Mälardalen University Press Licentiate Theses
No. 310

IMPROVING SOFT REAL-TIME PERFORMANCE OF FOG COMPUTING

Vaclav Struhar

2021

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISBN 978-91-7485-517-3
ISSN 1651-9256
Printed by E-Print AB, Stockholm, Sweden

Abstract

Fog computing is a distributed computing paradigm that brings data processing from remote cloud data centers into the vicinity of the edge of the network. The computation is performed closer to the source of the data, and thus it decreases the time unpredictability of cloud computing that stems from (i) the computation in shared multi-tenant remote data centers, and (ii) long distance data transfers between the source of the data and the data centers. The computation in fog computing provides fast response times and enables latency sensitive applications. However, industrial systems require time-bounded response times, also denoted as Real-Time (RT). The correctness of such systems depends not only on the logical results of the computations but also on the physical time instant at which these results are produced. Time-bounded responses in fog computing are attributed to two main aspects: computation and communication.

In this thesis, we explore both aspects targeting soft RT applications in fog computing in which the usefulness of the produced computational results degrades with real-time requirements violations. With regards to the computation, we provide a systematic literature survey on a novel lightweight RT container-based virtualization that ensures spatial and temporal isolation of co-located applications. Subsequently, we utilize a mechanism enabling RT container-based virtualization and propose a solution for orchestrating RT containers in a distributed environment. Concerning the communication aspect, we propose a solution for a dynamic bandwidth distribution in virtualized networks.

Sammanfattning

Fog computing är ett paradigm för distribuerad databehandling som innebär att databehandling från avlägsna molndatacenter kommer in i närheten av nätverkets kant. Beräkningen utförs närmare datakällan och minskar därmed den tidsmässiga oförutsägbarheten hos cloud computing som beror på i) beräkningen i gemensamma fjärrdatacenter med flera innehavare och ii) dataöverföringar på långa avstånd mellan datakällan och datacentren. Beräkningen i fog computing ger snabba svarstider och möjliggör latenskänsliga tillämpningar. Industriella system kräver dock tidsbundna svarstider, som också betecknas RT. Korrektheten hos sådana system beror inte bara på de logiska resultaten av beräkningarna utan också på det fysiska ögonblick då dessa resultat produceras. Tidsbegränsade svarstider inom fog computing beror på två huvudaspekter: beräkning och kommunikation.

I avhandlingen utforskar vi båda aspekterna med fokus på mjuka RT-tillämpningar inom fog computing där användbarheten av de producerade beräkningsresultaten försämras när tidsfristerna inte hålls vid missade deadlines. När det gäller beräkningen ger vi en systematisk litteraturstudie om en ny lättviktig RT-containerbaserad virtualisering som säkerställer rumslig och tidsmässig isolering av samlokaliserade tillämpningar. Därefter utnyttjar vi denna virtualiseringsteknik och föreslår en lösning för orkestrering av RT-containrar i en distribuerad miljö. När det gäller kommunikationsaspekten föreslår vi en lösning för dynamisk bandbreddsfördelning i virtualiserade nätverk.

Acknowledgment

Thank you Moris Behnam, Alessandro Papadopoulos, Mohammad Ashjaei and Silviu Craciunas, for supervising my work, for your support, valuable insights, and patience.

Apart from the knowledge gained throughout the course of my studies, I consider meeting countless inspiring individuals as the greatest asset. Thank you Robbert Jongeling, Salman Shaik, Gita Frasheri, Nitin Desai, Lan Van Dao, Jean Malm, Malvina Latifaj, Zeinab Bakhshi Valojerdi, Taufik Akbar Sitompul, Shahriar Hasan, Damir Bilic, Christoph Salomon, Zenepe Satka, Enxhi Ferko, Predrag Filipovikj, Elena Lisova, Muhammad Abbas, Filip Markovic, Soheila Sheikh Bahaei, Malina Adach, Sasikumar Punnekkat, Hans Hansson, Radu Dobrin, Mohammad Loni, Joana Silva, Branko Miloradovic, Jakob Danielsson, Mir Riyanul Islam, Shunmuga Priyan Selvaraju, Nandinbaatar Tsog, Le-Nam Hoang, Anna Friebe, Daniel Bujosa Mateu, Thomas Nolte, Saad Mubeen, Hossein Fotouhi, Leo Hatvani, Aida Causevic, Adnan Causevic, Paul Pop, Wilfried Steiner, Fereidoun Moradi, Auday Al-Dulaimy, Antonio Cicchetti, Adnan Ghaderi, Anders Lager, Bahar Houtan, Bahram Zarrin, Bjarne Johansson, Björn Leander, Neda Maleki, Eduard Paul Enoiu, Federico Ciccozzi, Fredrik Ekstrand, Gunnar Widforss, Jan Carlson, Svetlana Girs, Nikola Petrovic, Jan Ruh, Alexandre Silva Venito, Eleftherios Kyriakakis, Vasileios Karagiannis, Patrick Denzler, Mohamadreza Barzegaran, Cosmin Florin Avasalcai, Koen Pieter Tange, Marine Kadar, Jia Qian, Stefan Schulte, Guillermo Rodriguez-Navas, and many more.

Václav Struhár
Västerås, September 2021

List of Publications

Papers included in this thesis[1]

Paper A: 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. *Fog-based Industrial Robotic System: Applications and Challenges.* In the 25th International Conference on Emerging Technologies and Factory Automation (ETFA 2020).

Paper B: Václav Struhár, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos. *Real-Time Containers: A Survey.* In the 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020).

Paper C: Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos. *REACT: Enabling Real-Time Container Orchestration.* In the 26th International Conference on Emerging Technologies and Factory Automation (ETFA 2021).

Paper D: Václav Struhár, Mohammad Ashjaei, Moris Behnam, Silviu S. Craciunas, and Alessandro V. Papadopoulos. *DART: Dynamic Bandwidth Distribution Framework for Virtualized Software Defined Networks.* In the 45th Annual Conference of the IEEE Industrial Electronics Society (IECON 2019).

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

Related publications, not included in this thesis

Paper E: Shaik Mohammed Salman, Václav Struhár, Alessandro V. Papadopoulos, Moris Behnam, Thomas Nolte. "*Fogification of industrial robotic systems: research challenges.*" In the 1st Workshop on Fog Computing and the IoT (Fog-IoT 2019).

Paper F: Václav Struhár, Moris Behnam, Alessandro V. Papadopoulos. "*Work-in-Progress: Fog Computing for Adaptive Human-Robot Collaboration.*" ACM SIGBED International Conference on Embedded Software (EMSOFT 2018).

Paper G: Mirgita Frasheri, Václav Struhár, Alessandro V. Papadopoulos, Aida Čaušević. "*Ethics of Autonomous Collective Decision-Making:the CAESAR Method.*" Submitted to: Springer Journal, Science and Engineering Ethic

List of Acronyms

API	Application Programming Interface
BE	Best Effort
CLM	Container Level Metrics
COTS	Commercially available Off-The-Shelf
DART	Dynamic Bandwidth Distribution Framework
EDF	Earliest Deadline First
FEC	Fog and Edge Computing
FIFO	First In, First Out
HD	High Definition
IIoT	Industrial Internet of Things
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
NFV	Network Function Virtualization
OSLM	Operating System Level Metrics
OS	Operating System
PLC	Programmable Logic Controller
QoS	Quality of Service
QoS	Quality of Service
RPi	Raspberry Pi

RQ	Research Question
RTAI	Real Time Application Interface
RTOS	Real Time Operating System
RT	Real-Time
SDN	Software Defined Networking
TSN	Time Sensitive Networks
VLAN	Virtual LAN
VM	Virtual Machine
vCPU	virtual CPU

Contents

I

Thesis

Chapter 1

Introduction

Fog computing strives to extend the capabilities of the cloud to the edge of the network and overcomes some of the main limitations of cloud computing, for instance, tackling the limitations of unbounded communication latency [1], variance in bandwidth availability [2], security, and privacy aspects [3] of offloading data to remote data centers. Fog computing enables the benefits of offloading data collection and decision-making even in application domains that require low latency processing, for example, in industrial automation [4], smart manufacturing, or telecommunication [5]. Fog computing decreases the communication latency for critical data, which can then be processed on the decentralized computational devices on the network's edge. To meet stringent requirements demanded by industrial applications, fog computing has to provide time determinism that bounds the reaction time to environmental events [6], which is known as Real-Time (RT) systems. To bound the reaction time, fog computing requires both communication and computation to be time deterministic. While guaranteeing deterministic communication behavior between compute nodes can be achieved through the use of, for instance, TTEthernet [7] or Time Sensitive Networks (TSN) [8] enabled by specialized hardware, guaranteeing the time deterministic behavior of computation can be challenging.

Fog computing utilizes shared computational platforms to host several diverse applications that pose comprehensive resource and timing requirements. Fog computing consists of distributed heterogeneous computing devices based on Commercially available Off-The-Shelf (COTS) components and general-purpose Operating Systems (OSs) whose objectives are to make the average computation faster [9] and to provide good interactive performance while maximizing the overall utilization [10]. However, these technologies provide limited RT support. The employment of the COTS components may impair the timing predictability of co-located applications due to the simultaneous access to shared resources. Moreover, one of the main enablers of fog computing, resource virtualization, may introduce an additional source of unpredictability [11]. The presence of shared resources facilitates neither a composable analysis nor the computation of upper bound on the key timing parameters [9]. Moreover, the workload characterization of applications deployed in fog computing is complex and may change rapidly over time. Hence, providing time predictability in fog computing is challenging. The goal of this thesis is to explore both computational and communication areas of fog computing and introduce mechanisms that improve time predictability in such distributed systems.

In this thesis, we motivate the research problem of the need for RT fog computing. Subsequently, we provide a systematic literature survey on RT container-based virtualization as a lightweight technology that enables the co-existence of spatially and temporally isolated applications in a shared platform. We focus on the RT aspect of such virtualization. After that, we introduce a solution that enables the orchestration of RT containers across a cluster of hosts taking into account the timing requirements and possible timing interference. The solution includes the architecture, performance metrics of container-based virtualization, and implementation of an extension to the well-established orchestrating systems know as Kubernetes. Finally, to complement the computational part of fog computing, we focus on the communication part and we propose a solution for a dynamic distribution of network bandwidth in large virtualized networks to maintain a certain level of Quality of Service (QoS). Figure 1.1 depicts the context of the thesis. Although all the parts of the thesis (virtualization, orchestration, and bandwidth management) can stand separately, we frame the research under the fog computing theme.

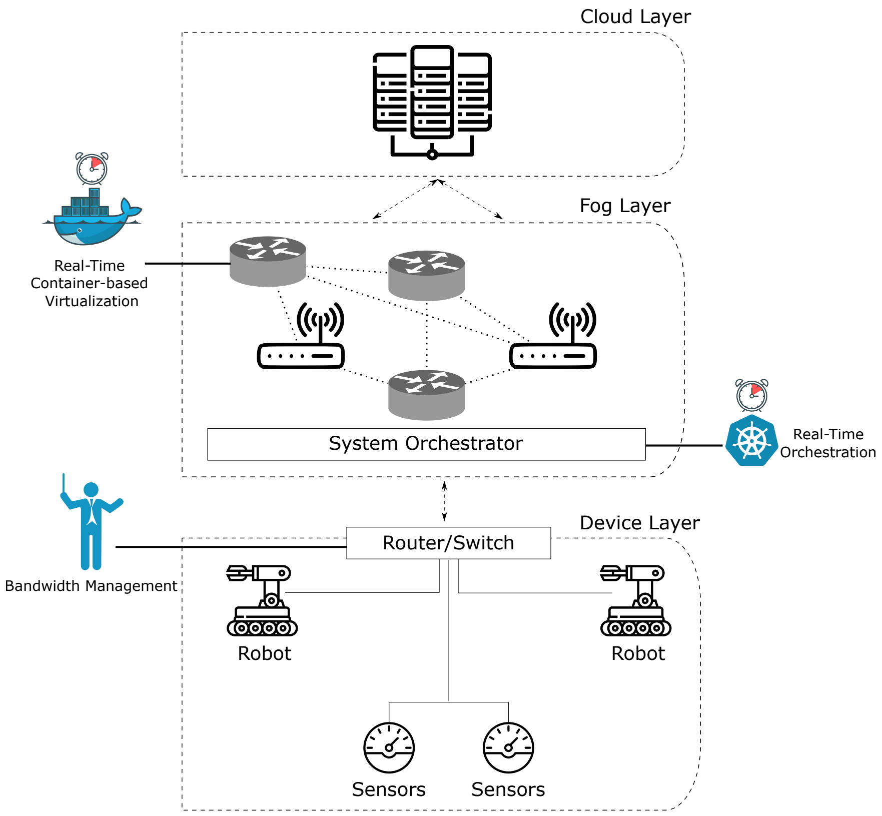


Figure 1.1. The context of the research carried out in this thesis.

1.1 Thesis Overview

This thesis is based on a collection of papers and is divided into two parts. The first part provides an introduction to the topic, the research overview, and the relation between included papers. Within the first part, Chapter 2 provides background and related work of fog computing, RT virtualization, RT orchestration, and communication bandwidth management. Chapter 3 defines research questions and introduces the research process adopted in this thesis. Chapter 4 summarizes the thesis contributions. Finally, Chapter 5 concludes the thesis, provides a discussion and presents the future work. The second part of this thesis includes a collection of published papers which their overview is provided below.

Paper A

Title: Enabling Fog-based Industrial Robotics Systems [12]
Authors: 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.
Status: Published in ETFA 2020
Abstract: Low latency and on-demand resource availability enable fog computing to host industrial applications in a cloud-like manner. One industrial domain which stands to benefit from the advantages of fog computing is robotics. However, the challenges in developing and implementing a fog-based robotic system are manifold. To illustrate this, in this paper we discuss a system involving robots and robot cells at a factory level and then highlight the main building blocks necessary for achieving such functionality in a fog-based system. Further, we elaborate on the challenges in implementing such an architecture, with emphasis on resource virtualization, memory interference management, RT communication, and system scalability, dependability, and safety. We then discuss the challenges from a system perspective where all these aspects are interrelated.
Paper contributions: The contribution of this paper is to identify the key research challenges in the implementation of fog computing in industrial areas. The paper discusses key aspects such as resource orchestration, network scalability, virtualization, and memory management techniques supported by RT communication paradigms. Further, the paper identifies inter-relations between the identified challenges.
My role: The paper is a joint work composed with colleagues from the FORA[1] project. Together with Salman, I was the main driver of the paper and we completed major parts of the paper together. I have completed the sections related to my research (section IV.A, virtualization, and section IV.B,

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

orchestration), and I have also defined the system architecture (Section II).

Paper B

Title: Real-Time Containers: A Survey [13]
Authors: <u>Václav Struhár</u>, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos
Status: Published in Fog-IoT 2020
Abstract: Container-based virtualization has gained significant importance in the deployment of software applications in cloud-based environments. The technology entirely relies on operating system features and does not require a virtualization layer (hypervisor) that introduces a performance degradation. Container-based virtualization allows to co-locate multiple isolated containers on a single computation node as well as to decompose an application into multiple containers distributed among several hosts. Such technology seems promising in other domains as well, e.g., in industrial automation, automotive, and aviation industry where mixed-criticality containerized applications from various vendors can be co-located on shared resources. However, such industrial domains often require RT behavior (i.e, a capability to meet predefined deadlines). These capabilities are not fully supported by container-based virtualization yet. In this work, we provide a systematic literature survey study that summarizes the effort of the research community on bringing RT properties in container-based virtualization. We categorize existing work into main research areas and identify possible immature points of the technology.

Paper contributions: The paper provides a systematic literature survey on the novel RT container-based virtualization. It provides an overview of enabling technologies for such virtualization. In The last part, the paper identifies pitfalls, challenges, and future research directions for RT container-based virtualization.

My role: I was the primary driver of the paper including formulating the problem and writing the paper. The supervising team provided their valuable feedback during the work.

Paper C

Title: REACT: Enabling Real-Time Container Orchestration
Authors: Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos
Status: Published in ETFA 2021
Abstract: Fog and edge computing offer the flexibility and decentralized architecture benefits of cloud computing without suffering from the latency and security issues inherent in the cloud. This makes fog computing very attractive in RT and safety-critical applications, especially if combined with container-based technologies. Whereas different orchestration mechanisms are available to manage the container placement based on their resource demand, no orchestration tool is considering RT requirements for containerized applications. In this paper, we present the architecture and design of an RT container orchestrator based on Kubernetes. Moreover, this paper defines metrics for the performance evaluation of RT containers and describes an initial model for allocating a mixture of RT and non-RT containers. We present an initial implementation of our RT container extension and evaluate its feasibility on Linux-based systems.
Paper contributions: The paper presents an orchestrating system that allows the distribution of a mixture of RT and non-RT containers. The system considers the timing requirements of RT containers in the container placement process. The paper includes the architecture of such a system, container performance metrics, a mathematical model of the orchestration problem, and implementation in an existing open-source orchestration system Kubernetes.
My role: I was the primary driver of the paper including formulating the problem, writing the paper, implementing the extension of Kubernetes, and performing the experiments. The supervising team provided their valuable feedback during the work.

Paper D

Title: DART: Dynamic bandwidth distribution framework for virtualized software-defined networks [14]
Authors: Václav Struhár, Mohammad Ashjaei, Moris Behnam, Silviu S. Craciunas, Alessandro V. Papadopoulos

Status: Published in IECON 2019
Abstract: In this paper we address a network architecture that uses a combination of network virtualization and software defined networking in order to reduce complexity of network management and at the same time support high quality of service. Within this network architecture, we propose a framework to be able to dynamically distribute the network bandwidth to various services such that the network resources are utilized efficiently. In many industrial domains, multiple services may use the same hardware platform for the sake of better resource utilization. Therefore, bandwidth distribution among the services should be done efficiently during run-time. We also develop an admission control in this framework which dynamically coordinates the bandwidth distributions based on the requested quality of services. We show the applicability of the proposed framework by implementing it on a common Software Defined Networking (SDN) controller. Moreover, we conduct a set of experiments to show the performance of the proposed framework.
Paper contributions: In this paper, we propose a framework, which we name *dynamic bandwidth distribution (DART)*, based on a virtualized SDN architecture that makes the fully dynamic bandwidth allocation on a physical network feasible. Moreover, we propose an admission control mechanism to distribute the network bandwidth during run-time based on the QoS level requested by the Industrial Internet of Things (IIoT) devices. The admission control mechanism resides within the proposed framework. We also show the applicability of the proposed framework on a use case study where the proposed admission control mechanism is implemented within an SDN controller, known as Floodlight. Finally, we conduct a set of experiments to present the performance of the implemented framework and mechanism.
My role: I was the primary driver of the paper including formulating the problem, writing the paper, implementing SDN controller, and performing the experiments. The supervising team provided their valuable feedback during the work.

Chapter 2

Background and related work

In this section, we provide a background and related work to the fog computing paradigm, RT virtualization that enables time deterministic computation, RT orchestration that allows distributing RT applications in a cluster of computing nodes, and a communication bandwidth management that improves communication related QoS in large computer networks.

2.1 Fog computing

The term fog computing has been coined in [1] as a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional cloud computing data centers near the edge of the network. The platform overcomes drawbacks of cloud computing issuing from bandwidth limitations and unpredictable network links towards the remote cloud data centers [15]. Fog computing similarly brings services as the cloud in the vicinity of the source of the data [16], it offers on-demand resource provisioning and elastic infrastructure for applications. The computation is distributed among participating devices located at the edge of the network, core network, and cloud. The interconnected devices that share part of their resources are denoted as fog nodes (sometimes also denoted as micro data centers and cloudlets [17]). The network of fog nodes, called the fog layer, comprises an intermediate layer between the cloud and device layer. The generic architecture of fog computing is depicted in Figure 2.1.

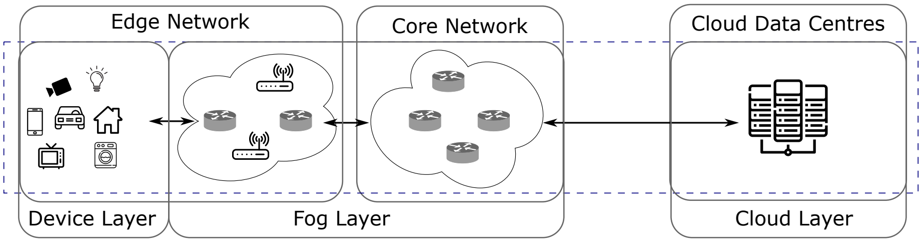


Figure 2.1. Generic fog computing architecture [16].

The participating devices are distributed between three layers consisting of (i) cloud layer, (ii) fog layer, and (iii) device layer. The cloud layer provides a high computing capacity but offers limited-time predictability due to varying data transmission latencies and unpredictable resource sharing amongst multiple tenants. The fog layer provides an elastic environment in the vicinity of the origin of the data. It consists of many interconnected physical fog nodes that are capable of hosting software applications. While the processing power of the fog layer is lower than that of the cloud layer, the network latencies, are shorter and more predictable. The device layer consists of resource-limited devices such as sensors and actuators that typically pre-process data and transmit it to fog nodes for further processing.

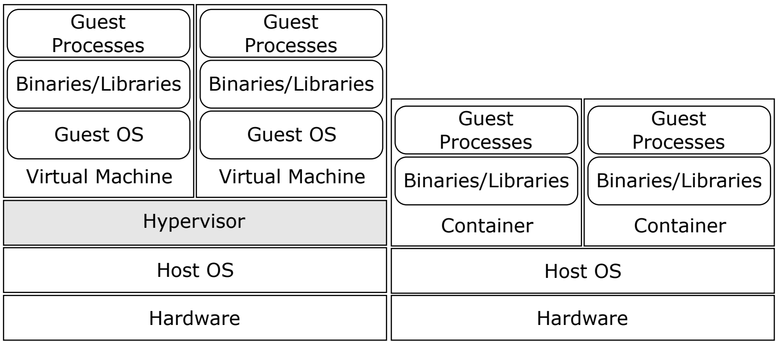
In industrial domains (e.g., in the context of industrial automation or smart manufacturing), fog computing strives to meet strict requirements of industrial applications where the time aspects are crucial. For example, robot control applications require reaction time in milliseconds [18] and once the timing requirements are not met, the manufacturing process may be severely impaired. In the current state of the practice, critical automation systems are controlled by dedicated controllers designed to meet timing constraints under any circumstance [19]. In contrast, applications in fog computing are distributed amongst fog nodes and a cloud and share resources with other co-located applications. The resource sharing between applications and data transmissions may lead to time unpredictability.

The correctness of an RT application depends not only upon its logical correctness but also upon the time in which it is performed [20]. RT systems are classified into three categories: hard, firm, and soft, distinguished by the consequences of missing a deadline. Hard RT systems have a set of strict deadlines, missing a deadline is considered a system failure and may cause a catastrophic consequence. Missing a deadline in soft and firm RT systems may lead to QoS degradation, however, the system failure does not occur. While the utility of the result becomes zero after the deadline miss in firm RT systems, the utility of the result decreases with the deadline miss in soft RT systems.

2.2 Resource virtualization

Resource virtualization is the key enabler for fog computing [21] as it enables the consolidation of applications with various requirements in a shared platform. It provides guest environments separated from the underlying hardware and gives the applications an illusion to be running on exclusive hardware. Virtualization enables several new capabilities, e.g., an entire environment including the OS, and all the applications running on it can be stopped, saved, and then restored and restarted on a different physical machine [22]. Virtualization provides spatial isolation that guarantees that an application can not alter the private data of other co-located applications. More specifically, virtualization offers the following three main features [22]: Isolation, Partitioning, and Encapsulation. Isolation ensures that the applications can not alter the data of other co-located applications in another partition. A crash of an application in one partition should not affect applications in other partitions. Partitioning ensures that several applications and OSs are supported in a single physical computing system by partitioning (separating) the available resources. Encapsulation provides the ability to encapsulate the entire environment (applications, their dependencies, and even the entire OS) into a distributable file.

There are two common virtualization technologies: hypervisor- and container-based virtualization. Hypervisor-based virtualization utilizes a hypervisor that is a software layer that creates the different partitions within which each virtualized instance of an OS runs. In contrast, container-based virtualization utilizes kernel features to create an isolated environment for processes [23]. In the following text, we describe the individual technologies. The overview of both technologies is depicted in Figure 2.2.



(a) Hypervisor-based virtualization (type 2). (b) Container-based virtualization.

Figure 2.2. Comparison of hypervisor-based and container-based virtualization.

2.2.1 Hypervisor-based virtualization

For a long time, the term virtualization implied talking about hypervisor-based virtualization [23]. This type of virtualization introduces a concept of a hypervisor, which is a software layer that creates and runs Virtual Machines (VMs) that emulate the complete hardware of a computer [23]. This technology allows one computer to host multiple guests VM. The hypervisor runs directly on the hardware (type 1 hypervisor) or on the top of a host's operating system (type 2 hypervisor) [24]. Inside of each VM instance runs a full OS and installed software [25].

The need for running an entire OS inside of each instance of VM results in higher overheads and worse performance in comparison to container-based virtualization. For example, Felter et al. [26] experience 40% slowdown when using Kernel-based Virtual Machine (KVM) virtualization, while container-based virtualization shows only a 2% slowdown in the experiment. Joy et al. [27] show that hypervisor-based virtualization achieves three times lower performance than container-based virtualization. Shichao et al. [28] provide a comparison between hypervisor-based virtualization and container-based virtualization showing that the instantiation time of a VM is 100 times slower and both image size and memory footprint is 20 times bigger than container-based virtualization.

Using VMs is not adaptable to fog computing. For example, the boot-up time of a VM is several minutes, which is too long for RT applications. For a large number of access points, the characteristics of fog computing are affected by the number of VMs because the performance of physical machines is degraded when the number of VMs increases. The overhead of a hypervisor exponentially increases when more VMs run within the same machine [29, 30]. Moreover, since VMs are resource-intensive, they are not the best virtualization approach for resource-constraint devices [30].

