figure 5: The number of messages issued to deploy an application on different network sizes and preexisting network workloads. This figure presents 8 *experiments, each with* 16 *replicates. Each bar presents the resulting mean and standard deviation.*

The data collected from these experiments was analyzed by: 1) comparing across network sizes when workload is at 30% 2) comparing across network sizes when workload is at 90% 3) comparing between 30% and 90% workload.

From the first analysis we can conclude that the difference in the number of messages issued to deploy an application is not statistically significant with a $p - value = 0.841$. From the second analysis, the conclusion is the same with a $p - value = 0.477$. As for the third analysis, the conclusion drawn is that the variation of the response variable is statistically significant as the preexisting network workload changes, given that the the null hypothesis can be rejected with a $p - value = 0$.

13.2 Application Deployment Scalability

The objective of this scenario is to explore how HYDRA behaves to parallel deployment of applications, that is, to which degree can a given network simultaneously deploy applications.

A total of 8 experiments were run, with 16 replicates each. Figure 6 presents the results of these experiments. The variables considered in this setup were the network size, at 5000 and 15 000 nodes, and the number of applications deployed to the network in parallel. For the 5000 node network, the number of applications simultaneously deployed were 10, 100, 1000 and 10 000. For the 15 000 node network, the applications deployed were 30, 300, 3000, and 30 000. In this setup the preexisting network workload was set to be 70%. The larger network is three times the smaller network, and thus has three times the resources. This is why, on the larger network, the number of applications simultaneously deployed per experiment is three times that of the smaller one.

When deploying 10 000 applications on the 5000 node network, on average 99.99% of the applications were successfully deployed with a standard deviation of 0.013. In the

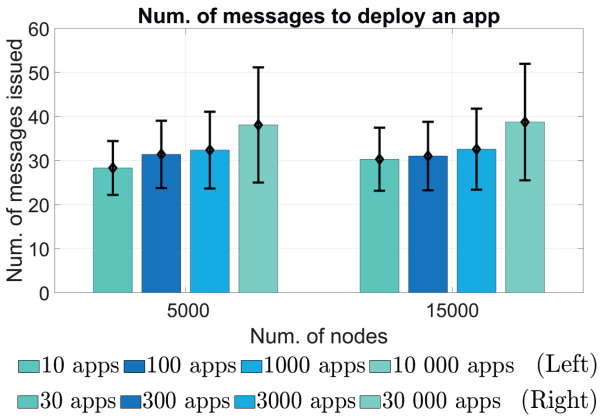

Figure 6: Each bar represents a different experiment (i.e., number of applications deployed simultaneously). On each network size, from left to right, the number of applications increases. The bar height presents the average number of messages needed to deploy an application in that experiment. The bar also presents the pooled standard deviation on that average.

case of 30 000 applications on the 15 000 node network, the average number of successfully deployed applications was 99.98% with a 0.008 standard deviation. All other experiments were 100% successful.

Launching 10 000 applications, of the type presented in the previous section, means that if the preexisting network workload was at 70% when the experiment began, after launching all those applications the network workload would the roughly at 90%. The same applies to the 30 000 application experiment on the 15 000 node network.

These results attest to HYDRA's ability to deal with a large volume of application deployment requests. Given HYDRA's decentralized design, the nodes that handle each application's deployment are fairly spread across the network and the nodes that are required for resources are too. This scenario highlights the benefits of employing a decentralized design in HYDRA as deployment requests are handled by many nodes on the network, according to the application IDs. In a traditional orchestrator these requests would go through a central entity that determines where to deploy an application given the resources of the cluster it manages, this sets a stringent limit on how much the orchestrator is able to handle.

The average number of messages issued to deploy an application does increase with the number of applications being simultaneously deployed to the network. However, this variation does not seem cumbersome. This change is likely due to an increased difficulty in finding resources as they become scarcer on the network, which increases the average number of messages needed to deploy an application. Moreover, collisions may be occurring such that various deployment nodes may be obtaining from a node an affirmative answer about resource availability, however, only one of the deployment nodes succeeds in making use of those resources. This is also more likely to occur when

the network workload is high.

It is clear that the number of messages exchanged to deploy these applications increases as resources become scarcer in the orchestrator network. Therefore, in HYDRA, evidence suggests that deployment scalability is hindered mostly due to resource availability, which is related to the size of the network, and not due to the processing of application deployment requests. This stands to reason as the design of the orchestrator is such that there is no centralized management that processes these requests, but rather they may be submitted to any node on the orchestrator network.

13.3 Performance of the Random Search Algorithm

The following scenario investigates the random search algorithm's ability to find resources across the network to deploy the services of an application. The algorithm is governed by a set of parameters (i.e., γ, δ, parallelism type, and reservation time). In this analysis, those parameters are kept constant to the values presented in Section 12, except for δ which is varied between 10 and 20.

To assess the algorithm's performance when it comes to locating resources, we have calculated an approximation of the maximum number of the defined applications that a given network size could potentially deploy given its initial state.

This approximation is rather loose, as the values taken to calculate this theoretical maximum are based on the average available resources for each simulated instance type and the average size of the services being deployed to the network. As explained in Section 12, the preexisting workload on the nodes and the size of each application's services are derived from a distribution. Further, there are many possible combinations when it comes to fitting services, as which services end up being hosted at which nodes is probabilistic, given the nature of the search algorithm and the design of HYDRA. This means that for each experiment replicate there are multiple possible values of the maximum number of applications that could potentially be deployed, which we seek to estimate via Formula 1.

In this scenario, 4 experiments, each with 16 replicates were performed. The experiments were carried out for three input parameters: network size, number of applications deployed, and δ. The network sizes were of 5000, 10 000, 15 000 and 20 000 nodes. The number of applications deployed during each experiment were the estimated maximum number of applications that could be deployed as per Formula 1, in an attempt to deploy applications that would utilize the resources available on the network to the fullest extent. δ represents the search depth, so when $\delta = 10$, at maximum, 50 nodes on the network are queried about resources when trying to deploy an application, while with $\delta = 20$, a maximum of 100 nodes are queried.

After running the experiment iterations, the estimated post-deployment network workload for each iteration was calculated with Formula 2. The averages of each experiment (i.e., $\overline{W_{post}}$) are provided in Table 2. The standard deviations on these averages are not presented in the table, but they are all between 0.036 and 0.074. Given that each experiment iteration provides the mean and standard deviation on the number of

Table 2: Performance of the Random Resource Search Algorithm

Network size	**Max. # of queries**	**Estimated max. # of applications**	**Average of applications deployed**	**SD of applications deployed**	**Avg. estimated post-deployment network workload**	**Combined avg. of msgs issued per app**	**Combined SD of msgs issued per app**
N	$\delta * \gamma$	max_{apps}	$\overline{n_{apps}}$	$S_{n_{apps}}$	$\overline{W_{post}}$ (%)	$\overline{N_{msgs_{(c)}}}$	$S_{msgs_{(c)}}$
5000	50	4200	3615.8	38.4	97.72 %	51.52	23.65
5000	100	4200	3995.4	36.1	98.52 %	62.78	42.48
10 000	50	8400	7233.1	64.2	97.71 %	51.41	23.09
10 000	100	8400	7929.8	45.7	98.46 %	62.22	41.57

messages issued to deploy an application in that iteration (i.e., $\overline{N_{msgs}}$, S_{msgs}), the combined arithmetic average ($\overline{N_{msgs_{(c)}}}$) and standard deviation ($S_{msgs_{(c)}}$) were computed for each of the experiments. Further, the average and standard deviation on the number of applications that were successfully deployed in each experiment were also calculated ($\overline{n_{apps}}$, $S_{n_{apps}}$).

$$\max_{\text{apps}} = n_s \times \sum_{i=1}^{h} \lfloor \frac{I_i(1-w)}{\overline{S}} \rfloor Nc_i \tag{1}$$

$$\text{W}_{\text{post}} = \frac{n_{apps} \times n_s \times \overline{S}}{\sum_{i=1}^{h} Nc_i I_i} + W_{pre} \tag{2}$$

n_s - Number of services per application (i.e., s=5).
h - Number of instance types simulated in a given experiment (i.e., h=2 in this paper).
I_i - Resources of the ith instance type simultated by HYDRA nodes.
w - Preexisting network workload (value between 0 and 1).
$\overline{S}$ - Average size of a service.
N - Number of HYDRA nodes on the network.
c_i - percent of the network that is simulating the ith instance type (between 0 and 1).
n_{apps} - Number of deployed applications.
W_{pre} - Preexisting network workload.
W_{post} - Estimated post-deployment network workload.

From the results presented in Table 2 we can conclude that the random search algorithm employed in HYDRA is efficient in its search for resources when deploying services as around 2% of the resources are not found. However, this result which seems too good, has be understood within its context.

The current resource vector that characterizes the services is composed of a single resource, RAM. Further, the services that are being deployed are small (i.e., in between 24MB and 2048MB), and given this range, the variability in service size is somewhat conservative, as can be seen in Figure 4. Further, all the experiments carried out in this paper simulate a heterogeneous network, but only two instance types are simulated by the nodes (i.e., 70% of nodes simulate the larger instance type and 30% of them the smaller one).

These reasons make the placement of services easier than it would be in a real-world scenario. However, it does provide a baseline comparison for more complex scenarios and other resource search algorithms. Moreover, this proves the potential viability of this search algorithm as if it did not perform well in these scenarios, then it would have been discarded as a possibly viable resource search algorithm in HYDRA. This algorithm would be the fallback search algorithm of HYDRA as it does work quite well, though a broader resource vector should be tested to have a better idea of its real performance.

However, other resource search algorithms, such as the other algorithm presented in Section 8, could potentially outperform the random search algorithm in some scenarios.

The reason for initially choosing this algorithm was to fulfill three conditions, apart from the obvious condition of successfully deploying applications. These were previously listed in Section 8: 1) benefit from the existing ID-based schema to find resources 2) ideally, the nodes found during a search to deploy an application's services should always be different 3) the search algorithm should distribute services across the network evenly, either throughout the whole network when there is no location awareness or within a network location when there is.

First, condition one is met by this search algorithm directly by design. Second, the number of nodes that are found more than once during a search algorithm run is a small but non-negligible number. A node can be seen more than once in a search when that node is returned as being one of the closest nodes to two or more of the random IDs generated during a given search. In the case of the first experiment of Table 2, the combined arithmetic mean and standard deviation of the number of nodes that are seen more than once is 1.17 and 1.87, respectively. While the combined average and SD of the total number of nodes found during a search where the application was successfully deployed are 26.12 and 12.94. On average, per successful application deployment, 4.48% of the nodes found are repeated nodes.

Third, in an experiment of a network of 1000 nodes where 3000 applications are evenly spread (i.e., 15 000 services), 15 services should be deployed per node. However, 70% of the nodes simulate an instance type, and the rest, a smaller type. Taking into account the sizes of these instances (the large instance is eight times the size of the small one) and their distribution across the network, to evenly spread services, 70% of the nodes should run on average 19.09 services per node, while 30% of the nodes should execute 5.57 services per node. After running the 1000 node and 3000 applications experiment 16 times, we conclude that for 70% of the nodes, the average and standard deviation of the number of services deployed per node are 19.44 and 8.39 while for 30% of the nodes, these are 7.765 and 4.91. Given the results, it seems that the search algorithm does manage to evenly spread the workload across the network.

The number of *Resources* messages that are issued while attempting to deploy an application is the number of nodes that get queried about their resources. This is set to a maximum of 50 nodes when $\delta = 10$ and to 100 nodes when $\delta = 20$. If these do not result in the deployment of the application, the search algorithm exits having failed to deploy the application. For example, in the experiment for a network size of 5000 nodes and a δ of 10, the combined arithmetic mean and standard deviation of the number of *Resources* messages that are issued to successfully deploy an application are 20.23 and 10.9, respectively.

13.4 Location-aware Deployment

The purpose of this scenario is to display HYDRA's ability to deploy applications to target locations on the network and analyze its performance in comparison to when

there is no location awareness. While in the previous scenarios, HYDRA was location-agnostic (i.e., IDs for nodes and applications were random), in this scenario, we define a location-aware network following the LID addressing strategy explained in Section 12.

The network is defined as being separated into three locations, which we will refer to as regions. This location-aware network is composed of 15 000 nodes evenly spread throughout the regions. There are 5000 nodes per region, as 5000 HYDRA nodes are simulated by each EC2 instance.

A single experiment is carried out in this scenario, of which 16 replicates are run. The experiment consists of simultaneously deploying 30 000 applications, where 10 000 applications are deployed to each region. The results for these experiment iterations can be found in Table 3 under "Location-aware".

To contrast the difference between when HYDRA executes with and without location-awareness, the results for the last experiment done in Scenario B are also included in this table. That experiment is of a network of 15 000 nodes where 30 000 applications are simultaneously deployed anywhere on that network as there is no defined network location in that setup.

