Mälardalen University Licentiate Thesis 309

Persistent Fault-tolerant Storage at the Fog Layer

Zeinab Bakhshi

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Licentiate Theses
No. 309

PERSISTENT FAULT-TOLERANT STORAGE AT THE FOG LAYER

Zeinab Bakhshi Valojerdi

2021

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISBN 978-91-7485-518-0
ISSN 1651-9256
Printed by E-Print AB, Stockholm, Sweden

Abstract

Clouds are powerful computer centers that provide computing and storage facilities that can be remotely accessed. The flexibility and cost-efficiency offered by clouds have made them very popular for business and web applications. The use of clouds is now being extended to safety-critical applications such as factories. However, cloud services do not provide time predictability which creates a hassle for such time-sensitive applications. Moreover, delays in the data communication between clouds and the devices the clouds control are unpredictable. Therefore, to increase predictability, an intermediate layer between devices and the cloud is introduced. This layer, the Fog layer, aims to provide computational resources closer to the edge of the network. However, the fog computing paradigm relies on resource-constrained nodes, creating new potential challenges in resource management, scalability, and reliability. Solutions such as lightweight virtualization technologies can be leveraged for solving the dichotomy between performance and reliability in fog computing. In this context, container-based virtualization is a key technology providing lightweight virtualization for cloud computing that can be applied in fog computing as well. Such container-based technologies provide fault tolerance mechanisms that improve the reliability and availability of application execution. By the study of a robotic use-case, we have realized that persistent data storage for stateful applications at the fog layer is particularly important. In addition, we identified the need to enhance the current container orchestration solution to fit fog applications executing in container-based architectures. In this thesis, we identify open challenges in achieving dependable fog platforms. Among these, we focus particularly on scalable, lightweight virtualization, auto-recovery, and re-integration solutions after failures in fog applications and nodes. We implement a testbed to deploy our use-case on a container-based fog platform and investigate the fulfillment of key dependability requirements. We enhance the architecture and identify the lack of persistent storage for stateful applications as an important impediment for the execution of control applications. We propose a solution for persistent fault-tolerant storage at the fog layer, which dissociates storage from applications to reduce application load and separates the concern of distributed storage. Our solution includes a replicated data structure supported by a consensus protocol that ensures distributed data consistency and fault tolerance in case of node failures. Finally, we use the UPPAAL verification tool to model and verify the fault tolerance and consistency of our solution.

Sammanfattning

Moln är kraftfulla datacenter som tillhandahåller beräknings- och lagringskapacitet som kan nås på distans. Molnens flexibilitet och kostnadseffektivitet har gjort dem mycket populära för affärs- och webbtillämpningar. När användningen av moln nu utökas till säkerhetskritiska tillämpningar såsom fabriker, uppstår problem eftersom molntjänster inte är tillräcklig förutsägbara tidsmässigt för den typen av tillämpningar. För att råda bot på detta införs ett mellanlager mellan enheterna och molnet. Detta lager, som kallas dim-lagret (Eng. ”the fog layer”), tillhandahåller beräkningsresurser närmare kanten av nätverket. Dim-paradigmet bygger dock på resursbegränsade noder, vilket skapar nya utmaningar relaterat till resurshantering, skalbarhet och tillförlitlighet.

Virtualisering kan användas för att minska motsättningen mellan prestanda och tillförlitlighet i dim-lagret. Här är containerbaserad virtualisering en nyckelteknik med ursprung i molntjänster som också kan användas i dim-lagret. Sådana tekniker tillhandahåller feltoleransmekanismer som förbättrar tillförlitlighet och tillgänglighet. Genom en fallstudie har vi identifierat att datalagring anpassad till tillämpningar där tillståndet måste sparas mellan exekveringarna är särskilt viktig för säkerhetskritiska tillämpningar. Dessutom identifierade vi behovet av förbättrad containerhantering, anpassad till dim-tillämpningar i containerbaserade arkitekturer. Bland dessa utmaningar för pålitliga dim-plattformar fokuserar vi särskilt på skalbara virtualiserings-, återställnings- och återintegreringslösningar efter fel i dim-tillämpningar och dim-noder. Vi implementerar en testbädd för vår fallstudie på en containerbaserad dim-plattform och undersöker om viktiga pålitlighetkrav är uppfyllda. Vi förbättrar arkitekturen och identifierar brister i hanteringen av tillämpningar där tillståndet måste sparas mellan exekveringarna. Vi föreslår även en lösning för feltolerant lagring av sådan tillståndsinformation i dim-skiktet. Lösningen separerar lagringen från tillämpningarna för att minska belastningen på tillämpningarna och göra dessa oberoende av den distribuerade lagringen. Vi använder en replikerad datastruktur som med stöd av ett konsensusprotokoll säkerställer distribuerad datakonsistens och feltolerans vid nodfel. Slutligen använder vi verifieringsverktyget UPPAAL för att modellera och verifiera lösningens feltolerans och konsistens.

To my mother ♡
To my late father ♥

Acknowledgments

There were many moments from the time I started my studies at Mälardalen University that the simple phrase “Thank you” was just not enough to show my overwhelming gratitude to those who helped and supported me unconditionally in my studies and still do. As such, I have eagerly been waiting to write this part of my thesis.

Starting with Hans Hansson, my main supervisor. He is the smartest person I have ever met who not only taught me to think outside of the box, but also to think as if there are no boxes or boundaries. I did not expect to be as lucky as I am now in our first interview when I met him online. However, I won my own personal lottery when I got the chance to work with him closely as his Ph.D. student. I believe my mom had prayed a lot for me in the background.

I can never find a better mentor than Guillermo Rodriguez-Navas. He is intelligent and perceptive, friendly, and continues to give me endless support. He has opened new doors in front of me each time we talked. Guillermo taught me to be confident and never fear making mistakes. I will never forget that even after he could have left this project when he got a full-time job in the industry, he continued to support and supervise me alongside his other responsibilities at work and at home as a father. I hope I was not too much of a burden.

Great insights and feedbacks are always received from Sasikumar Punnekkat. I am very grateful to him for finding time to provide his invaluable comments although he is the busiest person in the department.

Very special gratitude goes out to the one who brings the art to my life, Komeil Asli that has brought beauty to my papers and presentations by his unique designs.

I have nothing but love from the bottom of my heart for Maryam Kabiri, my mother who devoted her dreams so that her children can catch theirs. She has always provided me with moral and emotional support in all aspects of my life.

My brother, Mohammad Bakhshi is another special member of my family, who has always trusted and believed in me.
This work is supported and funded by the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 764785 FORA. I would like to thank all FORA members for their time and valuable input in our meetings.
I am also sincerely very grateful to all of you who have mentored me, have been patient with me, loved me unconditionally, and encouraged me to become the person I am today and continue to become. In no particular order, Elisabeth Uhlemann, Stephen Fitzpatrick, Carola Ryttersson, Jenny Hägglund, Malin Åshuvud, Melika Hozhabri, Soheila Sheikh Bahaei, Shahriar Hasan, Václav Struhár, Salman Shaik, Nitin Desai, Lan Van Dao, Taufik Akbar, Radu Dobrin, Gunnar Widforss, Patrick Denzler, Aida Causevic, Robbert Jongeling, Arash Ghareh Baghi, Ali Balador, Mohammad Loni, Bahram Zarrin, Nandinbaatar Tsog, Illiar Rabet, Hossein Fotouhi, Bahareh Agha Jafari, Sara Abbaspour Asadollah, Daniel Bujosa Mateu, Ali Zoljodi, Ramon Riazati, Ali Sedaghatbaf, Jean Malm, Leo Hatvani, Mahdi Momeni, Johan Relefors, Mimi Axelsson, Jan Carlson, Leila Bakhshi, Fateme Bakhshi, Hooman Khoshvaghti, Monadi Naderi, Nikoo Lajevardi, Shaik Shabbir, and Illi Aliu.

Zeinab Bakhshi, Västerås, October 2021

List of publications

Papers included in the thesis[1]

Paper A *Dependable Fog Computing: A Systematic Literature Review*, Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.
In Proceeding of 45thEuromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, IEEE. Chalkidiki, Greece, September 2018.

Paper B *A preliminary Roadmap for Dependability Research in Fog Computing*, Zeinab Bakhshi, Guillermo Rodriguez-Navas.
In 17th International Workshop on Real-Time Networks – RTN 2019, ACM, In conjunction with the 31st ECRTS 2019.

Paper C *Fault-tolerant Permanent Storage for Container-based Fog Architectures*, Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.
In 22nd IEEE International Conference on Industrial Technologies - ICIT 2021.

Paper D *Using UPPAAL to Verify Recovery in a Fault-tolerant Mechanism Providing Persistent State at the Edge*, Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.
In 26th International Conference on Emerging Technologies and Factory Automation, ETFA 2021, Västerås, Sweden

[1]The included articles have been reformatted to comply with the thesis layout.

Additional papers, not included in the thesis

1. *Cost-Aware Task Scheduling in Fog-Cloud Environment*,Tina Samizadeh Nikoui, Ali Balador, Amir Masoud Rahmani, Zeinab Bakhshi.
   In CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST) 2020, IEEE.

2. *An Overview on Security and Privacy Challenges and Their Solutions in Fog-Based Vehicular Application*, Zeinab Bakhshi, Ali Balador.
   In 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops) 2019, IEEE.