2.2.2 Container-based virtualization

In contrast, container-based virtualization does not emulate an entire VM, instead it utilizes kernel features to create an isolated environment for processes [23]. It introduces a concept of containers that are sets of resource-limited processes that are isolated from the rest of the system and other containers. This type of virtualization utilizes the resources more efficiently and permits the deployment of more virtualized applications [27]. Container-based virtualization does not impose any requirements on hardware support for virtualization, and hence, such virtualization can be utilized on a broader range of devices. The container-based virtualization is achieved by the host OS [31] (in this thesis we consider Linux), namely by utilizing two kernel features: namespaces and control groups (cgroups). Namespaces virtualize global resources (e.g., processes, network, inter-process communication) in the way that a group of processes can see and use one subset of resources while another group can use a different subset of resources. Cgroups provide a mechanism for aggregating and partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour[1]. It allows to organize processes hierarchically and distribution system resources (e.g., CPU, memory) along the hierarchy.

Containers co-located on a single computing node run as user-space isolated tasks and make use of functions of the host's OS. From the user's perspective, each container appears and executes like a standalone OS [31]. In comparison to hypervisor-based virtualization, container-based virtualization provides a

[1]`https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`

negligible overhead, is more resource-efficient (e.g., 29.4 times less memory use than a VM in a use-case presented in [32]), has higher flexibility, provides fast booting times [33], and enables a near-native performance [31, 34, 35, 26].

Fog computing can strongly benefit from this type of virtualization as the lower overhead allows to co-locate a higher number of applications, wider hardware support allows to employ more devices and low startup times allows to deal with elasticity (i.e, fast deployment of applications). On the other hand, the container-based virtualization provides a low level of isolation for memory, disk, and network operations [36]; hence, such operations may lead to degrading QoS of the virtualized applications and may damage their time predictability.

2.2.3 Real-time container-based virtualization

In addition to spatial isolation, RT application requires temporal isolation in order to minimize temporal interference between co-located applications. Several hypervisors support RT virtualization, for instance, RT-Xen [37], μRTZVisor [38], Hermes [39]. However, they do not solve high overheads and slow instantiation times, and high memory footprint [28] inherent to hypervisor-based virtualization.

The support of the RT capability of container-based virtualization is a novel research topic as our research shows [13]. The first research in this direction started in 2015 as shown in Figure 2.3, while the container-based virtualization itself appeared in the current form in 2006 with *process containers* developed by Google. The RT capabilities of container-based virtualization depend on the underlying OS. In this thesis, we consider the general-purpose OS Linux as it offers several benefits for fog computing: wide hardware support, virtualization capabilities, and light-weightness. While Linux was not originally designed for RT computing, there are considerable efforts to improve the time determinism on several levels, e.g., introducing RT scheduling policies as a standard kernel feature and providing time-predictable kernel behavior through full preemption. In recent years, Linux has gained several characteristics of a real-time operating system [40]. On the task scheduling level, there are RT scheduling policies in the Linux kernel: First In, First Out (FIFO) and Round Robin combined with priority queues to provide different priority levels for tasks, and Earliest Deadline First (EDF) and Constant-Bandwidth Server that prioritizes tasks dynamically

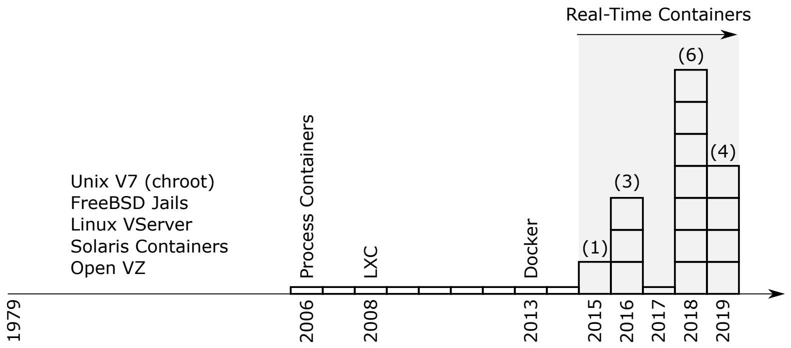


Figure 2.3. Timeline of RT container-based virtualization with important milestones. The numbers in brackets denote the number of publications related to RT container-based virtualization [13]
.

according to their deadlines [41]. As shown later in this thesis, there is an effort to connect the container-based virtualization with the improved Linux RT features.

2.3 Real-Time container orchestration

While the RT container-based virtualization enables co-location of spatially and temporally isolated applications on compute nodes, the container orchestration automates the deployment, execution, and maintenance of containers in clusters of heterogeneous computing nodes [42]. The orchestration consist of two major components: the orchestrator and a local container manager that provides functionalities for setting up containers in the local node. The Figure 2.4 shows phases of a container life cycle. A container manager (e.g., Docker) provides API to support the following phases: acquire, build, deliver containers, while the container orchestrator provides functions to automatize deployment, run, and maintenance of containers in a cluster of compute nodes [43]. The acquire phase gets the containers from a repository, build phase packs the application and libraries for a container image, deliver phase brings the application in production, deploy, run, maintain [43].

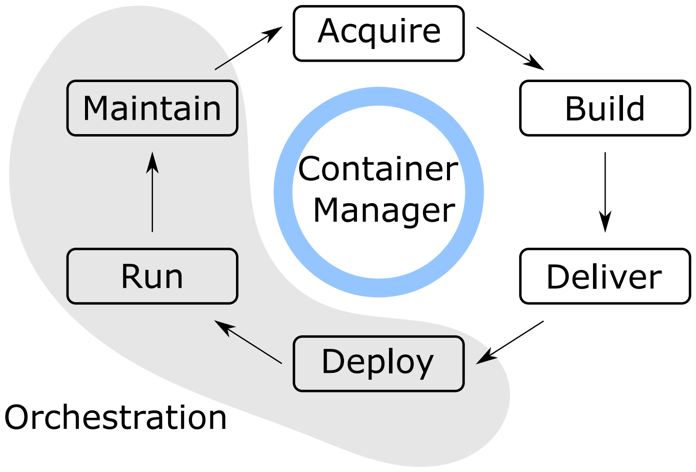


Figure 2.4. The container life cycle [43].

The principal functionality of container orchestrators is to automate the deployment of containers in a cluster of compute nodes following placement policies and user-supplied placement constraints. The ultimate goal of the orchestrators is to choose an optimal compute node to deploy the requested containers on. The orchestrator matches the resource requirements with the resource capacity of the nodes, e.g., CPU, memory, and disk storage capacity, and applies strategies to maximize the performance (e.g., the highest spread of containers). Additionally, orchestrators address fault-tolerance of the deployed containers, scaling or removing containers, load balancing, container health monitoring, and efficient resource utilization [42].

There are several available container orchestrating tools, for instance Kubernetes[2] and Docker Swarm[3]. The container orchestrators can be easily extended to be used in fog computing as shown in [44, 45]. However, the current orchestrators [42] and their fog computing extensions lack support for the deployment and maintenance of containers based on their RT requirements and RT-related QoS.

[2]https://kubernetes.io
[3]https://docs.docker.com/engine/swarm

Although container orchestration, per se, is a mature area from both research and practice perspectives, it still lacks some (even non-RT-related) functions [46]. For instance, Casalicchio et al. [46] emphasize in their container orchestrator survey the research challenges in the following areas: advanced monitoring and workload characterization, container performance models, and adaptation models for container orchestration. Such research challenges are amplified in RT computing where they add an additional dimension of the problem in terms of defining RT-related QoS metrics, their measurement, and prediction.

There are attempts to connect the virtualization and orchestration for instance in [47] Xi et al. use OpenStack[4] to orchestrate RT VMs based on RT-Xen. However, this work does not consider possible interference between VMs.

The challenge is to connect the emerging RT container-based virtualization technology with the orchestrating technology to enable matchmaking between RT containers and compute nodes taking into account RT requirements of the containers and RT capabilities of the compute nodes. Moreover, it is crucial to consider the inherently low isolation level of co-located containers that may influence the overall performance of the system.

2.4 Communication bandwidth management of large IoT networks

Computer infrastructures may consist of a large number of interconnected devices. It is estimated that the total number of devices will grow up to 80 billion in 2025[5]. With such many devices, the complexity of network management increases significantly. The networking and cloud community will have to find ways on many levels to deliver services that satisfy intrinsic demands on availability and QoS, while also requiring support for heterogeneous and multi-vendor equipment [48]. On the computing level, the shift to edge and fog computing is inevitable to alleviate the limitations of cloud computing. On the networking level, the promising technology to mitigate the complexity of the net-

[4]https://www.openstack.org/

[5]https://idc-cema.com/dwn/SF_177701/driving_the_digital_agenda_requires_strategic_architecture_rosen_idc.pdf

works and to provide dynamically QoS control is a mixture of SDN and network virtualization [49]. SDN decreases the complexity of network management by decoupling network control and forwarding functions [50] (Figure 2.5). Network virtualization provides isolation between multiple network services, simplifies network management, and abstracts the physical network resources [48].

The combination of SDN and network virtualization may not be fully compatible as network virtualization disrupts the centralized view of the network expected by SDN. Such a combination of the two technologies may introduce some issues, for instance, a problem of shared resources among multiple virtual networks [48] when it is not clear how to control the incoming traffic to the shared resources.

SDN is comprised of three layers: a) The *Application layer* consists of SDN business applications written in common languages controlling the underlying SDN enabled devices via the SDN controller, b) The *Control layer* fetches various statistics from the physical devices (usage statistic, topology details, state details) and enables the communication between SDN applications and SDN devices, and c) The *Infrastructure layer* is composed of physical SDN switches. The need for service isolation and diverse resource requirements within one physical network brings the topic of network virtualization into the focus of the researcher community [51, 52]. Network virtualization enables the coexistence of multiple logical networks sharing the same underlying physical network [53]. One technique in this context is network slicing [54] where there is a division of the shared physical network into multiple logical isolated sub-networks (slices). Besides being isolated from each other, slices may be optimized for different purposes (e.g., high bandwidth High Definition (HD) video streaming, low latency video gaming) [54]. Slicing allows infrastructure providers to adapt the sharing of the underlying physical network to customer requirements while at the same time providing isolation of the network resources. Network virtualization requires a network hypervisor that creates an abstraction layer on top of physical hardware and allows the creation of virtual networks.

There are several resource reservation techniques in processor and communication domains which most of which focus on a static reservation of bandwidth. For instance, in the processor domain, supporting multimedia applications [55] and hierarchical reservation techniques [56] are presented, whereas

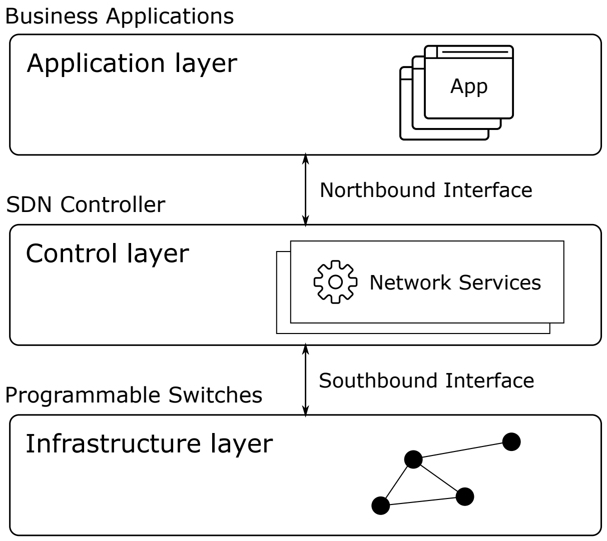


Figure 2.5. SDN Architecture separates a network into three layers: Application, Control, and Infrastructure layer.

in the distributed level communication bandwidth reservation for multimedia systems [57], adaptive QoS control [58] and a platform to support end-to-end timing [59] are presented.

Chapter 3

Research overview

3.1 Research goals and research Questions

The overall research goal of this thesis is *to improve soft RT predictability in fog computing systems* assuming COTS hardware-based systems and general-purpose OS Linux. We break down the overall research goal into three Research Questions (RQs). The outcomes of the RQs are published in relevant peer-reviewed conferences (e.g., ETFA, IECON) and workshops (e.g., Fog-IoT). The first RQ aims to identify technology, approaches, challenges, shortcomings of technology enabling RT computation in fog computing. The second RQ points to the problem of distribution of RT containers among a cluster of compute nodes. The third RQ focuses on a communication aspect to improve RT performance. The questions are elaborated below:

3.1.1 RQ1: *What are the main technologies, approaches, and challenges towards providing RT predictability in fog computing?*

Fog computing is a distributed paradigm that allows to seamlessly utilize resources at the edge of the network. Industrial domains often require a capability to meet predefined deadlines. However, fog computing, to the best of our knowledge, does not provide such capabilities yet. By answering this question, we identify the main technologies, approaches, challenges, and shortcomings of fog computing and RT container-based virtualization that is one of the main enablers of fog computing for the industrial domain.

3.1.2 RQ2: *How to distribute RT containers in fog computing to achieve soft RT behavior using Kubernetes?*

Following the previous question, the advance in RT container-based virtualization is not reflected in container orchestration systems. Existing orchestration systems do not take into account RT requirements of containerized applications and RT capabilities of compute nodes. In order to answer to this question, we propose an extension to an orchestration system Kubernetes that enables the orchestration of RT containers. We propose an architecture of the system, mathematical model, admission control mechanism, and validate the solution using experimental study.

3.1.3 RQ3: *How to improve a soft RT performance via quality of service management in virtualized SDN networks?*

The increasing number of network devices raises a need for network infrastructure sharing. An infrastructure can be shared between devices belonging to various domains having various QoS requirements. This setting calls for a combination of network virtualization to isolate individual domains and SDN that enables dynamic, programmatic control of a network. However, these two technologies are not fully compatible. SDN expects full knowledge of the network while the network virtualization breaks the network into smaller sub-networks. The answer to this question shows how to connect SDN and network virtualization and how to enable bandwidth management in case of overlapping resources present in multiple sub-networks.

3.2 Research Process

In this thesis, we use the hypothetico-deductive method described in [60] that includes steps depicted in Figure 3.1. To hypothesize the RQs and validate the proposed solutions, the research process adopts specific steps described

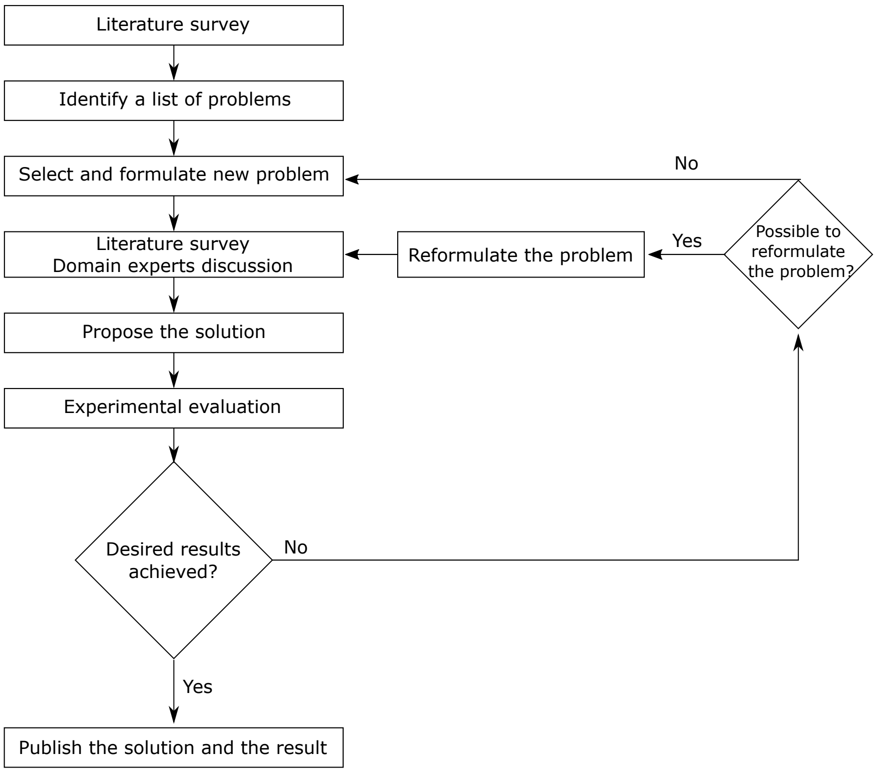

Literature survey
Identify a list of problems
Select and formulate new problem
Literature survey
Domain experts discussion
Propose the solution
Experimental evaluation
Desired results achieved?
No
Yes
Publish the solution and the result
Possible to reformulate the problem?
Yes
No
Reformulate the problem

Figure 3.1. The research methodology followed in this thesis.

below. The research is conducted in a collaboration with our industrial partners: TTTech[1] and with ABB[2]. We continuously discuss the research direction and results achieved in order to reflect industry needs and requirements. The process consists of the following steps:

1. **Literature survey**: Publications related to the thesis topic are reviewed and then a shortlist of the relevant literature is obtained as a result of this step.

[1]`https://www.tttech.com/`
[2]`https://abb.com`

2. **Identify a list of problems**: We identify issues in the emerging fog computing area related to soft RT behavior. The result of this step is a list of problems that can be solved with different techniques and technologies.
3. **Select and formulate a new problem**: From the list of problems identified in the previous step, we choose a specific problem that we clarify and formulate as a research question.
4. **Literature survey and domain expert discussion**: We perform a literature survey to identify the current state-of-the-art of each of the selected problems. We continuously discuss with the domain experts to assess the relevance of the problem and the feasibility of the possible solution from the industrial perspective.
5. **Propose the solution**: Based on the domain expert discussion, we propose a new solution that tackles the problem and overcomes the existing drawbacks of other solutions or improves the system performance compared to others.
6. **Experimental evaluation**: The proposed solution is implemented and evaluated through a set of experiments.
7. **Publish the solution**: After the evaluation of the solution, we publish it in a peer-reviewed article.

Chapter 4

Thesis contributions

This section provides a summary of the thesis contributions. Firstly, we study and compare methods and approaches to enable RT containers, we utilize the knowledge and propose a solution for the orchestration of RT containers across a cluster of hosts. Subsequently, we propose a solution for the dynamic distribution of network bandwidth in large virtualized networks to maintain a certain level of QoS.

4.1 C1: *Study and compare methods and approaches to enable RT container-based virtualization*

The first contribution is based on findings in Paper B [13] where we conduct a systematic literature survey on RT container-based virtualization. The research is motivated together with our industrial partner TTTech that is exploring possible techniques enabling co-location of multiple RT applications on a shared industrial edge computing platform[1]. An important aspect of such application co-location is the minimization of possible overheads and timing interference between the applications. The promising technique for that purpose is low footprint container-based virtualization, however, container-based virtualization does not support directly RT capabilities. In the survey, we extract available

[1] `https://www.tttech-industrial.com/products/nerve/`

literature related to RT and container-based virtualization. After the extraction, we aim to identify the techniques and approaches enabling RT container-based virtualization, the context where such virtualization is currently being used, and current shortcomings and immature aspects that prevent full adoption of RT container-based virtualization in the industry.

From the surveyed literature we identify and explain the main directions to enabling RT container-based virtualization, namely *PREEMPT_RT* patch-based, co-kernel based, hierarchical scheduling based. Below, we provide a brief description of these techniques:

- *PREEMPT_RT* patch-based approaches [61, 62, 63, 64, 65]: The research in this area aims to improve the time predictability of co-located containerized applications through the use of the *PREEMPT_RT* patch.
- Co-kernel methods: RT micro-kernel runs in parallel to Linux kernel. The RT co-kernel handles time-critical activities (for instance handling interrupts and scheduling RT threads), the standard Linux kernel runs only when the co-kernel is idle. In comparison to the *PREEMPT_RT*, the co-kernel approach offers lower latencies and lower jitter. On the other hand, it requires a special Application Programming Interface (API), tools, and libraries for application development. Additionally, there are impediments with scaling co-kernel solutions on large platforms
- Hierarchical scheduling: Inspired by a similar concept in the hypervisor-based virtualization where a global scheduler assigns CPU time for the VM, the second layer scheduler schedules the individual tasks of the VM.

In the surveyed literature, we recognize the following shortcomings and possible immature aspects in container-based virtualization related to the RT aspect that prevents such a technology to be adopted in the industry. It can be summarized into three categories: (i) tools, frameworks, and middleware support, (ii) RT communication support, and (iii) miscellaneous.

The literature included in the survey shows that there is a lack of tools, frameworks, and middlewares supporting RT container-based virtualization. For instance, there is a need for an orchestration tool that can schedule RT containers based on pre-configured capabilities [66]. Moga et al. [61] emphasize a lack of a middleware that is aware of both communication needs as well as run-time and performance isolation needs. Also, there is a need for a framework that exposes runtime requirements of RT applications running inside containers while optimizing the resources usages such that more applications can share resources. [61]. Related to the communication aspect, Goldschmidt et al. [63] and Moga et al. [61] put stress on insufficient communication methods and data management of RT containers. Hofer et al. [66] promote an investigation on container security restricted container access and intra-container communication. Additionally, there is a lack of latency and performance tests of recent releases of a patched Linux Kernel. As well as a proper analysis of the configuration of the Linux kernel parameters that may improve overall task determinism [66]. There is missing safety and security analysis of RT containers and vulnerability management for the acceptance in industry [67, 63].

4.2 C2: *Orchestration of RT containers*

Container orchestrators are well-established tools that are used in production environments to automate the deployment, management, networking, and scaling of containers. However, as the RT container-based virtualization is an emerging technology, the orchestrators do not consider RT requirements or RT capabilities of compute nodes [66, 61]. To bridge the gap, in Paper C, we propose an RT-aware container orchestration system that enables the scheduling of both RT and non-RT Best Effort (BE) containers while considering timing requirements defined for time-critical applications. We define a mathematical model for the RT components, performance metrics to evaluate the performance of RT containers, and we implement such a system.

4.2.1 Architecture

We propose an RT-aware orchestrating system that is based on the master-minion architecture that consists of a master node (denoted as RT orchestrator) and a set of minion compute nodes connected in a cluster as described in [42]. The core of the system is the master node that makes global decisions about the cluster; it receives users' requests for container deployments enhanced with RT requirements, continuously monitors states of compute nodes in the cluster, and schedules containers on computing nodes.

Every compute node, depicted in Figure. 4.1b, provides an environment for hosting RT containers[2] and contains a node agent that communicates with the master node through dedicated APIs. The node agent takes container deployment specifications defining container requirements and deployment parameters.

The master node depicted in Figure 4.1a is a central point in the architecture; it accepts user-defined container deployment specifications enhanced with RT interface and task annotations. It provides mechanisms for admission control and scheduling of containers. Additionally, it continuously collects performance Operating System Level Metrics (OSLM) and Container Level Metrics (CLM) metrics (described later).

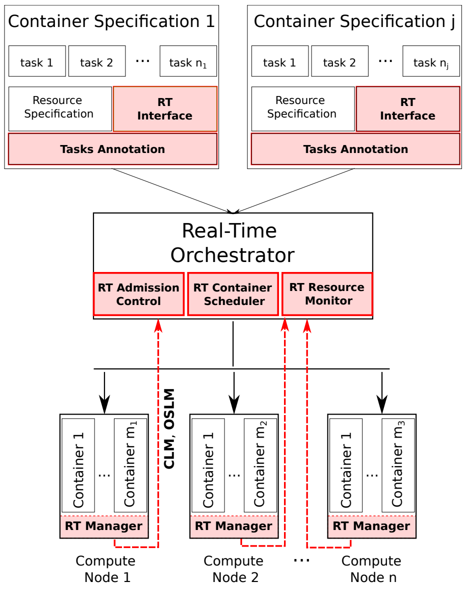
Each container deployment specification, which is supplied to the master node via a dedicated API, contains the specification of the RT interface and the container annotation. The RT interface specifies the CPU reservation (i.e., CPU budget over a time period) of the respective container following the periodic resource model of the hierarchical scheduling framework (c.f. [68]). The container specification contains the description of the tasks inside the container to compute the CLM during run-time.

The admission control determines if there are nodes in the cluster with enough available resources (e.g., memory and storage) to accommodate the resource demands of the new container. Moreover, it performs necessary utilization-bound schedulability tests that reject those nodes on which the RT timing requirements cannot be met.

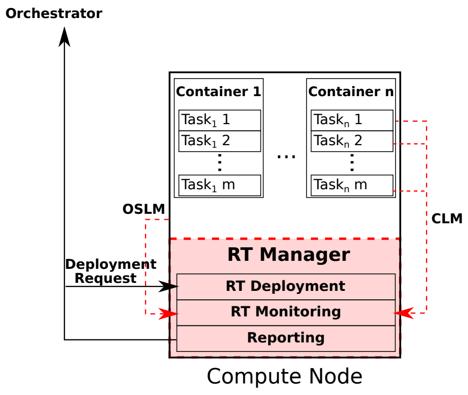
Performance metrics

To evaluate the performance of the system, we propose to use OSLM [69]. Such metrics are useful to estimate the suitability of the system to run RT tasks. For instance, Interrupts with a non-preemptable section that can influence the RT performance of the system [70], CPU Utilization, number of handled interrupts per second, number of I/O requests per second, and amount of data read/written.

[2]Enabled by the use of the hierarchical patch by Abeni et al. available at `https://github.com/lucabe72/LinuxPatches/tree/HCBS`.



(a) Real-time Aware Container Orchestration System.



(b) Compute Node.

Figure 4.1. A high-level architecture of the RT-aware orchestration system and the compute node

There are several tools for collecting performance data as shown in [71], for instance, *mpstat* and *iostat* tools collect the performance data from *proc* and *cgroups* directories to estimate the OSLM.