The number of applications that are successfully simultaneously deployed are statistically significant between the two experiments shown in Table 3. However, we can be 95% confident that the true difference between the population means of the number of applications deployed is contained by the interval (6.032, 11.968). This difference is very small in comparison to the number of applications being compared, so in percentage this 95% confidence interval is (0.017, 0.037). From the results of the combined averages and standard deviations of the number of messages issued per successfully deployed application, we can conclude that the difference between the two scenarios is small and within the interval (0.091, 0.199) with a 95% confidence.

These analyses show that the overhead of running HYDRA with location awareness versus without it, exists, but its impact is minor.

13.5 Location-aware Deployment - One Region Network Partitioned

The goal of this scenario is to analyze HYDRA's performance when it is running with location awareness and a region suffers a network partition from the other regions. In other words, we carry out an experiment to analyze how HYDRA behaves in such a scenario and to validate whether the orchestrator's regions manage to work autonomously so that HYDRA continues performing on a level that is comparable to when there is no network partition.

This scenario carries out the same experiment presented in the previous section, but in this case one of the regions (region A) is network partitioned from the other two regions (regions B and C). That is, there is 100% loss for all traffic going out from region A and all traffic going from the other two regions towards region A.

Given that the nodes for each region are simulated on a single host, the network loss has been implemented via NetEm. NetEm is a Network Emulator that can set delay,

Table 3: Location-agnostic vs.Location-aware - 15 000 nodes, 30 000 apps

	Location-agnostic	**Location-aware**
Avg. percentage of deployed apps	99.984	99.957
SD percentage of deployed apps	0.008	0.017
Avg. number of deployed apps	29 995	29 986
SD on number of deployed apps	2.33	5.22
Sample size	16	16
Combined arithmetic mean of # msgs per app	38.733	38.878
Combined SD of # msgs per app	13.231	13.713
Number of samples	479 925	479 794

packet loss, among other features, in a repeatable and unidirectional way, in this case, from one region to another [15].

For each of the 16 experiment iterations, NetEm is set up at each region instance after the 15 000 nodes on the network have reached convergence. Considering that the nodes are obtaining a random bootstrap node, if the network loss was established prior to convergence some nodes would be unable to successfully bootstrap into the network.

The results of this experiment are presented in Table 4. To determine whether the different locations in the HYDRA network manage to work independently, even under a network partition, we compare these results with those obtained in Scenario D where there was location awareness but no network loss. We conclude that there is statistical significance between these scenarios. Thus, in this case, the difference between population means for the total number of applications successfully deployed is represented by the interval (19.665, 19.735) with a 95% confidence, while in percentage the interval is (0.048, 0.09). However, for the combined average and standard deviation of the number of messages issued to successfully deploy an application, when performing analysis of variance, we cannot conclude that the difference between these two scenarios is statistically significant.

We expected a smaller difference in the number of successfully deployed applications between the two analyzed scenarios. It was observed that for Scenario E, a number of application deployment requests did not result in metrics, probably due to the Experimenter timing out before these were received and the time outs for each message being too long. This could explain the small disparity between the expected and the obtained difference in successfully deployed applications. However, this would have to be resolved before we can establish whether that is the case.

Table 4: Location-aware: One Region is Network Partitioned

	Scenario E	**# of samples**
Avg. percentage of deployed apps	99.888	16
SD percentage of deployed apps	0.037	
Avg. number of deployed apps	29 966.3	
SD number of deployed apps	11.2	
Combined mean of # of msgs per app	38.855	479 461
Combined SD of # of msgs per app	13.739	

13.6 HYDRA Design Considerations

While a hierarchical design of the LIDs may seem constraining in view of the issues encountered by IP, that is not the case here. A HYDRA network is not meant to address all internet-enabled devices. A HYDRA network would be deployed by a cloud provider, telecommunications operator or an organization on the resources they have control over, and on which they wish to deploy containerized microservice-oriented applications. This limits the scale considerably, particularly when compared to the case of IP and the Internet. Further, the location ID design is specific to each deployment of the HYDRA network as is the ID size, which should be carefully chosen according to the requirements of each setup.

To expand and ratify some of the conclusions drawn, particularly for the random resource search algorithm, it would be beneficial to characterize the network with a deeper variability in terms of the vector of resource types, the resource sizes, and the instance types being simulated by the HYDRA nodes.

14 Conclusions and Future Work

This paper explored HYDRA, a decentralized location-aware orchestrator for containerized microservice applications. This orchestrator design focuses on availability, resiliency and scalability of resources and applications through decentralization. Moreover, the proposed orchestrator provides churn resistance and failure prevention and recovery for the management of applications and for the applications themselves. In this paper, we explored various research questions posed in Section 1.

Thus, in reference to the question about HYDRA's capacity to scale, the empirical results show HYDRA successfully scaled to 20 000 nodes.

Further, relative to application deployment scalability, the largest experiment carried out in this paper showed that the orchestrator successfully managed the simultaneous deployment of 99.98% of the 30 000 applications deployed to a 15 000 node network.

As for the viability of the random resource search algorithm and its capacity to act

as a baseline algorithm, the experiments carried out show that the algorithm successfully finds resources to deploy 97% of the applications requesting deployment. This points to this algorithm's capacity to find resources. However, further experiments for larger application resource requirements in comparison to the defined instances available on the orchestrator network, and with multi-dimensional resource requirements, should be carried out to conclude with certainty that this algorithm is a valid option as a resource search algorithm for HYDRA. However, it is an algorithm that can be used as a baseline to compare against future resource search algorithms in HYDRA.

Further, different scheduling algorithms should also be analyzed as it does impact performance when deploying services. In these experiments, a FFD algorithm has been applied. This algorithm might not be a viable option for a multi-resource vector, as services cannot be sorted in order fully. In that case, one of the resources would have to be the main focus of the scheduling algorithm. Thus, HYDRA would benefit from exploring other scheduling algorithms going forward [24].

We can conclude that location awareness in the orchestrator has a negligible impact on orchestrator performance. Further, we have shown that the nodes that make up the orchestrator work independently from one another. The experiments run have shown that if part of the orchestrator network gets network partitioned the orchestrator as a whole still operates normally.

Future work should include experiments where more diverse applications are deployed, both in terms of size and the types of resources these need. Moreover, an analysis of the maximally distant resource search algorithm presented in this paper should be carried out, and compared to the current baseline algorithm. Furthermore, these resource search algorithms are unstructured and entirely decentralized, however, other self-organizing designs for the search of resources across the orchestrator network should be considered.

HYDRA is a decentralized location-aware orchestrator which makes it a more complex system than that of traditional orchestrators. Therefore, a monitoring solution that adapts to the orchestrator architecture is essential. Exploring different approaches [14] to monitoring the state of the orchestrator is a worthwhile endeavor that should be tackled.

We have presented a series of roles and behavior that provide resiliency during the life-cycle management of an application in HYDRA. A study that analyzes resiliency of an application in HYDRA under different conditions would be beneficial. Further, the consensus algorithm to be used in HYDRA, which has been mentioned in this paper, should be evaluated for its validity, and if applicable, compared to other distributed consensus algorithms.

Acknowledgment

This study was partly supported by the Swedish Energy Agency under grant 43090-2, Cloudberry Datacenters.

References

[1] AWS. Aws greengrass.

[2] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. pages 1–10, 01 2017.

[3] Mahantesh Birje, Praveen Challagidad, R.H. Goudar, and Manisha Tapale. Cloud computing review: Concepts, technology, challenges and security. *International Journal of Cloud Computing*, 6:32, 01 2017.

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[5] A. Chandra, J. Weissman, and B. Heintz. Decentralized edge clouds. *IEEE Internet Computing*, 17(5):70–73, 2013.

[6] Cisco visual networking index.

[7] Győrgy Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i) $\leq$ 11/9 opt(i) + 6/9. volume 4614, pages 1–11, 01 2007.

[8] C. Dupont, R. Giaffreda, and L. Capra. Edge computing in iot context: Horizontal and vertical linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–4, 2017.

[9] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes - Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, Inc, 2017.

[10] S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299, 2017.

[11] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[12] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23, 03 2014.

[13] L. L. Jiménez and O. Schelén. Docma: A decentralized orchestrator for containerized microservice applications. In *2019 IEEE Cloud Summit*, pages 45–51, 2019.

[14] Lara Lorna Jiménez., Miguel Gómez Simón., Olov Schelén., Johan Kristiansson., Kåre Synnes ., and Christer Åhlund. Coma: Resource monitoring of docker containers. In *Proceedings of the 5th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 145–154. INSTICC, SciTePress, 2015.

[15] A. Jurgelionis, J. Laulajainen, M. Hirvonen, and A. I. Wang. An empirical study of netem network emulation functionalities. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, July 2011.

[16] K8s. Kubernetes cluster federation.

[17] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. pages 615–629, 04 2017.

[18] Kubernetes. Scalability updates in kubernetes 1.6: 5,000 node and 150,000 pod clusters.

[19] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[20] Sam Newman. *Building Microservices - Designing fine-grained systems.* O'Reilly Media, Inc, 2015.

[21] Claus Pahl. Containerisation and the paas cloud. *IEEE Cloud Computing*, 2:24–31, 06 2015.

[22] Claus Pahl, Pooyan Jamshidi, and Danny Weyns. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *Journal of Software: Evolution and Process*, 29:e1849, 02 2017.

[23] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang. Orchestration of containerized microservices for iiot using docker. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, pages 1532–1536, 2017.

[24] Asser Tantawi and Malgorzata Steinder. Autonomic cloud placement of mixed workload: An adaptive bin packing algorithm. pages 187–193, 06 2019.

[25] William Tärneberg, Amardeep Mehta, Eddie Wadbro, Johan Tordsson, Johan Eker, Maria Kihl, and Erik Elmroth. Dynamic application placement in the mobile cloud network. *Future Generation Computer Systems*, 70:163 – 177, 2017.

[26] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[27] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289 – 330, 2019.

[28] J. Zhu, D. S. Chan, M. S. Prabhu, P. Natarajan, H. Hu, and F. Bonomi. Improving web sites performance using edge servers in fog computing architecture. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 320–323, 2013.

Paper D

A Computationally Inexpensive Classifier Merging Cellular Automata and MCP-Neurons

Authors:
Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg.

Reformatted version of paper originally published in:
The 10th International Conference on Ubiquitous Computing and Ambient Intelligence, 2016, Gran Canaria, Spain.

A Computationally Inexpensive Classifier Merging Cellular Automata and MCP-Neurons

Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg

Abstract

There is an increasing need for personalised and context-aware services in our everyday lives and we rely on mobile and wearable devices to provide such services. Context-aware applications often make use of machine-learning algorithms, but many of these are too complex or resource-consuming for implementation on some devices that are common in pervasive and mobile computing. The algorithm presented in this paper, named CAMP, has been developed to obtain a classifier that is suitable for resource-constrained devices such as FPGA:s, ASIC:s or microcontrollers. The algorithm uses a combination of the McCulloch-Pitts neuron model and Cellular Automata in order to produce a computationally inexpensive classifier with a small memory footprint. The algorithm consists of a sparse binary neural network where neurons are updated using a Cellular Automata rule as the activation function. Output of the classifier is depending on the selected rule and the interconnections between the neurons. Since solving the input-output mapping mathematically can not be performed using traditional optimization algorithms, the classifier is trained using a genetic algorithm. The results of the study show that CAMP, despite its minimalistic structure, has a comparable accuracy to that of more advanced algorithms for the datasets tested containing few classes, while performing poorly on the datasets with a higher amount of classes. CAMP could thus be a viable choice for solving classification problems in environments with extreme demands on low resource consumption.

1 Introduction

Neural networks are algorithms that have proven to perform with great accuracy in a wide variety of applications [16, 20, 23]. They have important and useful characteristics, such as the ability to approximate nonlinear functions, to use different activation functions, to perform both regression and classification, and to be executed in parallel. Neural networks, however, require that all weights between the layers are stored in memory. In addition, these weights are floating point values that need to be multiplied and summed together for being used in a discretization of a continuous function. This makes Neural Networks challenging to implement in hardware or in resource-constrained devices such as FPGA:s. Even for more sophisticated devices like microcontrollers, this can pose problems if there are hard constraints on memory or real-time response.

In order to implement a neural network in such resource-constrained devices, memory consumption, computational complexity and the use of floating point values must be addressed. This could, for example, be achieved by simplifying the neural network by using integer or even binary values and reducing the interconnectivity between neurons. The complexity of implementing the activation function in hardware is of course depending on the function used, but the common sigmoid and hyperbolic tangent functions are not easily implemented in devices such as FPGA:s [27]. This paper aims to find a simplified model addressing all these problems in order to produce a neural network that is suitable for hardware implementation, or for use on devices with limited resources.

1.1 The McCulloch-Pitts neuron model

The neuron model introduced by McCulloch-Pitts in 1943 is an interesting choice for our problem, since it uses only binary values for both neuron output and network weights. Inputs, that are of either inhibitory or excitatory type, are summed together and compared to a threshold value, which determines the output of the neuron. Inhibitory inputs override excitatory inputs, meaning that if there are any inhibitory inputs active, the neuron will always output zero.