3. *Enabling Fog-based Industrial Robotics Systems*, Mohammed Salman Shaik, Václav Struhár, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, Gerhard Fohler.
   In 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2020), 2020, IEEE

Contents

I

Thesis

Chapter 1

Introduction

The expansion of cloud services has led to an increase in the number of cloud users and variety of uses of clouds, from web applications to Cyber Physical Systems (CPS), including safety-critical applications. Many CPS applications manage data collection from devices at the edge of the network and send the collected data to clouds to be stored and analyzed for further decisions [1]. However, in many safety-critical applications like ”fire detection” from raw data, or collaborating industrial robots, it is necessary to have local computation and decision making closer to the network edge to reduce latencies and jitter related to network communication and cloud computation, as well as to increase reliability [2].

In some other cases there might be sufficient bandwidth and processing time to send data to the cloud while the edge devices suffer from limited resources to do the computation. In these cases, it is required to offload some computation and data storage tasks to the cloud or more powerful resources nearby known as “fog computing”. Such decisions to send data to the clouds or providing resources close to the edge of the network is application dependent based on the application requirements and available resources at the edge, fog and cloud layer. With the growth of Industrial Internet of Things (IIoT) applications, it is required to consider both fog and cloud computations for applications like industrial robotics. However, this brings some issues: (1) Portability and management of applications and processes between fog and cloud; (2) In some cases, it is required to provide computations only at the fog layer to meet processing deadlines and save the bandwidth for other sensors, devices and applications at the edge, in this case, data computation and data storage must be guaranteed at the fog layer [3]; and (3) Failure of applications and nodes at the fog layer with smaller capacities, and consequently less redundancy than the cloud, might result in data loss. It is therefore required to consider local fault-tolerance mechanisms at the edge of the network. Using containerization and container orchestration solutions can solve these issues by providing portable, scalable, and automatic application deployment to move between both the fog and the cloud layers. Containerization also saves a lot of resources at the resource constrained fog platforms due to its lightweight characteristic. Using cloud native container orchestration solutions, like Kubernetes [4], also provides automatic healing for its managed containers, that is, restarting the failed containers and replacing or rescheduling them when they or their hosts fail [4]. In addition to these recovery actions that naturally improve the availability of the containerized applications deployed with Kubernetes, providing redundancy mechanism by replicating applications remains the most important feature provided by Kubernetes to improve application availability [5].

In stateless applications, like web applications, the use of Kubernetes is expanding, since it brings high availability by its built-in reliability features, like idempotency, replication, auto recovery and hot swapping mechanisms [6].

Replication and hot swapping of stateless applications can be performed easily as they can be deployed as interchangeable instances. However, the same is not applicable for stateful applications. There are two important issues when it comes to deploying stateful applications using Kubernetes; (1) the state of the failed container is not restored, and (2) the behaviour of a stateful application is dependent on its current state and therefore, redeploying a new instance without providing the state of the instance it is replacing could lead to unpredictable, or even dangerous, behaviour. Hence, a synchronization mechanism is required to coordinate the different replicas when redeploying stateful applications.

Deploying stateful applications using Kubernetes StatefulSet [4] or leveraging other methods like customizing deployment controller [5] or even using third party solutions like Ceph storage [7] are among few approaches to solve the statefull application deployment issue. However, they are all implemented in the cloud and the states of the stateful applications are always kept outside the cluster in these approaches. In addition, consistency between replicas in stateful applications remains a research challenge in the proposed solutions that focus on stateful application deployment. Therefore, it is not possible to directly apply these solutions for stateful fog applications that require local or close to the edge processing and storage capacities.

Problem. This thesis is motivated by the identified problems in our study of a robotics use-case where containerizing and deploying a robotic application in a container-based fog architecture showed the critical need of managing states using fault-tolerant persistent storage for stateful applications in container-based solutions. As discussed, the cloud native container-based solution Kubernetes, suffers from certain issues when it comes to deploying stateful applications. In particular, stateful applications require persistent storage. A suitable storage system for fog must be co-located with the nodes at the fog layer and fulfill data consistency requirements when it is leveraged in distributed fog platforms which is the basis of the specific problems addressed in this thesis.

Summary of Contributions. This thesis is a collection of four research publications, forming four contributions. In the first contribution (C1), we identify open challenges related to dependability of fog computing. In the second contribution (C2), we focus on some of the identified challenges in C1. In particular, we focus on enhancement of a light weight fog platform by means of containerization that support self-healing, portability and scalability of fog application in a robotics use-case. In addition, we examine the suitability of the container-based fog platform and identify the limitations we face using the volatile storage for stateful applications. In the third contribution (C3), we provide solutions that overcome the volatile storage issue by proposing a fault-tolerance distributed persistent storage that provides: (1) Dissociation of data storage from application processes by introducing separate storage containers; (2) A replicated data structure to increase data availability and to provide fault-tolerant persistent storage in each node; and (3) Data consistency by adapting a consensus protocol to keep distributed replicated data structures synchronized. In the forth contribution (C4), we verify the proposed solution for fault-tolerance persistent distributed storage for fog stateful applications using the UPPAAL [8] verification tool.

Thesis Outline. This thesis is divided into two parts. Part I provides an overview of the thesis and is organized as follows. Chapter 1 introduces the main challenge, Chapter 2 provides an overview of the research process pursued, and Chapter 3 discusses the background and related work to this thesis. Chapter 4, gives an overview of the included papers and the contributions. Part I ends with Chapter 5, that concludes the thesis with a discussion on the current work and planned work for the doctoral dissertation. Part II includes the collection of included papers, reformatted to comply with the thesis layout.

Chapter 2

Research Overview

In this chapter, we present the overall goals of the thesis, the research process followed for achieving these goals, and the research methods used to realize the research goals.

2.1 Context and Research Goals

This research is mainly carried out in the context of dependability of fog computing platforms and targets design of fault-tolerance mechanism in the fog infrastructure prior to deploying applications on fog platforms. Fog computing is designed to carry out the processing and storage close to the edge devices. Traditional methods for achieving fault-tolerance and dependability include implementing redundancy techniques and providing replicas at different levels of a system. These methods have been used in both embedded systems and powerful cloud infrastructures. However, computational resources at the fog layer are constrained, therefore, the resources must be managed to serve all the requests from applications which in our context are assumed to be safety-critical applications. Resource limitations at the fog layer is an obstacle in applying the traditional redundancy techniques in which replicated components could take the place of failed ones. Therefore, we propose to enhance the existing dependability mechanisms and formulate our first research goal as follows:

Research Goal 1 (RG1): Identification of challenges and solutions related to dependability aspects of fog computing in the current state of the art.

RG1 also aims at finding novel dependability and fault-tolerance mechanisms to leverage on fog platforms. In addition, it focuses on identifying the challenges of applying traditional dependability techniques in fog computing and finding gaps between current solutions and fog platform requirements.

In our journey to achieve RG1, we identify the current challenges and solutions on fog computing dependability. We study the traditional dependability notions [9] and compare them with the current dependability requirements and fault-tolerance solutions. As a result, we figured out that there are two new dependability requirements, namely, Scalability and Quality of Service (QoS) which need to be considered while designing a dependable fog infrastructure. We also noticed that the most common solutions for fulfilling reliability and availability of a fog system are by redundancy. The source of threats (fault, error and failure) are identified as fog nodes, communication channels and application deployments. Our study also provided useful information about the topics that attracted more attentions in the research towards dependability aspects of fog computing. We noticed that the discussions are mainly about the trade-off between resource utilization and fault tolerance, the use of redundancy methods to increase availability, and lastly, the trade-off between reliability and timeliness, particularly for node replication schemes.

As a result of our study, we also identified several open challenges considering the fog platform and applications requirements together with the current solutions in the state of the art.

Different methods for fault detection, fault-tolerance, fault prevention and fault diagnostics have been investigated. However, in a long-lived system like the fog, there is also a need to develop methods that allow faulty components to recover and be reintegrated in the system operation. This can prevent system failure or shut down caused by fast redundancy attrition. Therefore, we identified a lack of study on fault-recovery and re-integration after failure.

The re-integration and fault-recovery of a system when a failure occurs, considering the limitations and requirements of fog platforms, brings the idea of leveraging self-healing mechanism and auto recovery techniques. Therefore, finding self-healing solutions to adapt in fog platforms becomes a key activity in the context of our research. This leads to the definition of our second research goal as:

Research Goal 2 (RG2): Design of a fault-tolerance fog architecture by leveraging auto-recovery and self-healing mechanisms.

RG2 mainly aims at reuse of current self-healing solutions in fault-tolerance mechanism and enhance them to be tailored and suitable for fog architectures, considering fog and application requirements. To achieve RG2, we define two main tasks:

(1) Identifying self-healing mechanisms and leverage them to deploy fog applications.

To perform the first task, we leverage the cloud native containerization and container orchestration solutions, since they fulfil many of the fog platform requirements. For instance, providing light weight virtualization at the OS level, portability, scalability, etc. In addition to these features, container orchestration solutions provide built-in fault-tolerance mechanisms by application replication and auto-recovery mechanisms. To test the features brought by containerization and container orchestration in our robotics use-case, we implemented a container-based platform using Kubernetes at the fog layer using resource constrained devices and partitioned a robotics use-case into containers and deployed our robotic application on this platform.

(2) Examining and evaluating the suitability of identified solutions for deploying applications at the fog layer in the presence of faults as per dependability aspects [9]. Specifically, node and application availability and reintegration after failure.

To perform the second task, we used two approaches, 1) fault-tree analysis, and 2) injecting faults to examine the behaviour of our system in the presence of faults and failures.

The result of our study shows that the volatile storage system of the implemented system is the most vulnerable point of failure. In addition, in our study we noticed that the built-in replication and auto-recovery mechanisms of the orchestration solution used suffer from the following issues when it comes to deploying stateful applications: (1) in the event of failure the state of the failed application cannot be restored, and (2) the behaviour of a stateful applications is dependent on its current state and therefore, redeploying a new instance without providing the state of the instance it is replacing could lead to unpredictable, or even dangerous behaviour.