On the container level, there is a lack of measurement methodology, tools, and best practices, as well as a lack of metrics on the characterization of the container overhead [71]. Available tools, e.g., *docker stats* and *cAdvisor* allow estimating the basic set of container-related metrics (e.g., CPU and memory utilization). However, when considering the RT performance evaluation of containers, the available tools are lacking such capabilities. Therefore, we define CLM that evaluates the RT performance of tasks inside RT containers, namely: the *total number of deadline misses*, *maximum lateness*, *maximum response time*. The total number of deadline misses characterizes the total number of deadline

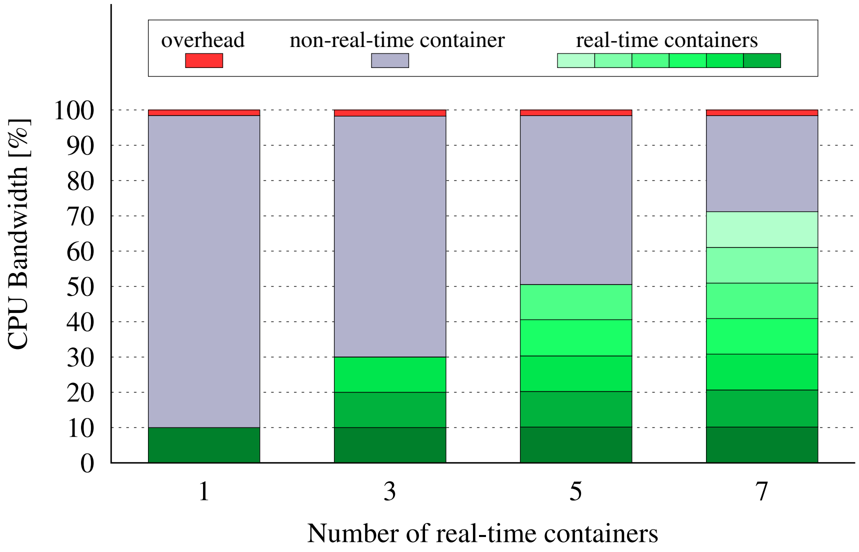


Figure 4.2. Distribution of CPU in a multi-container environment.

misses of tasks inside of the container within a time period, maximum lateness characterizes the maximum lateness encountered by a task inside the container within a time period, and response time characterizes the maximum response time encountered by a task inside the container within a time period.

We have performed a set of experiments that attempts to co-locate multiple RT and BE containers on a single compute node (Figure 4.2). We have attempted to attack the obtained CPU bandwidth that was reserved in the compute node by the orchestrator. One of the containers runs resource-intensive computation generating an excessive amount of I/O requests and interrupts. The results show that the RT containers are getting the reserved CPU resources. In future work, we plan to design more sophisticated experiments that will show the loss of performance and interference between containers.

4.3 C3: *Dynamic bandwidth management*

As the third contribution, we propose a framework Dynamic Bandwidth Distribution Framework (DART) based on a virtualized SDN architecture. It enables communication across SDN controllers in order to make the fully dynamic bandwidth allocation on a physical network feasible. The network virtualization splits the physical network into multiple sub-networks (also known as virtual networks or slices). Each sub-network is controlled by a single SDN controller that expects complete knowledge of the network. However, network virtualization limits the knowledge of the controllers to the corresponding sub-networks. Here, the problem emerges when two or more sub-networks share resources, for example, a file server or a fog node. The shared resources may experience network congestion as the access control is uncoordinated and divided between multiple SDN controllers. This problem is depicted in Figure 4.3.

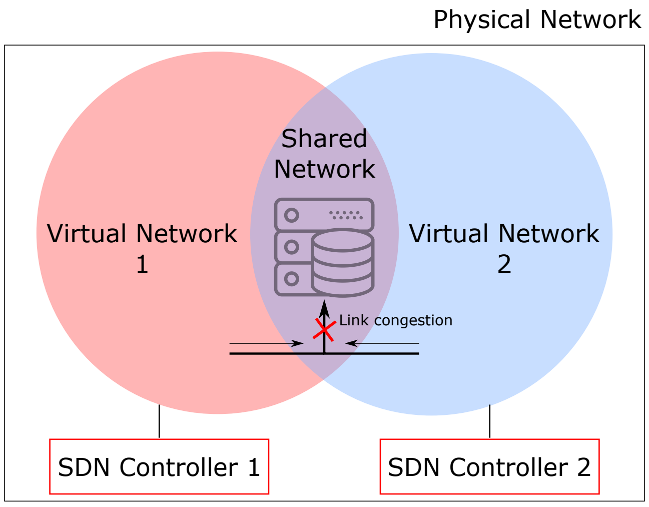


Figure 4.3. Overview of DART framework.

In Paper D, we propose the DART framework that enables the communication between SDN controllers. Moreover, we propose an admission control mechanism on the SDN controller level to distribute the network bandwidth during run-time that dynamically adapts the system based on the current need.

We also show the applicability of the proposed framework on a use case study where the proposed admission control mechanism is implemented within the Floodlight SDN controller[3].

The DART framework is a generic concept that can be applied to any virtualized SDN architecture. Figure 4.5 depicts the DART framework. On the bottom left side, an architecture with several sub-networks in a physical network is shown. Note that in this architecture we assume that the sub-networks can share a part of the physical network to increase the efficiency of utilizing the resources, however, the framework covers the cases with fully isolated sub-networks as well. The proposed framework is depicted on the right side of the architecture, which consists of two main components: a distributed component and a centralized component.

The distributed component deals with synchronizing the bandwidth management among the sub-network. As each SDN controller can only coordinate its own sub-network, it is essential to have a general view of the network status when allocating the bandwidth. The distributed component ensures that the SDN controllers collaborate on bandwidth management, leading to coherent bandwidth utilization.

The admission control mechanism is divided into centralized and distributed components. The centralized component contains the logic of bandwidth management. The distributed component, however, resides on top of the SDN controller to provide information about the corresponding sub-networks to the centralized component.

The sending nodes decompose the data into multiple data streams (see Figure 4.4). The data streams can have different priorities which are set by the sender nodes depending on the importance of the data. The primary goal of the admission control is to check the priority of the data streams and allocate bandwidth for the links that the data stream is transmitting. In order to do that the admission control defines a priority limit. Any received data stream with a priority higher than the priority limit will be forwarded to its destination, whereas the data streams with priority less than the priority limit will be prevented for transmission. The priority limit is defined in the centralized component of the

[3]`https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview`

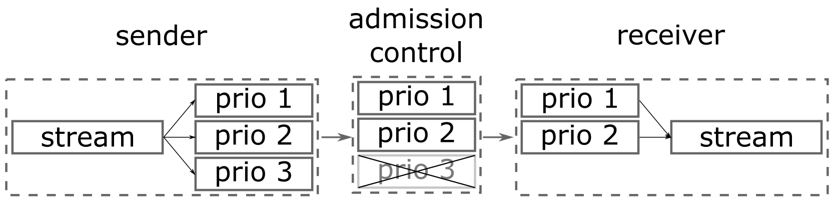


Figure 4.4. Admission control mechanism.

admission control for all sub-networks. In a normal case, priority limits are equal and thus the bandwidth is uniformly distributed among the sub-networks. If there is a request for priority limit change (detected by a distributed component), the centralized component adjusts the limits accordingly in order to a) increase bandwidth in the requesting sub-network, b) keep the total bandwidth used by all sub-networks constant.

We set up a testbed consisting of two virtual networks and we place two sending devices within each of the virtual networks. The receiving device is set on a shared segment of the two virtual networks and receives data from the sending devices. The result of the experiment is shown in Figure 4.6. The Figure shows the average network utilization, in the case of no bandwidth adaptation (Fig. 4.6a), and with bandwidth adaptation (Fig. 4.6b). In both cases, between time 60 and time 120, a motion is detected, and additional traffic is generated from the sending node 1. In Fig. 4.6a, the bandwidth limit is exceeded for all the periods when the motion is present, while Fig. 4.6b shows that the SDN controller 1 detects high importance traffic, and the system increases the priorities for sub-network 1 and decreases the priorities allowed for sub-network 2, resulting in a better allocation of the bandwidth over the high-priority traffic.

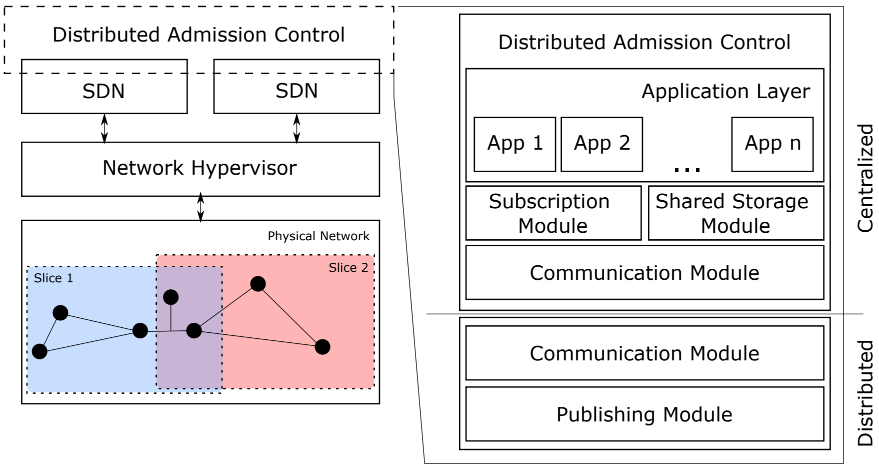
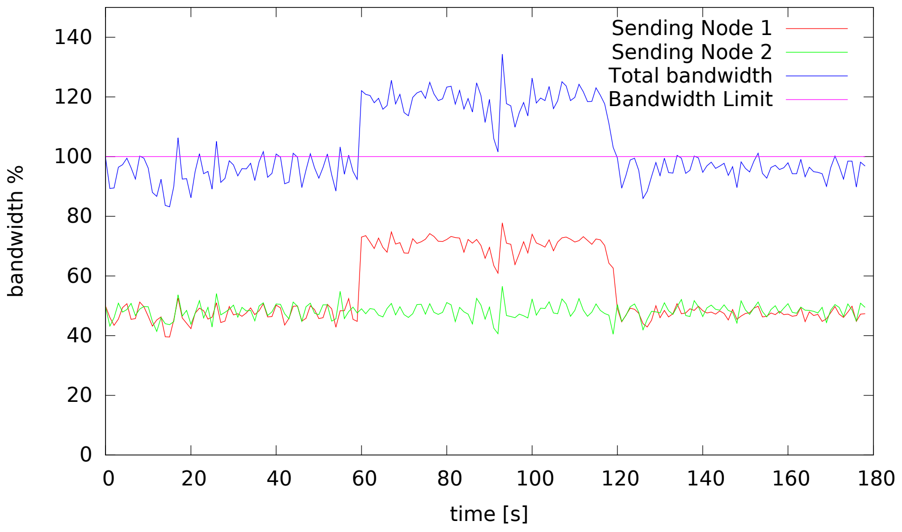
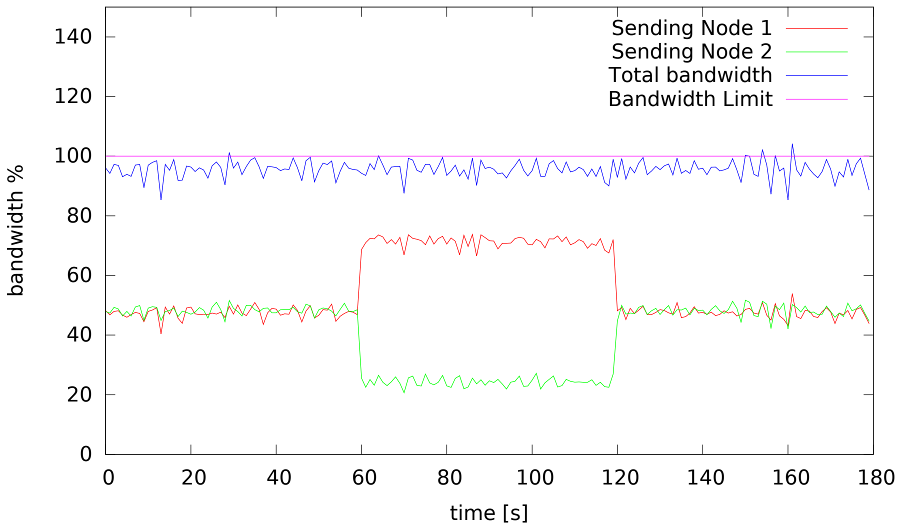


Figure 4.5. An architecture of a system using the DART framework.



(a) Without bandwidth adaptation.



(b) With bandwidth adaptation using the DART framework.

Figure 4.6. Average bandwidth utilization.

Chapter 5

Conclusions and future work

This Chapter concludes on the contributions and obtained results presented in the previous chapters, and outlines potential directions of research for future work.

5.1 Conclusions

Fog computing is an emerging paradigm that adds another layer between edge devices and the cloud. It helps to maintain the increasing number of devices that produce a massive amount of data that can not be properly handled by cloud computing. In comparison to cloud computing, one of the supreme benefits of fog computing is a reduction of network latencies and increasing time predictability. Such properties enable new applications that are not feasible with cloud computing. However, decreased latencies and predictability, per se, are not enough for RT systems that require stringent response time. The adoption of fog computing in such systems requires additional considerations and techniques to provide temporal isolation and guarantees of a certain level of time predictability.

In this thesis, we have explored container-based virtualization that has the potential to be the foundation of RT fog computing. It offers the desired properties for fog computing, namely: it is lightweight with low overheads (thus, it enables a large amount of co-located applications), and it provides rapid startup times (thus, it enables fast application migration). Nevertheless, container-based virtualization in general-purpose OS (e.g., Linux) lacks RT support. For that reason, we have performed a systematic literature survey on the topic of RT container-based virtualization and identified the main directions that the research community considers for enabling RT behavior of container-based virtualization.

As we have identified from the surveyed literature and from a survey of current container orchestrators, there is a lack of container orchestration tools and mechanisms that take into account containers' RT requirements during the orchestration processes. Thus, we have extended a container orchestration architecture, propose RT container performance metrics, and RT admission mechanism. Subsequently, we have provided an implementation in Kubernetes.

Finally, we have explored the communication aspect of large networks with a multitude of connected devices. We have introduced the DART framework that enables the collaboration of multiple SDN controllers among virtualized networks. Subsequently, we have implemented a use-case of a surveillance system that utilizes the framework. The results have shown that SDN controllers can cooperatively make decisions, prioritize, and distribute the bandwidth between virtual networks to mitigate congestion of shared resources. The framework introduced does not have to be restricted to bandwidth distribution only but it can be extended to support numerous applications that can benefit from the collaboration of virtual networks.

5.2 Limitation of the work

The RT enhancement of fog computing is a large research area concerning myriads of topics and sub-topics. In this thesis, we have investigated three major building blocks (i.e., virtualization, orchestration, and network management) of fog computing systems. However, we may not have captured all aspects of the building blocks.

For example, the RT container-based virtualization is not only a term containing one specific approach but a chain of interconnected underlying technologies and mechanisms ranging from scheduling theory, resource reservation, Linux-specific features, resource interference. Not all the aspects were fully mentioned in the thesis and the included papers. We have omitted some aspects of fog computing, e.g., the collaboration of fog nodes with a remote cloud. Fog computing serves in the thesis as a unifying topic that connects the building blocks. However, we have not completely connected all the mentioned topics from the technological point of view, for example, the SDN network management part stays apart from the rest of the thesis. However, it opens an interesting field for future work to connect these topics.

5.3 Future work

The future work follows the research presented in this thesis and aims to conduct research in the areas described below. We would like to utilize the developed infrastructure (e.g., the SDN and RT Kubernetes testbed) and knowledge gained while compiling this thesis. Additionally, we plan to explore the following:

- Identification, prediction, and prevention of interference of container-based virtualization. Paper C proposes a framework for orchestration and performance metrics of RT containers. However, we need to profoundly evaluate the proposed metrics and identify factors of interference between RT and BE containers.

- Advanced admission system for RT containers: In Paper C we propose an admission control system for RT containers. However, the admission control system is simple and needs to be extended with additional factors including OSLM and CLM performance metrics presented in Paper C.

- Online re-dimensioning of containers' resources: The container-based virtualization, as it is implemented now in the major OSs, does not take into account interference that originates due to concurrent use of shared resources, e.g.: memory, cache, and data buses. As a consequence, co-located containers may mutually influence the temporal isolation of each other and they may not deliver the requested performance. We plan to more sophisticated control system for resource (re-)allocation for container-based virtualization.

Bibliography

[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012.

[2] Jens Myrup Pedersen, M Tahir Riaz, Joaquim Celestino Junior, Bozydar Dubalski, Damian Ledzinski, and Ahmed Patel. Assessing measurements of qos for global cloud computing services. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2011.

[3] Frank Pallas, Philip Raschke, and David Bermbach. Fog computing as privacy enabler. *IEEE Internet Computing*, 2020.

[4] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner. Enabling fog computing for industrial automation through time-sensitive networking (tsn). *IEEE Communications Standards Magazine*, 2018.

[5] César Augusto García-Pérez and Pedro Merino. Experimental evaluation of fog computing techniques to reduce latency in lte networks. *Transactions on Emerging Telecommunications Technologies*, 2018.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd edition, 2011.

[7] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. *Int. Symp. on Object-oriented Real-time distr. Comp. (ISORC)*, 2005.

[8] Institute of Electrical and Electronics Engineers. Time-Sensitive Networking Task Group. `http://www.ieee802.org/1/pages/tsn.html`, 2016.

[9] Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *2013 8th IEEE international symposium on industrial embedded systems (SIES)*. IEEE, 2013.

[10] CS Wong, IKT Tan, RD Kumari, JW Lam, and W Fun. Fairness and interactive performance of o (1) and cfs linux kernel schedulers. In *2008 International Symposium on Information Technology*. IEEE, 2008.

[11] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, 2007.

[12] Mohammed Salman Shaik, Václav Struhár, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, et al. Enabling fog-based industrial robotics systems. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2020.

[13] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V Papadopoulos. Real-time containers: A survey. In *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[14] Václav Struhár, Mohammad Ashjaei, Moris Behnam, Silviu S Craciunas, and Alessandro V Papadopoulos. Dart: Dynamic bandwidth distribution framework for virtualized software defined networks. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2019.

[15] Mohamed Firdhous, Osman Ghazali, and Suhaidi Hassan. Fog computing: Will it be the future of cloud computing? The Third International Conference on Informatics & Applications (ICIA2014), 2014.

[16] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*. 2018.

[17] Mudassar Ali, Nida Riaz, Muhammad Ikram Ashraf, Saad Qaisar, and Muhammad Naeem. Joint cloudlet selection and latency minimization in fog networks. *IEEE Transactions on Industrial Informatics*, 2018.

[18] Torsten Kröger. *On-Line Trajectory Generation in Robotic Systems*. 2009.

[19] Shaik Mohammed Salman, Vaclav Struhar, Alessandro V Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: research challenges. In *1st Workshop on Fog Computing and the IoT (Fog-IoT 2019)*, 2019.

[20] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. 2011.

[21] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, 2015.

[22] Matthew Chapman, Daniel J Magenheimer, and Parthasarathy Ranganathan. Magixen: Combining binary translation and virtualization. *HP Enterprise Systems and Software Laboratory*, 2007.

[23] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 2016.

[24] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2013.

[25] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *IEEE Int. Conf. on Cloud Eng.*, 2015.

[26] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[27] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, 2015.

[28] Chen Shichao and Mengchu Zhou. Evolving container to unikernel for edge computing and applications in process industry. *Processes*, 2021.

[29] Juan Luo, Luxiu Yin, Jinyu Hu, Chun Wang, Xuan Liu, Xin Fan, and Haibo Luo. Container-based fog computing architecture and energy-balancing scheduling algorithm for energy iot. *Future Generation Computer Systems*, 2019.

[30] Olena Skarlat and Stefan Schulte. Fogframe: a framework for iot application execution in the fog. *PeerJ Computer Science*, 2021.

[31] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2013.

[32] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *IEEE Int. Conf. on Adv. Inform. Netw. and Appl. (AINA)*, 2012.

[33] Thuy Linh Nguyen and Adrien Lebre. Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time. Research Report RR-9221, INRIA, 2018.

[34] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[35] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[36] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. F. D. Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2015.

[37] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *ACM Int. Conf. Embedded Software*, 2011.

[38] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. μrtzvisor: A secure and safe real-time hypervisor. *Electronics*, 6(4), 2017.

[39] Neil Klingensmith and Suman Banerjee. Hermes: A real time hypervisor for mobile and iot systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018.

[40] Daniel Bristot De Oliveira and Romulo Silva De Oliveira. Timing analysis of the preempt rt linux kernel. *Software: Practice and Experience*, 2016.

[41] Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *SIGBED Rev.*, 2015.

[42] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 2019.

[43] Emiliano Casalicchio and Stefano Iannucci. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, 2020.

[44] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and management of fog computing applications. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018.

[45] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti. Foggy: A platform for workload orchestration in a fog computing environment. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.

[46] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, 2019.

[47] Sisu Xi, Chong Li, Chenyang Lu, Christopher D Gill, Meng Xu, Linh TX Phan, Insup Lee, and Oleg Sokolsky. Rt-open stack: Cpu resource management for real-time cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015.

[48] S. Aglianò et al. Resource management and control in virtualized SDN networks. In *RTEST*, 2018.

[49] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys & Tutorials*, 2015.

[50] B. A. A. Nunes et al. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Comm. Surv. Tut.*, 2014.

[51] L. Xingtao et al. Network virtualization by using software-defined networking controller based Docker. In *ITNEC*, 2016.

[52] A. Blenk et al. Pairing SDN with network virtualization: The network hypervisor placement problem. In *NFV-SDN*, 2015.

[53] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 2010.

[54] Rob Sherwood et al. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[55] Xiang (Alex) Feng. Towards real-time enabled microsoft windows. In *The 5th ACM International Conference on Embedded Software*, 2005.

[56] Saowanee Saewong et al. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, 2002.

[57] Michal Sojka et al. Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Sys. Arch.*, 2011.

[58] T. Cucinotta and L. Palopoli. QoS control for pipelines of tasks using multiple resources. *IEEE Trans. Computers*, 2010.

[59] A. B. Oliveira, A. Azim, S. Fischmeister, R. Marau, and L. Almeida. D-RES: Correct transitive distributed service sharing. In *IEEE Emerging Technology and Factory Automation*, 2014.

[60] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, 2002.

[61] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. OS-level virtualization for industrial automation systems: are we there yet? In *ACM Symp. on Applied Computing (SAC)*, 2016.

[62] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. of Syst. Architecture*, 2018.

[63] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[64] Philip Masek, Magnus Thulin, Hugo Andrade, Christian Berger, and Ola Benderius. Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016.

[65] C. Mao, M. Huang, S. Padhy, S. Wang, W. Chung, Y. Chung, and C. Hsu. Minimizing latency of real-time container cloud for software radio access

networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[66] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to IAAS. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[67] Kilian Telschig, Andreas Schonberger, and Alexander Knapp. A real-time container architecture for dependable distributed embedded applications. *IEEE Int. Conf. Automat. Science and Eng.*, 2018.

[68] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2008.

[69] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *IEEE Int. Conf. on Emerging Tech. and Factory Aut. (ETFA)*, 2017.

[70] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of Linux. In *IEEE Real-Time and Emb. Tech. and Appl. Symp. (RTAS)*, 2002.

[71] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *ACM/SPEC on Int. Conf. on Perf. Eng. (ICPE)*, 2017.

II

Included Papers

Chapter 6

Paper A: Fog-based Industrial Robotic System: Applications and Challenges

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 the 25th International Conference on Emerging Technologies and Factory Automation (ETFA 2020).

Abstract

Low latency and on demand resource availability enable fog computing to host industrial applications in a cloud like manner. One industrial domain which stands to benefit from the advantages of fog computing is robotics. However, the challenges in developing and implementing a fog-based robotic system are manifold. To illustrate this, in this paper we discuss a system involving robots and robot cells at a factory level, and then highlight the main building blocks necessary for achieving such functionality in a fog-based system. Further, we elaborate on the challenges in implementing such an architecture, with emphasis on resource virtualization, memory interference management, real-time communication and the system scalability, dependability and safety. We then discuss the challenges from a system perspective where all these aspects are interrelated.

Industrial robots are widely used in different automation applications such as painting and welding in automotive facilities and packaging in the food industry [1]. More recently, the domain of robotics has evolved to support warehouse automation with mobile robots and, at the same time, emphasis on collaborative robots has also gained significant attention [2]. However, existing solutions are limited in addressing the demands of such applications due to limited computational resources and the strong coupling of the software and the computing hardware [3]. The on-demand availability of resources and reduced communication latency as offered by the fog computing paradigm [4] makes an interesting case for investigating the use of fog computing for addressing existing limitations. For example, the localization and mapping tasks of mobile robots which may be computationally intensive, have been successfully implemented using fog computing resources, improving the computation time significantly [5]. Furthermore, an optimized offloading algorithm which targets fog computing resources has been designed for robotic mission planning to show the benefits of fog computing [6]. While these approaches show the benefit of using fog computing in industrial robots, practical implementation of fog-based robotics systems remains a challenge.

To facilitate further discussions on fog-based industrial robotics systems, and to identify the challenges that should be addressed to enable fog-based control of robots, in this paper, we provide an overview of the different technical aspects that are necessary for a practical realization of a fog network based on the OpenFog reference architecture [7] in Section 6.3. In Section 6.2, we describe a robotic cell-based factory automation environment and a robot control application. In Section 6.4 we use the factory automation environment to contextualize the technical aspects of the fog system and identify the potential challenges that should be addressed to enable a fog-based industrial robotics system. Here, we focus primarily on virtualization, resource orchestration, multi-core memory management and RT communication along with a discussion on challenges from the dependability, safety and scalability perspectives. Finally, Section 6.5 concludes the paper.

6.1 Related work

Industrie 4.0, intelligent manufacturing, factories of the future, smart factories, industrial internet of things(IIoT) etc. are used to describe the growing trend to digitize industrial domains that were once considered to be largely analogue. A considerable body of research has been devoted, in recent times, to advancing or bridging the digital divide in traditional manufacturing industries. Recent works such as [8] [9] discuss how control system architectures have changed in industrie 4.0 domains. One of the strongest drivers for flexible manufacturing systems is network connectivity to the Cloud. With the rapid advanced being made in the wireless 5G domain, manufacturing systems stand to benefit immensely from near zero latency connectivity which is the major bottleneck to implementation [10]. An alternate technological paradigm, Time Sensitive Networking or TSN is also being inducted into the Ethernet standards and is being touted as a RT variant of standard Ethernet [11]. However, a limitation of TSN thus far has been its lack of support for wireless networking which is seen as a requirement for mobility-based services. Research has been ongoing in this direction as well [12][13][14]. Of particular relevance is [15] where the authors discuss a novel method to integrate 5G with TSN which can be a game changer in the industry. Another work [16] gives a very detailed overview of how the two so-called competing technological paradigms can be integrated effectively. Challenges related to using these technologies in smart factories have been detailed in [17][18] where the authors discuss methods to use OPC-UA, a unified communications paradigm and Cloud computing for smart factories. With increasing data points, comes the need for intelligently handling such a deluge. Authors in [19] [20] elaborate on the role of data analytics and machine learning techniques in the smart factories. A rather in-depth study of various challenges involved in adopting TSN in industrial automation domain is provided in [21]. A performance study of industrial communication in the context of a real world use case is detailed in [22]. Readers interested in a survey of industrial communication trends in distributed systems are referred to [23] which provides a comprehensive background on the development of industrial communications in distributed environments such an automation facility or a smart factory.