Due to its binary-only values, the architecture of the MCP-network is much easier to implement in hardware than a traditional neural network. Although being structurally well-suited to run on primitive devices, there are two problems using the MCP-network for resource constrained classification:

1. It can only model linearly separable problems [21].
2. The computational complexity of three layered MCP-networks is quadratic when using an equal amount of neurons in each layer.

The first problem can be overcome by replacing the activation function with a non-linear one, while the second problem requires to reduce the number of connections between neurons. While there are many ways to solve these problems, this paper examines how principles of binary Cellular Automata could be used for doing so. Binary Cellular Automata has several interesting properties that fit our problem domain, such as having a low computational complexity, being parallelizable and having been proven to be useful in pattern recognition [8, 18].

1.2 Cellular Automata

Cellular Automata (CA) is known to work at the edge of chaos and can show complex behaviour emerging from seemingly simple rules [19]. A binary one-dimensional CA consists of binary cells on a lattice. The cells' states are updated in timesteps in the following way: the next state for each cell is calculated by looking at the cell's own state together with its left and right neighbourhood, called a *configuration*. This configuration, whose size is denoted by $\mathbf{r}$, is used for looking up the cell's next state in a rule table

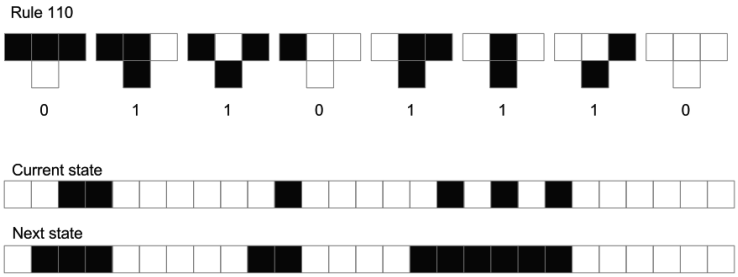

Figure 1: Rule 110 for a 1-dimensional binary cellular automata. The rule is named by translating the output pattern to an integer (0b01101110 = 110). At the bottom is an example of a CA using the rule to evolve one time step.

that holds all possible combinations of configurations. In the case of, for example, 3-neighborhood binary CA, one can construct 256 possible rules. Rules are commonly named by first permuting the possible configurations in an orderly fashion and then constructing a bit pattern from the according next states that is translated to an integer number [26].

Despite its deceitful simplicity, CA has been shown to be turing complete even for a 3-neighbourhood binary cellular automata using rule 110 [10], making it interesting to study this setup and its applications further (Fig. 1).

1.3 Research questions

The hypothesis assumed in this paper, is that using a CA with rule 110 as an activation function could increase the performance of the McCulloch-Pitts model while reducing the number of needed interconnections between neurons (Fig. 2). The research questions addressed are:

1. Can a 3-neighbourhood binary cellular automata using rule 110 be used as an activation function in a simplified MCP-network to create a computationally inexpensive classifier?

2. What is the accuracy of such an algorithm compared to common machine learning algorithms?

The paper is structured as follows: Sect. 2 reviews related work, Sect. 3 describes the CAMP classifier, Sect. 4 explains how CAMP is trained using a Genetic Algorithm, Sect. 5 explains the testing setup, Sect. 6 shows the performance of the classifier, Sect. 7 discusses the results, and Sect. 8 proposes future work and concludes the paper.

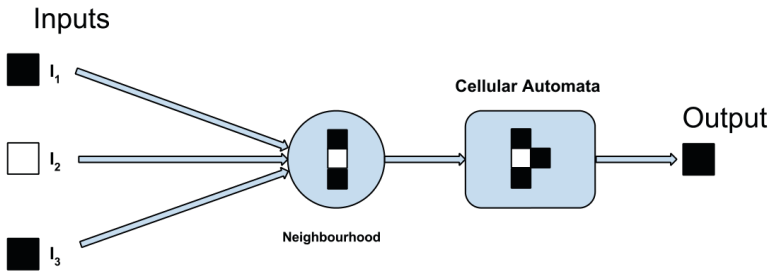

Figure 2: The CAMP neuron. Each neuron has 3 incoming connections. Both inputs and outputs are binary-valued.

2 Related work

3-neighbourhood binary cellular automata has shown promising results in pattern recognition. Chaudhuri et al. have done extensive work on additive cellular automata [9]. Among their work we find a multi-class pattern classifier using Multiple Attractor Cellular Automata (MACA). This classifier saved on average 34% of computer memory using pseudo-exhaustive patterns compared to the straight-forward approach of storing the class of each pattern directly [8]. Maji et. al performed pattern recognition with CA:s using General Multiple Attractor Cellular Automata (GMACA). They proposed an associative memory using CA that was found to be 33% more effective than a hopfield network [18].

The pattern recognition algorithms using (G)MACA:s are highly computationally efficient, but they only act as content-addressable memories and do not create generalised models over data. They also require that the learnt patterns are close to each other in Hamming Distance [15], which is problematic since data from e.g sensors and such need to be encoded into binary patterns in a suitable way (e.g. gray code [6], thermometer encoding [12] or scatter code [22]). The CAMP algorithm, however, does not require such binary encoding of input data due to its non-constant neighborhood configuration inspired by traditional feedforward neural networks.

Combining principles from Neural Networks with Cellular Automata is not in itself a novel approach. Cellular Automata Neural Networks (CNN:s, note this is not to be confused with Convolutional Neural Networks), for example, are also neural networks using principles from CA. The focus of CNN research has however mainly been to improve the accuracy of classification and regression. In a CNN, decimal values are used for weights and neuron outputs like in a traditional neural network. Cells are often connected in a 2-dimensional grid with an 8-neighbor configuration, and are updated using their neighbor's values as inputs to an output function. Cells' outputs are connected to other cells and/or the output of the algorithm. CNN:s are promising and have shown to be successful in, for example, image processing [5], fluid dynamics [17] and statistical physics [14]. The CAMP algorithm differs from a CNN in the way that each cell can

connect to any **r** cells in the CA, not just adjacent ones, and it uses only binary values and relies on binary logic operations instead of floating point value computations.

Another field of research related to the work in this paper is chaotic neural networks. There is research showing that chaotic dynamics may exist in biological neurons [2–4, 13] which have led to research with using chaotic activation functions in neural networks. These types of networks can have advantages regarding computational time and memory usage, since the complex dynamics of the neurons are described by simple and deterministic difference equations rather than models of differential equations or stochastic processes [1].

3 The CAMP algorithm

This work tries to utilize the fact that CA works at the edge of chaos, and is turing complete even for a binary neighborhood of three cells, in order to create a simplistic neural network. In traditional Cellular Automata, a configuration consists of adjacent cells, but in order to create the proposed algorithm, this principle is loosened to allowing cells to get their neighbourhood from *any* 3 cells, not just adjacent ones. Another way of looking at the algorithm, is considering it to be a MCP-neural network, where each neuron has 3 connections to neurons in the previous layer and all connections are of excitatory types (unit weight) only.

A CAMP neuron gets updated in the following two steps:

1. Determination of its neighbourhood configuration.
2. Calculation of the next state using the CA rule for that configuration.

In essence, the CAMP algorithm works in the same manner as a feed-forward neural network. Since the aim of this paper is to keep resource consumptions at a minimum, a three layered network structure was used (input, hidden, output). All layers were chosen to have equal size. To perform the classification, the output layer's pattern is compared to chosen output patterns for the different classes.

The memory M and time T required by the CAMP algorithm for a number of neurons n in "Big O" notation evaluates to (assuming n neurons in each layer):

$$\begin{aligned} M(n) &= \Theta(n) \\ T(n) &= \Theta(n) \end{aligned} \tag{1}$$

This can be compared to a "traditional" feed-forward Neural Network (assuming n neurons in each layer):

$$\begin{aligned} M(n) &= \Theta(n^2) \\ T(n) &= O(n^2) \end{aligned} \tag{2}$$

It should be noted though, that CAMP requires we create a neuron *for each bit* in an input variable. This means that the number of neurons n is always higher for CAMP

than it is for a Neural Network by a multiplier that has the size of the input variables bit-resolution. This number is however constant, so the Big-O notation comparison still holds.

The CPU operations required to update each CAMP neuron can easily be calculated due to the simplistic structure of the classifier (assuming AND, OR and NOT are part of the CPU instruction set and are atomic). The operations consists of evaluating the boolean expression that is the activation function. For rule 110 this expression is:

$$q = (\neg a \wedge b) \vee (b \wedge \neg c) \vee (\neg b \wedge c)$$

Where q is the next state for a neuron with incoming connections a, b, c. In the case of rule 110, each such update requires 8 instructions which means that in order to calculate the activation functions of n neurons we must perform 8n CPU instructions.

As we can see, the CAMP algorithm holds several interesting properties that are important for machine learning algorithms running in resource-constrained environments:

1. It has a small memory footprint.
2. It is computationally inexpensive.
3. It does not make use of floating point values, making it faster to compute on less sophisticated processors [11].
4. Updates of neurons are done locally, i.e. does not need access to a global state making parallel computations possible.
5. It has a simple structure and logic making it suitable for FPGA or ASIC implementation.

4 Training of the classifier

The output of the classifier depends solely on the set of connections between the cells, and training the classifier consists of finding the best set of connections among the nodes that will map an input set to an desired output set. The chaotic behaviour of the CA, however, makes it problematic to solve this mapping mathematically. Exhaustive search is not an option either, since there are C possible combinations of connections for a three layer network with I, J, K number of nodes in each layer:

$$C = \left(\frac{I!}{r!(I-r)!}\right)^J \left(\frac{J!}{r!(J-r)!}\right)^K \tag{3}$$

This means that even for a limited neighbourhood of $\mathbf{r} = 3$, the number of permutations of connections escalates quickly. In order to find a connection set, we instead rely on a genetic algorithm to converge towards a solution. This is unfortunate, since this does not guarantee finding the optimal solution. The following sections describes the genetic algorithm's four steps:

1. Chromosome encoding.
2. Fitness calculation.
3. Selection.
4. Crossover and mutation.

4.1 Chromosome encoding

The input connections to a neuron from the previous layer, is represented using a bit string, or a connection-vector, where each bit corresponds to a neuron in the previous layer. A set bit indicates there's a connection to the corresponding neuron. Each connection-vector becomes a gene, and chromosomes are created by concatenating all the connection-vectors (genes) into one binary string using a lexicographical order.

4.2 Fitness calculation

The fitness F for each chromosome is simply calculated by running the algorithm with the chromosome's connections and counting the amount of correct classifications among the q training examples.

$$F = \sum_{1}^{q} (classifierOutput_q \wedge correctClassPattern_q) \tag{4}$$

4.3 Selection of individuals

Roulette wheel selection was used to select mating individuals from the current population based on their fitness results. The 10 most fit individuals were added to the next generation unmodified (elitism).

4.4 Crossover and mutation

Uniform crossover was used where both parents' connection vectors for each cell was equally probable to be used in the offspring chromosome. Mutation rate was set to 5 percent and mutation was performed by randomly changing one of the offspring connections. Each offspring could only mutate once.

5 Test setup

In order to evaluate the CAMP algorithm the WEKA 3.7.12 platform was used [25]. WEKA is an open source tool for machine learning and data mining, that supports several standard machine learning algorithms of various types along with other useful tools such as validation methods, pre-processing, visualization, and feature selection.

The CAMP algorithm was implemented as a classifier in WEKA since that allowed for the algorithm to be benchmarked against validated implementations of other machine learning algorithms. This also makes it easier for other researchers to put the results presented here into perspective.

5.1 Datasets and feature selection

Six different classification datasets with numerical features were used in order to evaluate the algorithm's performance (see Table 1). These datasets are all included with the WEKA distribution and are thus publically available. Further, they have all been part of published scientific work, making it possible for researchers to compare our results with other work on the same datasets.

For each dataset, the three highest ranked features were selected using InfoGain [7] to avoid overfitting. All results were obtained using 10-fold cross validation.

5.2 CAMP setup

Dataset instances' binary representations were used to setup the input layer (each bit corresponding to a CAMP-neuron). Both the hidden layer and the output layer were chosen to have the same size as the input layer. Each output neuron corresponds to a choice of a class. For example, a three class dataset would have the output configurations; "100...." = class A, "010...." = class B, "001...." = class C. Classification is then performed by evaluating the Hamming Distance between the CAMP output and these output configurations. Training was performed by running the genetic algorithm for 1000 generations with a population size of 100 and with a 10 individual elite.

5.3 Comparative algorithms setup

To put the CAMP performance in perspective, five of WEKA's standard machine learning algorithms of different types (Bayesian, Tree, Neural Networks, Ensemble) were used as a comparison. Since this was performed only to have an indicative comparison of CAMP's performance, all the comparative algorithms used their default settings in WEKA in order for the experiments to be easily repeated. This means that neither CAMP nor the comparative algorithms were adapted in any way to fit the datasets and it is therefore important to note that the results presented here are not to be considered the best performance possible for neither CAMP nor the comparative algorithms. The results from the different classifiers are shown in Table 2.