As mentioned, these are the consequences of the volatile storage. In our research towards identifying persistent storage solutions for stateful applications in container-based architectures we noticed that there are available solutions providing persistent storage solutions at the cloud layer, however, we also noticed that these solutions still have limitations when it comes to node failure and data consistency between replicas and nodes. In addition, data access at the fog layer is crucial for safety-critical applications like robotics applications.

In this case, the first solution is to design persistent storage at the fog layer. However, in a distributed system designing a storage solution must be tightly bound to data consistency as well. We also need to consider fault-tolerance mechanism in the design of the persistent storage to ensure data availability in the system.

This leads us to formulate our third research goal as follows:

Research Goal 3 (RG3): Design of a fault tolerance persistent storage solution for stateful applications in container-based architectures.

To achieve RG3, we need to consider the following requirements: 1) ensure that states, libraries, input and result data indefinitely remain accessible to a stateful application and 2) guarantee data consistency between different distributed storages. In addition to these two characteristics, a fault tolerant persistent storage could deliver the following services to fit into a container-based fog architecture: 1) maintaining scalability of containerized applications, 2) portable enough to migrate between different nodes in the cluster, and 3) self-healing when a fault occurs in the system. Considering these characteristics and services, we propose a storage system realized by a containerized data storage application. The proposed data storage is more than a directory that can store data. Our solution is bound to a storage application that can dynamically manage storage capacity, data transfer between applications, automatically recover itself and the gathered data, and provide data consistency despite of failure of nodes, application and the storage. This means that by assuming a storage system that delivers data storage and fault-tolerance services can be containerized like any other application, we can take advantage of the containerization to deliver a storage service. To achieve data consistency we integrate the RAFT consensus protocol [10] in the storage container (SC) system. To investigate if the objectives of the proposed solution can be achieved, we need to verify it. Therefore we formulated the last goal as follows:

Research Goal 4 (RG4): Evaluation of the correctness of the proposed solution in presence of faults.

To achieve RG4, we use the UPPAAL formal verification tool to model and verify our proposed solution. We verify the correctness of defined properties of our system and evaluate our system behaviour in presence of faults and failures. We also verify system recovery and re-integration after failure at application and node level as well as showing data eventual consistency in the system (explained in Section 3.7).

The defined high level properties are divided into properties that verify the fault-tolerance behaviour of system, which we call Fault-Tolerance Properties (FTP) and properties that verify the data consistency which we call Consistency Properties (CP).

FTP includes Application failure recovery, Node failure recovery, Data availability after application failure, and Data availability after node failure. CP includes data consistency in the distributed replicated data structure.

2.2 Research Methodology

The initial objective of this research is derived from the objectives of the FORA (Fog Computing for Robotics and Industrial Automation) project [1]. In accordance with FORA, a dependable infrastructure is required to implement a fog computing platform. We conduct a scientific research to narrow down this high level objective and fulfil the dependability requirements of implementing a fog platform for robotics applications.

To conduct a scientific research and achieve a concrete method to apply as a solution to the problem of the thesis, leveraging a research methodology is crucial.

Holz et al. [11] discuss the four major steps to conduct a research including problem formulation, propose solution, implementation and evaluation. We tried to follow a similar approach [11] in our research.

Figure 2.1 shows the process of our research in this thesis. Numbers in this figure indicate the sequence of activities.

We first conducted a systematic literature review (SLR) and then formulated the problem based on the findings of the SLR study. In the next step we explored and extended our research in a related case study. Then, we proposed a solution and finally we evaluated it through formal verification. The details of our research methodology are summarized in the following:

[1]https://www.fora-etn.eu/

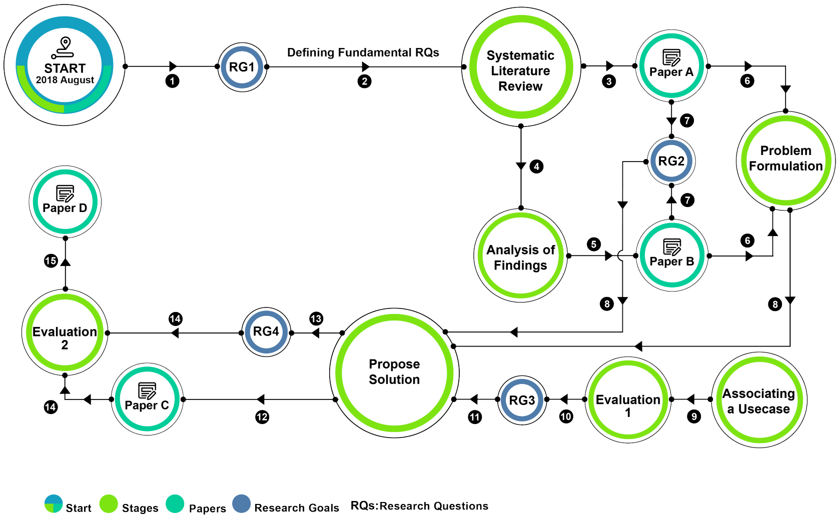

Figure 2.1: Research process followed in this thesis

2.2.1 Systematic Literature Review (SLR)

As the first step, we have done a systematic literature review to answer a number of fundamental research questions as well as finding the current state of the art and problems for initiation of our research. We followed the guidelines and process proposed in [12], [13] and [14]. We first investigated current dependability solutions in both distributed systems and clouds and then defined four fundamental research questions. We formulated a search string knowing the research questions and collected papers in a defined time frame. The relevant papers in the collected papers are included in our survey study. In Paper A and B, we discuss the gaps between the approaches studied in the state of the art and fog computing requirements for designing dependable fog systems. We answer four fundamental research questions. Figure 2.2 summarizes the research method we used in the SLR and Figure 2.3 shows the numerical and details of the SLR process.

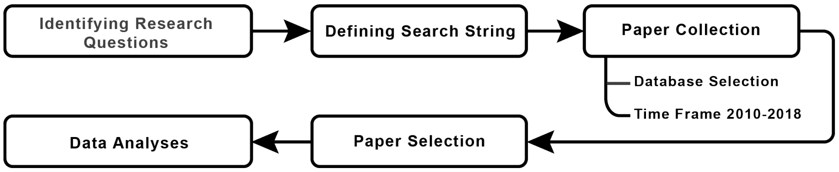

Figure 2.2: SLR Research Method (from Paper A).

2.2.2 Problem Formulation

The results of Paper A inspired in bringing more clarity to our research problem formulation. A number of identified challenges in Paper A were considered for further investigation. The problem was formulated considering the finding of Paper A and the study of trending technologies was refined in several iterations. An exhaustive list of problems found in Paper A that formed the specific problem of this thesis is also published in Paper B [15]. This motivated the formulation of RG2.

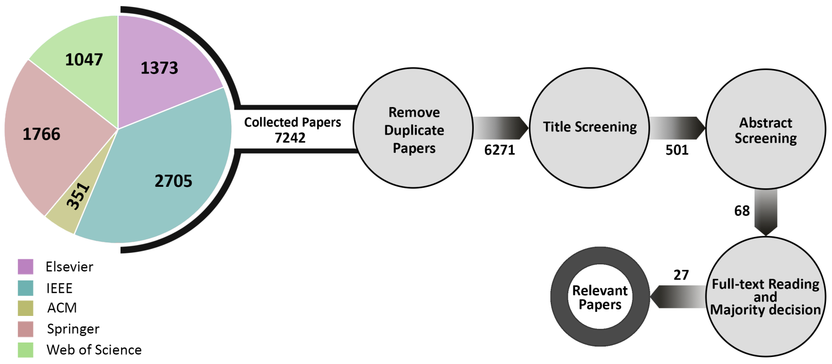

Figure 2.3: Study Selection Process (from Paper A).

2.2.3 Case Study

After problem formulation, we focused on the most relevant and important papers found in the SLR to consolidate our ideas. Then, we summarized our findings as RG2. In Papers C, we considered the main gaps found in paper A and

B and worked on a use-case to implement a container-based fog-robotics architecture. Our use-case comes from a cloud-native architecture. We extended this solution to lower capacity resources down to the edge of the network to implement it as a fog architecture. Next, we leveraged an industrial robotic application originally implemented on ROS (Robot Operating System). We deployed the robotic application on a distributed fog network (details described in the next subsection).

2.2.4 Implementation

The practical implementation in our study refers to the implementation of a test-bed for the case study by containerizing a robotic application and deploying it on the edge of the network. Details of the design and deployment are explained in Section 3.4. Observations and measurements based on practical experience helped us understand the real impact of deploying the robotic application at the edge using cloud native solutions.

2.2.5 Evaluation

By the practical implementation, we were able to identify probable faults in the transient storage system and its impacts on the stateful robotic applications. Evaluation in our study consists of two steps. First, injecting faults on the system and investigating the results of the fault injection on the system after implementation reported in (Paper C). Fault injection and fault tree analysis helped us identify certain problems with the volatile storage issue. Therefore, we proposed the use of containerized storage systems to provide fault-tolerant persistent storage. In the second step of evaluation, we modeled the proposed solution in the well-known verification tool, UPPAAL [8] which helped us understand the real impact of our solution on the system design and prove that the identified problems with the volatile storage are addressed and solved by our proposed solution. The results/outcomes of each step are presented in Paper C and Paper D.

Chapter 3

Background & Related Work

This chapter presents and discusses background and related work relevant to our research in this thesis and is structured as follows. First, we present the background on the most popular open source cloud-native containerization and container orchestration solutions, namely Docker and Kubernetes. Then, motivated by a robotic use-case to find the dependability and fault-tolerance requirements of the application level, we present background Robot Operating System (ROS) and implementation of the robot application (our use-case) in ROS. Next, we provide an explanation of the design rational for containerizing the robot application, implemented and integrated at the fog layer. Finally, we end the chapter with a short discussion on the related work.