6.2 Factory Automation Environment

A common factory automation setup is composed of a set of robotic cells [24]. A robotic cell consists of either a single robot or a set of robots grouped together with additional non-robotic machines to accomplish a task such as painting and welding. We categorize multi-robot cells as (i) coordinated cell, (ii) uncoordinated cell and (iii) mixed cell. In a coordinated cell, all the robots work in a synchronized manner on a single object. In an uncoordinated cell, the robot may work on different objects and need not be synchronized. For example, a pick and place cell with two robots operating on different objects on two different conveyors need not be in sync with each other. In many cases, a supervisory controller may control the workflow between the robots and other machines within the cell.

Each robot within a cell may be fitted with additional sensors and actuators, such as seam tracking lasers, force sensors, or vision systems, while actuators are typically end-effector tools, such as an arc torch [25]. The sensors and actuators are physically connected to an interface device, which processes the sensor data for transmission over a RT network. It is possible that some of the sensors and actuators can be directly connected to the robot controller through a wired or a wireless connection. In some cases, sensors and actuators are still physically attached to the interface device but the interface device itself can communicate wirelessly with the controller. Fig. 6.2 illustrates the different possible connections within a fog network.

At the robot level, the runtime robot behaviour is directed through a task specification interface, where a user typically specifies the way-points that the robot should pass through, the maximum speed the robot is allowed to take, along with other attributes such as, if the robot should pass through the way-points or just within a range of the way-points. The user is also able to define logical behaviour such as to wait until a specific signal is set or a timer has expired before moving to the next way-point. Finally, depending on the configuration and the user task specification, the robot software determines the trajectory of the robot using motion planning and trajectory generation algorithms [26, 27]. The information from the trajectory generation is fed to a low-level controller that runs periodically, usually, with a fixed cycle time having a typical value of

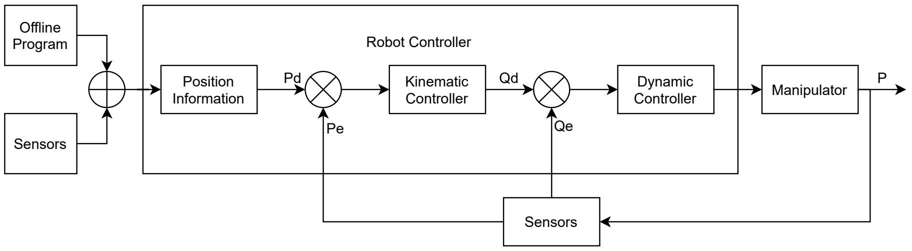


Figure 6.1. Robot controller block diagram.

1 millisecond to control the joints of the robots [27]. To achieve this, a robot controller software, composed of diverse components, is systematically put together to provide a coherent mechanism for manipulator control supported by a RT operating system. Fig. 6.1 illustrates the block diagram of a robot controller. The position information component of the robot controller reads the desired values either from sensors or a pre-defined program and forwards it to the kinematic controller. The kinematic controller generates the necessary reference values which are used by the dynamic controller to generate the required actuator commands for each joint. The sensors component provides the feedback information to the kinematic controller and the dynamic controllers.

6.3 System architecture

While several fog computing architectures have been proposed in the literature, [7, 28, 29], for our discussion, we adopt the IEEE OpenFog reference architecture and deployment model for fog computing [7]. Here, we assume that the participating devices are distributed between three layers (see Fig. 6.2) consisting of (i) cloud layer, (ii) fog layer, and (iii) device layer. The cloud layer provides a high computing capacity, but offers limited time predictability due to varying data transmission latencies. The fog layer provides an elastic environment in the vicinity of the origin of the data. It consists of a number of interconnected physical hardware devices (fog nodes) that are capable of hosting software applications on the shared node resources. While the processing power of the fog layer is less than that of the cloud layer, the network latency, however,

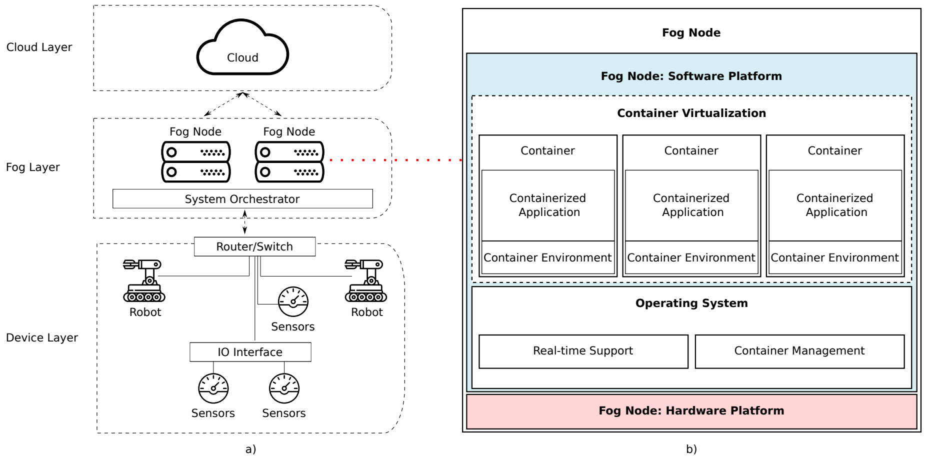


Figure 6.2. a) The three-layered fog network with the device layer consisting mainly of sensors and actuators and a fog layer with a RT orchestrator managing the distribution of workload between the fog nodes. b) An abstract view of a fog node. The fog hardware platform is supported by the software platform composed of container virtualization on top of an operating system.

is improved. The device layer consists of resource limited devices such as sensor and actuator devices that typically pre-process data from sensors and transmit it to the fog nodes, but also smart sensors and actuators, that are capable of communicating directly with the fog nodes.

Fog Software Components and Services:

We assume a fog computing architecture to be composed of a network of fog nodes, which can be viewed as a single logical entity [30]. The network is assumed to be hybrid of wired and wireless networks to exploit the benefits of both advanced wired and wireless technologies under the practical constraints of reliability, timeliness, and security for industrial environments. We briefly discuss some of the software components and services necessary for such a fog architecture below:

- System Orchestration: The system level orchestrator is responsible for ensuring that application requirements such as latency and memory are met by assignment of the applications to fog nodes. It takes into account the hardware capabilities, applications already running, task schedulability and application latency requirements. Additionally, it provides interfaces that enable seamless connection and disconnection of devices (e.g., additional fog nodes, robots, sensors and actuators) as well as interfaces for application providers for deploying applications in the fog system. Moreover, it continuously monitors the status of fog nodes in terms of availability, resource usage, and communication status and assesses the quality of service provided to the applications.

- Application Virtualization: The application virtualization component provides necessary functionality that allows to co-locate multiple independent applications on a single physical device in such a way that interference between the applications is minimized. It ensures proper allocation of resources and isolation between applications on the shared hardware building virtualized environments.

- Memory Management: Memory management component is responsible for ensuring spatial isolation among the tasks running on the same hardware. Applications have bounded memory space and cannot exceed these limits. Shared memory is allowed as long as it is explicitly declared by the applications and allowed by the operating system.

- Real-time Communication: The communication component is responsible for ensuring connectivity between the nodes and the sensor and actuator interfaces to ensure RT data freshness, correlation and separation constraints of the applications [31]. It is also responsible for non RT communication and supports both wired and wireless communication.

- Scalability: Services related to scalability are responsible for adding new physical and virtual compute nodes (as well as sensing and actuating devices) to the system in a dynamic manner. Due to such scalability services, the system level orchestrator is agnostic of discovery and integration mechanisms, but is able to consider all the available computational resources for the execution of the applications.

- Dependability and Safety: The dependability services provide a set of functionality for ensuring system dependability including the safety aspects by providing means for fault prevention, fault tolerance, fault removal, and fault forecasting [32] .

6.4 Use-case Aspects

Cyber physical systems such as industrial robots are dependent not only on the logical correctness of the computations but also the time at which the computations are completed. For example, the dynamic controller as shown in Fig. 6.1 is dependent on new reference values from the kinematic controller to be available during each new cycle of the control loop for better control. Using stale information may result in the robot deviating from the expected path. In order to cover such scenarios, as a fog-based industrial robotic system, we denote a fog computing system enhanced with RT capabilities, i.e., capabilities to guarantee that a computational task is finished and its result is transmitted within a predefined amount of time. Further, such a system should have the following properties: timeliness (the results of the computation must be available and transmitted within a predefined time), predictability (the system must be analyzable to guarantee performance of the applications), efficiency (the system should efficiently utilize available resources), scalability (the system should be able to grow in size dynamically when new cloud/fog nodes are discovered) and fault tolerance (the system should provide mechanisms to deal with unpredictable failures). We note that security is a critical property for enabling fog-based robotic systems, however, it is beyond the scope of this paper, and interested readers are referred to [33, 34, 35, 36, 37]. In the following sections, we elaborate in detail on the elementary set of aspects of fog computing in the context of factory automation environments.

6.4.1 Virtualization

Fog computing allows co-location of independent applications on a shared fog node in a fog network. To host such applications, for example, the different kinematic controllers of different robots, the fog nodes must provide virtual environments that ensure a proper isolation, resource allocation and the fulfilment of the demands of the applications. Virtualization abstracts physical hardware from the applications running on that hardware, and thus, emulates computing environments in such a way that it appears for the applications that they are executed exclusively on a dedicated hardware. Virtualization allows to host multiple isolated applications and their software dependencies on a single physical device, and thus, reducing resource wastage. However, sharing resources may lead to (time) unpredictability, and consequently, the timing constraints may be violated.

There are two main classes of virtualization technologies: Hypervisor-based virtualization and container-based virtualization [38]. Hypervisor-based virtualization utilizes a hypervisor that distributes resources among virtual machines that behaves like independent virtual computers containing dedicated operating systems and scheduling mechanisms. This solution requires support from hardware and, additionally introduces non-negligible overheads [39] and performance degradation. The container-based virtualization is provided purely by the host OS and it offers near-native performance, rapid startup times and low overhead, however the resource isolation may be weaker [40].

The solutions addressing RT in hypervisor-based virtualization, e.g., RT-Xen [41] or PikeOS[1]. However, time predictability in container-based virtualization is a novel topic. As summarized in [42], RT behavior of container-based virtualization must be supported by a predictable host operating system and RT scheduling policies that are container-aware. The former is addressed by the application of a RT patch that makes the Linux Kernel fully preemptive [43] or by the use of a RT co-kernel that runs side-by-side with a standard Kernel. Real-time aware scheduling policy for containers is addressed by utilization of hierarchical scheduling that provides temporal isolation of containers [44].

For a full adoption of RT virtualization, we see the following challenges: (i) Minimizing the interference between virtualized applications (e.g., co-located memory or cache-intensive applications may experience performance degradation of the physical fog nodes). (ii) Lack of RT communication mechanisms between virtualized applications (that may be executed on different devices) and enabling the communication in a time-predictable, secure, and safe manner. (iii)

[1]`https://www.sysgo.com/products/pikeos-hypervisor`

The possibility of supporting a mixture of both hypervisor-based virtualization (to satisfy hard RT requirements) and container-based virtualization (to satisfy less stringent requirements) in a single fog computing system should be explored. (iv) Dealing with unpredictability of communication between virtualized applications in a fog computing architecture due to network performance.

Multi-core Platforms

Additional set of problems hampering timing predictability of virtualized applications is caused by the use of a Multi-Core Processor (MCP). General Commercial Off-The-Shelf (COTS) MCPs share hardware resources like cache and main memory. Sharing such resources is one of the primary sources of Worst Case Execution Time (WCET) unpredictability [45] of a task. In an MCP, the task execution time not only depends on the task itself, but it is also significantly influenced by the applications running on the other cores (intercore interference). Nowotsch *et al.* [46] showed that the latency of a single memory store request can increase up to 25.82 times when the number of active cores increases from 1 to 8.

Virtualization aims for resource optimization and allows co-execution of independent applications such as the control tasks of different robots on the same hardware. Since sharing hardware resources leads to execution time unpredictability, bounding the WCET is necessary for the use of schedulability analysis and admission tests by the orchestrator. Such analysis allows the orchestrator to optimally allocate the resources for the applications while meeting the timing constraints imposed by the applications.

There are some proposed solutions based on a memory bandwidth management system to improve, and also to guarantee the tasks' WCET on an MCP in the context of RT systems, like in Yun *et al.* [47] and Agrawal *et al.* [48]. However, it is necessary to know the WCET of a task to schedule a set of tasks, and since the WCET depends on the inter-core interference, and which in turn is dependent on the schedule, we need solutions that address both memory bandwidth and scheduling problems together.

The solutions based on time and memory bandwidth management are well suited for time-triggered applications in a context where we know the number of active cores at the same time, and the maximal inter-core interference introduced by these cores. It becomes unrealistic for a industrial robotic system where the number of applications and their requirements change dynamically and during runtime. As an example, unlike in regular RT systems, where the execution behaviour can be modelled via different task models based on WCETs, modelling the execution time of motion planning tasks is complicated. The reason for this variability is twofold. One, the user of the robotic system is free to program the robot motion as desired. Two, the non-deterministic nature of the motion planning and trajectory generation algorithms. Some commonly used planning algorithms such as the Probabilistic Road Map (PRM) sample the joint space to find a collision free path [26]. In the best case, if a connection between initial point and the target point is established without collisions in the first iteration, no further computation is required. The execution time in this case can be minimal. While in the worst case, possibly in the presence of multiple obstacles, the number of samples that need to be checked for connectivity and collision can be huge, requiring larger computational time. This makes the motion and trajectory planning algorithms non-deterministic [49]. Given such a scenario, using traditional WCET-based analysis and design can result in significant wastage of resources. Therefore, to achieve RT guarantees in a system running on a MCP, we have (i) to ensure the temporal isolation of tasks and containers taking the platform resource contentions into account, and (ii) to design a resource sharing mechanisms for managing access to shared resources, such as bus and memory controller, taking dynamic applications scenarios into account.

The worst-case number of memory accesses that an application can issue depends on many different factors, for instance, hardware like in 32 or 64 bits platform, system configuration, and operating system. However, it is also unrealistic to assume that the maximum memory bandwidth necessary for each critical task is known. To address the lack of this data, a solution is to bind the containers that run critical tasks to a specific core and monitor their execution progress in predefined checkpoints along the time, to check if the execution is going according to the schedule. In case execution is late, the other non-critical containers can be paused to reduce the inter-core interference preventing the critical one from missing a deadline.

COTS MCP are designed primarily for the average-case performance and that is not enough to meet the RT requirements of robotic applications. Therefore, to address the loss of predictability in such platforms, we need to apply new mechanisms and know the application resource needs better.

6.4.2 Real-Time Aware Orchestration

The role of the orchestrator in the fog-based industrial computing systems is to deploy and maintain virtualized applications in the shared fog and cloud computing environment in such a way that the resource and timing requirements are fulfilled. Although there has been extensive research on orchestration, taking into account various resources and optimization goals, and there are several mature orchestrator systems available[2], none address timing-related requirements [50]. Therefore, we envision the following RT enhancements of the orchestrator that can serve in fog computing systems. The RT orchestrators should provide functionality for resource selection, RT deployment, RT aware service monitoring and RT resource control.

Resource Selection The orchestrator must be aware of timing requirements of applications as well as RT capabilities provided by fog nodes. The orchestrator must perform schedulability tests that ensure that the virtualized applications will be granted enough CPU time for performing time-critical actions. Additionally, the orchestrator should be aware of interference between virtualized applications and try to minimize such impacts during the resource selection phase.

Real-time Deployment The orchestrator should provide a bounded time for the deployment of virtualized applications. It should take into account transmission times of the applications from the repository to the fog node and the startup time of the application. This enhancement is important for safety and dependability aspects, e.g., during the re-deployment of a failed application.

Real-time Aware Service Monitoring Due to imperfections of the underlying operating systems (e.g., Linux) that may not provide accurate temporal isolation for virtualized environments, the orchestrator must monitor the quality

[2]e.g., Kubernetes (`https://kubernetes.io/`), Docker Swarm (`https://docs.docker.com/engine/swarm/`), or OpenStack(`https://www.openstack.org/`)

of service delivered by the virtualized applications (e.g., deadline misses or lateness[3]). The orchestrator should use this information during the resource selection phase.

Real-time Aware Resource Control Based on RT related metrics mentioned previously, the orchestrator should perform migration of virtualized applications in order to improve their RT behavior. This can be a case of a memory or cache-intensive application that may experience performance degradation when it is co-located with another memory or cache-intensive application on a single node.

6.4.3 Timely and Reliable Communication

The fog nodes should communicate with a number of sensors, actuators and other devices in different layers, using both wired and wireless connections to ensure the smooth operation of the robot tasks. In many instances, the communication should be in RT. For example, the sensors providing the feedback information as shown in Fig. 6.1, may be wireless, and for the feedback based dynamic control algorithms, the sensor data should be available before the beginning of the computation of next cycle of the control loop. However, when the probability of losing a packet goes up, especially in a wireless network, for example, due to the noise in industrial environments combined with the Doppler effect, multi-path fading, and dynamic wireless channel [51], a re-transmission technique may be necessary to improve communication reliability. This can lead to an increase of end-to-end latency resulting in unavailability of the sensor values in a timely manner. This in turn may affect the path accuracy of the robots.

Therefore, to enable RT robot control, the wireless communication protocols should be designed to fulfil both strict timeliness and reliability requirements. For example, the wireless protocols should be able to guarantee an upper bound end-to-end latency of 1 millisecond along with a probability that a packet does not reach its destination before the deadline to not exceed 10^{-7} [52].

For wired networks, more recently, Time Sensitive Networking (TSN) has emerged as a front-runner and a competing technology to the well established

[3]The delay of a task completion with respect to its deadline.

field-bus standards that have been the staple of industrial and automotive networking [53]. TSN is a set of standards that provide RT guarantees over standard Ethernet. Hence, current IEEE 802.1Q standards [54] come integrated with TSN so that vendors can directly provide properties such as timeliness, fault-tolerance, reliability and availability to their networking products.

Since wired networks can offer deterministic communications, wireless ones should also guarantee deterministic reliable communications at the same level. In this context, Medium Access Control (MAC) protocols and relaying strategies are central in achieving the desired requirements. Rajandekar *et al.* [55] concluded that the hybrid MAC protocols can meet the stringent requirements of reliability and timeliness. Furthermore, since the fog nodes need to support for a large number of IoT devices in a massive IoT scenario such as in a factory-to-factory communication (Section 6.4.4), the proposed MAC protocols must support a large number of simultaneous connections. Li *et al.* [56] proposed a hybrid Time Division Multiple Access (TDMA) Non Orthogonal Multiple Access (NOMA) scheme for cellular-enabled machine-to-machine communications. With NOMA, multiple nodes can be served simultaneously utilizing the same time-frequency resources but different power levels. The proposed time-sharing scheme is introduced to deal with the massive deployment of devices, while improving the total transmission time and energy efficiencies. This solution is suitable for fog networks where a NOMA transceiver may be deployed at the fog nodes [57]. Another approach, Hoang *et al.* [58] proposed a relaying sequence that considers all cases that can happen in each time slot. Hence, the probability of an error is an exact value compared to an upper bound value as is the case in other solutions related to multi-hop communication. Moreover, the authors introduced a method of group based relaying on a hybrid TDMA-CSMA (Carrier-Sense Multiple Access) protocol to address the drawback in relaying sequencing where a relayer keeps silent if it does not have any correct copy of the packet [59]. Therefore, the relaying strategies combined with packet aggregation are practical techniques that can be implemented on the fog nodes. However, further research on these techniques based on a hybrid protocol with NOMA is needed to minimize the end-to-end latency, especially in the case of the large number of connections.

To obtain timely and reliable communication necessary for the robot control, we observe several challenges including: (i) the placement of fog nodes' transceiver must minimize the probability of error and end-to-end latency, (ii) hybrid protocols with NOMA should be further studied and improved to deal with the massive connection, and (iii) in industrial environments, such as those of the factory automation environments, there are strict constraints, i.e., a limited number of relay nodes and re-transmissions are possible, and therefore, specific relaying strategies should be defined.

6.4.4 Scalability

Another aspect of the proposed use case is scalability [60]. In an industrial setting (as discussed in Section 6.2), the number of participating compute nodes can become very large depending on the size of the factory, because compute nodes may include cloud and fog nodes as well as the interface devices. Furthermore, apart from interconnecting the compute nodes of one factory, there is also the possibility of interconnecting various factories with each other. This can be beneficial when, for instance, the production of an item in a factory is reduced or halted due to faults in the hierarchical or vertical communication of the nodes (see Section 6.4.5). In such cases, another factory which produces the same product (and is able to increase production) can change its production plans in order to compensate for the faults. This can be achieved by coordinating the function of multiple factories through fog and cloud nodes.

Along with the potential benefits of interconnected factories, and the advantages that fog computing will bring to industrial environments, there are also related concerns. For example, the communication overhead from the interactions among the distributed cloud and fog nodes, needs to remain limited so that it does not interfere with the operation of the applications [60], and it does not compromise the functionality of nodes which execute tasks with strict deadlines (see Section 6.4.1). Most applicable fog computing approaches aim at enabling computing close to the edge of the network [61]. However, they do not consider scalability metrics (e.g., communication overhead) which show that scaling a fog computing system to a large degree (i.e., adding many compute nodes), does not compromise the performance of the applications [62].

In an industrial fog computing setting (as discussed in Section 6.2), when adding new compute nodes (e.g., new controllers, fog, and cloud nodes) these

Table 6.1. Overview of the aspects and their inter-relation and challenges.

Aspect/Relation	**Virtualization & Orchestration**	**Multi-Core Platform**	**Communication**	**Safety & Dependability**	**Scalability**
Virtualization & Orchestration	–	Platform abstraction and efficient resource management.	Resource allocation for achieving tight end-to-end communication.	Provide predictability and fault tolerance.	Complexity.
Multi-Core Platform	Provide RT guarantees and predictability.	–	Virtualization & Orchestration isolates the platform.	Intra-core isolation and predictability.	Challenges wrt. hierarchical architectures & legacy.
Communication	Heterogeneity, combination of wired and wireless networks.	Virtualization & Orchestration isolates the platform.	–	Trade-off between reliability and safety.	Large numbers along with big data communication.
Safety & Dependability	Heterogeneous safety levels.	Certification.	Tight end-to-end communication and fault tolerance.	–	Redundancy overhead.
Scalability	Runtime reconfiguration.	Hardware architecture.	Congestion.	Increased complexity.	–

nodes need to be discovered and integrated in the system [63]. This means that the existing nodes need to store the new nodes' information (e.g., IP addresses, amount of computational resources, etc.) so that they can communicate (e.g., using widely-used communication protocols [64]). This creates the problem of determining which nodes a new node should connect to [65]. A simple solution to this problem would be that each node maintains a global view of the system, i.e., stores the information of all the other nodes. However, this might be impractical since the compute nodes at the edge (e.g., the fog nodes) may be able to store only a small number of other compute nodes due to having limited computational resources [66]. For this reason, scalability still poses a challenge in the proposed use case.

6.4.5 Dependability and Safety

Dependability

While fog-based systems address some of the limitations of existing architectures [3], they introduce new challenges for ensuring dependability attributes, for instance, reliability and availability. For example, the fog-based software architecture demands frequent data exchange between the different nodes of the fog layer to accomplish a functionality such as trajectory generation and control if we assume that the trajectory generators are placed on a different fog nodes. This puts more focus on ensuring reliability and availability of the fog platforms.

The attributes of the fog services, such as compute and communication, are challenged by dependability threats viz., errors, faults and failures [32], which in turn might disrupt the entire functionality of the system. Threats related to computational resources include the occurrence of faults in fog nodes. Dependability threats associated with the orchestrator may lead to either a performance failure or wastage of resources. Another example is A failure in data storage that might result in data loss. This can result in loss of user-specified tasks or the configuration settings mapping different sensors to the user task specifications and data variables. Additionally, data stored for future analysis in the cloud may not be accessible. Threats related to communication are faults that might disrupt or prevent the connectivity between the different nodes as well as the different layers in the architecture. Loss of communication between the nodes and the edge devices can result in stoppage of robots as the new trajectory parameters are not available for the low level controllers. Such failures can have a cascading effect within a robotic cell. For example, if a robot stops in the worskpace of another robot, the safety system may force the independent robot to take mitigating actions such as slowing down or stopping altogether. This situation may be undesirable It is therefore critical to preserve a tight end-to-end communication, while providing suitable fault tolerance mechanisms.

Dependability approaches for fog computing in the literature are mainly proposed to address dependability requirements using fault management solutions and redundancy techniques [67]. Fault management solutions proposed are mainly considering threat detection tools like monitoring [68] for fault prevention and fault removal, or failure recovery solutions like reconfiguration upon failure [69]. Redundancy techniques proposed in the literature focus on addressing different dependability requirements of the fog platform. For instance, improving reliability [70], and availability [71], of the system by introducing redundancy in the form of a redundant node [70], in a network, redundant communication channels [72] and task offloading [73] or application migration [74].

The proposed dependability solutions are mainly tied to redundancy methods and replication techniques, for instance, using passive/ active replicas of the system components. However, using replicas for each component will result in overhead of cost and consumption of resources [75], and it can also impact the scalability of the overall system. Therefore, we need further research to improve the system dependability.

Safety

Safety in robotics involves multiple domains such as the design of the manipulator arm and the layout of the cell. We focus on safety from the software perspective. We define safety in the present use-case as the property of the system which guarantees the timely and correct execution of safety-critical tasks (with hard RT deadlines) under *all* operational conditions [76] [77]. This encompasses scenarios wherein the system reverts to a safe state in the event of a safety violation or hazard. For instance, applications using motion control algorithms for robot arm movements need safety guarantees in terms of the range of motion that is allowed to prevent hazardous motions. Typically a safety function (or task) is responsible for ensuring the said guarantees. Catastrophic consequences (such as loss of life, danger to the surrounding environment) can ensue in case of a failed execution of safety-critical tasks occur [78].