6 Results

The results in Table 2 show that CAMP works as a classifier. In 2 out of 6 datasets CAMP shows less than 4 percent worse accuracy compared to a Multilayer Perceptron

Table 1: Description of the tested datasets.

Dataset	Classes	Instances	Features selected	Description	Creator
Glass	7	214	Mg, Al, K	Dataset for classification of types of glass motivated by criminological investigation.	B. German – Central Research Establishment Home Office Forensic Science Service Aldermaston, Reading, Berkshire RG7 4PN
Ionosphere	2	351	a05, a06, a33	Radar data classifying good or bad radar returns. Good radar returns are those showing evidence of some type of structure in the ionosphere. Bad returns are those that do not; their signals pass through the ionosphere	Space Physics Group Applied Physics Laboratory Johns Hopkins University Sigillito, V. G., Wing, S. P., Hutton, L. V., and Baker, K. B. (1989)
Segment-Challenge	7	1500	Intensity-mean, rawred-mean, rawgreen-mean	The instances were drawn randomly from a database of 7 outdoor images. The images were hand segmented to create a classification for every pixel. Each instance is a 3x3 region	Vision Group, University of Massachusetts
Labour	2	57	Wage-increase-first-year, wage-increase-second-year, statutoru-holidays	Data includes all collective agreements reached in the business and personal services sector for locals with at least 500 members (teachers, nurses, university staff, police, etc) in Canada in 87 and first quarter of 88.	Collective Barganing Review, montly publication, Labour Canada, Industrial Relations Information Service, Ottawa, Ontario, K1A 0J2, Canada
Diabetes	2	768	plas, mass, age	A diagnostic, binary-valued variable is investigated to show whether the patient shows signs of diabetes according to World Health Organization criteria.	National Institute of Diabetes and Digestive and Kidney Diseases, Vincent Sigillito (vgs@aplcen.apl.jhu.edu) Research Center, RMI Group Leader Applied Physics Laboratory The Johns Hopkins University

which, while in half the tests this figure is less than 10 percent. Also, for 2 out of the 6 datasets, the CAMP algorithm even performs marginally better than 3 of the other comparative algorithms. This is promising and shows that the algorithm could prove

beneficial, at least for certain kinds of classification problems. The datasets for which the algorithm has the highest performance have in common a low number of classes. Similarly, the algorithm seems to perform poorly on datasets with a higher amount of classes.

Table 2: The classification results for the algorithms on the different datasets.

	CAMP	Naive Bayes	Bayes Net	Multi-layered Perceptron	K-star	J48	Random Forest
Glass	**43.45%**	49.06%	63.55%	63.08%	64.49%	64.49%	67.29%
Ionosphere	**80.34%**	86.32%	89.46%	87.46%	89.17%	90.60%	90.03%
Segment-challenge	**41.80%**	60.60%	70.53%	78.47%	76.20%	81.00%	86.00%
Labour	**64.91%**	84.21%	78.90%	84.20%	80.70%	80.70%	84.20%
Unbalanced	**98.60%**	98.60%	98.60%	98.60%	98.60%	98.60%	97.90%
Diabetes	**72.91%**	76.43%	74.22%	76.43%	72.40%	74.61%	72.53 %

7 Discussion

It is important to look at the CAMP results with its computational complexity and simple structure in mind. In this light, the performance of the algorithm is impressive. Also, results could most likely be improved since there were no efforts made to adjust the algorithm to fit each dataset. By using different feature selections and feature scaling or changing parameters such as bit resolution, the algorithm could be better matched against a certain dataset.

Results suggest that the algorithm performs better on binary classification problems. In Cellular Automata there can exist states that can't be reached from some previous state given restrictions of rules and timesteps. Therefore, increasing the number of output patterns (states) in a classification problem will reduce the solution space for the CAMP input-output mapping. It is therefore possible that reducing the single multi-class classification problems into multiple binary classification problems (similar to e.g. multiclass Support Vector Machines [24]), could improve the accuracy of the algorithm. In the same way, increasing the number of output patterns per class could improve the accuracy by increasing the input-output mapping solution space. Both these solutions, however, could impose a much higher level of implementation complexity for ASIC:s and FPGA:s.

Another possible way of increasing the accuracy of CAMP would be to use an expanded neighbourhood size in order to allow for more complex computations. This would, however, result in both a larger memory footprint and a higher computational complexity, since the number of connections between neurons would increase. Using a deeper network, with two or more hidden layers, could also allow for a higher complexity of computations, but with the same tradeoff as an expanded neighborhood.

A final way that could improve the classification performance, would be to use different rules. The CA rule 110 was chosen for this work based on its ability of universal

computation, but further studies of the algorithm should include the use of other rules as well. Another interesting study would be to expand the algorithm to use non-uniform cellular automata. Each neuron would then have its own update rule, thereby drastically expanding the solution space for the input-output mapping. This would, however, increase the memory usage of the algorithm and could increase the number of atomic operations needed to update the neurons.

8 Conclusions and future work

This paper proposes a binary-valued classifier based on a combination of Cellular Automata and the McCulloch-Pitts neuron model. The classifier is designed to be computationally inexpensive and highly suitable for microcontrollers, FPGA:s and ASIC:s. Evaluation was performed using six different datasets with their three highest ranked features selected using InfoGain. Results show that the classifier is a viable choice for certain problems where resource constraints are high.

Future work should include studying how reducing the single multiclass classification problems into multiple binary classification problems affect the accuracy of the algorithm. Another study would be to allow a higher amount of connections between neurons, as this allows for more complex input-output computations. Furthermore, the use of hybrid CA's (each neuron/cell having its own update rule) could also be studied, but both these studies would make the algorithm itself more complex and thus less suited for the initial purpose.

References

[1] M. Adachi and K. Aihara. Associative dynamics in a chaotic neural network. *Neural Netw.*, 10(1):83–98, Jan 1997.

[2] K. Aihara and G. Matsumoto. *Forced oscillations and routes to chaos in the Hodgkin-Huxley axons and squid giant axons*, pages 121–131. Chaos in biological systems. Springer, 1987.

[3] Kazuyuki Aihara and Gen Matsumoto. Chaotic oscillations and bifurcations in squid giant axons. *Chaos*, 12:257–269, 1986.

[4] Erol Basar. *Chaos in Brain Function: Containing Original Chapters by E. Basar and TH Bullock and Topical Articles Reprinted from the Springer Series in Brain Dynamics*. Springer Science & Business Media, 2012.

[5] Alper Başt˝urk and Enis G˝unay. Efficient edge detection in digital images using a cellular neural network optimized by differential evolution algorithm. *Expert Systems with Applications*, 36(2):2645–2650, Mar 2009.

[6] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, Sep 1976.

[7] Veronica Bolon-Canedo, Noelia Sanchez-Marono, and Amparo Alonso-Betanzos. A review of feature selection methods on synthetic data. *Knowledge and information systems*, 34(3):483–519, 2013.

[8] S. Chattopadhyay, S. Adhikari, S. Sengupta, and M. Pal. Highly regular, modular, and cascadable design of cellular automata-based pattern classifier. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 8(6):724–735, Dec 2000.

[9] Parimal Pal Chaudhuri. *Additive cellular automata: theory and applications*, volume 1. John Wiley & Sons, 1997.

[10] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[11] Nicholas J. Cotton, Bogdan M. Wilamowski, and Gunhan Dundar. A neural network implementation on an inexpensive eight bit microcontroller. pages 109–114. IEEE, Feb 2008.

[12] Paul Crook, Stephen Marsland, Gillian Hayes, Ulrich Nehmzow, et al. A tale of two filters-on-line novelty detection. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 3894–3899. IEEE, 2002.

[13] Dennis W. Duke and Walter S. Pritchard. *Proceedings of the Conference on Measuring Chaos in the Human Brain, April 3-5, 1991, at the Supercomputer Computations Research Institute, Florida State University, Tallahassee, FL.* World Scientific, 1991.

[14] M. Ercsey-Ravasz, T. Roska, and Z. NÃ©da. Statistical physics on cellular neural network computers. *Physica D Nonlinear Phenomena*, 237(9):1226–1234, Jul 2008.

[15] N. Ganguly, P. Maji, B.K. Sikdar, and P.P. Chaudhuri. Design and characterization of cellular automata based associative memory for pattern recognition. *IEEE Transactions on Systems Man and Cybernetics Part B (Cybernetics)*, 34(1):672–678, Feb 2004.

[16] K. J. Hunt. Neural networks for control systems - a survey. *Automatica*, 28:1083–1112, 1992.

[17] Sándor Kocsárdi, Zoltán Nagy, Árpád Csík, and Péter Szolgay. Simulation of 2d inviscid, adiabatic, compressible flows on emulated digital cnn-um. *International Journal of Circuit Theory and Applications*, 37(4):569–585, May 2009.

[18] Pradipta Maji, Niloy Ganguly, Sourav Saha, Anup K. Roy, and P. Pal Chaudhuri. *Cellular automata machine for pattern recognition*, pages 270–281. Cellular Automata. Springer, 2002.

[19] Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys*, 32(1):80–107, Mar 2000.

[20] J. David Schaffer, David Whitley, and Larry J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37. IEEE, 1992.

[21] Michael Schmitt. On the size of weights for mcculloch-pitts neurons. In *Proceedings of the Sixth Italian Workshop on Neural Nets WIRN VIETRI-93*, pages 241–246. Citeseer, 1994.

[22] D. Smith and P. Stanford. A random walk in hamming space. pages 465–470 vol.2. IEEE, 1990.

[23] A. Vellido, P. J. G. Lisboa, and J. Vaughan. Neural networks in business: A survey of applications (1992-1998). *Expert Syst. Appl.*, 17:51–70, 1999.

[24] Jason Weston and Chris Watkins. Multi-class support vector machines. Technical report, Citeseer, 1998.

[25] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2005.

[26] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, 2002.

[27] Xie Zhen-zhen and Zhang Su-yu. A non-linear approximation of the sigmoid function based fpga. In *Proceedings of the 2011, International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE2011) November 19–20, 2011, Melbourne, Australia*, pages 125–132. Springer, 2012.

Paper E

Classifier Optimized for Resource-constrained Pervasive Systems and Energy-efficiency

Authors:
Niklas Karvonen, Lara Lorna Jiménez, Miguel Gómez Simón, Joakim Nilsson, Basel Kikhia, and Josef Hallberg.

Reformatted version of paper originally published in:
International Journal of Computational Intelligence Systems, 2017.

Classifier Optimized for Resource-constrained Pervasive Systems and Energy-efficiency

Niklas Karvonen, Lara Lorna Jiménez, Miguel Gómez Simón, Joakim Nilsson, Basel Kikhia, and Josef Hallberg

Abstract

Computational intelligence is often used in smart environment applications in order to determine a user's context. Many computational intelligence algorithms are complex and resource-consuming which can be problematic for implementation devices such as FPGA:s, ASIC:s and low-level microcontrollers. These types of devices are, however, highly useful in pervasive and mobile computing due to their small size, energy-efficiency and ability to provide fast real-time responses. In this paper, we propose a classifier, CORPSE, specifically targeted for implementation in FPGA:s, ASIC:s or low-level microcontrollers. CORPSE has a small memory footprint, is computationally inexpensive, and is suitable for parallel processing. The classifier was evaluated on eight different datasets of various types. Our results show that CORPSE, despite its simplistic design, has comparable performance to some common machine learning algorithms. This makes the classifier a viable choice for use in pervasive systems that have limited resources, requires energy-efficiency, or have the need for fast real-time responses.

1 Introduction

Pervasive devices are often kept small in order for them to be unobtrusive. Reducing the physical size of a system, however, typically also reduces computational resources (e.g. memory and CPU) which affects the choices of hardware and software. It is also common for pervasive devices to be battery-powered which enforces even harder constraints upon their hardware and software design. These resource-constraints can be challenging when creating context-aware devices or services, since such systems often rely on machine learning (ML). ML algorithms often require significant computational resources (e.g. memory, CPU) that makes them difficult to implement in devices such as Field-Programmable Gate Arrays (FPGA), Application-Specific Integrated Circuits (ASIC) and low-level microcontrollers. Many such algorithms (e.g. neural networks, Support Vector Machines) also rely on floating point operations, which further complicate such implementations [24]. This is unfortunate, since low-level components such as FPGA:s are typically highly useful for pervasive computing due to their small size, energy efficiency, fast response times, and high throughput (i.e. when pipelined).

In this work we extend our previous efforts on creating a classifier specifically targeted at low-level and resource-constrained devices [13]. We propose a Classifier Optimized for

Resource-constrained Pervasive Systems and Energy-efficiency, CORPSE for short. The classifier is based on Elementary Cellular Automata (ECA) which can show complex behaviour emerging from seemingly simple rules [21]. This makes it an interesting underlying model for a classifier, since it exhibits nonlinear behaviour and it can work at "the edge of chaos"; a powerful realm for computations [17]. Additionally, ECA has been proven capable of universal computation (Rule 110) [7] which shows that the model holds extensive expressive power.