3.1 Docker Containers

Docker is a set of ”Platform as a Service” products that uses OS-level virtualization to deliver software in packages called *containers*. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels [16]. Docker automates the packaging, shipping, communication and deployment of containers [17]. Containerization exploits the kernel of the host Operating System (OS) for running multiple isolated applications in the containers. This concept of sharing the kernel of the host OS reduces the overall overhead of executing the application as a guest on the shared OS resources. Therefore, containerization is known as a light-weight virtualization technology. Containers are portable instances of applications that can be executed on any resources they are deployed on and they provide consistent operation. They provide agile application deployment which accelerates system development and simplifies the test and debug process. In addition to agile deployment, containers exhibit good modularity and scalability for IT applications. For instance, the number of containers can be increased or decreased depending on the current traffic and processing demands.

Isolated containers protect applications from attack attempts and possible malicious activities on the host OS, but in this work we will not consider the security aspects of the platform. Containers also support microservice applications due to their flexibility in deployment and high resilience [18].

3.2 Kubernetes

Docker containers can reduce the complexities of developing distributed application software like, e.g. web applications, and it can help tolerate certain failures by regenerating faulty containers. But it still requires a significant effort in order to manage and orchestrate these containerised applications, managing their dependencies, scaling up/down the application automatically, etc. Kubernetes is an open-source container orchestration system for solving these issues, i.e. automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation [4].

There are other solutions, such as Docker Swarm, Docker Compose, that we initially considered. But after reviewing the state of the art [19, 20, 21, 22, 18, 23, 1], we concluded that even if implementation and adaptation of Kuberenetes seem more complicated, the features it provides for container orchestration better meet the fault-tolerance requirements of an architecture at the fog layer, considering the resource constrained, low-latency, scalability, mobility, geo-distributed, etc, characteristics of fog [24].

Figure 3.1 presents the structure of a Kuberenetes architecture. In the following we will describe the components of the architecture to later map them to our proposed architecture.

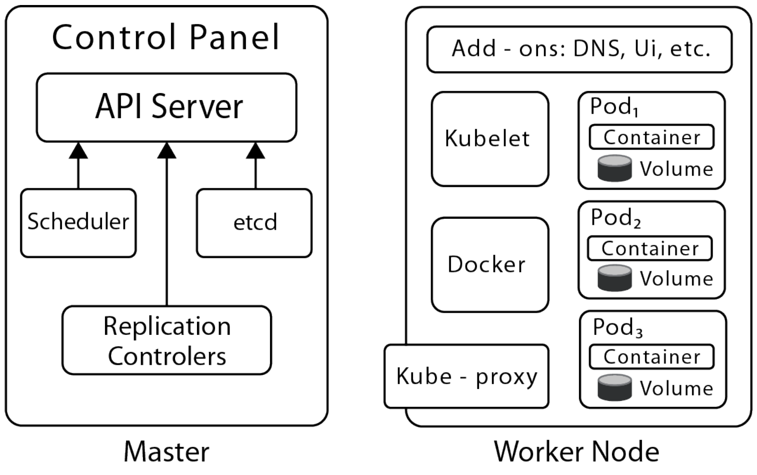

Figure 3.1: Kubernetes Architectures

Figure 3.1 depicts a cluster based Kubernetes architecture including one master and a worker node, known as node. A Kubernetes cluster can consist of a single node (a master) or multiple nodes (one master and several workers). We follow the terminologies given at the Kubernetes foundation website [4] for explaining each of its cluster elements.

Kubernetes Worker Node

Worker nodes are the hosts to entities named *Pod*. Pods are the smallest components of Kubernetes providing execution units for the applications. The containerised applications in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated. Containers inside the Pods are managed by container runtime which is a software responsible to run and manage the containers. We use the Docker engine as Container runtime in our use-case. When a container fails or terminates, the Docker engine will automatically regenerate a new container inside the Pod. Worker nodes communication to the master node is maintained through two different components: (1) kubelet and (2) kube-proxy. Kubelet works as an agent which is running on all the nodes in a Kubernetes cluster, it ensures that the containers inside the Pods are healthy and working; it checks container availability using the Pod specifications, received by different mechanisms in the deployment process. Kublet only takes care of the containers which are created in the Kubernetes cluster. Kub-proxy also runs in each node in the cluster. It acts as a communication module and maintains network rules.

There are also a list of Addons in each node in Kubernetes, that use resources to implement features in a cluster. For instance, the DNS addon acts as a specific Domain Name System server inside the Kubernetes cluster.

Kubernetes Master Node

The master node is the control plane of the whole Kubernetes cluster, managing nodes and Pods. It consists of different components to make decisions about the cluster, including:

- The API server that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

- Etcd, a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

- Kube-scheduler that watches for newly created Pods with no assigned node, and selects a node for them to run on.

- Replication Controller which is responsible for maintaining the correct number of pods to make sure pods and containers are running in case a pod or node fails.

3.3 Robotics Use-case

Our study on container-based architectures for fog computing is based on a use-case that integrates ROS, Docker Containers, and Kubernetes. Section 3.1 and 3.2 explained about Docker and Kubernetes. Here, we provide a brief introduction to ROS and continue with a detailed description of the robotics application.

3.3.1 Robotic Application Implemented in ROS

The Robot Operating System (ROS) is a well-known open source robotic meta-operating system (middleware) that has become the de-facto standard in the industry. ROS provides services similar to those of an operating system (OS), such as hardware abstraction, low-level device control, implementation of commonly-used functionality and message-passing between processes, but it has to work alongside a real operating system that provides the services that ROS does not include [25].

The essential components of a ROS application are ROS packages, nodes and topics. Communication among ROS nodes is implemented with a Publish/Subscribe mechanism via shared topics; where a *topic* in ROS is the communication module over which nodes exchange messages.

The implemented application in ROS is a navigator robot application that constantly moves towards newly set goals, while avoiding obstacles. The behavior is illustrated in Figure 3.2, showing that the robot first aims to reach goal 1, while it should avoid the obstacles. It plans the optimum path to the goal and when it reaches there, a new goal is set, as shown in Figure 3.2B. Now the robot should reach goal number 2 without hitting the obstacles. The robot starts execution in its base station and turns around to observe the environment before deciding on the best route to the goal. This process is repeated every time the robot reaches the goal and a new goal is set for it, as shown in Figure 3.2C.

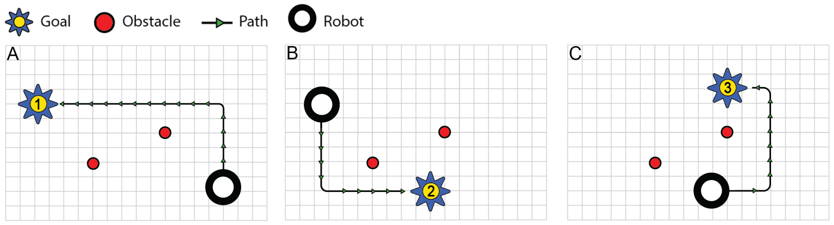

Figure 3.2: How the Robot works (from Paper C).

Our robotic application includes the following nodes:

Navigate Node: This node works as the main part of our robot application. It receives/sends data from the movebase node through topics and makes the robot repeatedly navigate toward new goals. It basically subscribes to the feedback topic and publishes the optimal path plan toward a random goal.

Movebase Node: This node has a fundamental role, as it makes the robot turn around itself and observe the environment, including obstacles and goals [26]. It publishes feedback topics to the navigate and simulation nodes. It subscribes to topics from map, obstacle, broadcaster and simulation nodes.

Mover Node: The Mover node subscribes to the movebase feedback topic to make the robot move. It publishes the speed, distance and movement data of the robot so that other nodes will know if the movement was backward or forward, etc. It is used for simulation in our scenario.

Simulation Node: This node handles simulation. It enables controllers to be integrated into motion enabled behaviour and processes for the robot(s). To run a simulation in ROS, this node (named *stageros* in ROS) is required for interpretion of the simulation data [27].

Broadcaster Node: Broadcasts the changes of the coordinate frame of the Robot as it moves around. It only publishes coordinate frame topics and does not subscribe to other topics from other nodes.

Obstacle Node: This node generates obstacles in random places in the environment that the robot is moving through. It publishes transformation of coordinate frames of different obstacles to the movebase node so that the robot can detect the obstacles and re-plan to navigate to its goal.

Mapserver Node: This node dynamically publishes updated map data as topics to the movebase node [28].

Following the example given in Figure 3.2, the interaction of the nodes would be the following. First, the Robot moves around itself and checks the environment and publishes feedback to the navigate node, checking out the new goal location and subscribing to the map data and the obstacle's coordinates frames. The simulation node also subscribes to the feedback topic publishes by movebase to simulate sensor data and exchange data with the mover node. Mover node publishes the distance and speed and the step direction. The broadcaster node publishes the coordinate frames of the robot on each move. The Mapserver node updates and publishes the map. Obstacle publishes obstacles coordinate frames whereas Mapserver and Navigate subscribe to it. A graph showing the relations between different nodes in this setup is presented in Figure 3.3.