To ensure that such failure scenarios are well handled, it is necessary that timing analysis, schedulability tests and the network schedule are designed considering the potential threats to safety. The fact that a single failure scenario can pose a threat to safety is the principal challenge for the verification and validation of multi-core platforms, and more in general for fog architectures.

Therefore to ensure safety of users and of the equipment, we need to ensure such threats are handled at the design phase itself, even in presence of heterogeneous safety levels. Additionally, as in all safety systems, there needs to be no single point of failure. Along with these challenges, it is necessary to investigate if RT applications running in the fog provide the same level of safety guarantees that the existing safety-certified robotics systems provide.

6.5 Conclusion

Fog-computing brings cloud-like capabilities to low latency applications but the practical implementation of a fog architecture for RT applications such as robot control is non-trivial. To move a step forward in this direction, we need a holistic approach that considers different technical aspects independently but also in conjunction with each other. We summarize the relationship between the different aspects in Table 6.1.

To make effective use of the fog resources, we need virtualization techniques such as hypervisors and containers with RT capabilities to support the timing requirements of robotic cells and robot motion. To provide temporal isolation, hypervisors and containers need memory management techniques to limit the interference of shared caches and buses within the multi-core architectures. By providing bounds on the interference, we can enable the orchestrator with the capabilities to use RT schedulability analysis and appropriate scheduling algorithms to allocate applications to fog nodes, to ensure timing predictability. Since the fog platform is a distributed system involving fog nodes and edge devices such as low-level controller and sensors, we need RT communication mechanisms. Such communication can be wired or wireless. The constraints imposed by communication technologies further guide the scheduling and allocation of resources by the orchestrator. For robotic applications, dependability and safety are important attributes to prevent any damage to the equipment and more importantly, to safeguard the health of the operators working in close proximity to such robots. To this end, we need solutions that consider the requirements of the applications as well as the new challenges imposed by the fog platforms. In this paper, we briefly discussed a robotic cell environment to highlight the usefulness of fog-based solutions and discussed key aspects such as resource orchestration and network scalability, virtualization and memory management techniques supported by RT communication paradigms. Further we discussed the dependability and safety issues that need to be considered when moving towards the fog-based architectures for robotic applications.

Bibliography

[1] Martin Hägele, Klas Nilsson, J. Norberto Pires, and Rainer Bischoff. Industrial robotics. In *Springer Handbook of Robotics*. 2016.

[2] Valeria Villani, Fabio Pini, Francesco Leali, and Cristian Secchi. Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 2018.

[3] Shaik Mohammed Salman, Vaclav Struhar, Alessandro V. Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: Research challenges. In *Proceedings of the Workshop on Fog Computing and the IoT*, 2019.

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012.

[5] Swarnava Dey and Arijit Mukherjee. Robotic slam: a review from fog computing and mobile edge computing perspective. In *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*, 2016.

[6] Ajay Kattepur, Hemant Kumar Rath, and Anantha Simha. A-priori estimation of computation times in fog networked robotics. In *2017 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2017.

[7] IEEE. Ieee adoption of openfog reference architecture for fog computing, 2018.

[8] S. S. P. Olaya, M. Wollschlaeger, and S. S. Perez Olaya. Control as an industrie 4.0 component: Network-adaptive applications for control. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.

[9] T. Kuhn, P. O. Antonino de Assis, M. Damm, A. Morgenstern, D. Schulz, C. Ziesche, and T. Müller. Poster: Industrie 4.0 virtual automation bus. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018.

[10] M. Gundall, J. Schneider, H. D. Schotten, M. Aleksy, D. Schulz, N. Franchi, N. Schwarzenberg, C. Markwart, R. Halfmann, P. Rost, D. Wübben, A. Neumann, M. Düngen, T. Neugebauer, R. Blunk, M. Kus, and J. Grießbach. 5g as enabler for industrie 4.0 use cases: Challenges and concepts. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

[11] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner. Enabling fog computing for industrial automation through time-sensitive networking (tsn). *IEEE Communications Standards Magazine*, 2018.

[12] C. Cruces, R. Torrego, A. Arriola, and I. Val. Deterministic hybrid architecture with time sensitive network and wireless capabilities. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

[13] H. Wang, L. Zeng, M. Li, and C. Yang. A protocol conversion scheme between wia-pa networks and time-sensitive networks. In *2019 Chinese Automation Congress (CAC)*, 2019.

[14] O. Seijo, Z. Fernández, I. Val, and J. A. López-Fernández. Sharp: Towards the integration of time-sensitive communications in legacy lan/wlan. In *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018.

[15] D. Ginthör, J. von Hoyningen-Huene, R. Guillaume, and H. Schotten. Analysis of multi-user scheduling in a tsn-enabled 5g system for industrial

applications. In *2019 IEEE International Conference on Industrial Internet (ICII)*, 2019.

[16] A. Neumann, L. Wisniewski, R. S. Ganesan, P. Rost, and J. Jasperneite. Towards integration of industrial ethernet with 5g mobile networks. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, 2018.

[17] T. Kobzan, S. Heymann, S. Schriegel, and J. Jasperneite. Utilizing sdn infrastructure to provide smart services from the factory to the cloud. In *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*, 2019.

[18] C. Nagpal, P. K. Upadhyay, S. Shahzeb Hussain, A. C. Bimal, and S. Jain. Iiot based smart factory 4.0 over the cloud. In *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*, 2019.

[19] D. J. Ahn, J. Jeong, and S. Lee. A novel cloud-based fog computing network architecture for smart factory big data applications. In *2018 South-Eastern European Design Automation, Computer Engineering, Computer Networks and Society Media Conference (SEEDA_CECNSM)*, 2018.

[20] N. Desai and S. Punnekkat. Enhancing fault detection in time sensitive networks using machine learning. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, 2020.

[21] L. Lo Bello and W. Steiner. A perspective on ieee time-sensitive networking for industrial communication and automation systems. *Proceedings of the IEEE*, 2019.

[22] L. Seno, F. Tramarin, and S. Vitturi. Performance of industrial communication systems: Real application contexts. *IEEE Industrial Electronics Magazine*, 2012.

[23] P. Gaj, J. Jasperneite, and M. Felser. Computer communication within industrial distributed environment—a survey. *IEEE Transactions on Industrial Informatics*, 2013.

[24] Jiafan Zhang and Xinyu Fang. Challenges and key technologies in robotic cell layout design and optimization. *J. Mech. Eng. Science*, 2017.

[25] P. Kah, M. Shrestha, E. Hiltunen, and J. Martikainen. Robotic arc welding sensors and programming in industrial applications. *International Journal of Mechanical and Materials Engineering*, 2015.

[26] Steven M. LaValle. *Planning Algorithms*. 2006.

[27] Torsten Kröger. *On-Line Trajectory Generation in Robotic Systems*. 2010.

[28] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*. 2014.

[29] A.V. Dastjerdi, H. Gupta, R.N. Calheiros, S.K. Ghosh, and R. Buyya. Chapter 4 - fog computing: principles, architectures, and applications. In Rajkumar Buyya and Amir [Vahid Dastjerdi], editors, *Internet of Things*. 2016.

[30] Eva Marín Tordera, Xavi Masip-Bruin, Jordi Garcia-Alminana, Admela Jukan, Guang-Jie Ren, Jiafeng Zhu, and Josep Farré. What is a fog node a tutorial on current concepts towards a common definition. 2016.

[31] R. Gerber, Seongsoo Hong, and M. Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. on Softw. Eng.*, 1995.

[32] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure comput. *IEEE Trans. on Dep. and Secure Comp.*, 2004.

[33] I. Stojmenovic, S. Wen, X. Huang, and H. Luan. An overview of fog computing and its security issues. *Concurrency Computation*, 2016.

[34] S. Yi, Z. Qin, and Q. Li. Security and privacy issues of fog computing: A survey. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[35] M. Mukherjee, R. Matam, L. Shu, L. Maglaras, M.A. Ferrag, N. Choudhury, and V. Kumar. Security and privacy in fog computing: Challenges. *IEEE Access*, 2017.

[36] P. Zhang, M. Zhou, and G. Fortino. Security and trust issues in fog computing: A survey. *Future Generation Computer Systems*, 2018.

[37] A. Khalid, P. Kirisci, Z.H. Khan, Z. Ghrairi, K.-D. Thoben, and J. Pannek. Security framework for industrial collaborative robotic cyber-physical systems. *Computers in Industry*, 2018.

[38] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *IEEE Int. Conf. on Cloud Eng.*, 2015.

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

[40] Miguel G Xavier, Israel C De Oliveira, Fabio D Rossi, Robson D Dos Passos, Kassiano J Matteussi, and César AF De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

[41] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *ACM Int. Conf. Embedded Software*, 2011.

[42] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V Papadopoulos. Real-time containers: A survey. In *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[43] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. OS-level virtualization for industrial automation systems: are we there yet? In *ACM Symp. on Applied Computing (SAC)*, 2016.

[44] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 2019.

[45] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2010.

[46] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, 2014.

[47] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Trans. on Comp.*, 2016.

[48] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *Euromicro Conf. on Real-Time Syst.*, 2017.

[49] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *IEEE Real-Time and Embedded Technol. and Appl. Symp.*, 2020.

[50] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 2019.

[51] A. Willig, K. Matheus, and A. Wolisz. Wireless technology in industrial networks. *Proceedings of the IEEE*, 2005.

[52] R. Candell and M. Kashef. Industrial wireless: Problem space, success considerations, technologies, and future direction. In *Resilience Week*, 2017.

[53] N. Finn. Introduction to Time-Sensitive Networking. *IEEE Commun. Standards Mag.*, 2018.

[54] IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks - Redline. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014) - Redline*, 2018.

[55] A. Rajandekar and B. Sikdar. A survey of MAC layer issues and protocols for machine-to-machine communications. *IEEE Internet of Things J.*, 2015.

[56] Z. Li and J. Gui. Energy-efficient resource allocation with hybrid tdma–noma for cellular-enabled machine-to-machine communications. *IEEE Access*, 2019.

[57] H. Tezuka, M. Moriyama, K. Takizawa, and F. Kojima. A UL-NOMA system providing low E2E latency. In *IEEE VTS Asia Pacific Wireless Commun. Symp.*, 2019.

[58] Le-Nam Hoang. *Relaying for Timely and Reliable Applications in Wireless Networks*. PhD thesis, Halmstad University, 2017.

[59] L. Hoang, E. Uhlemann, and M. Jonsson. Relay grouping to guarantee timeliness and reliability in wireless networks. *IEEE Commun. Lett.*, 2019.

[60] V. Karagiannis, S. Schulte, J. Leitão, and N. Preguiça. Enabling fog computing using self-organizing compute nodes. In *IEEE Int. Conf. Fog and Edge Comput.*, 2019.

[61] Vasileios Karagiannis and Apostolos Papageorgiou. Network-integrated edge computing orchestrator for application placement. In *Int. Conf. Netw. and Service Manage.*, 2017.

[62] V. Karagiannis and S. Schulte. Comparison of alternative architectures in fog computing. In *IEEE Int. Conf. Fog and Edge Comput.*, 2020.

[63] Vasileios Karagiannis, Nitin Desai, Stefan Schulte, and Sasikumar Punnekkat. Addressing the node discovery problem in fog computing. In *Workshop Fog Comput. and IoT*, 2020.

[64] Vasileios Karagiannis, Periklis Chatzimisios, Francisco Vazquez-Gallego, and Jesus Alonso-Zarate. A survey on application layer protocols for the internet of things. *Trans. on IoT and Cloud comput.*, 2015.

[65] Vasileios Karagiannis. Compute node communication in the fog: Survey and research challenges. In *Workshop on Fog Comput. and IoT*, 2019.

[66] Rajiv Ranjan, Omer Rana, Surya Nepal, Mazin Yousif, Philip James, Zhenya Wen, Stuart Barr, Paul Watson, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, et al. The next grand challenges: Integrating the internet of things and data science. *IEEE Cloud Computing*, 2018.

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

[68] Xiao Yuan, Chimay J. Anumba, and M. Kevin Parfitt. Cyber-physical systems for temporary structure monitoring. *Automation in Construction*, 2016.

[69] 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 *IEEE Int. Symp. on Personal, Indoor, and Mobile Radio Communications*, 2017.

[70] A. Aral and I. Brandic. Quality of service channelling for latency sensitive edge applications. In *IEEE Int. Conf. Edge Computing*, 2017.

[71] X. Chen, X. Wen, L. Wang, and W. Jing. A fault-tolerant data acquisition scheme with mds and dynamic clustering in energy internet. In *IEEE Int. Conf. Energy Internet*, 2018.

[72] K. E. Benson, G. Wang, N. Venkatasubramanian, and Y. Kim. Ride: A resilient IoT data exchange middleware leveraging SDN and edge cloud resources. In *IEEE/ACM Third Int. Conf. Internet-of-Things Design and Implementation*, 2018.

[73] M. T. Saqib and M. A. Hamid. FogR: A highly reliable and intelligent computation offloading on the internet of things. In *IEEE Region 10 Conf. (TENCON)*, 2016.

[74] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K. R. Choo, and M. Dlodlo. From cloud to fog computing: A review and a conceptual live vm migration framework. *IEEE Access*, 2017.

[75] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson. Dependable fog computing: A systematic literature review. In *Euromicro Conf. Software Eng. and Adv. Appl.*, 2019.

[76] Radu Dobrin, Nitin Desai, and Sasikumar Punnekkat. On fault-tolerant scheduling of time sensitive networks. In *Int. Workshop on Security and Dependability of Critical Embedded Real-Time Syst.*, 2019.

[77] Nitin Desai and Sasikumar Punnekkat. Safety of fog-based industrial automation systems. In *Proceedings of the Workshop on Fog Computing and the IoT*, 2019.

[78] N. Desai and S. Punnekkat. Safety-oriented flexible design of autonomous mobile robot systems. In *2019 International Symposium on Systems Engineering (ISSE)*, 2019.

Chapter 7

Paper B: Real-Time Containers: A Survey

Václav Struhár, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos.

In the 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020).

Abstract

Container-based virtualization has gained a significant importance in a deployment of software applications in cloud-based environments. The technology fully relies on operating system features and does not require a virtualization layer (hypervisor) that introduces a performance degradation.

Container-based virtualization allows to co-locate multiple isolated containers on a single computation node as well as to decompose an application into multiple containers distributed among several hosts (e.g., in fog computing layer). Such a technology seems very promising in other domains as well, e.g., in industrial automation, automotive, and aviation industry where mixed criticality containerized applications from various vendors can be co-located on shared resources.

However, such industrial domains often require RT behavior (i.e, a capability to meet predefined deadlines). These capabilities are not fully supported by the container-based virtualization yet. In this work, we provide a systematic literature survey study that summarizes the effort of the research community on bringing RT properties in container-based virtualization. We categorize existing work into main research areas and identify possible immature points of the technology.

Fog Computing as well as cloud computing relies extensively on resource virtualization. In this area, the container-based virtualization is gaining its importance as a lightweight alternative of hypervisor-based virtualization. The container technology allows to execute applications and their software dependencies in a virtual environment independently on the software ecosystem of their hosts. A host can accommodate multiple containers at a time, providing means for container isolation and resource control for the containers. Container-based virtualization (sometimes referred as an OS level virtualization) does not require a hypervisor and therefore it provides near-native performance [1, 2], rapid deployment times and a low overhead while still retaining a certain level of resource isolation and resource control. The containers are a de-facto standard for development of large scale web applications adopted by a number of companies [3].

The benefits of the container-based virtualization are aligned with the strive of the companies in other areas such as in industrial and robot control, automotive and aviation. In these industrial domains, there are strong requirements to (i) consolidate computational resources (Electronic Control Units, physical controllers) and (ii) provide a flexible environment for running (RT) applications. Additionally, container-based virtualization can enable interruption-free hardware and software maintenance, dynamic system redundancy change and system redundancy healing [4]. However, in such fields stringent RT requirements are often needed. This means that the applications inside of a container should meet predefined deadlines independently on other co-located containers.

In this survey, we summarize the research carried out in the area of RT containers since the introduction of containers in Linux (i.e. 2008[1]).

The main contributions of this paper include:

- Systematic literature survey of the RT containers.
- Overview of the approaches and technology enabling RT behavior of containers.
- Identification of pitfalls, challenges and future research directions for RT containers.

[1]`https://lwn.net/Articles/256389/`

7.1 The Review Process

The systematic literature survey is carried out with the guidance in [5]. The research questions are defined together with search queries and sources of the studies and, subsequently, we extract the data and answer to the questions. We apply the snowballing [6] method to identify relevant papers outside the search query. Databases used: *Scopus* and *IEEE*. Only full peer review papers published between 2008-2019 are considered. We search the databases using the following search queries:

(***Real-time*** OR ***RT***) AND (***Containers*** OR ***Container***)

The search string extracts 1855 articles in *Scopus* and 609 articles in *IEEE*. Out of that, we identify 38 and 23 potentially relevant articles by the title. As the number of articles is low, we fully screen each potentially relevant paper (abstract and full text) to make the decision for inclusion/exclusion into this survey. In total, we include 14 papers as seen in Table 7.1.

7.1.1 Question Formalization

In this work, we elaborate the following questions:

- **RQ1**: *Why and in which context have RT containers been used?* Answering this question will give an overview of the motivation behind the use of RT containers, expected benefits and areas where RT containers are used.
- **RQ2**: *What approaches are used for enabling RT behavior of containers?* The answer will give an overview of the approaches and technologies, their combinations and their usages for the RT container-based virtualization.
- **RQ3**: *What are the pitfalls and weak points of using RT containers that prevent full adoption of such technology in industry?* The answer for this question will give a list of research challenges and problems for RT container computing.

7.2 Container-based Virtualization

From the runtime perspective, a container is a set of resource-limited processes that are isolated from the rest of the system and from other containers. This is achieved by utilizing two Linux kernel features: (i) Namespaces and (ii) Control groups (cgroups). Namespaces virtualize global resources (e.g., processes, network, inter-process communication) in the way that a group of processes can see and use one set of resources while another group can use different set of resources. Cgroups provide a mechanism for aggregating and partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour[2]. It allows to organize processes hierarchically and distribute system resources along the hierarchy.

7.2.1 Container Platforms

There are several container solutions, all of them rely on cgroups and namespaces. Thus, all the platforms pose similar options and performance [7]. The philosophy of using the two most commonly used container platforms Linux Containers (LXC) and Docker differs. Docker containers are microservice-based (each container should contain a single application), whereas LXC, similarly to Virtual Machines, allows to run a complex ecosystem of applications which is beneficial for emulation of legacy systems.

7.2.2 Real-Time Containers

The term RT implies that the correctness of the system depends not only on the results of the computation but also on the time at which the results are produced [8]. Real-time systems can be categorized into three groups: hard, firm and soft RT. Missing a deadline in a hard RT system may cause catastrophic consequences, whereas missing deadline in a firm RT system leads to the complete loss of the utility of the result. Missing deadline in a soft RT system just degrades the utility of the result.

[2]`cgroupsdocumentation`

A RT container is a container that provides resource isolation, resource control and additionally provides time deterministic and predictable behavior for the containerized application.

7.2.3 Real-time Support of Linux

To ensure the time predictable behavior of the containers, the operating systems must provide such capability. Default (Vanilla) Linux does not give any time guarantees on execution of tasks and therefore the predictability is low [9]. However, there are several approaches to improve the predictability: the RT patch that improves preemptability of the Linux kernel and co-kernel approaches that run a RT micro kernel in parallel to the Linux kernel. Containerized applications are scheduled in the same way as native applications using the host's scheduler, the Default Linux kernel provides three schedulers: *(i) Completely Fair Scheduler*: Aims to maximize CPU utilization while also maximizing interactive performance. It does not give any time guarantees. *(ii) Real-Time scheduler*: The scheduler allows to schedule tasks in the fixed priority manner using First In First Out or Round Robin policies. The tasks run till they yield or are preempted by higher priority tasks. The Real-Time group scheduling[3] extension allows to divide and allocate CPU time between RT and non RT tasks. *(iii) Earliest Deadline First Scheduler* Uses Constant Bandwidth Server [10] and allows to associate to each task a budget and a period.

7.3 Survey Results

In this section, we summarise relevant papers. There are four main directions for enabling RT behavior of containers: (i) RT patch based, (ii) co-kernel based, (iii) hierarchical scheduling based and (iv) custom approach. A short summary is provided in Table: 7.1.

[3]https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

Study	Main focus	Approach & Technology	Communication aspects
Cinque et al. [11, 12]	· Architecture definition · Faulty tasks monitoring · Implementation details	· Real Time Application Interface (RTAI) · Docker · Fixed priority scheduling	-
Cucinotta et al. [13, 14, 15]	· Temporal Interference between containers	· Hierarchical Scheduling	-
Tasci et al. [16]	· Architecture definition · Real-time communication between containers	· Combination of Real-Time patch and Xenomai · Docker	Design of messaging system based on ZeroMQ.
Moga et al. [17]	· Feasability study · Communication between containers · Communication overheads	· Docker · Real-Time patch	Network performance and overhead measurements between containers using default Docker Linux NAT Bridge.
Hofer et al. [18]	· Experimental comparison between Real-Time patch, Xenomai, Vanilla Linux	· Real-Time patch · Xenomai · Vanilla Linux	-
Goldschmidt et al. [19, 4]	· Architecture definition · Feasibility study	· Real-Time patch · Legacy systems emulation in RT containers	-
Telschig et al. [20]	· Model-based architecture and analysis · Dependable RT container computing.	· LXC	-
Mao et al. [21]	· Minimizing latencies in software-based Radio Access Networks	· Docker · Real-Time patch	Application of fast packet processing using Intel Data Plane Development Kit.
Masek et al. [22]	· Systematic evaluation of sandboxed software	· Real-Time patch	-
Wu et al. [23]	· Dynamic CPU allocation for mixed-criticality RT systems	· Custom scheduling mechanism · Docker	-

Table 7.1. Summary of studies elaborating on RT containers.

7.3.1 Methods Based on PREEMPT_RT Patch

The RT patch (PREEMPT_RT) improves the kernel's locking primitives to maximize preemptible sections. The advantage of the patch is that there is no need for special libraries or API needed by the application developers.

Moga et al. [17] considers RT containers in the context of industrial automation systems that works with RT data and have RT deadlines on detection and response to events. The paper emphasises the need for OS-level virtualization in an industrial automation and gives examples of timing requirements of industrial applications (e.g. motor drive typically requires cycle time between 1ms to 250μs) and a need for synchronization between the containers. The evaluation of the effects of containers on performance of industrial automation systems is provided in two cases: (i) Cyclic behavior of a containerized application, (ii) Virtual networking performance for communications between containers. Cyclic behavior test evaluates the ability to execute application logic at pre-defined intervals, measures accuracy and jitter. Virtual networking test evaluates the ability to communicate between co-located containers in a time-bounded manner. The researches see the RT container computing as a promising technology, however communication mechanisms between containers are not clear.

The work in [4] (and previously [19]) addresses a container based architecture for RT controllers that allow a flexible function deployment and a support of legacy control applications. Such architecture is needed to preserve a functionality of legacy control programs and to reduce maintenance cost of legacy systems (in which the software is often bounded to a specific hardware and software ecosystem). The researchers investigate the feasibility of building a RT capable system (for legacy systems) based on RT containers, they target Programmable Logic Controller (PLC) and automation controllers with the cycle time between 100ms to 1s. They perform a set of tests under various load scenarios (i) using containerized applications inside of Docker and (ii) running complete operating system (PowerPC) inside LXC. They conclude that a containerized execution of control applications can meet requirements of PLC and automation controllers.

From the latency point of view, Masek et al. [22] performed a literature review on sandboxed RT software on the example of self-driving vehicles. The researchers were interested in the question: How does the execution environment influence the scheduling precision and input/output performance of a given application? The result shows that docker does not impose additional overhead (similarly to [24]) for scheduling and input/output performance. However, selecting the correct kernel has a greater impact on the scheduling precision and input/output performance of containers.

Mao et al. [21] uses RT containers to enable software-based RAN (Radio Access Network) in order to avoid high capital and operating expenditures during deployment of new standards. However, the software based RAN has strict deadlines to satisfy (1ms). The researchers use RT patch to decrease the latency, interestingly they improve the latency 13.9 times by applying the patch in comparison to the vanilla Kernel.

7.3.2 Methods based on Real-time Co-Kernel

In this approach, a RT micro-kernel runs in parallel to Linux kernel. The RT co-kernel handles time critical activities (e.g., handling interrupts and scheduling RT threads), standard Linux kernel runs only when the co-kernel is idle. In comparison to the RT patch, the co-kernel approach offers lower latencies and lower jitter. On the other hand, it requires special APIs, tools and libraries for the application development. Additionally, there are impediments with scaling co-kernel solutions on large platforms (e.g., many cores platforms). There are two co-kernel alternatives: Real Time Application Interface (RTAI) and Xenomai.

RTAI aims to minimize latencies to the lowest technically possible values. Real-time tasks are compiled as kernel modules and ran in the kernel space. Xenomai [25] is a fork of RTAI. Its mission is to enable RT tasks in the user space. It consists of an emulation layer that is capable of reusing code from other Real Time Operating System (RTOSs).

Tasci et al. [16] elaborates on modularization of RT control applications into RT containers. Such modular architecture needs two essential parts: (i) Computational part, enabled by a RT operating system (combination of Xenomai and RT patch), and (ii) Messaging part that allows passing messages between containers in a RT manner. Traditional monolithic architectures communicate through function calls and shared memory, the containers do not make the assumption if they are running on the same host or in a distributed environment (they communicate through standard OS networking stack), therefore direct passing messages through shared memory is not directly supported. Hence, the researchers provide a design and implementation of a custom made RT messaging system for containers based on ZeroMQ[4].

Hofer et al. [18] use the RT containers in the context of control applications. The paper presents comparison between type 1 hypervisor, Vanilla Linux, Xenomai co-kernel and Linux with RT patch for various idle and stress scenarios.

7.3.3 Method Based on Hierarchical Scheduling of Containers

Inspired by a similar concept in the hypervisor-based virtualization where a global scheduler assigns CPU time for the virtual machines, the second layer scheduler schedules the individual tasks of the VM.