Another interesting property of ECA is that it does not rely on a global state for making computations. This makes it suitable for parallel processing, which can be utilized for reducing power consumption or decreasing real-time response times in pervasive computing. Furthermore, ECA requires only a binary lattice and a set of boolean algebra rules to perform its computations which allows for efficient implementations on low-level devices such as FPGA:s and ASIC:s.

Given the attractive properties of ECA combined with the challenges of performing context-recognition on low-level devices, the research questions addressed in this paper are:

RQ1: *Can Elementary Cellular Automata be extended to create a computationally inexpensive classifier by allowing varying neighborhoods across space and time?*

RQ2: *Which are the important parameters in terms of setup and training to optimize the performance of such an algorithm?*

This paper is structured as follows: section 2 reviews related work, section 3 describes the CORPSE classifier and explains how it is trained using a Genetic Algorithm, section 4 shows the evaluation of the algorithm, section 5 shows the results, followed by discussion in section 6, and conclusions and future work in section 7.

2 Related Work

Being able to run machine learning algorithms on resource-constrained devices allows for richer context-aware applications, such as determining behavioural patterns of a user [11,15]. Several efforts have been made to adapt existing machine learning algorithms for such systems. Lee et. al [18] stated that using machine learning offer promising tools for analyzing physiological signals, but that the computations involved are not well handled by traditional DSP:s. In their work they propose a biomedical processor with configurable machine-learning accelerators for low-energy and real-time detection algorithms.

Kane et. al [12] state that the computational process of Support Vector Machine classification suffers greatly from a large number of iterative mathematical operations and an overall complex algorithmic structure. Their work proposed a fully pipelined, floating point based, multi-use reconfigurable hardware architecture designed to act in conjunction with embedded processing as an accelerator for multiclass SVM classification. Other work such as DianNao [5] and PuDianNao [19], has also focused on hardware

accelerating the most resource-consuming computations of common machine learning algorithms.

While most research on machine learning for use on resource-constrained devices has been focused on adopting and optimizing existing algorithms for implementations in FPGA:s, the classifier proposed in this paper is, however, designed specifically for this purpose. The simple principles of CORPSE yields a low computational complexity while using only rudimentary binary operations [13]. These properties makes it easy to implement on digital circuits and depending on the configurations of the classifier it can also be fully or partially pipelined.

The use of ECA for classification on resource-constrained devices is not a novel approach. Several researchers have utilized the attractive properties of ECA in order to create computationally efficient classifiers. Chaudhuri et al. [4] did extensive work on additive cellular automata. Among their work we find a multiclass pattern classifier using Multiple Attractor Cellular Automata (MACA) [20]. Ganguly et. al [10] performed pattern recognition with CA:s using Generalized Multiple Attractor Cellular Automata (GMACA). Their study confirmed the potential of GMACA to perform complex computations, like pattern recognition, at the edge of chaos.

While these pattern recognition algorithms using (G)MACA:s are highly computationally efficient, they act as content-addressable memories and do not necessarily create generalized models over data. They also require that patterns belonging to the same class must be close to each other in terms of Hamming Distance [10]. This can be problematic for use in pervasive systems since data from e.g sensors or other peripherals first need to be encoded into binary patterns in a way that upholds this requirement (e.g. gray code [2], thermometer encoding [8] or scatter code [22]). Our proposed algorithm does not require any such encoding so the input data can be fed straight from any sensors to the classifier.

Cellular Automata Neural Networks (CANN) are another example of where principles of cellular automata are used to create classifiers [6]. CANN:s have shown to be successful in, for example, image processing [1], fluid dynamics [16] and statistical physics [9]. Unfortunately, CANN:s still rely on floating point values to perform its computations, which complicates the implementations in digital logic. However, the CORPSE classifier is not limited in this way.

3 CORPSE Algorithm

CORPSE working principles

CORPSE's basic principle is similar to an ECA with the difference that each cell's neighborhood can vary spatio-temporally (See figure 1). The rationale behind this is that it creates a function similar to a sparse neural network where each neuron has three incoming connections of unit weight and uses the ECA rule as an activation function. This creates a novel combination of neural networks and cellular automata that aims to add powerful computational properties from neural networks to ECA. Since the ECA neigh-

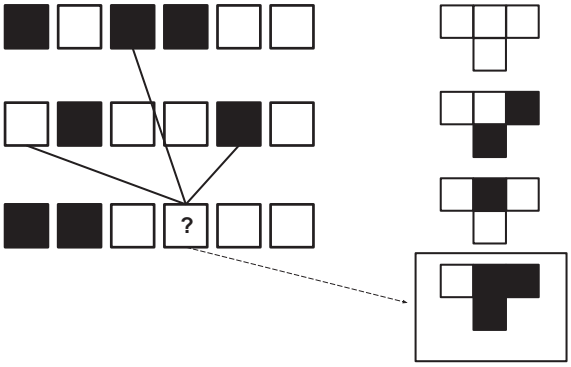

Figure 1: The principle of determining neighborhood and updating of a CORPSE neuron/cell.

borhood varies in space, it enables a cell A to "shortcut" its interactions with another cell B in the automata without having to propagate changes through the cells between them during multiple timesteps. By allowing the neighborhood to also connect across timesteps, cell A can also interact with any previous state of cell B. This enables to have temporal dependencies between cells in the network. Each cell also has the possibility to connect to bias cells which have a constant value of zero or one.

Since most researchers are likely more familiar with neural networks than with cellular automata, we will introduce the notion of CORPSE as a binary neural network. CORPSE can be viewed as a sparsely connected binary neural network where each neuron has three connections to neurons in previous layers. Each connection has unit weight, i.e. only copies the value from the connected cell. The activation function used for the network can be any ECA rule.

The CPU operations required to calculate a neuron's state, consists of evaluating a boolean expression (the activation function). E.g. for rule 110 that is used in this work:

$$q = (\neg a \wedge b) \vee (b \wedge \neg c) \vee (\neg b \wedge c) \tag{1}$$

Where q is the next state for a neuron with incoming connections a, b, c. Each update thus requires 8 instructions when implemented on a CPU (assuming AND, OR and NOT are part of the CPU instruction set and are atomic).

To perform a classification, all features' bit patterns are loaded onto the input layer. Then the network is run in a forward-propagation manner which will produce a bit pattern at the output layer. This output pattern is then compared to a set of bit patterns that corresponds to each class. The comparison between the output pattern and the classes' bit patterns is performed using a binary similarity measure. The class with the highest similarity to the output pattern is chosen.

CORPSE Hyper parameters

There are five different setup parameters for CORPSE:

1. The number of network layers.
2. Whether to allow neurons' connections to skip over layers (i.e. connect to a neuron in any previous layer)
3. The bit pattern for each class (that the output pattern will be compared to).
4. The similarity measure used to compare the output pattern to the classes' bit patterns.
5. The cellular automata rule to use as activation function for the network.

Using a deeper network model is assumed to allow for a higher degree of expressiveness for the classifier model. A deeper model, however, results in a larger search space for the Genetic Algorithm to explore during training. The number of layers also affect the computational resources required for running the algorithm.

Since CORPSE is based on ECA, the computational expressiveness could be limited when only a small number of layers is used. By allowing neurons to connect to a neuron in any previous layer and not just the adjacent (previous) layer, the expressive power of the classifier could increase due to introducing the ability of having temporal dependencies between neurons. It should be noted, however, that using this setting complicates a fully pipelined implementation of the classifier.

The bit patterns assigned to the classes interacts with the GA fitness model and the output similarity measure. Together, these parameters make up the discriminative function of the classifier. In order to achieve optimal results, these parameters should be tuned to fit the classification problem at hand.

There exists 256 different rules for ECA that can be used for CORPSE. While these rules have known characteristics in ECA, it is not clear how they affect the behaviour of CORPSE when the neighborhood configurations are different. Most likely, the activation rule should be chosen with regards to the comparative bit patterns, the output similarity measure, and the dataset.

Training

Training is performed using a Genetic Algorithm (GA) where an individual represents all connections between cells in the network. The fitness of an individual is the total amount of correct classifications the individual's network gets on the training set. Regeneration is performed using tournament selection and elitism is used to prevent degeneration. The least fitted individuals are culled and replaced with random individuals. Uniform crossover is performed by randomly picking connections from the parents and assigning these to the offspring. The probability for an individual to mutate is based on the mutation rate. If an individual is chosen to mutate, a selected amount of its connections

will be randomly changed (with the constraint that each cell must connect to a cell in a previous layer).

4 Evaluation of CORPSE

4.1 Evaluation setup for the CORPSE algorithm

Implementation

In order to evaluate the algorithm, CORPSE was implemented in Java to work with the WEKA machine learning suite [23]. This choice allowed the algorithm to be compared with other machine learning algorithms whose implementation and performance is known within the research community. Five classifiers of different types (Bayesian, Tree, Neural network, Ensemble) were chosen to be used for comparison with CORPSE's results. The choice of having multiple algorithms as a baseline was based on the fact that some datasets are better classified with a certain type of classifier. Using several algorithms for comparison reduces the effect of this while also giving readers a better bases for further reasoning about the performance of CORPSE.

GA Training Parameters

In an effort to find well-performing parameters for the GA, a series of 192 tests were performed varying: number of generations, elite size, tournament size, crossover type(uniform, single-point), and mutation rate. Average fitness plots were used to validate convergence of the GA. From these experiments, the best overall performing parameters were selected:

- 400 generations
- 100 population size
- 85 tournament selected individuals
- 2 elites
- 13 individuals culled
- Uniform crossover
- 90 percent mutation rate (amount of connections changed in a mutation was randomly chosen between 1-5)

4.2 Datasets and Feature Selection

Due to our earlier work and findings [13], this article focuses on binary classification problems. 8 different datasets (See Table 1.) were used for evaluating the proposed algorithm. Datasets 1-7 were chosen since they included with WEKA, which makes it easy

to reproduce our results. They also allow for a comparison with our previous efforts [13]. For more information about these datasets, please refer to the WEKA documentation. Dataset 8 was chosen based on that it consists of an activity recognition task using accelerometer data from a wrist-worn device [14]. This was considered a good use-case of the proposed algorithm. It is also significantly larger than datasets 1-7. In order to avoid overfitting and to put a resource-constraint on the feature selection, the three highest ranked features for each dataset were selected using InfoGain [3].

4.3 Evaluating hyper parameters

A series of tests were performed varying different hyper parameters of CORPSE to analyze its performance. All these tests were performed using 10-fold cross validation. Following are the the different tests and their results.

Allowing connections to skip layers

This test aimed at exploring if an increased expressive power of the classifier could enhance performance. Results showed that allowing neurons to connect (skip) over layers did, in fact, have a positive effect on the performance.

Using bias neurons

Since the GA can choose whether to connect "regular" neurons to bias neurons or not, the hypothesis in this test was that using bias neurons should always yield a better result. Our tests, however, showed that using bias could actually degrade performance slightly.

Using different bit patterns for classes

Two different models were used for generating the bit patterns for each class. The first one was to randomly choose the bit pattern belonging to each class. This choice was based on the assumption from hyper-dimensional computing that two random binary vectors is approximately orthogonal if their dimensionality is in the thousands. Although the dimensionality in this paper only reaches the hundreds, this model was still considered an interesting setup since the two patterns will still likely be discriminative in hamming space.

The second choice was to use a equidistant and maximized hamming distances between each pattern. This was achieved by dividing the total pattern length in two and filling the first half with ones for class A and the second half with ones for class B. This setup was also chosen to make the two patterns be discriminative in hamming space.

The test's results indicates that the difference between using maximum hamming distance patterns compared to using random patterns, is very small. This parameter gave an inconclusive result and also had an varying effect depending on the dataset.

Using different output compare models

Two different similarity measures, Hamming Distance and Jaccard Similarity Coefficient, were used in order to determine which class the output layer is closest to. Surprisingly, the difference between using the two output compare models showed little differences in our tests.

ECA rule (activation function)

There was no evaluation of how different rules affect the performance of the algorithm. Instead, the ECA rule was kept fixed to rule 110 throughout this paper. This choice was based on the fact that rule 110 is the only ECA rule that has been shown capable of universal computing.

Varying the amount of network layers

In this last test, the best found hyper parameters from the previous tests were used while varying the network layer size between 2-6 layers (See Table 2). The hypothesis was that a larger amount of layers could yield a better performance due to a higher expressive power for the classifier. As can be seen from the results, however, the optimal number of layers changes with the dataset and no clear conclusion can be drawn from this.