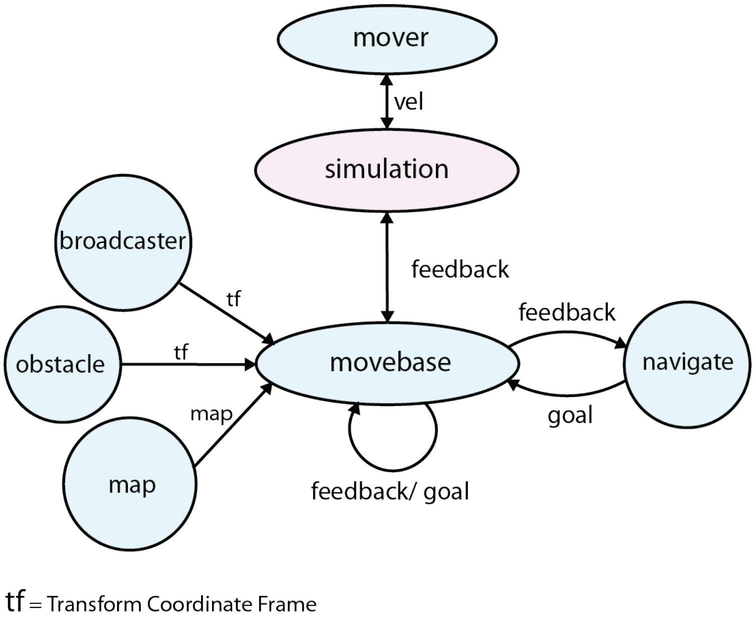

Figure 3.3: ROS Nodes Graph

3.4 Integration of ROS and Kubernetes at the Fog: design options

Building a well-designed system using Docker/Kubernetes and ROS together is a challenging task. There are several ways to design such a system, which will end up in configurations with different nodes and different dependencies.

Considering the ROS nodes we mentioned earlier, there are three reasonable design choices for this robot system: (1) Putting all ROS nodes in the same Docker container. This will result in a huge and heavy container. (2) Allocating the ROS nodes to a number of different containers. In this design, we put nodes that are working together in the same container. (3) Putting each ROS node in a single container. The latter would result in a huge number of containers, especially when the robot system is complicated with a large number of nodes. Setting up and managing such a complex system adds complexity to the design.

Therefore, we choose design model (2) and decompose our ROS application into three different containers: (A) ROS Core application (B) Navigation application (C) Simulator application.

The ROS core application container (A), includes the ROS core nodes, e.g., the movebase, map, broadcaster, and mover nodes. The Navigate application container (B), includes navigate nodes to repeatedly setup new goals for the robot and the obstacle node which publishes the obstacles coordinate frames. The Simulator application container (C), consist of simulation and graphical components, required to simulate and display robot movements and sensor data.

To containerise the ROS application, we used so-called *dockerized images* of ROS from Open Source Robotic Foundation (OSRF) [29], which simplifies the installation and setup of the ROS system.

We built a Kubernetes cluster (Version 1.16) in our local network without using any cloud platform. The cluster we implemented has three nodes. One Kubernetes master node and two worker nodes. The Master node and one worker node are running under Linux Ubuntu 18.04.

The other node is a Raspberry Pi (RPi) running Rasbian OS, which is a Linux based OS.

To deploy each container (ROS Master, Navigate and Simulator) in Pods, we need to consider the communication between the ROS nodes that are now inside the containers. Communication between nodes in ROS is based on publish/subscribe topics and the ROS master dynamically allocates unique port numbers to each publisher [30]. However, dynamic port allocation is not available in Kubernetes. Thus, to configure the communication between containers that contain ROS nodes using publish and subscribe topics, we use the headless services function that enables publisher and subscribers to directly communicate with each others by name, rather than by IP address, and we statically define the same port for all Pods as the one defined in the ROS master.

In Kubernetes, configuration and the container dependencies need to be carried out through a YAML file, in which we define communication, storage, number of replicas, etc.

The Pods hosting the containers are deployed to the worker nodes (not the master) in the Kubernetes cluster. We set the number of replicas for this scenario to one (1) to have the minimum resource consumption.

3.5 Related Works on Fault-tolerant Container-based Fog

A key factor of maintaining computing resources closer to the edge devices is to make sure they function correctly in the design stage of the implementation. In that case, applying fault-tolerance mechanism in the architecture design of fog computing can provide guarantee of correct service and application execution [31, 32].

In designing and implementing a fault-tolerance system at the fog layer, using container-based virtualization has gained traction because of its light-weight characteristic and support of a variety of edge devices that increase system performance compared to hypervisor-based virtualization [33, 34].

There are a number of works in the literature proposing fog architectures, particularly container-based fog architectures focusing on different aspects and benefits that containerization brings [35]. To the best of our knowledge, no work has addressed fault-tolerance to the extent that is required for critical fog applications. Most of the works are focused on performance, data collection from unreliable resources at the edge and task allocation [36]. Here we review a number of studies that propose container-based architectures and then we discuss in which way they fall short in addressing dependability.

Pahl et al. [19] describe the requirements for an edge cloud Platform as a Service (PaaS). Different solutions and technologies, like, Kubernetes, Docker and Mesos are evaluated to investigate which one can meet the edge cloud PaaS requirements. The evaluation results indicate a need to develop an own cluster management tool. The tool they propose use topology-based service orchestration together with Docker Swarm for application orchestration on an edge PaaS with a focus on improving the system performance without considering dependability requirements, such as availability and reliability.

Ismail et al. [20] evaluate Docker Swarm to enable an edge computing platform. They considered four fundamental criteria 1) deployment and termination, 2) resource & service management, 3) fault tolerance and 4) caching. The result of their evaluation is that Docker Swarm and Consul can meet the first and second criteria. For the third criteria, the Docker import/export feature is used as a fault-tolerance solution using the built-in Docker back-up solution. However, they do not propose any unified system fault-tolerance mechanism and disregard application re-integration.

The same approach for comparing different orchestration and management tools and technologies is used by Tosatto et al. [21]. They compare different container orchestration solutions, Kubernetes, Docker compose, etc., and conclude that each orchestration solution can be helpful in specific use-case scenarios, but that there is no solution that fit all scenarios. This work does not consider dependability.

In a recent work by Toffetti & Bohnert [22], a state of the art for cloud robotics is presented. The authors discuss the needs for an architecture based on Kubernetes for cloud application orchestration. The focus of this work is to investigate the requirements and challenges for a cloud robotic system, and considers dependability requirements, such as reliability, availability and robustness. This determines the need for a dependable system architecture to address fault-tolerance in robotics applications. However, there is no solution suggested in this work. We address a number of challenges explained in this paper in our implementation and proposed architecture.

A three layered architecture for containerized micro-services for Industrial Internet of Things (IIoT) is proposed in [18]. The architecture is composed of three basic layers, 1) Cyber physical system, 2) Gateway and 3) Enterprise system. The authors discuss reliability aspects by explaining how modularity of independent container-based micro-services can improve the system resilience to probable failures. They show that the overall system can continue working even if some applications fail in this architecture. However, they do not, as we do, consider how to tolerate application failures.

Hoque et al. [23] evaluate different container orchestration tools with the aim of comparing the impact of each tool on the overall performance in fog nodes. They propose a container orchestration framework for fog computing infrastructures using the ”Open IoT Fog Toolkit”. However, they do not, as we do, consider fault-tolerant aspects of the whole system and specifically not for fog nodes.

A fault-tolerance architecture for edge and cloud is proposed by Javed et al. [1]. They integrate Kafka and Kubernetes for data replication and cluster configuration. The proposed architecture has three abstracted layers: 1) Application isolation using containers, 2) Data transport with the help of Kafka and 3) Multi-cluster management by implementing Kubernetes. Their proposal for tolerating faults is based on replicating data collected from edge devices. Fault detection mechanisms and node failure recovery are not addressed.

The study of the state of the art shows that there is growing interest in the application of container-based solutions for fog. In general, the light-weight of these applications is seen as a very appealing property, but there is consensus about the need to provide new orchestration services and, in particular, support for replication and redeployment of containers. However, an orchestration architecture with focus on tolerating faults in different levels of the system (application level, node level, etc.) is still missing in the literature. We should also consider the fact that as containerization is a cloud-native solution, there is a lack of study on container's transient storage system as there are already a number of solutions providing persistent storage systems in the cloud. However, this is a cost and time inefficient solution while deploying applications on the fog layer.

3.6 Related Works on Fault-tolerant Distributed Storage Systems

Although containers increase application execution reliability by ease of creating replication, there still remains the issue that the state of the failed container cannot be restored. Applications use container volumes to maintain their states, however, volumes are transient storage systems which cannot be recovered when a container fails.

As container-based fog architectures are distributed systems, we review the works in the literature with a focus on fault-tolerant distributed storage systems both in container-based architectures and traditional distributed systems.

Proposed solutions for distributed fault-tolerant distributed storage systems in fog, edge and cloud computing show that authors have more focused on two fundamental problems in distributed systems, (1) Fault tolerant, permanent data storage [15], (2) Achieving a decentralised consensus. The first one has a focus on distributed data storage systems and optimal allocation of redundancy, to reduce utilization and techniques for error detection and reconfiguration upon failure. The latter one is applying consensus protocols based on system requirements [37].

In a work by Shahaab et al. [37], 66 consensus protocols such as RAFT

[10], Byzantine Fault Tolerance [38], Sieve [39], etc. are studied and analysed based on different objectives, like, sustainability, efficiency, etc. The authors conclude their work by stating that there is no single consensus protocol to apply as a solution for all the requirements in a distributed system. A consensus solution must be leveraged based on the network and types of nodes and the whole system requirements.

There are also a number of recent works proposing persistent storage solutions for container-based architectures in cloud platforms. Sharma et al. [40] propose a distributed storage system using storage application deployment on Kubernetes. However, they do not address the transient storage issue for the Pods in the orchestration solution. This work also lacks a consideration of any consensus algorithm.