Cucinotta, Abeni et al. [14, 15, 13] proposed the use of RT containers on the field of Network Function Virtualization (NFV), where the functionality of traditional physical network devices (e.g., firewalls) is transformed into software components (in containers) that are consolidated in a single computing device. NFV has critical latency requirements inducted by the need of time critical per-packet processing. The researchers modified the Linux scheduling mechanism to provide two levels hierarchical scheduling. First level Earliest Deadline First scheduler selects the container to be scheduled on each CPU. Subsequently the second level Fixed Priority scheduler selects a task in the container. CPU reservation (runtime quota and period) is assigned to each of the containers.

7.3.4 Custom Methods

Wu et al. [23] proposed the Flexible Deferrable Scheduler for containerized mixed-criticality RT systems that consist of RT and non RT containers. The scheduler guarantees the allocated CPU capacity to RT containers and dynamically distributes the unused capacity to non RT containers. The work supplements Completely Fair Scheduler with a Workload Adjustment Module that collects CPU utilization by containers and Dynamic Adjustment Module that allocates CPU to the container.

Cinque et al. [12] (previously [11]) implemented RT containers using Linux patched with RT co-kernel (RTAI) and utilizing custom made monitoring and

[4]`https://zeromq.org/`

policy enforcing modules. Their solution allows to co-habit containers with different criticality levels and to prevent fixed-priority hard RT periodic tasks inside of the containers to affect the temporal guarantees of other containers. The temporal guarantees are provided by two mechanisms: (i) proper tasks priority assignments to tasks inside the containers and (ii) monitoring and enforcing temporal protection policies. The former ensures that tasks inside of the high-criticality containers are assigned higher priorities than tasks in the lower-criticality containers and thus they are never preempted by tasks of lower criticality containers. The latter monitors the tasks and, in case of overruns or overtimes, it enforces one of the temporal protection policy (i.e., kill or suspend the faulty task, suspend the task until the next period).

7.4 Challenges of Real-time Container-based Virtualization

In the reviewed papers, we identified shortcomings and immature aspects of RT container virtualization that prevents the expansion of the technology. Below, we listed them categorized in three groups: (i) tools support, (ii) RT communication support, and (iii) miscellaneous.

Lack of tools for RT container management: The reviewed papers emphasises a need for supporting tools for RT containers. Tools that enable deployment on containers taking into account RT requirements of containers and properties of computational nodes.

- The need for an orchestration tool that can schedule RT containers based on pre-configured capabilities [18].
- Middleware that is aware of both communication needs as well as run-time and performance isolation needs [17].
- Framework to expose the runtime requirements of RT application running inside containers and to enforce an optimal allocation of containers to resources [17].

Communication between RT containers: Real-time communication between a container and its environment has to be further researched. Currently, the reviewed papers emphasize the following issues:

- Need for a RT communication among containers [17].
- Further investigation on container security restricted container access and intra-container communication [18].
- A research on data management shared across containers [19].

Miscellaneous: In addition to generic issues that may harm the RT behaviour (e.g., shared caches, memory and I/O), the studies reviewed highlight the following points and questions:

- Lack of safety, security analysis of RT containers and vulnerability management for the acceptance in industry [20, 19].
- Lack of latency and performance tests of recent releases of a patched Linux Kernel. As well as a proper analysis of configuration of the Linux kernel parameters that may improve overall task determinism. [18].
- The measurements of memory overhead of the container solution and is it acceptable for real world applications [19].
- Processes in different containers may use the same resources in the same way because of their independent views of the system (i.e., processes are not aware of a resource-limited isolated environment co-located with other containers). This results in poor resource utilization as well as a potential violation of the RT execution assumptions [17].
- Container approaches are a new technology. Will this create problems due to its possible immaturity [19]?

7.5 Conclusion

Container-based virtualization has become popular as a lightweight alternative of hypervisor-based virtualization. The technology has proven its viability in large cloud-based systems, it has been adopted by a number of enterprise companies and it is supported by a large scale of tools (e.g., container orchestration and monitoring tools).

However, in industrial domains where the RT behavior is required, the container-based virtualization seems not to be mature enough. In this paper, we summarize the research carried out in the field of RT containers. We show in what contexts, what approaches and technologies are used, and what are the possible immaturity points of the RT container-based virtualization.

Bibliography

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

[2] Cristian Spoiala, Alin Calinciuc, Cornel Turcu, and Constantin Filote. Performance comparison of a webrtc server on docker versus virtual machine. *13th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS, Suceava, Romania, May 19-21, 2016*, 2016.

[3] Thanh Bui. Analysis of docker security. *ArXiv*, 2015.

[4] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. of Syst. Architecture*, 2018.

[5] Jo Hannay, Dag Sjøberg, and Tore Dybå. A systematic review of theory use in software engineering experiments. *Software Engineering, IEEE Transactions on*, 2007.

[6] Claes Wohlin. *ACM International Conference Proceeding Series*, 2014.

[7] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *IEEE Int. Conf. on Cloud Eng.*, 2015.

[8] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. 2011.

[9] Claudio Scordino and Giuseppe Lipari. Linux and real-time: Current approaches and future opportunities. In *IEEE Internafional Congress ANIPLA*, 2006.

[10] Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *Acm Sigbed Review*, 2015.

[11] Marcello Cinque and Domenico Cotroneo. Towards lightweight temporal and fault isolation in mixed-criticality systems with real-time containers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018.

[12] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *Euromicro Conf. Real-Time Syst.*, 2019.

[13] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. In *IEEE Int. Conf. on Edge Comp. (EDGE)*, 2019.

[14] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Virtual network functions as real-time containers in private clouds. In *IEEE Int. Conf. on Cloud Comp. (CLOUD)*, 2018.

[15] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *SIGBED Rev.*, 2019.

[16] Timur Tasci, Jan Melcher, and Alexander Verl. A container-based architecture for real-time control applications. In *IEEE Int. Conf. on Eng., Tech. and Innov. (ICE/ITMC)*, 2018.

[17] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. OS-level virtualization for industrial automation systems: are we there yet? In *ACM Symp. on Applied Computing (SAC)*, 2016.

[18] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to IAAS. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[19] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[20] Kilian Telschig, Andreas Schonberger, and Alexander Knapp. A real-time container architecture for dependable distributed embedded applications. *IEEE Int. Conf. Automat. Science and Eng.*, 2018.

[21] C. Mao, M. Huang, S. Padhy, S. Wang, W. Chung, Y. Chung, and C. Hsu. Minimizing latency of real-time container cloud for software radio access networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[22] Philip Masek, Magnus Thulin, Hugo Andrade, Christian Berger, and Ola Benderius. Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016.

[23] J. Wu and T. Yang. Dynamic cpu allocation for docker containerized mixed-criticality real-time systems. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, 2018.

[24] A. Krylovskiy. Internet of things gateways meet linux containers: Performance evaluation and discussion. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015.

[25] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai*, 2004.

Chapter 8

Paper C: REACT: Enabling Real-Time Container Orchestration.

Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos.

In the 26th International Conference on Emerging Technologies and Factory Automation (ETFA 2021).

Abstract

Fog and edge computing offer the flexibility and decentralized architecture benefits of cloud computing without suffering from the latency issues inherent in the cloud. This makes fog computing very attractive in real-time and safety-critical applications, especially if combined with container-based technologies. Whereas different orchestration systems are available to manage the container placement based on their resource demand, no orchestration system is considering real-time requirements for containerized applications.

In this paper, we present the architecture and design of a real-time container orchestrator based on Kubernetes. Moreover, this paper defines metrics for the performance evaluation of real-time containers, and describe an initial model for allocating a mixture of real-time and non-real-time containers. We present an initial implementation of our real-time container extension and evaluate its feasibility on Linux-based systems.

8.1 Introduction

Fog and Edge Computing (FEC) are conceived to overcome some of the main limitations of cloud computing, e.g., tackling the limitations of unbounded communication latency and variance in bandwidth availability [1]. FEC enables the benefits of offloading data collection and decision-making even in application domains that require RT guarantees, e.g., in industrial automation [2, 3]. FEC systems decrease the communication latency for critical data, which can then be processed on the decentralized computational devices on the network's edge. While guaranteeing deterministic communication behavior between nodes can be achieved through the use of, e.g., TTEthernet or TSN, guaranteeing the RT behavior of computation functions can be challenging. The RT execution of functions on nodes cannot always be maintained in the same way due to the complexity and non-determinism of hardware artifacts (e.g., caches), and software layers within the compute nodes (e.g., OS layers, network stack, interrupts). Hence, to maintain the RT behavior of functions, isolation and adaptation/re-allocation of functions at run-time is necessary. Typically, the re-allocation and the isolation of functions have been facilitated through the use of virtualization technologies.

Container-based virtualization is gaining importance in industrial domains as a lightweight alternative to full-blown virtualization [4, 5]. Containers are standalone self-containing software packages, comprising applications and their software dependencies, that simplify the deployment of software [6]. The technology enables sharing of computation resources among several containers while preserving isolation. Recently, there has been an effort to enhance container-based virtualization with time predictability (i.e., RT containers), enabling container applications with RT requirements [7, 8, 9]. Through these enhancements, containers may be used for time-critical applications required by many industrial systems such as robot control in fog computing [2, 3]. In parallel, due to the continuous change in resource usage by applications running in fog and the availability of the resources, there is a need to dynamically manage and deploy containers in a cluster of compute nodes necessitating container orchestrators that take into account both resource requirements and resource availability. Additionally, container orchestrators offer further benefits, e.g., scalability and availability, fault-tolerance, and efficient resource utilization [10].

RT systems require additional constraints regarding the reaction time to environmental events [11]. So far, container orchestrators do not consider RT requirements of applications running inside containers, neither from the theoretical nor practical points of view. There are no mechanisms of the distribution of containers based on their RT requirements without violating these requirements and how to manage the co-execution of RT and BE containers on the same node. RT containers are a novel topic [9]. There are no strategies for dealing with dynamic changes in container workloads that may change interference between RT containers. Containers do not provide strict resource isolation as they share not only physical resources but also OS kernel [12], and hence, they are prone to performance degradation in the presence of other competing containers. For example, disk-intensive workloads can induce performance degradation up to 35% [13]. Therefore, if the benefits of containers are to be exploited for RT systems, run-time monitoring and container orchestration that continuously evaluates QoS metrics and uses them in container scheduling decisions is necessary.

This paper proposes a container orchestration architecture to support the co-existence of RT and BE containers with temporal isolation. Specifically, the contributions of this paper are as follows:

- The design of a container orchestration architecture enabling the deployment and online adaptation of both RT and BE containers considering the timing requirements defined for time-critical applications;
- The definition of a mathematical model for the RT components of the designed orchestration architecture, and of the performance metrics needed to implement run-time monitoring and orchestration of the RT containers;
- The implementation of the proposed orchestration over the existing open-source container orchestrator Kubernetes.

8.2 Background and Prior Work

RT systems are computing systems that require deterministic temporal behavior since the correctness of their functionality depends not only on the value but also on the timing of the computed result [11]. Many software applications utilize periodic tasks that are cyclically executed [14]. Typically, RT systems require that tasks finish their execution in every period instance to be considered correct (assuming an implicit deadline for the tasks).

Virtualization is a technology that allows multiple virtualized applications or systems to be co-located onto the same shared platform [15]. There are two prevalent virtualization technologies: hypervisor- and container-based virtualization. Hypervisor-based virtualization utilizes a hypervisor that is a software layer that creates the different partitions within which each virtualized instance of an OS runs. In contrast, container-based virtualization utilizes OS features to create an isolated environment for processes. Container-based virtualization does not impose any requirements on hardware support for virtualization. Hence, such technology can be utilized on a broad range of devices. Containers co-located on a single computing node run as user-space isolated tasks and share functions of the host's OS (e.g., scheduling, memory allocation). From the user's view, each container appears and executes like a stand-alone OS. In comparison to the hypervisor-based virtualization, container-based virtualization has a negligible overhead, is more resource efficient (e.g., 29.4 times less RAM usage than a VM [16]), has higher flexibility, provides fast booting times [17], and enables a near-native performance [6, 4, 5]. However, container-based virtualization provides a low level of isolation for memory, disk, and network operations [13]; hence, such operations may lead to degrading QoS of the applications.

The container-based virtualization relies on *namespaces* and *control groups*, referred to as *cgroups*, implemented in the host OS [18]. *Namespaces* partition global resources, e.g., tasks, network, inter-process communication, into different sets, only visible by different task groups [9]. *cgroups* enable the organisation of tasks into hierarchical subgroups with various configurable runtime properties, that allows a dynamic resources redistribution among the subgroups [19].

Looking at the RT support for container-based virtualization, three major directions are aiming to improve RT behavior of containers: hard RT co-kernel that co-exists with a standard Linux Kernel [20, 21, 22], solutions based on the *preempt_rt* patch for Linux that aims to minimize the latency in the Kernel [23, 24, 25], and solutions that employ hierarchical scheduling [7, 18, 8]. The RT properties of container-based virtualization depend on the underlying OS. In this paper, we consider the general-purpose OS Linux. While Linux was not designed for RT operation, there are considerable efforts to improve the time determinism on several levels, e.g., introducing RT scheduling as a standard kernel feature and providing time-predictable kernel behavior through full preemption. On the task scheduling level, there are RT scheduling policies in the Linux kernel: First-In-First-Out and Round Robin combined with priority queues to provide different priority levels for tasks, and Earliest Deadline First/Constant-Bandwidth Server that prioritizes tasks dynamically according to their deadlines.

RT scheduling support for containers has been added in [18] through the implementation of hierarchical scheduling combining standard fixed priority scheduler (*SCHED_FIFO* or *SCHED_RR* scheduling policy) and *SCHED_DEADLINE*, consistent with the multiprocessor periodic resource analysis from [26]. The kernel is extended with a reservation-based scheduling policy in which each virtual CPU (vCPU) is assigned a quota and a period, bounding the execution of the vCPU to the respective quota in each time interval of length equal to the period. On the container level, the global *SCHED_DEADLINE* policy selects at each time instant the container with the earliest deadline, while at the task level, the *SCHED_FIFO/SCHED_RR* policy is used to schedule tasks within containers. Once there is no task to be scheduled by the *SCHED_FIFO*, other BE tasks, are executed via the default scheduling policy. This enables the co-existence of RT and BE containers. We use this hierarchical patch as the implementation basis for our environment to host containers in compute nodes.

Container orchestrators automate the deployment, management, and scaling of containers in clusters of heterogeneous computing nodes. The main functionality of container orchestrators is the placement of containers in a cluster of compute nodes following placement policies and user-supplied placement constraints. The goal is to choose an optimal compute node to start the requested containers on. The orchestrator matches the resources requirements with the resource capacity of the nodes, e.g., CPU, memory, and disk storage capacity, and applies strategies to maximize the performance (e.g., the highest spread of containers). Additionally, orchestrators address fault-tolerance of the deployed containers, scaling or removing containers based, load balancing, container health monitoring, and efficient resource utilization. There are several container orchestrators available: Kubernetes, Docker Swarm, None of them support the deployment of containers based on their timing requirements [10]. Closest to our work is [27], where Xi et al. utilizes OpenStack to orchestrate RT VMs.

8.3 Orchestration of real-time containers

This paper introduces a solution to enable the orchestration of RT containers so that the RT requirements are taken into account during the scheduling process. We define a set of scheduling policies, run-time monitoring mechanisms, performance metrics, and an implementation that supports the deployment and execution of RT containers. The RT container orchestrator provides the following functionalities:

- *Placement of RT and BE containers*: The orchestrator places the RT containers according to their RT requirements (i.e., quota and budget). The orchestrator prevents over-reservation of the resources at the container scheduling level by performing a utilization-bound schedulability test.
- *Run-time monitoring of RT QoS of the containers*: The orchestrator continuously monitors resource usage and the delivered QoS expressed by a set of metrics of the containers deployed in the compute nodes administrated by the orchestrator. As the OS does not enforce strong container isolation, there may be an interference with other noise-perturbing containers that may influence the timing behavior of RT containers. The orchestrator takes the information into account in the next scheduling decisions.

8.3.1 System Model

In this section, we define our system model including infrastructure, compute node, container, and task elements.
Infrastructure: We consider a system $\mathbb{S}$ consisting of a container orchestrator F and n connected compute nodes denoted by $f_1, \ldots, f_n$.
Compute node: Each node f_j offers memory and storage resources of a given capacity, denoted with f_j^M and f_j^S, respectively. Each compute node f_j is capable of hosting a mixture of RT containers and BE containers. We define the set of RT and BE containers on node f_j with Π_j^{rt} and Π_j^{be}, respectively. We introduce the OSLM property of a node f_j, denoted by $OSLM_j$, to quantify its performance.

Container: Each container π_k has memory (π_k^M) and storage (π_k^S) demands. An RT container $\pi_k \in \Pi_j^{rt}$ has RT interface expressed as (P_k, Q_k) where Q_k is a CPU quota over period P_k. We introduce the metric of a container π_k, denoted by CLM_k, to capture the performance of RT containers.
Task: Each container π_k accommodates a set of tasks denoted by $\mathcal{T}_k$. Each RT task τ_i is defined by the tuple $\langle a_i, C_i, T_i, D_i \rangle$, where a_i represents the release or arrival time (i.e., the time relative to the period when the task becomes active), C_i is the worst-case execution time bound, T_i is the period, and D_i is the relative deadline of the task. We use two functions $L_i(t_0, t_1)$, and $R_i(t_0, t_1)$ to capture the maximum lateness and maximum response time of the task τ_i in the interval $[t_0, t_1)$. The lateness represents the delay of the task's completion with respect to its deadline, while the response time measures the difference between the finishing and arrival time of a task. The functions return a set of lateness and response time values, respectively, corresponding to the task instances executed within the given time interval. We also use a counter defined as $\mu_i(t_0, t_1)$ to capture the number of deadline misses for task τ_i in the time interval $[t_0, t_1)$. Tasks assigned to BE containers do not have any timing requirements.

There are methods [28] to abstract the timing requirements of a set of RT tasks under certain scheduling algorithms into a periodic resource model in a form similar to the RT interface (Q_k, P_k) of the containers. If the appropriate scheduling mechanisms are in place, the abstracted resource guarantees that when the resource receives an allocation of Q_k over a period P_k, all RT tasks within the resource will meet their RT requirements [28].

8.3.2 Performance Metrics

There are several metrics, collectively defined as OSLM [29], to evaluate the performance of a system in terms of task execution. Such metrics are useful to estimate the suitability of the system to run RT tasks. E.g., Interrupts with a non-preemptable section that can influence the RT performance of the system [30], CPU Utilization, number of handled interrupts per second, number of I/O requests per second, and amount of data read/written. There is a number of tools for collecting performance data [31]: *mpstat*, *iostat*. These tools collect the performance data of tasks from *proc* and *cgroups* directories to estimate the OSLM.

On the container level, there is a lack of measurement methodology, tools, and best practices, as well as a lack of metrics on the characterization of the container overhead [31]. Available tools, e.g., *docker stats* and *cAdvisor* allow to estimate the basic set of container-related metrics (e.g., CPU and memory utilization). However, when considering the RT QoS evaluation of containers, the available tools are lacking such capabilities. In this paper, we define CLM that helps to evaluate the performance of RT containers.

8.3.3 Container Level Metrics

Several metrics are used to evaluate the timeliness [11] of tasks running in the system, which we adapt to assess the RT performance of containers. The following properties characterize the RT performance of a task:

- *Number of deadline misses*: represents the number of times that a deadline of a task was exceeded.
- *Lateness*: represents the delay of a task with respect to its deadline [11]. If the task finishes before its deadline, the lateness is negative.
- *Response time*: represents the difference between the finishing time and the start time of a task. [11]

We define CLM to evaluate the RT performance of tasks running in the respective container. For each of the tasks in a container π_k, the CLM are:

- *Number of deadline misses*: characterizes the total number of deadline misses of tasks inside of the container π_k between time t_0 and t_1:

$$\mathcal{M}_k(t_0, t_1) = \sum_{\tau_i \in \mathcal{T}_k} \mu_i(t_0, t_1).$$

- *Maximum lateness*: characterizes the maximum lateness encountered by a task inside the container between time t_0 and t_1:

$$\mathcal{L}_k^{\max}(t_0, t_1) = \max_{\tau_i \in \mathcal{T}_k} \{L_i(t_0, t_1)\}$$

- Maximum response time: characterizes the maximum response time encountered by a task inside the container between time t_0 and t:

$$\mathcal{R}_k^{\max}(t_0, t) = \max_{\tau_i \in \mathcal{T}_k} \{R_i(t_0, t_1)\}.$$

We express CLM within the observation interval $[t_0, t_1)$ (usually from system start at $t_0 = 0$ until the current time) as follows:

$$CLM_k(t_0, t_1) = [\mathcal{M}_k(t_0, t_1), \mathcal{L}_k^{\max}(t_0, t_1), \mathcal{R}_k^{\max}(t_0, t_1)]$$

8.4 Design of the RT Orchestrator

The orchestration system is based on the master-minion architecture that consists of a master node and a set of minion compute nodes connected in a cluster as described in [10]. The core of the system is the master node that makes global decisions about the cluster; it receives users' requests for container deployments, continuously monitors states of compute nodes in the cluster and schedules containers on computing nodes. The master node's functionality can be distributed across several physical machines to avoid a single point of failure [10].

The compute nodes, depicted in Figure 8.1, provide an environment for hosting containers and run a node agent that communicates with the master node through defined APIs. The node agent takes container deployment specifications defining container requirements and deployment parameters.

8.4.1 RT extension of the master node

The master node depicted in Fig. 8.2 is a central point in the architecture; it accepts user-defined container deployment specification enhanced with RT interface and task annotations. It provides mechanisms for admission control and scheduling of containers. Additionally, it continuously collects performance metrics of compute nodes and containers. In the following text, we elaborate on the proposed enhancements:

Container Deployment Specification

Each container deployment specification, which is supplied to the master node via a dedicated API, contains the specification of the RT interface and the container annotation. The RT interface specifies the CPU reservation (P_k, Q_k) of the respective container following the periodic resource model of the hierarchical scheduling framework (c.f. [32]). The container specification contains the description of the tasks inside the container to compute the CLM during runtime.

RT Resource Monitor

RT Resource Monitor stores the OSLM and CLM from individual compute nodes. The data are accessible to RT Admission Controller and RT Container Scheduler to support the scheduling decisions.

RT Admission Control

The admission control determines if there are nodes in the cluster with enough available resources to accommodate the resource demands of the new container. Moreover, it performs necessary utilization-bound schedulability tests that reject those nodes on which the RT timing requirements cannot be met.

We perform the checks as defined below to decide if a new container, denoted by π_{new}, can be allocated to a certain node.
1) We check if the available resources (memory, storage), when considering both RT and BE containers, are enough to fit the resource demands of the new container:

$$\forall f_i \in \mathcal{S} : \begin{cases} \pi_{new}^M + \sum\limits_{\pi_k \in \Pi_i^{rt} \cup \Pi_i^{be}} \pi_k^M \leq f_i^M \\ \pi_{new}^S + \sum\limits_{\pi_k \in \Pi_i^{rt} \cup \Pi_i^{be}} \pi_k^S \leq f_i^S \end{cases} \tag{8.1}$$

2) We check that the new container will not make the existing RT containers unschedulable by performing a necessary utilization-based test [11]:

$$\forall f_i \in \mathcal{S} : \frac{Q_{new}}{P_{new}} + \sum_{\pi_k \in \Pi_i^{rt}} \frac{Q_k}{P_k} \leq 1 - \delta_i \tag{8.2}$$

where, δ_i refers to the system overhead of node f_i (discussed in detail below), which reduces the amount of CPU bandwidth available to the containers/tasks. We remind the reader that (P_k, Q_k) is the RT interface of a RT container π_k, which defines that the container will be scheduled for at most Q_k time units over a period of P_k time units. Hence, this utilization-based test (c.f. [14]) defines that if the utilization of the RT containers (including the utilization of the new container) exceeds the bandwidth of the CPU, then the RT requirements of the new container cannot be guaranteed. Please note that the check is not sufficient, i.e., if the check is passed, it does not mean that the RT behavior will always be guaranteed since the actual execution of the RT tasks diverges from the ideal RT schedule due to e.g., timer resolution, scheduling jitter, locking mechanisms (c.f. [30]) or the overhead introduced by the container mechanism.

RT Container Scheduler

The Container Scheduler decides which of the feasible nodes in the cluster, the feasibility being determined through the admission control tests defined above, to assign the new container to. The decision is based on the CLM and OSLM introduced above, which are constantly monitored at run-time. Additionally, the scheduler might decide to change some properties of already running containers (e.g., the RT interface) to be able to fit the new container on a node. Hence, the problem of where to place new containers according to their RT requirements or which containers to modify can be viewed as a multi-objective optimization problem. While the exact mechanism on how to assign (and re-dimension) containers is out of the scope of this paper, we give a brief discussion on the important aspects of this orchestration problem and leave the definition and solution of this optimization problem for future work. Furthermore, selecting the best mix of metrics and optimization objectives to use is also not in the scope of this paper. However, we will briefly discuss several strategies below.

In terms of the observed OSLM properties, we identify several important aspects and strategies which we briefly discuss below.

The *system overhead* (δ) reduces the amount of CPU bandwidth that the containers can use. We show in the experiment section (Sect. 8.6) that the overhead when co-locating RT and BE containers influences only the BE containers while the RT containers are guaranteed their allotted budget (c.f. Fig. 8.4). We see that the overhead remains constant and jitters around the constant value both when changing the RT container period and when increasing the number of RT containers. Please note that the overhead analysis needs to be extended in future work to create a more accurate overhead model that produces a safe upper bound for admission control. However, even though the overhead model is deduced empirically and not analytically, we can still use it in the admission control since any overhead impact on RT containers will be corrected by the online reconfiguration algorithm.

Another objective may be to perform load balancing on nodes to not over-utilize some of the nodes. This decreases the probability of deadline misses and lateness for tasks and increases the probability of fitting future container requests.

The optimization function may also select nodes with a low number of context switches as this also influences the system utilization, leaving more CPU bandwidth available for container execution. Furthermore, nodes with a high number of interrupts/sec are more likely to lead to deadline misses due to the irregular nature of interrupt arrivals. Hence, they might not be the best selection for placing RT containers.