5 Results

The comparison between CORPSE and the comparative classifiers (see Table 1) was performed using the following hyper parameter setup found by the hyper parameter experiments:

- Allowing connections across layers
- Using bias nodes
- Using random bit patterns for class comparisons
- Output compare model set to Jaccard similarity coefficient

Results show that CORPSE is performing with an accuracy and kappa statistics similar to the comparative algorithms. In all datasets, CORPSE is outperforming one or more of the comparative algorithm. For dataset 1, CORPSE is even outperforming all of the comparative algorithms. It should be noted, however, that all classifiers used a standard setting and were not setup in any way to fit each dataset. The results for the classifiers in this paper, including CORPSE, should therefore not be considered to be optimal.

Given the results, it is clear that an extended ECA can be used in order to create a computationally inexpensive classifier (RQ1). Allowing neurons to connect across layers showed an improvement in performance, while adding bias nodes sometimes reduced

Table 1: The classification results for the algorithms on the different datasets.

Datasets	**CORPSE**	Naive Bayes	Bayes Net	Multi-layered Perceptron	K-star	J48	Random Forest
1 Breast Cancer	**73.77%**	71.8%	72.08%	70.14%	73.36%	71.86%	69.30%
2 Labour	**78.94%**	85.17%	78.5%	84.37%	81.90%	82.33%	84.30%
3 Vote	**94.71%**	94.71%	94.71%	95.08%	94.71%	95.63%	95.24%
4 Diabetes	**73.82%**	76.39%	75.57%	76.08%	73.52%	74.65%	72.21 %
5 Ionosphere	**84.61%**	86.82%	89.15%	87.56%	89.66%	90.17%	90.06%
6 Supermarket	**65.24%**	63.71%	63.71%	63.71%	63.71%	63.71%	63.71%
7 German-Credit	**73.60%**	73.94%	73.01%	73.5%	72.86%	72.06%	70.97 %
8 Strong and Light	**76.45%**	75.59%	79.23%	77.86%	81.30%	82.41%	81.68%

*Table 2: **CORPSE**: Results of Accuracy / Kappa statistic from Layer2 - Layer6*

Datasets	LAYER2	LAYER3	LAYER4	LAYER5	LAYER6
1 Breast Cancer	73.37%/0.2379	72.72%/0.2446	73.42%/0.2522	70.97%/0.2055	**73.77%/0.2080**
2 Labour	**78.94%/0.5378**	64.91%/0.2297	78.94%/0.5270	64.91%/0.2470	70.17%/0.3377
3 Vote	**94.71%/0.8896**	94.71%/0.8896	94.71%/0.8896	94.71%/0.8896	94.48%/0.8846
4 Diabetes	73.82%/0.3749	72.91%0.3747/	**73.82%/0.3985**	73.43%/0.3933	73.43%/0.3845
5 Ionosphere	**84.61%/0.6610**	84.04%/0.6434	82.33%/0.6121	84.33%0.6504	83.76%/0.6389
6 Supermarket	**65.24%/0.119**	65.16%/0.1181	65.16%/0.1181	65.24%/0.119	65.24%/0.119
7 German-Credit	72.40%/0.2367	**73.6%/0.2731**	73.40%/0.2756	72.70%/0.2573	72.70%/0.2573

performance slightly. No clear conclusion could be drawn from how the number of layers, the output patterns and the output compare models affect the performance. (RQ2).

6 Discussion

In the light of how simplistic the CORPSE algorithm's working principle is, its results in comparison with other more advanced ML algorithms are remarkable. It should also be noted that the performance could be further increased by using other output compare models, bit patterns for classes, or fitness functions. These could also be tailored to fit the underlying classification problem.

Results also show that our different choices of output compare models and classes' bit patterns, had a negligible effect on the results. This result is counter-intuitive but could possibly be explained by that the discriminative ability of each of these choices are similar. E.g. if two sets of points are sufficiently far apart, it makes little difference if one uses a line or a curve to separate them.

Another counter-intuitive result is that using bias nodes actually could degrade the performance. The reason for this might be that the bias nodes could be utilized by the GA to quickly achieve a relatively high fitness, but ending up in a local optima. This reveals a possible flaw in the fitness function since, theoretically, the addition of bias nodes should not degrade the performance. If the fitness function and GA works properly, it should simply disregard connecting neurons to the bias nodes if it is not beneficial for the final solution.

In our results no clear conclusion could be drawn from how the amount of layers affect the performance of the algorithm. A reason for this could be that the fitness function of the GA was too coarse. We can consider an example where there exists two different candidates which both would increase the number of correct classifications by an equal amount. One of these could, however, be closer to the global optima but our fitness function would fail to recognize this and consider them equal. If the numbers of layers are few, the effect of this could be less significant due to the amount of candidates that give rise to equal fitness are potentially fewer. With this in mind, a different fitness function should be designed in order to properly evaluate the possibilities of using deeper network structures.

7 Conclusion

In this paper we present the CORPSE algorithm, a computationally inexpensive classifier that is targeted for resource-constrained devices. We have shown that it has comparable performance to other well known algorithms using eight different datasets. The computational properties of the algorithm are interesting and it is possible that the algorithm could be used also for regression, filtering, or feature extraction. Our experiments also revealed a possible flaw in the fitness function of the genetic algorithm used for training. Future work should therefore primarily focus on designing a better fitness function or

finding alternative training methods. This could possibly improve performance by, for example, better utilizing deeper network structures.

References

[1] Alper Basturk and Enis Gunay. Efficient edge detection in digital images using a cellular neural network optimized by differential evolution algorithm. *Expert Systems with Applications*, 36(2):2645–2650, Mar 2009.

[2] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, Sep 1976.

[3] Veronica Bolon-Canedo, Noelia Sanchez-Marono, and Amparo Alonso-Betanzos. A review of feature selection methods on synthetic data. *Knowledge and information systems*, 34(3):483–519, 2013.

[4] Parimal Pal Chaudhuri. *Additive cellular automata: theory and applications*, volume 1. John Wiley & Sons, 1997.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[6] Leon O Chua and Lin Yang. Cellular neural networks: Applications. *IEEE Transactions on circuits and systems*, 35(10):1273–1290, 1988.

[7] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[8] Paul Crook, Stephen Marsland, Gillian Hayes, Ulrich Nehmzow, et al. A tale of two filters-on-line novelty detection. In *Robotics and Automation, 2002. Proceedings. ICRA 02. IEEE International Conference on*, volume 4, pages 3894–3899. IEEE, 2002.

[9] M. Ercsey-Ravasz, T. Roska, and Z. NÃ©da. Statistical physics on cellular neural network computers. *Physica D Nonlinear Phenomena*, 237(9):1226–1234, Jul 2008.

[10] N. Ganguly, P. Maji, B.K. Sikdar, and P.P. Chaudhuri. Design and characterization of cellular automata based associative memory for pattern recognition. *IEEE Transactions on Systems Man and Cybernetics Part B (Cybernetics)*, 34(1):672–678, Feb 2004.

[11] Anders Hedman, Niklas Karvonen, Josef Hallberg, and Juho Merilahti. Designing ict for health and wellbeing. In *International Workshop on Ambient Assisted Living*, pages 244–251. Springer, 2014.

[12] Jason Kane, Robert Hernandez, and Qing Yang. A reconfigurable multiclass support vector machine architecture for real-time embedded systems classification. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 244–251. IEEE, 2015.

[13] Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg. A computationally inexpensive classifier merging cellular automata and mcp-neurons. In *Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Part II 10*, pages 368–379. Springer, 2016.

[14] Basel Kikhia, Miguel Gomez, Lara Lorna Jiménez, Josef Hallberg, Niklas Karvonen, and Kåre Synnes. Analyzing body movements within the laban effort framework using a single accelerometer. *Sensors*, 14(3):5725–5741, 2014.

[15] Basel Kikhia, Thanos G Stavropoulos, Stelios Andreadis, Niklas Karvonen, Ioannis Kompatsiaris, Stefan Sävenstedt, Marten Pijl, and Catharina Melander. Utilizing a wristband sensor to measure the stress level for people with dementia. *Sensors*, 16(12):1989, 2016.

[16] SÃ¡ndor Kocsardi, Zoltan Nagy, Arpad Csik, and Peter Szolgay. Simulation of 2d inviscid, adiabatic, compressible flows on emulated digital cnn-um. *International Journal of Circuit Theory and Applications*, 37(4):569–585, May 2009.

[17] Chris G Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1-3):12–37, 1990.

[18] Kyong Ho Lee and Naveen Verma. A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals. *IEEE Journal of Solid-State Circuits*, 48(7):1625–1637, 2013.

[19] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[20] Pradipta Maji, Niloy Ganguly, Sourav Saha, Anup K. Roy, and P. Pal Chaudhuri. *Cellular automata machine for pattern recognition*, pages 270–281. Cellular Automata. Springer, 2002.

[21] Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys*, 32(1):80–107, Mar 2000.

[22] D. Smith and P. Stanford. A random walk in hamming space. pages 465–470 vol.2. IEEE, 1990.

[23] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[24] Xie Zhen-zhen and Zhang Su-yu. A non-linear approximation of the sigmoid function based fpga. In *Proceedings of the 2011, International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE2011) November 19–20, 2011, Melbourne, Australia*, pages 125–132. Springer, 2012.

Paper F

Low-Power Classification using FPGA—An Approach based on Cellular Automata, Neural Networks, and Hyperdimensional Computing

Authors:
Niklas Karvonen, Joakim Nilsson, Denis Kleyko, and Lara Lorna Jiménez

Reformatted version of paper originally published in:
The 18th IEEE International Conference On Machine Learning And Applications, 2019, Florida, USA.

Low-Power Classification using FPGA—An Approach based on Cellular Automata, Neural Networks, and Hyperdimensional Computing

Niklas Karvonen, Joakim Nilsson, Denis Kleyko, and Lara Lorna Jiménez

Abstract

Field-Programmable Gate Arrays (FPGA) are hardware components that holds several desirable properties for pervasive computing. They offer hardware implementations of algorithms using parallel computing, which can be used to increase battery life or achieve short response-times. Further, they are re-programmable and can be made small, power-efficient and inexpensive. In this paper we propose a classifier targeted specifically for implementation on FPGAs by using principles from hyperdimensional computing and cellular automata. The proposed algorithm is shown to perform on par with Naive Bayes for two benchmark datasets while also being robust to noise. It is also synthesized to a commercially available off-the-shelf FPGA reaching over 57.1 million classifications per second for a 3 class problem using 40 input features of 8 bits each. The results in this paper show that the proposed classifier could be a viable option for applications demanding low power-consumption, fast real-time responses, or a robustness against post-training noise.

1 Introduction

Wearable computing devices are becoming increasingly popular and are now accepted or even embraced as a part of everyday life. Many of these devices, such as smart watches and bracelets, are equipped with rich sensors that can be used together with machine learning to provide context-awareness such as activity recognition or stress detection [8, 26]. In order for wearable devices to be used in everyday life it is important for them to be unobtrusive. One key factor that impacts unobtrusiveness is the size of the device. While a small-sized device is often desirable, a small size will also affect the possible choices of sensors, battery, and computer hardware on the device. The requirements of a pervasive application being unobtrusive thus affects the choice of hardware, which in its turn will affect the setup for machine learning (M.L). (See Figure (1)). One technology that offers several interesting properties for wearable computing is Field-Programmable Gate Arrays (FPGAs). These are small and inexpensive hardware components that can offer high performance in terms of computations (i.e. short response times or increased battery life) due to their parallel nature. Further, they are also re-programmable which could enable the deployment of different hardware-implementations during runtime. These

powerful properties can be of use when designing unobtrusive pervasive systems.

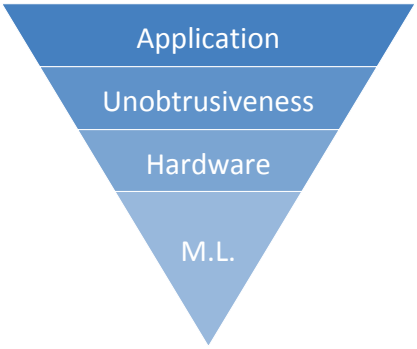

Figure 1: A hierarchical view of requirements for an unobtrusive pervasive application.

In this paper we propose a classifier specifically designed for hardware implementations on FPGAs or Application-Specific Integrated Circuits (ASICs). This work extends our previous efforts [6,7] by combining principles from hyperdimensional computing with Elementary Cellular Automata (ECA) to create a novel classifier named *HyperCorpse*. Hyperdimensional computing has shown promising results in various fields of machine learning [20], and its computations are well-suited for FPGAs. Moreover, its distributed representation of data can provide robustness against noise [21].

Elementary Cellular Automata is an interesting field for our target scope since it holds significant computational power despite its simplicity [1,29]. Also, like hyperdimensional computing, the computations involved in ECA are well-suited for implementations on FPGAs. Given these properties together with previous results from these fields, we pose the following research question:

RQ: Can we create a classifier by mapping input data to a high-dimensional space using Elementary Cellular Automata, and use the resulting high-dimensional vector for distinguishing between classes?