To ensure that operations are executed on all the containers and their replicas, state-machine replication in containers is proposed by Netto et al. [41]. This work uses the DORADO protocol which uses shared memory to project communication and persistent data. Requests can be sent to any replicated container, matching the cloud in regard to load balancing. However, the overhead of this solution on containers violates the light-weight nature and increases the container image size.

Kristiani et al. [42] propose a persistent volume for container-based architectures using Openstack and Kubernetes. Although in this work the container-based applications are running on the edge devices in the network, the persistent storage solution is still located in the Cloud which increases delay for each application data access request.

3.7 Eventual Consistency

A distributed system exhibits the property *data consistency* if all the data concurrent read operations are guaranteed to return the same value [43]. In distributed database systems committed write is defined as a write operation that stores a value permanently in the possibly distributed and replicated database and makes it accessible to all the nodes. In this context, strong consistency [44] means that the data returned must be exactly the same on each node. It therefore implies that a read data operation will return a value only if that value has been committed by all nodes, and that the operation will be delayed in case the value has not been committed, until it is either accepted or rejected by all nodes. In particular, with strong consistency it is impossible for a read data operation executed by two nodes at the same time to return two different values. Eventual consistency [44] is more relaxed than strong consistency. Eventual consistency means that when no updates are executed on the data, all read operations to that data will eventually return the same value, yet it is possible that in some intermediate states, some simultaneous read operations in different nodes will return different values (normally a previous write).

The advantage of eventual consistency compared to strong consistency is that the eventual consistency will be fulfilled even if some part of the system is not working, and this could be important in safety-critical applications.

According to the Consistency, Availability and Partition tolerance (CAP) theorem [45], access to data in distributed systems can only satisfy two out of the three attributes consistency, availability and partition tolerance where (1) consistency is when all the data read return the most recent write; (2) availability is when all the read requests return a response but without guaranteeing that the response is the most recent write; and (3) partition tolerance is when a system continues to provide service despite an arbitrary number of nodes are not returning any response due to a network failure or the failure of one of the components or any failures that causes the node to partition from the network.

When a failure occurs in a distributed system and causes node unavailability, we could decide to reduce availability by stopping the operation and achieve data consistency or achieve data availability by proceeding with the operation and achieve eventual consistency [44].

Strong consistency is a requirement in applications like financial transactions where data accuracy is more important than timing. However, in other applications where low latency and timeliness is a requirement, like in factory automation, eventual data consistency which favors continued operation and low latency over strong consistency is often sufficient.

When the priority is data availability, the data might not be updated on all the nodes simultaneously. In this case the data access is faster, but comes at the cost of reduced data accuracy. In applications where low latency is a requirement, for example, in industrial automation, if the latency is high, the reaction time required to process a response for an event could be slower than the rate at which the events are generated. In safety-critical applications late reactions might have hazardous consequences.

Bailis and Ghodsi [44] explain that there are two important metrics, namely,

Table 3.1: Summary of Most Related Works

Aspects	**Our solution**	**[5]**	**[20]**	**[46]**	**[41]**	**[47]**
Persistent storage	✓	✓	—	—	—	—
Stateful application	✓	✓	✓	✓	✓	✓
Data consistency between storages	✓	—	—	✓	✓	✓
Fault-tolerant storage	✓	—	Flocker	—	—	—
Node failure recovery	✓	—	✓	—	—	✓
Self-healing	✓	—	—	—	—	—
Timing requirements	—	✓	—	✓	✓	✓
Network level	Fog	Cloud	Edge	Cloud	Cloud	Cloud
Application	Industrial robotics	Video streaming	Not specified	Log producer	Log producer	Log producer

time and version, when designing a system that fulfils eventual consistency. Time refers to the time when the latest write is accessible for a read request and version refers how many versions of the written data a read request could have access to, i.e. in the worst case, how many versions back in time from the most recent write the value read could be.

We decided to choose eventual consistency in our system design as it has a non-blocking character, which simplifies meeting the timing requirements of safety-critical applications. Furthermore, the low rate of node failure compared to application failure in a container-based architecture reduces periods of temporary inconsistency of eventual consistency. Still, as some applications may require strong consistency, for correct operations, it is important that application developers consider that it is eventual consistency we provide. However, when an application in a node reads data from another application in the same node, strong consistency is guaranteed even in the case of application failure, thanks to the use of a replicated data structure.

3.8 Reflection on Related Works

Studying the literature provided us with a better view of the existing solutions, as well as broadened our knowledge about our system requirements. The need for persistent storage comes from the stateful application we aim at dealing with in our use-case. In addition, we need to investigate what can be done at the network edge in the fog layer to reduce data latency and provide immediate response to data access requests in the fog layer.

In this work we take advantage of the light-weight and self-healing characteristics of containers and propose container-based storage applications which can provide a distributed storage system for the containerized robotic applications in our use-case. This storage system can be recovered upon failure just like other containerized applications. Regarding the consensus protocol we decided to choose RAFT as we found it more suitable for our use-case because it is designed to deliver correct data and fulfills safety, liveliness and fault-tolerance properties for data and states [37].

Table 3.1 provides a summary of the most related works to ours with a comparison to aspects each provides for managing states using container-based solutions.

Chapter 4

Research Results

In this chapter, we discuss our results and present a summary of our contributions. We highlight the specific contributions of the included papers by a discussion of the results.

4.1 Thesis Contributions

This thesis focuses on four main goals, realized by four corresponding contributions. We re-present the research goals as follows:
RG1: Identification of challenges and solutions on dependability aspects of fog computing in the current state of the art.
RG2: Design of a fault-tolerance fog architecture by leverage auto-recovery and self-healing mechanisms.
RG3: Design of a fault tolerance persistent storage solution for stateful applications in container-based architectures.
RG4: Evaluation of the correctness of the proposed solution in presence of faults.

Our research goals are realized by the following four main contributions:

- **C1:** We identify open challenges related to dependability of fog computing and the gap between the existing dependability solutions and dependability requirements for fog computing. (Fog Dependability Challenges)

- **C2:** We enhance current container-based architectures, tailored them for fog applications specifically considering robotic and other control applications. (Fog Container-based Architecture)
- **C3:** We develop a fault-tolerant persistent distributed container-based storage (SC) solution, characterized by self-healing and consistency mechanisms. (Fault-Tolerant Distributed Persistent SC)
- **C4:** We formally verify the properties of the proposed container storage (SC) solution. (Formal Verification of SC)

4.1.1 C1: Fog Dependability Challenges

Paper A [36] and Paper B [15] form this thesis contribution. Paper A, presents a systematic literature review that answers four fundamental research questions (RQ), formed at the beginning of our research. The research questions aim at finding: 1) dependability attributes, 2) source of failure, 3) dependability means and techniques to implement them, and 4) relations between dependability and security in the current solutions proposed for fog computing.

In Paper A, we provide answers to these research questions based on our observations as follows:

1. Reliability and availability are the dependability attributes that most authors are focusing on and QoS and Scalability are new requirements which attract the attention of authors in ensuring dependability in fog computing.
2. Node failure and link or path failure are the main source of failures. In addition, failures due to resource allocations and application placement are causes for application execution failure.
3. Redundancy techniques are the most common methods to increase dependability level in fog computing.
4. Security in fog is mainly discussed from the perspective of privacy and confidentially. However, the existence of malicious faults and the design of fault-tolerant security solutions are not addressed in the literature.

We also observe that the two problems that are in main focus in the current state of the art are: guaranteeing reliable data storage/collection in systems with unreliable and untrusted nodes, and guaranteeing efficient task allocation in the presence of varying computing load.

The outcome of Paper A, in addition to the answers to the fundamental research questions, is a non-exhaustive list of research challenges discussed in both Paper A and Paper B. The highlight of the challenges in Paper A and B that formed basis for RG 2 are: 1) the lack of domain specific dependability solutions that are designed based on the requirements of specific applications, 2) unsuitability of the current physical and virtual platforms for resource constrained fog devices, 3) shortage of solutions for reintegration after fault recovery in distributed systems, 4) absence of self-healing and auto recovery mechanism for resource constrain fog platforms, and 5) negligence of scalability and portability of application at the node level.

4.1.2 C2: Fog Container-based Architecture

C2 is realized by Paper C [48]. In the non-exhaustive list of challenges found in C1, the need of light-weight platforms that support automatic re-integration after failure realized by self-healing, portability and scalability characteristics of fog applications is identified as a major requirement for forming a dependable fog platform. Therefore, we explored the current cloud-native solutions and identified the containerization solutions as light-weight virtualization platforms that support portability, self-healing and scalability features. We examined the possibilities to implement fog computing infrastructure for our robotics use-case as a fog application using container-based solutions. We performed two approaches to identify the vulnerable points of failures and the shortages of extension of container-based solutions to the fog layer considering the application requirements coming from our use-case. The first approach is a fault tree analysis, and the second approach is fault injection. The fault tree analysis shows that the most vulnerable point of failure for our stateful application is the volatile storage of container-based architecture. The fault injection approach confirmed this. In addition, through the fault injection approach, we identified that although containerization brings built-in fault-tolerance by auto-recovery of applications, the current container-based architecture suffers from limitations that are threats to dependability, although, they are not considered as failure. These limitations are the lack of monitoring and resource management mechanisms.

To solve the aforementioned volatile storage issue for stateful applications we formed RG3 which is achieved by C3. To solve the monitoring and resource management issues identified by the fault injection approach, we propose enhancing the container-based architecture with monitoring and resource management components, together with another layer of fault-tolerance mechanism that handles application and node failure. This platform is also designed to be suitable to host the storage solution we explain in C3.

4.1.3 C3: Fault-tolerant Distributed Persistent SC

Motivated by C2, Paper C [48] presents the main thesis contribution complemented with some low-level details in Paper D [49]. When we containerized our robotics use-case and deploy it at the test-bed implemented using the container-based platform, we realized that the main differences between our application and cloud-native applications are their statefulness and idempotency characteristics.