In terms of CLM properties, nodes with few (or zero) deadline misses are better candidates for new RT containers. However, adding additional RT containers might increase the number of deadline misses or the lateness/response times of tasks. The container scheduler needs to consider if an increased rate of timing failures is acceptable, depending on the nature of the running RT containers.

When using EDF to schedule containers, the admission test above also becomes necessary, i.e., if the test is passed, the new container can, in theory, be scheduled on the respective node. However, the divergence from the real schedule (described in [30]) can lead to deadline misses and/or negative effects on the lateness and response times of tasks. Hence, the run-time monitoring of the CLM properties can give hints about which containers to re-assign or re-dimension. The observation interval in the CLM properties can be dynamically adjusted to identify which of the new containers has a negative impact on the RT properties of the already running containers.

Feedback-based resource reservation mechanisms can be used to adapt the CLM properties at run-time while keeping hard RT guarantees for all the RT containers [33]. Such mechanisms can be implemented at the orchestration level to compensate for potential unforeseen over-or under-runs, providing a limited impact on the other RT containers. The overhead for feedback-based resource reservation is minimal since it requires implementing an integral control strategy for each container. Such approaches have proven successful also in other domains, such as mixed-critical systems [34].

8.4.2 RT Extension of Compute Nodes

We enable RT containers on node-level through the use of the *preempt_rt* patch, hierarchical scheduling of containers [18], hard RT co-kernel, or a combination of these technologies. The overview of the compute node extension is depicted in Fig. 8.1. A module responsible for the RT related functionality is denoted as *RT manager*. The process of deployment of RT containers is as follows: Upon receiving a deployment request from the orchestrator, the RT manager locate a requested image (either locally or in a remote repository), the container image is fetched, and instantiated with the requested parameters that set-up resources for the container. Concerning the RT functionality, the *RT Manager* offers the following:

- Deployment of RT containers: The containers have to be instantiated in RT mode and must be assigned the requested quota and period.
- Monitoring of RT performance: The monitoring functionality assesses the wellness and the performance of the compute node and the containers deployed. There are two monitoring parts: the OSLM monitor and the CLM monitor. The OSLM monitor collects data through the *proc* and *cgroup* file-system, which contain information related to interrupts, memory usage, CPU utilization. The CLM monitor evaluates the timelines of the containerized tasks in the RT containers.
- Reporting of the performance metrics to the orchestrator: The compute nodes report the OSLM and CLM to the master node, that uses the collected data for the admission control and container scheduling and, additionally, it can take decisions whether to re-allocate and/or re-assign resources of the containers.

8.5 Implementation

In order to show the feasibility of the proposed system, we have extended the existing open-source orchestrator Kubernetes with the ability to schedule containers onto compute nodes while taking into account their RT requirements. The RT behavior of containers on compute nodes is enabled by using the hierarchical patch[1] presented in [18] which we have extended with the monitoring capabilities. The system allows users to define RT requirements of the containers which need to be deployed (i.e., quota and period) and ensures that at the container scheduling level, the allocation to compute nodes respects the given requirements and does not lead to an overload in the respective node. Additionally, as the containers do not provide strong resource isolation, the system provides run-time monitoring of the container's performance. The implemented extension consists of the following components:

- *The RT Scheduler Extender* on the Master node is an extension of the Kubernetes control plane, which provides admission control and scheduling of RT containers onto compute nodes in the cluster.
- *The RT Manager* on compute nodes provides functionality to deploy RT containers and periodically evaluate and report the RT performance to the Master node.

The architecture of the Kubernetes extension is described in Fig. 8.3. The Kubernetes Master receives a deployment request to deploy a set of containers (denoted as a Pod in the context of Kubernetes). A new Pod is placed in a queue with other unscheduled pods, the Kubernetes scheduler (*kube-scheduler*) periodically checks the queue. If there is an unassigned pod, the Kubernetes scheduler attempts to place the Pod in a suitable node. First, the scheduler filters out infeasible nodes with insufficient available resources (e.g., insufficient memory or storage) as described in Section 8.4.1. Subsequently, so-called custom webhooks are triggered during each schedule polling cycle. The webhooks permit to attach custom actions to scheduling events (e.g., filtering and prioritizing events). We have implemented a custom *rt-filter* webhook that takes into account the utilization of the node as well as the RT interface as requested in the container deployment specification. If the utilization, including the new RT container, is

[1]Available at `https://github.com/lucabe72/LinuxPatches/tree/HCBS`

above a certain level, the node filtered out as infeasible and not used for hosting the container. In this way, we avoid overloading the host.

The *RT Scheduler Extender* performs a secondary filtering as described in Section 9.3 and ranks the nodes with the custom *rt-scoring* webhook. The *rt-scoring* ranks the nodes according to their suitability to host the RT container. For this work, the scoring is computed as the remaining unreserved CPU capacity. However, the rt-scoring step can incorporate additional properties of the compute node given in the OSLM and CLM to minimize the number of, e.g., deadline misses on compute nodes. As the orchestration is not part of the current paper, we leave this extension to future work.

The Pod is then assigned to the node with the highest score. The node agent on the compute node identifies that the Pod is assigned to its node and deploys it with the given RT parameters. The *RT Manager* continuously monitors the node's state beyond the understanding of the default Kubernetes monitoring metrics and reports them back to the master node. The RT manager monitors the previously described CLM and OSLM: i.e., the number of context switches, interrupts per second, I/O access, as well as task deadline misses, response time, and lateness.

As an input, the Kubernetes master accepts a Pod deployment request that contains the deployment specification of a group of one or more containers. If there are multiple containers in a pod, these are assigned and scheduled on the same node and run in a shared context (i.e., sharing memory and network). As a simplification, we assume that each Pod contains at most one RT container. We amend the deployment configuration (stored in the annotation part of the deployment file) to contain the RT interface (Q_k, P_k) as well as the list of tasks and their $\langle C_i, T_i, D_i \rangle$ parameters.

The *RT manager* runs as a Kubernetes daemon set and provides functionality for run-time monitoring and reporting of the performance of the RT containers and the system. A daemon runs on every compute node in the cluster. The monitoring part continuously monitors OSLM and CLM. The OSLM are computed by utilizing the Linux special *procfs* file system (*/proc* and */cgroups*) that contain information about tasks, similar to [31]. The CLM are derived from the custom tracepoints that we injected into the schedulers implementing the *SCHED_FIFO* and *SCHED_DEADLINE* policies in the Linux Kernel. The following events are recorded and periodically evaluated by the daemon:

- Container Started: The container is in a running state, the quota/period has been allocated, and the container's tasks are ready to run.
- Container Throttled: The container used more CPU quota than the allocated one and therefore, the scheduler throttled the container.
- Task Instance started: The start of j^{th} instance of task τ_i.
- Task Instance finished: The end of j^{th} instance of task τ_i.

From these tracepoints, we compute deadline misses, lateness, and response times of tasks within the RT containers.

8.6 Evaluation

In this section, we show the system's behavior for co-located RT and BE containers on a single compute node. The set of experiments illustrates the distribution of the CPU time amongst co-located containers. Tables 8.1, 8.2 and 8.3 show the feasibility of having a mixture of RT and BE containers on a single compute node. We change the reservation period and budget from values 0.1ms to 1000ms and measure the overhead (described below). We show the behavior of low utilization containers (10%) and high utilization (90%). We investigate if RT containers with very short periods introduce significantly more overhead than those with large periods. In the experiments, we instantiate an RT container and a heavy load BE container. The experiments for Table 8.1, 8.2 use containers that run CPU intensive operations. We measure the actual time that the containers spend on the CPU. The rest of the CPU that is not used by any container is considered as an overhead. Table 8.3 shows a similar experiment where the BE container runs *stress-ng* to generate an excessive workload aiming to affect the assigned CPU time of the RT containers. The stress generating BE containers execute 10 CPU intensive threads, 10 HDD intensive threads, 10 threads generating I/O stress, and 10 threads generating context switches.

We use an Intel i5 machine with 8GB RAM running Debian Linux, Kernel 5.2. patched with Hierarchical Scheduling Patch [18], and Docker v20.10.

We consider the overhead to be the part of the CPU capacity that is not used for any computation of the containerized tasks. It is caused by system-related

Table 8.1. RT containers (10% utilization) on a single core with noise generated by 10 CPU intensive containers.

	(1ms, 0.1ms)	**(10ms, 1ms)**	**(100ms, 10ms)**	**(1000ms, 100ms)**
RT Containers	10.0607% ± 0.00193%	10.0021% ± 0.00002%	9.9983% ± 0.00002%	9.9978% ± 0.00002%
NRT Containers	88.7870% ± 0.00406%	88.8963% ± 0.00280%	88.8713% ± 0.00222%	88.8906% ± 0.00264%
overhead	1.1522% ± 0.0035%	1.1016% ± 0.0028%	1.1304% ± 0.0022%	1.1115% ± 0.0025%

Table 8.2. RT containers (90% utilization) on a single core with noise generated by 10 CPU intensive containers.

	(1ms, 0.9ms)	**(10ms, 9ms)**	**(100ms, 90ms)**	**(1000ms, 900ms)**
RT Containers	89.9037% ± 0.00031	89.8597% ± 0.00042%	89.8818% ± 0.00066%	89.7780% ± 0.00316%
NRT Containers	9.0345% ± 0.00212	9.0368% ± 0.00193%	9.0349% ± 0.00045%	9.0658% ± 0.00164%
overhead	1.0618% ± 0.0022	1.1035% ± 0.0019%	1.0833% ± 0.0005%	1.1563% ± 0.0018%

tasks, context switches or docker-related processes, etc. In the experiments, we execute RT and BE containers simultaneously. Each of the containers executing a loop with a CPU-heavy computation. In theory, the containerized processes should fully utilize the processor; however, the full CPU capacity is not used exclusively for these processes. We can see that the overhead stays the same even when using an RT container with a very short period.

To investigate the overhead, we utilize *systemTap*, which is an instrumentation framework for Linux-based kernels. SystemTap allows instrumenting Linux events with user-defined code in form of loadable kernel modules. We monitor the following events in the Linux kernel:

- *scheduler.cpu_on*: The process is beginning execution on a CPU.
- *scheduler.cpu_off*: The process leaving the CPU.

From the recorded events, we are getting the total measurement time (from the first event to the last one) and each containerized task's total time.

The utilization test in Fig. 8.4 shows the distribution of CPU time on a single

Table 8.3. RT containers (10% utilization) on one core with noise generated by BE containers executing *stress-ng*.

	(1ms, 0.1ms)	**(10ms, 1ms)**	**(100ms, 10ms)**	**(1000ms, 100ms)**
RT Containers	10.0508% ± 0.00045%	10.0065% ± 0.00005%	9.9956% ± 0.00006%	9.9956% ± 0.00002%
NRT Containers	89.0251% ± 0.00091%	88.9333% ± 0.00034%	88.8779% ± 0.00296%	88.9798% ± 0.00142%
overhead	0.9241% ±0.0037%	1.0602% ±0.0031%	1.1265% ±0.0024%	1.0246% ± 0.0029%

core amongst multiple RT and BE containers when increasing the number of RT containers from 1 up to 7. Each RT container has a RT demand of 10% of the CPU bandwidth (RT period = 100ms, RT budget = 10ms). The experiments indicate that hierarchically scheduled containers using the Hierarchical Scheduling Patch [18] can keep the allocated CPU resource even when competing with BE containers under heavy load. Moreover, the RT containers keep their reserved resource allocation (CPU budget over the periods) with very low run-time jitter on a single core. The system overhead does not influence the RT containers but reduces the remaining CPU utilization used by BE containers.

The experiments indicate that RT containers maintain the target resource reservation even in the presence of heavy RT and BE load. The overhead (indicated in red in Fig. 8.4) remains relatively constant when increasing the number of containers scheduled on the same node.

8.7 Conclusion

In this paper, we have introduced a container orchestration for RT systems designed to prevent over-reservation of the CPU at the container scheduling level. Additionally, we considered the weak isolation inherent in container-based virtualization (e.g., the effects of the use of shared resources or context switches), which can lead to an interference and thereby harm temporal guarantees of RT containers. We have proposed metrics for measuring the RT performance at the container and node levels, which can be used for both the admission control and the online re-configuration of container deployment in order to guarantee timely behavior. We have implemented a scheduler extension and node monitoring system on top of an orchestrating system Kubernetes and have shown the feasibility of co-locating RT and BE containers on the same node in a series of experiments.

We aim to address the optimization problem arising from the orchestration needs in RT containerized systems in future work. This orchestration includes both the admission/allocation of new container requests as well as the online resource re-dimensioning of already deployed RT containers in case of run-time performance drops.

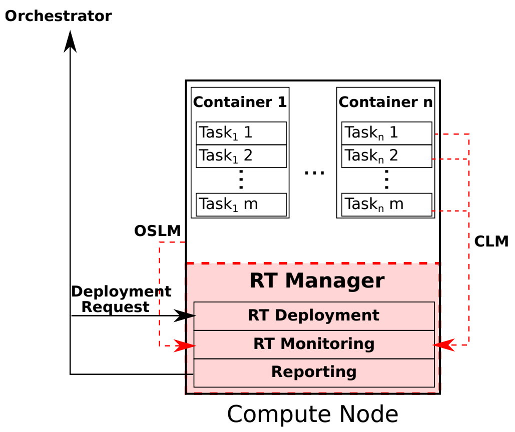


Figure 8.1. Compute node.

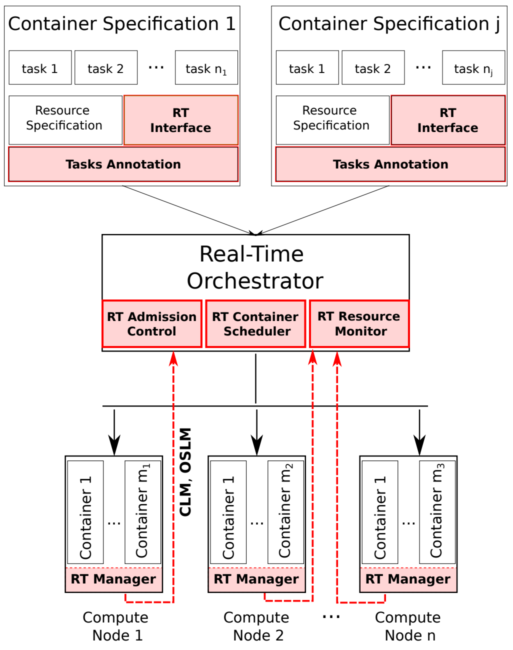


Figure 8.2. A high-level container orchestration system architecture enhanced with RT capabilities.

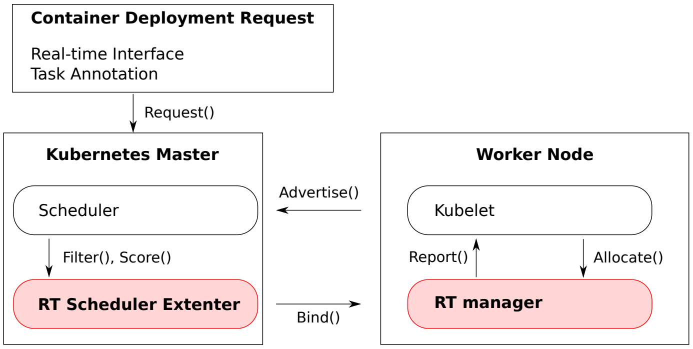


Figure 8.3. Scheduling process of Kubernetes.

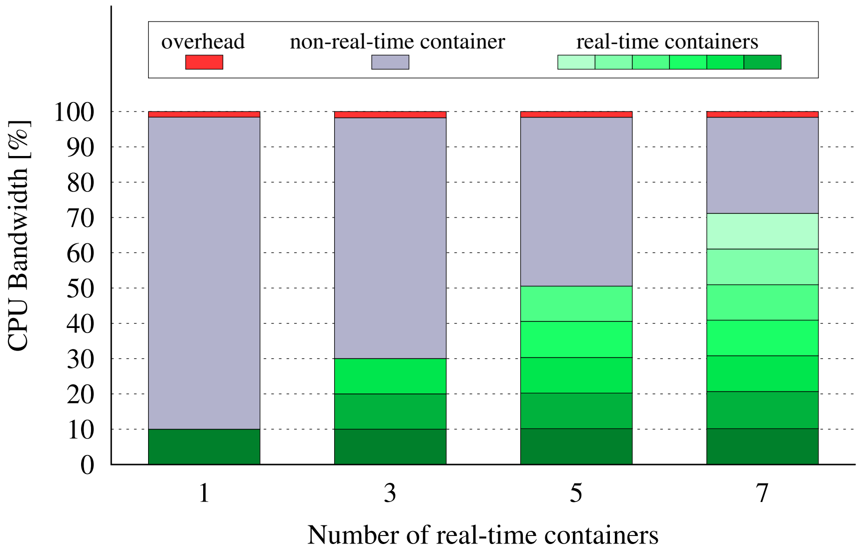


Figure 8.4. Distribution of CPU in a multi-container environment.

Bibliography

[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012.

[2] Shaik Mohammed Salman, Václav Struhár, Alessandro V. Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: Research challenges. In *W. on Fog Comp. and IoT (Fog-IoT)*, 2019.

[3] M. S. Shaik, V. Struhár, Z. Bakhshi, V. L. Dao, N. Desai, A. V. Papadopoulos, T. Nolte, V. Karagiannis, S. Schulte, A. Venito, and G. Fohler. Enabling fog-based industrial robotics systems. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020.

[4] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[6] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[7] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Virtual network functions as real-time containers in private

clouds. In *IEEE Int. Conf. on Cloud Comp. (CLOUD)*, 2018.

[8] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. In *IEEE Int. Conf. on Edge Comp. (EDGE)*, 2019.

[9] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V Papadopoulos. Real-time containers: A survey. In *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[10] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 2019.

[11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd edition, 2011.

[12] Chang Zhao, Yusen Wu, Zujie Ren, Weisong Shi, Yongjian Ren, and Jian Wan. Quantifying the isolation characteristics in container environments. In *Net. and Par. Comp. (NPC)*, 2017.

[13] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. F. D. Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2015.

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973.

[15] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in Xen. In *ACM Int. Conf. on Emb. Soft. (EMSOFT)*, 2011.

[16] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *IEEE Int. Conf. on Adv. Inform. Netw. and Appl. (AINA)*, 2012.

[17] Thuy Linh Nguyen and Adrien Lebre. Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time. Research Report RR-9221, INRIA, 2018.

[18] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 2019.

[19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software*, 2015.

[20] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai*, 2004.

[21] Timur Tasci, Jan Melcher, and Alexander Verl. A container-based architecture for real-time control applications. In *IEEE Int. Conf. on Eng., Tech. and Innov. (ICE/ITMC)*, 2018.

[22] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to IAAS. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[23] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. OS-level virtualization for industrial automation systems: are we there yet? In *ACM Symp. on Applied Computing (SAC)*, 2016.

[24] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. of Syst. Architecture*, 2018.

[25] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[26] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, 2009.

[27] Sisu Xi, Chong Li, Chenyang Lu, Christopher D Gill, Meng Xu, Linh TX Phan, Insup Lee, and Oleg Sokolsky. Rt-open stack: Cpu resource management for real-time cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015.

[28] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *IEEE Real-Time Syst. Symp. (RTSS)*, 2003.

[29] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *IEEE Int. Conf. on Emerging Tech. and Factory Aut. (ETFA)*, 2017.

[30] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of Linux. In *IEEE Real-Time and Emb. Tech. and Appl. Symp. (RTAS)*, 2002.

[31] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *ACM/SPEC on Int. Conf. on Perf. Eng. (ICPE)*, 2017.

[32] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2008.

[33] Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Syst.*, 2015.

[34] Alessandro Vittorio Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems. In *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2018.

Chapter 9

Paper D: DART: Dynamic Bandwidth Distribution Framework for Virtualized Software Defined Networks

Václav Struhár, Mohammad Ashjaei, Moris Behnam, Silviu S. Craciunas, and Alessandro V. Papadopoulos

In the 45th Annual Conference of the IEEE Industrial Electronics Society (IECON 2019).

Abstract

In this paper we address a network architecture that uses a combination of network virtualization and software defined networking in order to reduce complexity of network management and at the same time support high quality of service. Within this network architecture, we propose a framework to be able to dynamically distribute the network bandwidth to various services such that the network resources are utilized efficiently. In many industrial domains, multiple services may use the same hardware platform for the sake of a better resource utilization. Therefore, bandwidth distribution among the services should be done in an efficient way during run-time. We also develop an admission control in this framework which dynamically coordinates the bandwidth distributions based on requested quality of services. We show the applicability of the proposed framework by implementing it on a common SDN controller. Moreover, we conduct a set of experiments to show the performance of the proposed framework.

9.1 Introduction

With the advent of IIoT, where manufacturing processes are monitored and supported by a tremendous number of devices, the need for flexible and efficient resource management in industrial networks is gaining its importance [1]. The emergence of IIoT brings intensification of resource sharing in physical networks, introduce new challenges in flexible network reconfiguration and challenges in providing various QoS levels in a physical network. To address these issues, both industry and research community are considering a combination of two emerging technologies, which are network virtualization [2] and SDN [3]. Network virtualization finds its roots in computing virtualization mechanisms where multiple virtual machines are running on a same hardware platform. Through network virtualization a physical network is partitioned into several logical networks, known as *slices*, which are isolated and managed separately. Moreover, SDN on top of the network virtualization provides an architecture in which the slices are managed via a centralized point without knowing the underlying physical network details. We use the term *virtualized SDN* for the described architecture throughout this paper. In this architecture, each network slice is managed by an SDN controller, and commonly the SDN controllers have different requirements when coordinating their associated slices.

Motivation. Commonly, in IIoT applications resources are limited, both in computation and in communication resources. Therefore, in the context of communication resources, several IIoT devices share a physical network to communicate. Although, the virtualized SDN architecture provides means in managing network resources, not much attention has been paid in supporting dynamicity of resource utilization which is a prominent factor in IIoT applications. Several works have proposed similar architectures focusing on QoS provisioning [4], network timing properties in RT communication [5] and creating network slices according to application requests [6]. On the other hand, there are very few works addressing a fully dynamic network resource allocation in industrial systems. In this paper, we only focus on the network bandwidth as the resources could be energy or other constraints. For instance, the proposal in [7] attempts to develop an admission control in an SDN controller in order to bound the network bandwidth utilization for multiple network services. Nevertheless, the proposal is limited to a small network architecture without any `smart` decision making algorithms for efficient bandwidth allocation.

Contributions. In this paper, we propose a framework, which we name it *dynamic bandwidth distribution (DART)*, based on a virtualized SDN architecture which makes the fully dynamic bandwidth allocation on a physical network feasible. Moreover, we propose an admission control mechanism to distribute the network bandwidth during run-time based on the QoS level requested by the IIoT devices. The admission control mechanism resides within the proposed framework. We also show the applicability of the proposed framework on a use case study where the proposed admission control mechanism is implemented within a well-known SDN controller. Finally, we conduct a set of experiments to present the performance of the implemented framework and mechanism.

Organization. The rest of the paper is organized as follows. Section 9.2 briefly describes the background and related work. Section 9.3 presents DART framework and the bandwidth distribution mechanism. Section 9.4 presents the use case, while Section 9.5 shows a set of experiments. Finally, Section 9.6 concludes the paper with future directions.

9.2 Background and Related Work

In this section we introduce the basic concepts of SDN and network virtualization as well as a survey in the area of dynamic bandwidth management.

9.2.1 Software Defined Network

SDN helps to decrease the complexity of network management by decoupling network control and forwarding functions [8]. Network control is handled in a centralized manner by an SDN controller that has a complete knowledge of the network. SDN is comprised of three layers (Fig. 9.1): a) The *Application layer* consists of SDN business applications written in common languages controlling the underlying SDN enabled devices via the SDN controller, b) The *Control layer* fetches various statistics from the physical devices (usage statistic, topology details, state details) and enables communication between SDN applications and SDN devices, and c) The *Infrastructure layer* is composed of physical

SDN switches. The OpenFlow [9, 10] protocol enables communication between SDN controllers and network devices. The aim of the OpenFlow protocol is to overcome the proprietary systems of network hardware vendors and create a set of communication instructions for the interconnection of multiple vendors devices.

SDN switches contain hierarchically-chained flow tables defining rules and actions for handling incoming network traffic by SDN controllers. Flow tables are comprised of the following fields [10]:

- **Match fields**: The match fields consist of ingress ports, packet header fields, Virtual LAN (VLAN), priority and metadata.
- **Actions**: Actions to be performed for the matched data frame (e.g., forward data frame to a predefined port, drop the frame, send the frame to the controller). The actions can be chained in more complex actions.
- **Counter**: Statistic for matching data frames including count of data frames and their total sizes.
- **Priority**: Used if the incoming frame satisfies multiple match fields.

Every time a frame is received by an SDN enabled switch, the frame header is extracted and matched with records in local flow tables. If the match is found, the corresponding action is triggered. Otherwise, a copy of the frame is forwarded to the SDN controller that decides an action for the incoming frame. The action together with the matching rule is returned to the switch that stores in the flow table.