The paper is structured as follows: section 2 reviews related work, section 3 provides some background and describes the proposed classifier, section 4 describes the evaluation of the algorithm, section 5 shows the performance of the algorithm and validates its implementation on an FPGA, section 6 discusses the results, and section 7 concludes the paper and proposes future work.

2 Related work

A number of studies can be found on different ways of using FPGAs for accelerating classifier computation times. Some of these have focused on adapting and implementing conventional classifiers such as Support Vector Machines [17,18], Decision Trees [15], [16], Random forests [27], and K-Nearest Neighbor acceleration [19]. Many of these works, however, only implement parts of the classifiers and also make use of Digital Signal Processor (DSP) Slices such as Multiply Accumulate (MAC) modules or floating point units. One exception to this is the work by Meng et. al [13], who implemented a Naive Bayes classifier on an FPGA for object recognition. Their classifier was evaluated on a binarized MNIST [11] dataset and showed an accuracy of 84.50% while only consuming a small amount of FPGA resources and without the use of any DSP slices.

Implementing conventional machine learning algorithms or other data processing algorithms on FPGAs can be challenging since the parallel nature of an FPGA is fundamentally different compared to the von Neumann architecture for which many of these algorithms were created. If an algorithm is not parallelizable, it will not be able to fully utilize the power of the FPGA. Further, the limited resources available on an FPGA can make floating-point representations not be as feasible compared to more area-efficient numeric representations, such as 16 or 32 bit fixed-point [14]. In the light of this, we turn towards two interesting fields for machine learning on FPGAs: hyperdimensional computing and cellular automata.

Hyperdimensional computing classifiers has shown promising results on a wide variety of problems such as classification of biomedical signals [20, 22], classification of medical images [9], sensory data fusion [23], fault isolation in industrial systems [10], and word embedding [24]. Due to the nature of hyperdimensional computing it is often easy to make large parts of such algorithms run in parallel. Further, their binary representations is typically not bound to fixed amount of bits (e.g. 8, 16, or 32 bits) which allows for a higher flexibility when designing solutions. Another advantage is that the distributed representation of data also can provide some tolerance for noise (e.g. caused by failing components such as memory or sensors).

Elementary Cellular Automata is another interesting field for efficient hardware computations. ECAs use of local neighborhoods make them parallel in nature, which makes them interesting for FPGA implementations. Further, previous works have found them to be capable of performing pattern recognition and acting content-addressable memories [3, 12].

3 Background

Both ECA and Hyperdimensional computing has been used in machine learning and pattern recognition tasks. Further they hold properties that are suitable for the target devices. In this section we begin by providing a brief introduction to ECA and Hyperdimensional computing and how they relate to the proposed classifier. Then the proposed classifier is introduced along with the training procedure used in this paper.

3.1 Elementary Cellular Automata

Elementary Cellular Automata (ECA) consists of binary cells on a lattice. The cells' states are updated by applying a rule where the next state for each cell is calculated by looking at the cell's own state together with its left and right neighborhood, called a configuration (See figure 2). Some ECA are known to work at the edge of chaos and can show complex behavior emerging from seemingly simple rules [29]. An interesting property of ECA is that they do not rely on a global state for making computations which makes them suitable for parallel processing. This could be utilized for reducing power consumption or decreasing real-time response times. Further, since ECA requires only a binary lattice and a set of boolean algebra rules to perform its computations, they allow for efficient implementations on low-level devices such as FPGAs and ASICs.

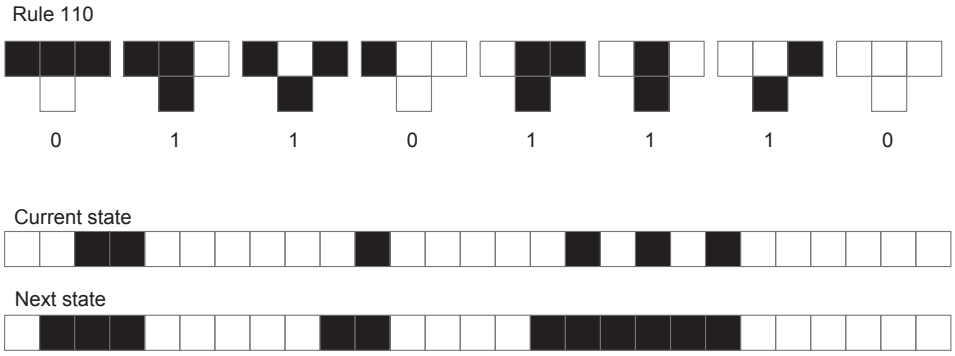

Figure 2: A cellular automata using rule 110 to move from one state to the next.

3.2 Hyperdimensional Computing

Another computational paradigm that uses binary lattices (i.e. binary vectors) and boolean algebra is Hyperdimensional Computing. Hyperdimensional computing [5] is a bio-inspired family of methods for representing concepts (letters, phonemes, features), their meanings and allows implementing sophisticated reasoning based on simple operations. In all general-purpose computers, single bits or small groups of bits can be interpreted without reference to the other bits. Such data representation is called localist. An alternative approach is a distributed representation, in which on the contrary, only the pattern in the total set of bits can be interpreted. Hyperdimensional computing operates with such distributed representations and utilizes statistical properties of representations of data in spaces with very high dimensionality (high-dimensional vectors or HD vectors), which is fixed for a given application. There are several types of distributed representations used for HD computing, e.g. real or complex numbers, but in this paper

we consider only binary representations [5]. Usually, the values of each element of an HD vector are independently equiprobable. The similarity between two binary HD vectors is characterized by the Hamming distance (normalized by the dimensionality), which measures the proportion of elements in which they differ.

3.3 Hypothesis of HyperCorpse

While the proposed classifier uses principles from both ECA and hyperdimensional computing, it is also loosely based on results found in Artificial Neural Networks (ANN). An ANN essentially performs a non-linear mapping from its input-space to a high-dimensional space (the last hidden layer), and in that high dimensional space classes become easier to linearly separate. Supporting this, there are recent studies suggesting that the weights of the output layer does not even need to be tuned [4]. Our hypothesis is that we can use a similar approach using ECA in order to map the input data to a high-dimensional vector (HD vector). ECA thus serves as a nonlinear mapping from input-space to a high-dimensional binary space, where classes then can be separated. In a binary high-dimensional space, a vast amount of the HD vectors are quasi-orthogonal, which means that the Hamming distances from any arbitrary chosen HD vector to more than 99.99% of all other HD vectors in the high-dimensional space are concentrated around 0.5 normalized Hamming distance. This implies that two random HD vectors are very unlikely to be similar by chance. This is utilized in our algorithm by comparing the resulting HD vector from the ECA mapping to that of random class vectors of the same dimensionality. The comparison between HD vectors is performed using Hamming Distance.

Since hamming distance is used as a similarity metric in a high-dimensional space, we need a way to preserve distance relationships between points in input space when we move to binary high-dimensional space. Our approach to this is to scatter code [25] the input data before mapping it to the output HD vector. Using scatter coding has benefits compared to other codes (e.g., thermometer coding), since it provides dense HD vectors offering a higher resolution using fewer bits, but at the trade-off of only preserving distances within a certain radius with a limited resolution. This radius is determined by the amount of bits flipped between consecutive numbers divided by the total amount of bits in the vector. Due to the use of hamming distance comparisons in our algorithm, we controlled the density of the first scatter code as well as the density of the random class vectors to have a balance of ones and zeros.

3.4 HyperCorpse Algorithm

The different operations of HyperCorpse are very simple. In order to map the scatter coded input data onto the output HD vector, we use ECA and rule 90. The mapping is performed as follows: Each cell in the output layer is connected to three cells in the input layer. In order to set the state for an output cell, the ECA rule 90 is applied to the configuration of the input cells it is connected to in a lexicographical order (See figure 3). The set of connections between the input and output layer is thus the most significant

part of the model. Once all the output cells are updated, the resulting output HD vector is compared to a set of random class HD vectors where each represents a class. The closest match in terms of hamming distance represents the classification result.

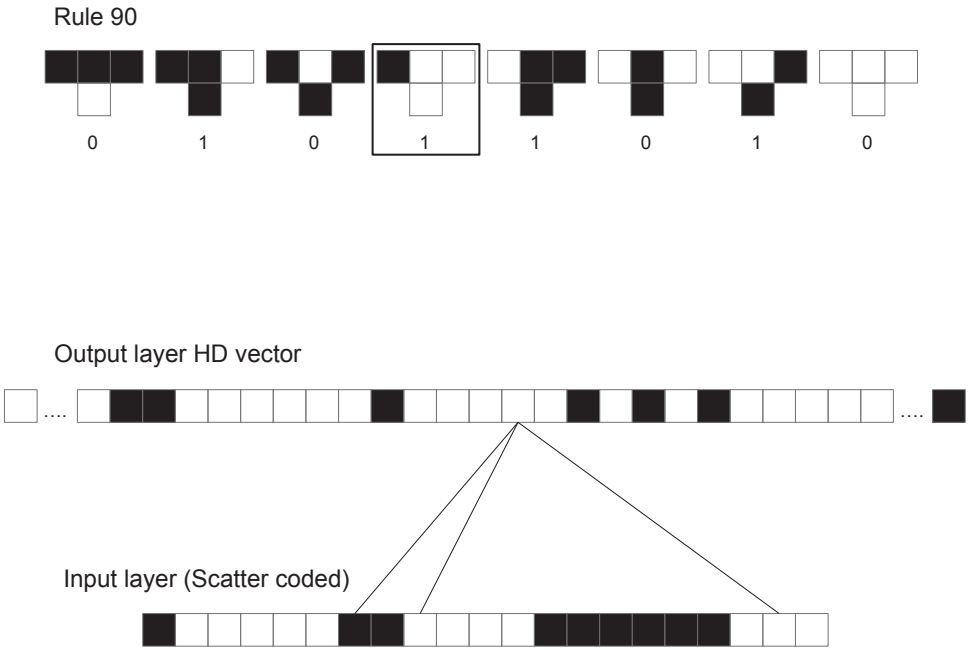

Figure 3: The HyperCorpse algorithm updating an output cell. Each cell has 3 connections to the input layer and the set of all connections make up the model.

3.5 Training

In order to train the model we try to find the best set of connections between input and output layer. This training is performed using a Genetic Algorithm where each individual represents a set of connections between output cells and input cells. The fitness of an individual is the total amount of correct classifications the individual's network gets on the training set. Regeneration is performed using tournament selection, and elitism is used to prevent degeneration. The least fitted individuals are culled and replaced with random individuals. Uniform crossover is performed by randomly picking each connection from one of the two parents and assigning it to the offspring. If an individual is chosen to

mutate, one of its cells connections will be randomly changed. Training was performed using randomly selected batches of instances for each generation (i.e. every generation the individuals' fitness was determined using a fixed amount of randomly selected training examples). This was performed in order to speed up the training process of the algorithm. The settings for the genetic algorithms for both data sets were:

- Population size: 100
- Elites: 15
- Culled individuals: 5
- Tournament size: 5
- Crossover: Uniform
- Mutation rate: 70% chance of mutation

4 Evaluation

In order to evaluate the proposed classifier, it was compared to a Naive Bayes classifier on two benchmark datasets, MNIST [11] and Waveform5000 [2]. Both datasets are publicly available and used in several other works which allows for comparing the results in this paper to other research. In our evaluation, MNIST served as a benchmark to assess HyperCorpse's predictive performance compared to the baseline, while Waveform5000 was used to assess the algorithm's performance from a resource-constrained scenario.

4.1 MNIST

For MNIST the input data was used in its original format when scatter coded (i.e. pixel values between 0-255). Only 8 bits were used for the scatter coding with 4 bits flipped between consecutive codes (note that for scatter coding this is a small bit size). This choice was made in order to obtain an equally sized bit representation as the one required by the baseline. The output layer size (HD vector dimensionality) was set to 1000 bits. When noise was added (See section 4.4) it was added to the original dataset prior to scatter coding. Noise was added by randomly flipping one bit in the 8 bit features. The supplied test set was used for evaluation and the full training set was used for training. Batches of 500 randomly chosen examples were used in each generation of the training.

4.2 Waveform5000

Before Waveform5000 was scatter coded, it was normalized to have values between 0-1 and then rounded to only include 1 decimal. Also here, batches of 500 randomly chosen examples were used in each generation of the training. The output layer size was set to 500 bits, and each of the 40 features was scatter coded using 8 bits with 4 bit flips

between consecutive codes. These choices were made to evaluate a resource-constrained scenario where sensor resolution and computational resources were low. Since the dataset only contains 5000 labelled examples, a 10-fold cross-validation was performed in order to evaluate the performance. The algorithm was synthesized for hardware implementation using Verilog and Xilinx ISE (version 14.7). This allowed us to verify that the algorithm does not pose any architectural challenges for FPGAs. The target device was a commercially available off-the-shelf FPGA, Xilinx Spartan 3E. This choice was based on that this device is popular and does not offer any extended functionality (e.g. hardware floating-point units, Multiply-Accumulate etc.).