We realize this by the two fault tree analysis and fault injection explained in C2.

When an application fails and recovers, its access to the volatile storage is lost, therefore, the application will start working again from the beginning. However, in safety-critical applications which are not idempotent, access to the latest state is critical, otherwise, there could be hazardous consequences.

Therefore, a persistent storage is required to solve the volatile storage issue of containerization and container orchestration. Although there are already some solutions providing persistent storage for container-based architecture, they still suffer from some limitations: 1) They are all implemented using cloud storage, while we aim to provide a fault-tolerant storage system at the fog level; 2) Solutions proposed for data consistency between different nodes and applications inside nodes require at least two replicas of each application and all the load of execution is always on one container. (All other containers in the cluster will remain standby, apart from forwarding the task and result to/from the leader.); 3) Node failure in a cluster has not been investigated; and 4) Data consistency between containers of different kinds has not been investigated.

In this contribution, we propose a storage solution to address the volatile storage issue in container-based architectures in fog platforms by taking the advantages of containerization: scalability, self-healing and portability. In this solution, the use of storage containers (SC) provides suitable mechanisms for: (1) Outsourcing data storing and data retrieving upon failure to containerized storage mechanisms, thereby reducing the application load; (2) Storing data/states of stateful application (execution) locally inside each node; (3) Tolerating failures at two levels, application (software) and node failures (Software and Hardware); and (4) Achieving data consistency between nodes and applications inside nodes using the RAFT consensus protocol.

4.1.4 C4: Formal Verification of SC

This thesis contribution is presented in Paper D [49] and motivated by C3.

In this contribution we verify our proposed SC solution in C3 using the UPPAAL verification tool. We present a formal verification of three categories of key properties: 1) model properties, 2) fault-tolerance properties, and 3) consistency properties of the proposed container-based persistent storage and the data consistency protocol used in this solution between nodes is provided in this contribution.

4.2 Overview of the Included Papers

In this section, we present the abstracts and short descriptions of the contributions of the included papers.

The mapping of contributions to the included papers are shown in Table 4.1.

Table 4.1: Mapping of contributions to the included papers.

	C 1	**C 2**	**C 3**	**C 4**
Paper A	✓			
Paper B	✓			
Paper C		✓	✓	
Paper D				✓

Individual Contributions

I have been the initiator and main author for the included papers. The supervision team participated in the brainstorming and planning sessions for the research and provided valuable feedback and helped in writing few sections in some of the included papers.

4.2.1 Paper A

Title: Dependable Fog Computing: A Systematic Literature Review [36].
Authors: Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.

Abstract. Fog computing has been recently introduced to bridge the gap between cloud resources and the network edge. Fog enables low latency and location awareness, which is considered instrumental for the realization of IoT, but also faces reliability and dependability issues due to node mobility and resource constraints. This paper focuses on the latter, and surveys the state of the art concerning dependability and fog computing, by means of a systematic literature review. Our findings show the growing interest in the topic but the relative immaturity of the technology, without any leading research group. Two problems have attracted special interest: guaranteeing reliable data storage/collection in systems with unreliable and untrusted nodes, and guaranteeing efficient task allocation in the presence of varying computing load. Redundancy-based techniques, both static and dynamic, dominate the architectures of such systems. Reliability, availability and QoS are the most important dependability requirements for fog, whereas aspects such as safety and security, and their important interplay, have not been investigated in depth.

Paper contribution: The main contribution of this paper is to answer four fundamental research questions and finding the current state of the art related to implementation of dependable fog networks, understanding future trends, and identifying the gap between current approaches and solution and fog computing requirements.

4.2.2 Paper B

Title: A preliminary Roadmap for Dependability Research in Fog Computing [15].
Authors: Zeinab Bakhshi, Guillermo Rodriguez-Navas.

Abstract. Fog computing aims to support novel real-time applications by extending cloud resources to the network edge. This technology is highly heterogeneous and comprises a wide variety of devices interconnected through the so-called fog layer. Compared to traditional cloud infrastructure, fog presents more varied reliability challenges, due to its constrained resources and mobility of nodes. This paper summarizes current research efforts on fault tolerance and dependability in fog computing and identifies less investigated open problems, which constitute interesting research directions to make fogs more dependable.

Paper contribution: The main contribution of this paper are: (1) identification and classification of current research approaches for dependability in fog computing, (2) comparison of different proposed solutions, considering traditional dependability notions for critical systems, and (3) a discussion of research gaps related to fog computing dependability.

4.2.3 Paper C

Title: Fault-tolerant Permanent Storage for Container-based Fog Architectures [48].
Authors: Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.

Abstract. Container-based architectures are widely used for cloud computing and can have an important role in the implementation of fog computing infrastructures. However, there are some crucial dependability aspects that must be addressed to make containerization suitable for critical fog applications, e.g.,in automation and robotics. This paper discusses challenges in applying containerization at the fog layer, and focuses on one of those challenges: provision of fault-tolerant permanent storage.The paper also presents a container-based fog architecture utilizing so-called storage containers, which combine built-in fault-tolerance mechanisms of containers with a distributed consensus protocol to achieve data consistency.

Paper contribution: The main contributions of this paper are (1) we identify key limitations of directly applying existing containerization and orchestration solutions in the resource constrained fog layer, and (2) we propose remedies in addressing related dependability challenges.

4.2.4 Paper D

Title: Using UPPAAL to Verify Recovery in a Fault-tolerant Mechanism Providing Persistent State at the Edge [49].
Authors: Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson.

Abstract. In our previous work we proposed a fault-tolerant persistent storage for container-based fog architecture. We leveraged the use of containerization to provide storage as a containerized application working along with other containers. As a fault-tolerance mechanism we introduced a replicated data structure and to solve consistency issue between the replicas distributed in the cluster of nodes, we used the RAFT consensus protocol. In this paper, we verify our proposed solution using the UPPAAL model checker. We explain how our solution is modeled in UPPAAL and present a formal verification of key properties related to persistent storage and data consistency between nodes.

Paper contribution: Evaluation and verification of the solution proposed for permanent storage application in container-based fog architectures.

Chapter 5

Discussion, Conclusion and Future Work

In this chapter, we discuss our experimental results and conclude the thesis with a list of potential future research directions.

5.1 Conclusion and Summary

Fog computing helps to provide computing resources closer to the edge of the network and has potential to increase predictability and time to respond, important for safety-critical applications like robotics where prompt decision making is required, and sending data to clouds might increase response time and latency and leads to hazardous consequences. However, there are many aspects to consider in designing a fog architecture. For instance, fog resources are constrained resources. Therefore, light-weight platforms are desirable for hosting fog applications. Other aspects to consider are the portability, scalability and fault-tolerance required when deploying applications at the fog layer. This thesis focuses on four research goals: (RG1) identifying dependability challenges in fog; (RG2) designing a light-weight fault-tolerance fog architecture; (RG3) designing a fault tolerance persistent storage for stateful applications in container-based architectures; and (RG4) evaluating the correctness of the proposed solution. The goals are met by four contributions, packaged into the four included papers. In Paper A, we present a systematic literature review to answer four fundamental research questions. Paper A shows that reliability and availability aspects of dependability are the main focus of authors dealing with the design of fog-based architectures [50, 51, 52, 53, 54]. To address reliability and availability, redundancy techniques are leveraged in the proposed solutions [50, 55, 51, 52]. However, considering limitations in resources in fog computing, using replicas as redundant components is identified as a challenge for our further studies. Moreover, most of the proposed solutions aim at providing dependability means by developing solutions on fault-tolerance and failure recovery [56, 57]. However, the way the system will re-integrate after failure is not discussed, nor evaluated in the literature,and is thus identified as a challenge for our research.

In paper B, we refine our previous study in Paper A and categorized the challenges identified in Paper A into different groups based on the sources where the failure is initiated. We present a non-exhaustive list of challenges which helped us formulate our research problem on designing a fault-tolerance fog-base architecture by leveraging auto-recovery mechanisms.

In Paper C, we present a design of an enhanced container-based architecture tailored for fog applications. The requirements of the fog applications in this study are derived from a robotics use-case. In the study of our use-case we identify two critical aspects related to handling failures, namely the lack of persistent storage for stateful applications and challenges in resource management while deploying applications. We propose remedies realized by adding new components and services to the current container orchestration architecture. In addition, we propose using a container-based storage system integrated with a consensus protocol to provide fault-tolerant persistent storage for distributed stateful applications at the fog layer.

In Paper D, we evaluated our solution presented in Paper C using the UPPAAL formal verification tool. We model our solution and define the properties of the proposed solution and verify these type of properties: (1) Model properties; (2) Fault-tolerance properties; and (3) Data consistency properties. We also add a safety properties to show the system is deadlock free. Since all properties are proved to be satisfied using UPPAAL, we can conclude that our proposed solution fulfils the consistency and fault-tolerance requirements in the modeled container-based fog application.

The solution proposed for the persistent storage in our study is realized by a container-based storage system that uses RAFT. This solution is based on three principles: (1) Dissociation of data storage from application processes; (2) Using a replicated data structure to increase data availability and to provide fault-tolerant persistent storage in each node of the cluster; and (3) Adding a consistency protocol to keep the replicated data structures that are distributed in the cluster of nodes synchronized.

5.2 Discussion and Future Work

In the future, we plan to address some of the challenges encountered in the included papers. This section provides a brief overview of identified limitations of our work, in addition to the possible future directions and extensions for the work presented in this thesis.