4.3 Baseline classifier

We chose to evaluate the HyperCorpse algorithm by comparing its performance with that of Naive Bayes. Choosing Naive Bayes as the baseline was based on the following:

1. Naive Bayes has, despite its simplicity, achieved good results for a wide variety of machine learning problems

2. It requires only small resources for performing the classifications once trained, making it a suitable baseline for resource-constrained comparisons

3. Naive Bayes has few hyperparameters, thus minimizing the risk of skewing the results by poor hyperparameter optimization

4. It can perform well even with smaller datasets, making the study less sensitive for skewed results due to insufficient dataset size

The proposed classifier was implemented in Java as a WEKA framework [28] classifier (version 3.9.2). All results presented in terms of machine learning, for both the baseline and HyperCorpse, was thus obtained from running the classifiers in WEKA.

4.4 Sensitivity to post-training noise

Apart from evaluating the classifiers based on accuracy, they were also evaluated when exposed to noisy input data after training, here called post-training noise. Post-training noise could happen, for example, if memory or sensors fail. Also, in small hardware implementations unintentional bit flips can occur due to e.g. inductive or capacitive crosstalk, by cosmic rays, or by Alpha particles from package decay. Checksums and error-correcting codes can be used to remedy this but at the cost of increased computational load (depending on the tolerance for errors). In order to evaluate the classifiers' performances when exposed to post-training noise, five noisy versions of the MNIST dataset was created. These were created by having each bit in the input data being flipped with a certain probability (i.e. 0.1%, 0.5%, 1.0%, 5.0%, 10.0%).

Table 1: Classifier performance

Dataset	Classifier	Accuracy
MNIST	Naive Bayes	69.65%
MNIST	HyperCorpse	74.06%
Waveform	Naive Bayes	79.58%
Waveform	HyperCorpse	78.88%

Table 2: HyperCorpse confusion matrix for MNIST

a	b	c	d	e	f	g	h	i	j	class
883	1	7	8	13	0	34	8	25	1	a=0
0	1052	6	13	8	0	8	1	45	2	b=1
27	34	768	22	35	0	46	23	64	13	c=2
21	19	50	779	18	0	15	17	62	29	d=3
4	6	1	5	826	0	39	7	23	71	e=4
110	50	18	283	83	0	44	50	208	46	f=5
31	13	25	4	43	1	803	1	28	9	g=6
4	35	24	4	27	0	5	846	18	65	h=7
17	25	31	59	21	0	16	15	758	32	i=8
15	6	12	18	146	0	9	57	55	691	j=9

5 Results

Our comparison of the HyperCorpse algorithm to the baseline shows that the proposed algorithm's performance is on par with the baseline in terms of accuracy for both the evaluated datasets (see Table 1). The synthesis of the algorithm in Verilog verifies that it can be implemented on the target Xilinx Spartan 3E device (See Table 4), and the algorithm achieves a maximum classification rate of over 57.1 million classifications per second at an accuracy of 78.88% for the Waveform problem. Further, the algorithm shows indications of being resilient to noisy input data caused by random bit-flips. Even with high amounts of noise, 10% chance of bit flip for each feature, HyperCorpse's performance only drops 10.89% in accuracy while the baseline's accuracy drops with 37.76% at this noise level.

Table 3: Classification accuracies on MNIST with post-training noise

	0.1%	0.5%	1.0%	5.0%	10.0%
Naive Bayes	68.12%	58.3%	42.89%	13.59%	10.3%
HyperCorpse	73.41%	70.8%	67.18%	41.28%	23.13%

Table 4: HyperCorpse FPGA synthesis to Xilinx Spartan 3E

```
Logic Utilization:
  Number of Slice Flip Flops:
    516 out of      9,312    5%
  Number of 4 input LUTs:
    5,345 out of    9,312   57%
Logic Distribution:
  Number of occupied Slices:
    3,082 out of   4,656   66%
    Number of Slices containing only related logic:
        3,082 out of   3,082 100%
    Number of Slices containing unrelated logic:
        0 out of   3,082   0%
  Total Number of 4 input LUTs:
    5,351 out of   9,312   57%
    Number used as logic:
        5,345
    Number used as a route-thru:
        6

  Number of bonded IOBs:
    18 out of      66   27%
    IOB Flip Flops:
        2
  Number of BUFGMUXs:
    1 out of      24    4%

Design statistics:
   Minimum period:  17.485ns
   (Maximum frequency:  57.192MHz)
```

6 Discussion

The HyperCorpse algorithm performs on par with Naive Bayes for the MNIST classification problem. Both these algorithms, however, are far behind the state-of-the-art for

MNIST where classifiers are achieving over 99 % accuracy. This is however achieved using more advanced algorithms and by applying different pre-processing steps. Perhaps a more interesting comparison is to that of Lecun et al. [11]. They showed that a simple linear model, where each input pixel value (and a bias) contributes to a weighted sum for each output unit and the highest-valued output is selected, performs at 88% accuracy. This is significantly higher than both the classifiers in this paper. By looking at HyperCorpse's confusion matrix for the MNIST problem, however, it can be observed that out of the 10,000 test examples, the character "5" is only predicted once (also erroneously). Most of character "5" examples gets classified as character "6". This error significantly reduces the classifier's performance and should be further investigated. Possible reasons for this include using an unfortunate scatter coding or that the genetic algorithm get stuck in a local optima.

The Waveform5000 case shows that the classifier could be useful for problems requiring a high classification throughput, or for problems requiring a low power consumption. The latter could be achieved by reducing the clock speed of the FPGA (e.g. from 57.2 MHz down to 1 Hz) which could result in a low energy cost for classifications.

Further, HyperCorpse outperforms the baseline for all tests where noise is added. This could make HyperCorpse an interesting choice where computational resources are scarce and noise-levels are expected to be high, e.g. in space applications.

Another interesting and less obvious advantage of the algorithm is that the scatter coding of the input allows for custom encoding of context. Due to its distributed representation of data it is possible to utilize operations like binding and bundling in hyperdimensional computing to create rich input features that can be used for classification or to make other interesting predictions that are not straight-forward to implement on most conventional machine learning algorithms, e.g. "What is the dollar of mexico?" [5].

While HyperCorpse has advantages, it also has drawbacks. Scatter coding the input can be problematic for some real-world implementations since it makes the algorithm not being compatible with common existing analog-to-digital converters (AD converters) like other algorithms (e.g. Naive Bayes). The scatter coding step could, however, be implemented on the FPGA before the classification step, but at the cost of further FPGA resources.

In this paper, the algorithm is trained using a genetic algorithm which is has several drawbacks. Genetic algorithms are susceptible to being stuck in local optima and have non-deterministic behavior due to their random nature. This means that there are no guarantees neither for the training performance nor for the training time. While genetic algorithms can be run in parallel, the authors still believe that more efficient ways of training the algorithm should be investigated.

7 Conclusions and Future Work

In this paper a novel classifier was proposed that is targeted for implementation on resource-constrained devices. The results show that the classifier performs on par with Naive Bayes for the two datasets while outperforming the baseline for noisy input data.

Results revealed possible problems with the training algorithm since one class of the MNIST problem was virtually left out. Future work should focus on improving the classification performance of the algorithm, possibly by improving the training of the algorithm. When synthesized to a commercially available off-the-shelf FPGA, Xilinx Spartan 3E, the classifier was found capable of performing over 57.1 million classifications per second at an accuracy of 78.88% for a compressed version of the Waveform5000 dataset. For the target device this consumed 57% of the Look-Up-Tables and 5% of the Flip-flops when using 40 8-bit input features. The results in this paper show that the proposed classifier could be a viable option for applications demanding low power-consumption, fast real-time responses, or a robustness against post-training noise.

References

[1] Parimal Pal Chaudhuri, Dipanwita Roy Chowdhury, Sukumar Nandi, and Santanu Chattopadhyay. *Additive cellular automata: theory and applications*, volume 1. John Wiley & Sons, 1997.

[2] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.

[3] Niloy Ganguly, P Pal Chaudhuri, Biplab K Sikdar, and Pradipta Maji. Design and characterization of cellular automata based associative memory for pattern recognition. 2004.

[4] Elad Hoffer, Itay Hubara, and Daniel Soudry. Fix your classifier: the marginal value of training the last weight layer. *arXiv preprint arXiv:1801.04540*, 2018.

[5] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.

[6] Niklas Karvonen, Lara Lorna Jimenez, Miguel Gomez Simon, Joakim Nilsson, Basel Kikhia, and Josef Hallberg. Classifier optimized for resource-constrained pervasive systems and energy-efficiency. In *International Journal of Computational Intelligence Systems*, volume 10, pages 1272–1279. Atlantis Press, 2017.

[7] Niklas Karvonen, Basel Kikhia, Lara Lorna Jiménez, Miguel Gómez Simón, and Josef Hallberg. A computationally inexpensive classifier merging cellular automata and mcp-neurons. In *Ubiquitous Computing and Ambient Intelligence*, pages 368–379. Springer, 2016.

[8] Basel Kikhia, Thanos G Stavropoulos, Stelios Andreadis, Niklas Karvonen, Ioannis Kompatsiaris, Stefan Sävenstedt, Marten Pijl, and Catharina Melander. Utilizing a wristband sensor to measure the stress level for people with dementia. *Sensors*, 16(12):1989, 2016.

[9] Denis Kleyko, Sumeer Khan, Evgeny Osipov, and Suet-Peng Yong. Modality classification of medical images with distributed representations based on cellular automata reservoir computing. In *Biomedical Imaging (ISBI 2017), 2017 IEEE 14th International Symposium on*, pages 1053–1056. IEEE, 2017.

[10] Denis Kleyko, Evgeny Osipov, Nikolaos Papakonstantinou, and Valeriy Vyatkin. Hyperdimensional computing in industrial systems: the use-case of distributed fault isolation in a power plant. *IEEE Access*, 2018.

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] Pradipta Maji, Niloy Ganguly, Sourav Saha, Anup K Roy, and P Pal Chaudhuri. Cellular automata machine for pattern recognition. In *International Conference on Cellular Automata*, pages 270–281. Springer, 2002.

[13] Hongying Meng, Kofi Appiah, Andrew Hunter, and Patrick Dickinson. Fpga implementation of naive bayes classifier for visual object recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 123–128. IEEE, 2011.

[14] Medhat Moussa, Shawki Areibi, and Kristian Nichols. On the arithmetic precision for implementing back-propagation networks on fpga: a case study. In *FPGA Implementations of Neural Networks*, pages 37–61. Springer, 2006.

[15] Ramanathan Narayanan, Daniel Honbo, Gokhan Memik, Alok Choudhary, and Joseph Zambreno. Interactive presentation: An fpga implementation of decision tree classification. In *Proceedings of the conference on Design, automation and test in Europe*, pages 189–194. EDA Consortium, 2007.

[16] Jason Oberg, Ken Eguro, Ray Bittner, and Alessandro Forin. Random decision tree body part recognition using fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 330–337. IEEE, 2012.

[17] Markos Papadonikolakis and Christos-Savvas Bouganis. A novel fpga-based svm classifier. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 283–286. IEEE, 2010.

[18] Markos Papadonikolakis and Christos-Savvas Bouganis. Novel cascade fpga accelerator for support vector machines classification. *IEEE transactions on neural networks and learning systems*, 23(7):1040–1052, 2012.

[19] Yuliang Pu, Jun Peng, Letian Huang, and John Chen. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 167–170. IEEE, 2015.

[20] Abbas Rahimi, Simone Benatti, Pentti Kanerva, Luca Benini, and Jan M Rabaey. Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition. In *Rebooting Computing (ICRC), IEEE International Conference on*, pages 1–8. IEEE, 2016.

[21] Abbas Rahimi, Sohum Datta, Denis Kleyko, Edward Paxon Frady, Bruno Olshausen, Pentti Kanerva, and Jan M Rabaey. High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2508–2521, 2017.

[22] Abbas Rahimi, Artiom Tchouprina, Pentti Kanerva, José del R Millán, and Jan M Rabaey. Hyperdimensional computing for blind and one-shot classification of eeg error-related potentials. *Mobile Networks and Applications*, pages 1–12, 2017.

[23] Okko Rasanen and Sofoklis Kakouros. Modeling dependencies in multiple parallel data streams with hyperdimensional computing. *IEEE Signal Processing Letters*, 21(7):899–903, 2014.

[24] Gabriel Recchia, Magnus Sahlgren, Pentti Kanerva, and Michael N Jones. Encoding sequential information in semantic space models: Comparing holographic reduced representation and random permutation. *Computational intelligence and neuroscience*, 2015:58, 2015.

[25] Derek Smith and Paul Stanford. A random walk in hamming space. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 465–470. IEEE, 1990.

[26] Jaakko Suutala, Susanna Pirttikangas, and Juha Röning. Discriminative temporal smoothing for activity recognition from wearable sensors. In *International Symposium on Ubiquitious Computing Systems*, pages 182–195. Springer, 2007.

[27] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 232–239. IEEE, 2012.

[28] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[29] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, IL, 2002.