5.2.1 Timing Aspects

In the studied use-case, the application is redesigned from a ROS implemented application to a containerised ROS application. At the same time, its dependability and fault-tolerance requirements have always been the point of focus. Therefore, the main focus in our study is the correctness of the application execution. However, when re-designing the application to fit it into the container-based architecture, there are some timing requirements that are not specifically addressed in our study. For instance, application startup delay after container failure and application redeployment delay after a node failure. Although these delays are known to be very short in the approach we leveraged in the deployment, we are aware of the possible effects that these delays might have on the functionality and service delivery of the application.

Another timing aspect that has not been investigated in our work is the communication delays. The studied use-case in ROS uses publish/subscribe methods for communication and when redesigning the architecture we use a communication model based on the services in Kubernetes. In our experiments we did not observe delays comparing these two design models when containers communicate with each other. However, by adding the storage system that handles the states and their consistency in the distributed network, there might be delays in communication between SCs and the SC leader. We consider that we can rely on the RAFT timeouts which are defined to guarantee that if a node does not provide response in the defined deadline it will be excluded in the election process. However, a timing analysis is required to make sure if these delays are tolerable considering the application timing requirements.

5.2.2 Performance Degradation

The solution proposed in our study for the persistent storage by leveraging replicated data structure and disassociation of application from storage system aims to reduce the load of data synchronization from the application by off-loading it to containerized storage applications. However, adding the RAFT consensus protocol to the storage system might result in degradation of performance for two reasons: 1) overhead related to the extra storage containers (SCs) that need to access replicated data structures and consume resources in each node and 2) the integration of RAFT that might result in performance overhead for election and state management.

At the same time, adding SCs reduces the load of application communication and data synchronization in applications as well as reduced number of replicas, since the need for unnecessary application replication is reduced compared to other solutions that require more replicas of application for data synchronization [47].

Regarding the use of RAFT protocol in our solution we need to evaluate the system performance in our future work.

5.2.3 Extensions to the Included Papers

In Paper C we presented an approach to provide fault-tolerant persistent storage for container-based architectures and integrated this solution with a consensus protocol. We extended this work in Paper D by formally verifying the proposed solution using UPPAAL. However, two important aspects of evaluations remain for further investigations, namely, timing and performance. In our future works we aim to consider timing requirements of application, communication, application recovery and application redeployment on another node in the network. In addition, we will evaluate the performance of a system considering resource consumption by comparing our solution to pure Kubernetes implementation explained in Section 3.4.

Bibliography

[1] A. Javed, K. Heljanko, A. Buda, and K. Främling. Cefiot: A fault-tolerant IoT architecture for edge and cloud. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 813–818, 2018.

[2] Nan Tian, Ajay Kummar Tanwani, Jinfa Chen, Mas Ma, Robert Zhang, Bill Huang, Ken Goldberg, and Somayeh Sojoudi. A fog robotic system for dynamic visual servoing. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1982–1988. IEEE, 2019.

[3] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. Computing on the edge: A platform for replicating internet applications. In *Web content caching and distribution*, pages 57–77. Springer, 2004.

[4] Kubernetes Foundation, Kubernetes Documentation. `https://kubernetes.io/`.

[5] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019.

[6] Gigi Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[7] Lubos Mercl and Jakub Pavlik. Public cloud kubernetes storage performance analysis. In *International Conference on Computational Collective Intelligence*, pages 649–660. Springer, 2019.

[8] UPPAAL Model Checker, UPPAAL Official Website. `https://https://uppaal.org/`.

[9] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: a taxonomy. In *Building the Information Society*, pages 91–120. Springer, 2004.

[10] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[11] Hilary J Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: what are they, and how should we teach them? *ACM SIGCSE Bulletin*, 38(4):96–114, 2006.

[12] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *EASE*, volume 8, pages 68–77, 2008.

[13] Barbara Kitchenham, Rialette Pretorius, David Budgen, O. Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen Linkman. Systematic literature reviews in software engineering: A tertiary study. *Information and Software Technology*, 52(8):792 – 805, 2010.

[14] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, 25(1):49–82, Mar 2017.

[15] Zeinab Bakhshi and Guillermo Rodriguez-Navas. A preliminary roadmap for dependability research in fog computing. *ACM SIGBED Review*, (4), 2020.

[16] Docker Hub, Docker Description. `https://www.docker.com/resources/what-container`.

[17] Pethuru Raj, Jeeva S Chelladhurai, and Vinod Singh. *Learning Docker*. Packt Publishing Ltd, 2015.

[18] 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.

[19] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 117–124, Aug 2016.

[20] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135, 2015.

[21] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: state of the art and challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 70–75. IEEE, 2015.

[22] Giovanni Toffetti and Thomas Michael Bohnert. *Cloud Robotics with ROS*, pages 119–146. Springer International Publishing, Cham, 2020.

[23] S. Hoque, M. S. d. 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.

[24] Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, ARM 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Jason M O'Kane. A gentle introduction to ros, independently published, 2013. *Electronic copies freely available from the authors website.*

[26] Movebase, ROS . `http://wiki.ros.org/move_base/`.

[27] Simulation, Stageros . `http://wiki.ros.org/move_base/`.

[28] Map Server Node, ROS. `http://wiki.ros.org/map_server/`.

[29] Ruffin White and Henrik Christensen. *ROS and Docker*, pages 285–307. Springer International Publishing, Cham, 2017.

[30] Nicholas DeMarinis, Stefanie Tellex, Vasileios P Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8514–8521. IEEE, 2019.

[31] Algirdas Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, 1997.

[32] A Aviziens. Fault-tolerant systems. *IEEE Transactions on Computers*, 100(12):1304–1312, 1976.

[33] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015.

[34] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[35] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson. Dependable fog computing: A systematic literature review. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 395–403, Aug 2019.

[37] Ali Shahaab, Ben Lidgey, Chaminda Hewage, and Imtiaz Khan. Applicability and appropriateness of distributed ledgers consensus protocols in public and private sectors: A systematic review. *IEEE Access*, 7:43622–43636, 2019.

[38] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[39] Christian Cachin, Simon Schubert, and Marko Vukolić. Non-determinism in byzantine fault-tolerant replication. *arXiv preprint arXiv:1603.07351*, 2016.

[40] Ashish Sharma, Sarita Yadav, Neha Gupta, Shafali Dhall, and Shikha Rastogi. Proposed model for distributed storage automation system using kubernetes operators. In *Advances in Data Sciences, Security and Applications*, pages 341–351. Springer, 2020.

[41] Hylson V Netto, Lau Cheuk Lung, Miguel Correia, Aldelir Fernando Luiz, and Luciana Moreira Sá de Souza. State machine replication in containers managed by kubernetes. *Journal of Systems Architecture*, 73:53–59, 2017.

[42] Endah Kristiani, Chao-Tung Yang, Yuan Ting Wang, and Chin-Yin Huang. Implementation of an edge computing architecture using openstack and kubernetes. In *International Conference on Information Science and Applications*, pages 675–685. Springer, 2018.

[43] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, volume 10, pages 1–15, 2010.

[44] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[45] Eric Brewer. Cap twelve years later: How the ”rules” have changed. *Computer*, 45(2):23–29, 2012.

[46] Caio Oliveira, Lau Cheuk Lung, Hylson Netto, and Luciana Rech. Evaluating raft in docker on kubernetes. In *International Conference on Systems Science*, pages 123–130. Springer, 2016.

[47] Hylson Netto, Caio Pereira Oliveira, Luciana de Oliveira Rech, and Eduardo Alchieri. Incorporating the raft consensus protocol in containers managed by kubernetes: an evaluation. *International Journal of Parallel, Emergent and Distributed Systems*, 35(4):433–453, 2020.

[48] Zeinab Bakhshi Valojerdi, Guillermo Rodriguez-Navas, and Hans Hansson. Fault-tolerant permanent storage for container-based fog architectures. In *Proceedings of the 2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, 2021.

[49] Zeinab Bakhshi, Guillermo Rodriguez-Navas, and Hans Hansson. Using UPPAAL to verify recovery in a fault-tolerant mechanism providing persistent state at the edge. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021.

[50] A. Aral and I. Brandic. Quality of service channelling for latency sensitive edge applications. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 166–173, June 2017.

[51] K. E. Benson, G. Wang, N. Venkatasubramanian, and Y. Kim. Ride: A resilient IoT data exchange middleware leveraging SDN and edge cloud resources. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 72–83, April 2018.

[52] H. P. Breivold and K. Sandström. Internet of things for industrial automation – challenges and technical solutions. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pages 532–539, 2015.

[53] X. Chen, X. Wen, L. Wang, and W. Jing. A fault-tolerant data acquisition scheme with mds and dynamic clustering in energy internet. In *2018 IEEE International Conference on Energy Internet (ICEI)*, pages 175–180, May 2018.

[54] Nikolay Chervyakov, Mikhail Babenko, Andrei Tchernykh, Nikolay Kucherov, Vanessa Miranda-López, and Jorge M. Cortes-Mendoza. AR-RRNS: Configurable reliable distributed data storage systems for internet of things to ensure security. *Future Generation Computer Systems*, 92:1080 – 1092, 2019.

[55] J. A. Cabrera, D. E. Lucani., and F. H. P. Fitzek. On network coded distributed storage: How to repair in a fog of unreliable peers. In *2016 International Symposium on Wireless Communication Systems (ISWCS)*, pages 188–193, Sep. 2016.

[56] Y. Xiao, Z. Ren, H. Zhang, C. Chen, and C. Shi. A novel task allocation for maximizing reliability considering fault-tolerant in VANET real time systems. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–7, Oct 2017.

[57] Xiao Yuan, Chimay J. Anumba, and M. Kevin Parfitt. Cyber-physical systems for temporary structure monitoring. *Automation in Construction*, 66:1 – 14, 2016.