9.2.2 Network Virtualization

The need for service isolation and diverse resource requirements within one physical network brings the topic of network virtualization into the focus of the researcher community [11, 12]. Network virtualization enables the coexistence of multiple logical networks sharing the same underlying physical network [2]. One technique in this context is network slicing [13] where there is a division of the shared physical network into multiple logical isolated sub-networks (slices). Besides being isolated from each other, slices may be optimized for

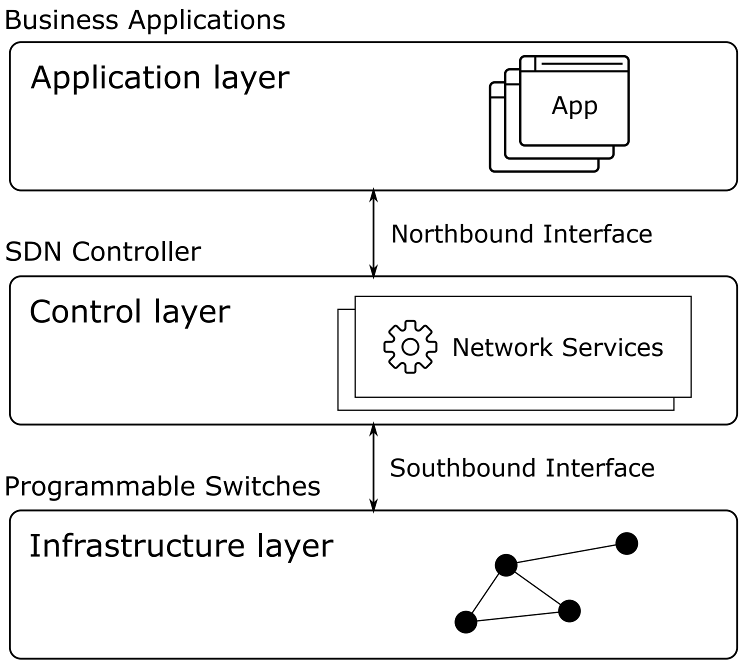


Figure 9.1. SDN Architecture separates a network into tree layers: Application, Control and Infrastructure layer.

different purposes (e.g., high bandwidth HD video streaming, low latency video gaming) [13]. Slicing allows infrastructure providers to adapt the sharing of the underlying physical network to customer requirements while at the same time providing isolation of the network resources. Network virtualization requires a Network Hypervisor which creates an abstraction layer on the top of physical hardware and allows the creation of virtual networks.

9.2.3 Bandwidth management

There are several resource reservation techniques in processor and communication domains which most of them focus on static reservation of bandwidth. For instance, in the processor domain, supporting multimedia applications [14] and hierarchical reservation techniques [15] are presented, whereas in the distributed level communication bandwidth reservation for multimedia systems [16], adaptive QoS control [17] and D-RES platform to support end-to-end timing [18] are presented.

In the context of using SDN architectures, Seokhong et al. [6] extended FlowVisor, as the network hypervisor, to guarantee the bandwidth requirements with an admission control and traffic scheduling. Moreover, Tomovic et al. [19] present a new SDN/OpenFlow control environment for dynamic adjustment based on Quality of Service (QoS) provisioning. In this solution, a centralized QoS SDN control system monitors the state of the network and automatically manages and configures network devices to provide the required QoS level for multimedia applications. There are several works, like HyperFlow [20], Onix [21], Kandoo [22], and devolved controllers [23], that use multiple SDN controllers on a single physical network (or a slice), either in a distributed or layered approach, where the orchestration goal between controllers is on controller redundancy and load balancing. The main focus is therefore to address the challenges that centralized SDN architectures introduce (c.f. [24]) rather than address network management. The closest work to this paper is a resource management solution for virtualized SDN networks in which each slice is governed by an SDN controller in terms of admission control for incoming traffic [7]. However, the dynamicity was limited to static parameters and the SDN controllers are not communicating for a better decision on bandwidth allocation. Nevertheless, the mentioned works either focused on static bandwidth reservation or they proposed a level of dynamicity in bandwidth management with different goal than reducing resource utilization, which is the primary objective of DART.

9.3 DART: Dynamic Bandwidth Distribution Framework

This section presents the concept of DART framework as well as the proposed admission control mechanism. The framework defines an overall concept to enable dynamic bandwidth management of networks, while the admission control mechanism, as part of the framework, facilitates the functionality of dynamic bandwidth redistribution.

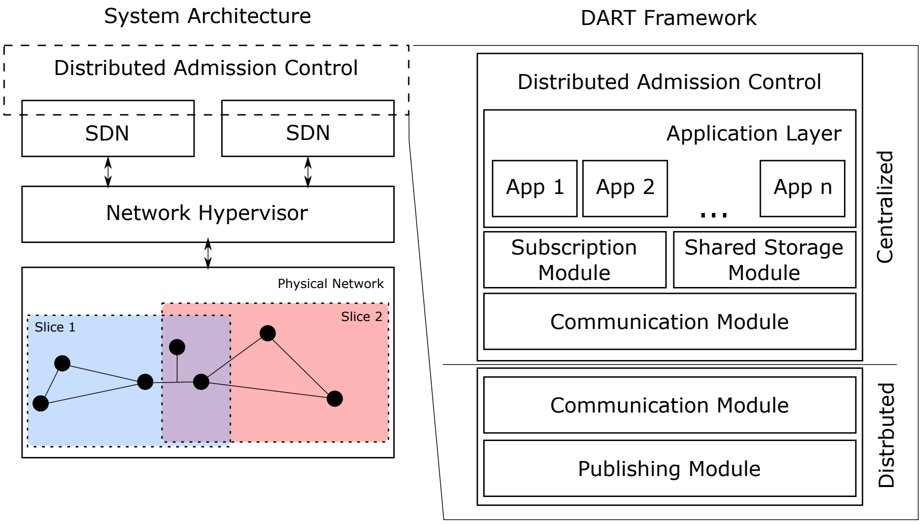


Figure 9.2. An architecture of a system using DART framework.

9.3.1 The DART Framework

The DART framework is a generic concept that can be applied on any virtualized SDN architecture. Fig. 9.2 depicts the DART framework on a virtualized SDN architecture. On the left side of the figure, an architecture with several slices in a physical network is shown. Note that in this architecture we assume that the slices can share a part of the physical network to increase the efficiency of utilizing the resources, however the framework covers the cases with fully isolated slices as well. The proposed framework is depicted on the right side of the architecture, which consists of two main components: a distributed component and a centralized component. Following we describe the components in details.

The distributed component deals with synchronizing the bandwidth management among the network slices. As each SDN controller can only coordinate its own slice, it is essential to have a general view of the network status when allocating the bandwidth. The distributed component ensures that the SDN controllers collaborate on bandwidth management, leading to a coherent bandwidth utilization. This component contains two main modules, which are communication and publishing modules. The communication module is responsible to send and receive information from and to the centralized component. The publishing module is the counterpart of the subscription module resided in the centralized component. Moreover, the coordination between SDN controllers are done using this component.

Another component in the framework is the centralized component which is responsible to coordinate the decision making for bandwidth allocation over the entire network. In case the changes in the bandwidth is requesting locally in one slice, the centralized component will decide on the allocation. However, if there is a shared part of the network that requires a change, SDN controllers need to negotiate on the bandwidth allocation from their slices. The centralized component is then responsible to advice the best possible bandwidth.

The admission control mechanism in this framework can be activated by any signal from various sources, including the load, priority and packet loss ratio, to start redistributing the network bandwidth. In this paper, as an initial phase, we specify priorities for the traffic to be sources of initiating the redistribution. Each IIoT device can transmit traffic with various priority levels depending on the QoS level it requires. Based on the traffic priority, the device can request for higher bandwidth during run-time. The details of this mechanism is presented in the following section.

9.3.2 Admission Control Mechanism

Following the DART framework, the admission control mechanism is divided into the centralized and distributed components. The centralized component contains the logic of the bandwidth management. The distributed component, however, resides on top of the SDN controller to provide information about the corresponding slices to the centralized component.

The sending nodes decompose the data into multiple data streams (see Fig. 9.3). The data streams can have different priorities which are set by the sender nodes depending on the importance of the data. Note that we consider only eight priority levels accepted by Ethernet frame. The primary goal of the admission control is to check the priority of the data streams and allocate bandwidth for the links that the data stream is transmitting. In order to do that

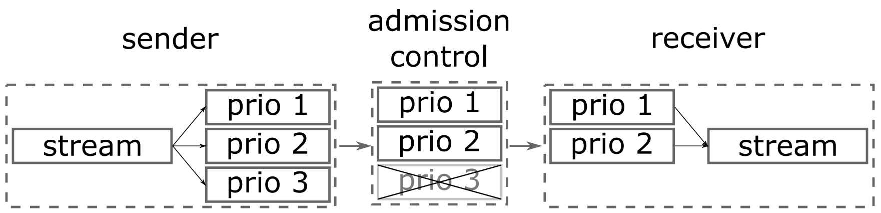


Figure 9.3. Admission Control Mechanism.

the admission control defines a priority limit. Any received data stream with priority higher than the priority limit will be forwarded to its destination, whereas the data streams with priority less than the priority limit will be prevented for transmission. The priority limit is defined in the centralized component of the admission control for all slices. In a normal case, priority limits are equal and thus the bandwidth is uniformly distributed among the slices. If there is a request for priority limit change (detected by a distributed component), the centralized component adjusts the limits accordingly in order to a) increase bandwidth in the requesting slice, b) keep the total bandwidth used by all slices constant.

9.4 Use Case: Surveillance System

To analyse the feasibility of the cooperation between the SDN controllers and to show the application of the DART framework, we present a use case of surveillance system in which the bandwidth is cooperatively adjusted based on traffic priorities. We present a factory-wide surveillance system (see Fig. 9.4), where the network is partitioned into several slices in order to keep virtual network domains separated (e.g., shop floor, administrative offices) and to allow to differentiate QoS levels across the slices. However, some of the services needed in both slices overlaps and must be available for both slices simultaneously. In the surveillance system, devices (senders) with attached cameras are transmitting video streams to a centralized location (receiver) where the video streams are processed and monitored by security guards in RT. Senders have limited processing capacity that allows to perform simple motion detection algorithms. Based on the result they are able to ask for higher bandwidth by

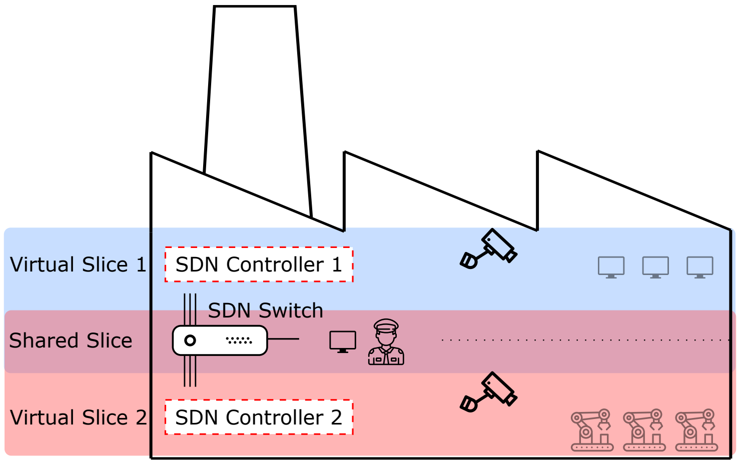


Figure 9.4. The use case of a factory-wide surveillance system.

increasing priority of the traffic. SDN controllers monitor the corresponding virtual network slices and allow only traffic of certain priority levels to pass.

We have simplified the use case and implemented it as shown in Fig. 9.5. We consider three nodes. Two of them are sending nodes that are transmitting a video stream to the receiving node. The sending nodes are separated into two virtual network slices, these devices have cameras connected and periodically send sequences of images to the receiving node that stores and analyses the images. The receiving node resides in shared network slice and thus provides services for both of the slices.

The bandwidth in the shared slices is limited. Thus, there must be a traffic control in order to prevent network congestion in the shared slice.

9.4.1 System Setup

For the implementation, the following hardware and software components are used (see Fig. 9.5):

- **Raspberry Pi (RPi) devices**: One RPi located on shared network slice acts as a data receiver, the rest of the RPi devices are separated on different network slices used as data senders. The data senders are connected to RPi cameras.

- **Aruba SDN enabled switch**: Three ports are used for connecting RPi devices, the fourth port is used for connection to the laptop that provides network hypervisor (FlowVisor) and SDN controllers (FloodLight).
- **FlowVisor**[1]: FlowVisor is an OpenFlow controller serving as a transparent proxy between OpenFlow switches and SDN Controllers. It creates rich network slices defined by any combination of switch ports, src/dst ethernet address or type, src/dst IP address or type, and src/dst TCP/UDP port or ICMP code/type.
- **FloodLight**: FloodLight is an Java based open-source SDN controller offering a modular architecture that allows to extend its functionality with custom tailored applications.

9.4.2 System Implementation

The system requires implementation of four components: sending node, centralized bandwidth controller, SDN controller module and receiving node. With respect to DART, the bandwidth controller acts as a centralized component used for sharing states between the SDN controllers, and the SDN controller module represents the distributed part of the framework. The scheme of the implementation of the SDN controller module and the bandwidth controller is depicted in Figure 9.6.

The sending nodes are detecting motion in the video stream and transmitting video frames with corresponding priorities. The SDN controller modules are detecting priority changes in received data, reporting it to the centralized bandwidth controller, that decides the priority thresholds for corresponding SDN controllers in order to deliver higher importance video frames (with detected motion) with higher data rate to the receiving node.

[1]`https://github.com/OPENNETWORKINGLAB/flowvisor/wiki`

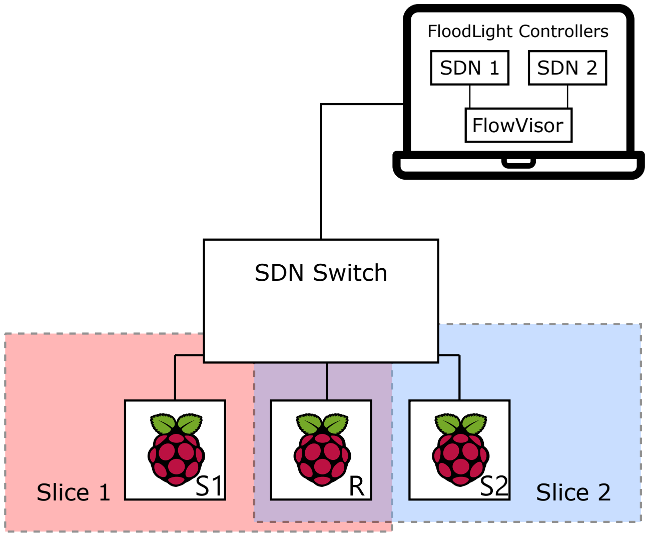


Figure 9.5. The experimental setup.

Sending nodes The sending nodes are continuously obtaining video frames, detecting movement, fragmenting the video frames into smaller segments and sending the segments to the receiving node at a constant rate. Based on the detected movement, the sender changes priorities of sent frames. The following steps are taken:

- Motion detection: The implemented motion detection algorithm compares a current video frame with the previous one, measures the amount of changed pixels and compares it with a pre-set threshold. If a motion is detected, the priorities in the sent data are increased.
- Video frame fragmentation: Due to payload limitation of IEEE 802.1Q Ethernet Frames (1500 Bytes), the video frame has to be fragmented into several frames. Payload of each frame contains a header (id of video frame, sequence of the the fragment, total) and video frame fragment data.
- Traffic differentiation: To differentiate the traffic to help SDN controllers to filter the data, the frames are uniformly distributed into data streams having

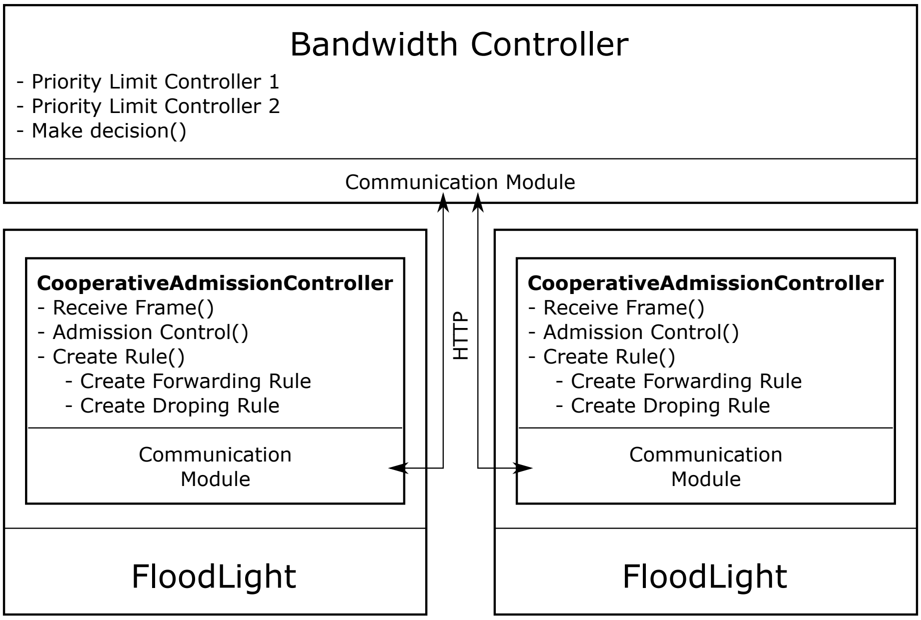


Figure 9.6. Overview of the collaborative admission control using DART framework.

priorities (0-5). If a video frame is detected, the sender increases the priority level of the traffic to (1-6).

SDN control The FloodLight SDN controller is extended by a Java module that is governing incoming video frame streams. Based on priorities of the incoming data and limits imposed by the shared bandwidth controller, the data is either accepted and a forwarding rule to the shared slice is created, or the data transmission for particular priority level is blocked. The module performs the following:

- Receiving frames and creating rules: The module is receiving Ethernet frames from the SDN switch that needs actions to be resolved. Based on frame fields and custom data header, the the action the frame (MAC address of the sender, MAC address of the receiver, VLAN, priority) is established. There are two actions that the module can take: a) Create forwarding rule:

Forward matching packets to receiving node, b) Create dropping rule: Drop matching packets.

- Collaborative admission control: The controller filters data frames by creating flow rules based with cooperation with the bandwidth controller.

Bandwidth controller The bandwidth controller is implemented as a Java based application that provides functionality for a centralized bandwidth control, it provides REST API to enable communication between the bandwidth controller and the components distributed among SDN controllers. The application keeps track of the traffic in virtual networks and assigns priorities limits to distributed components residing on the top of the SDN controllers. The priority limits are assigned according to the Table 9.1. The priority limits (S1 and S2 priority) are changed based on detected movement (S1 and S2 state).

Receiving node The receiving node, located on the shared network slices, is collecting video frame fragments from the multiple slices, verifying data consistency and merging video frame parts in order to reconstruct the original images.

9.5 Experimental results

The purpose of the experiment is to show the behavior of the dynamic bandwidth allocation in sliced networks by a set of SDN controllers that are communicating through shared entity (the bandwidth controller). We consider transmissions

Table 9.1. Priority limits.

S1 state	S2 state	S1 priority	S2 priority
no movement	no movement	4	4
movement	no movement	6	2
no movement	movement	2	6
movement	movement	4	4

from each slice with different importance that is changing dynamically during the experiment, in an event-based fashion. We assume that the traffic triggered by specific events, e.g., motion detection or alerts, are high priority traffic.

The sending nodes are transmitting sequence of video frames at a constant rate to the receiving node. The video frames have been prerecorded. The network hypervisor and two FloodLight controllers together with the shared bandwidth controller application run in a single computer, thus the network communication overhead between these entities is negligible. The node on a shared slice is receiving and reconstructing data. Also, the receiving node is recording all the arriving frames. The size and the sender of the frame is known, thus the network utilization can be reconstructed. The parameters of the system are the following: Image size $\sim$ 40kB (depends on the scene, the image sizes may differ slightly).

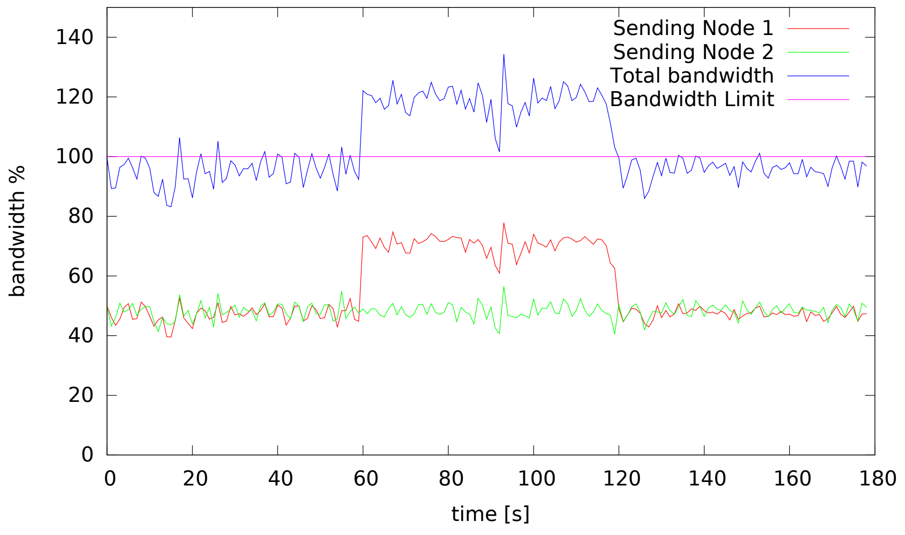
Fig. 9.7 shows the average network utilization over 20 runs of the same experiment, in the case of no bandwidth adaptation (Fig. 9.7a), and with bandwidth adaptation (Fig. 9.7b). In both cases, between time 60 and time 120, a motion is detected, and additional traffic is generated from the sending node 1. In Fig. 9.7a, the bandwidth limit is exceeded for all the period when the motion is present, while Fig. 9.7b shows that the SDN controller 1 detects high importance traffic, and the system increases the priorities for slice 1 and decreases the priorities allowed for slice 2, resulting in a better allocation of the bandwidth over the high-priority traffic. The reaction time from the frame reception by the SDN controller, decision making by the bandwidth controller to the alteration of the rules in SDN switch is 60ms. The communication time between the SDN controller and bandwidth controller is 3ms.

9.6 Conclusion and Future Work

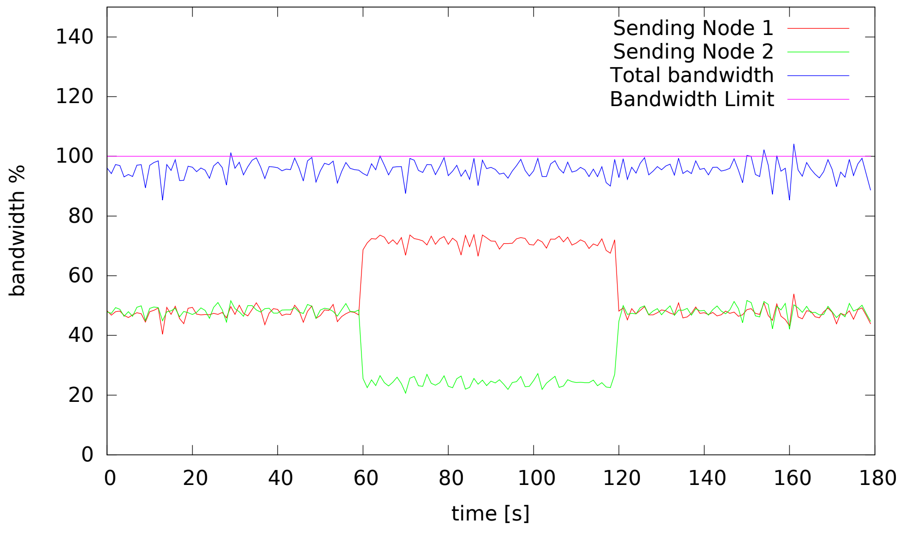
Connection between network virtualization and Software Defined Networking plays an important role in Industrial Internet of Things due to the need for domain separation, provision of various levels of QoS in a single physical network and ability of dynamic network reconfiguration. Shared segments of virtualized networks can be used for sharing services and resources among separated segments. However, in order for efficient usage and utilization of shared resources, a collaborative approach must be set.

In this work, we introduced the DART Framework that enables collaboration multiple SDN controllers among virtualized networks. Subsequently, we implemented an use case of a surveillance system that utilizes the Framework. he results shows that SDN controllers can cooperatively take decisions and prioritize and distribute the bandwidth between slices to mitigate a congestion of shared resources. The Framework introduced here does not have to be restricted to bandwidth distribution only but it can be extended to support numerous application that will benefit from the inter-slice collaboration, e.g.: distributed access control lists, firewall and traffic scheduling.

The DART Framework opens the door for a design of smart algorithms for dynamic reconfiguration of the network that can utilize proactive monitoring of the flow/port usages in the separated virtual network.



(a) Without bandwidth adaptation.



(b) With bandwidth adaptation.

Figure 9.7. Average bandwidth utilization over 20 runs.

Bibliography

[1] E. Sisinni et al. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Trans. Ind. Inf.*, 2018.

[2] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 2010.

[3] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Comm. Mag.*, 2013.

[4] Murat Karakus and Arjan Durresi. Quality of service (QoS) in software defined networking (SDN). *Journal of Net. and Comp. Appl.*, 2017.

[5] K. Ahmed, J. O. Blech, M. A. Gregory, and H. Schmidt. Software defined networking for communication and control of cyber-physical systems. In *ICPADS*, 2015.

[6] Seokhong Min et al. Implementation of an OpenFlow network virtualization for multi-controller environment. In *ICACT*, 2012.

[7] S. Aglianò et al. Resource management and control in virtualized SDN networks. In *RTEST*, 2018.

[8] B. A. A. Nunes et al. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Comm. Surv. Tut.*, 2014.

[9] Nick McKeown et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[10] ONF. OpenFlow switch specification. Technical report, Open Networking Foundation, 2015.

[11] L. Xingtao et al. Network virtualization by using software-defined networking controller based Docker. In *ITNEC*, 2016.

[12] A. Blenk et al. Pairing SDN with network virtualization: The network hypervisor placement problem. In *NFV-SDN*, 2015.

[13] Rob Sherwood et al. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[14] Xiang (Alex) Feng. Towards real-time enabled microsoft windows. In *The 5th ACM International Conference on Embedded Software*, 2005.

[15] Saowanee Saewong et al. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, 2002.

[16] Michal Sojka et al. Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Sys. Arch.*, 2011.

[17] T. Cucinotta and L. Palopoli. QoS control for pipelines of tasks using multiple resources. *IEEE Trans. Computers*, 2010.

[18] A. B. Oliveira, A. Azim, S. Fischmeister, R. Marau, and L. Almeida. D-RES: Correct transitive distributed service sharing. In *IEEE Emerging Technology and Factory Automation*, 2014.

[19] S. Tomovic, N. Prasad, and I. Radusinovic. SDN control framework for QoS provisioning. In *TELFOR*, 2014.

[20] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *INM/WREN*, 2010.

[21] Teemu Koponen et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[22] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *HotSDN*, 2012.

[23] A. S. . Tam, Kang Xi, and H. J. Chao. Use of devolved controllers in data center networks. In *INFOCOM WKSHPS*, 2011.

[24] Akram Hakiri et al. Software-defined networking: Challenges and research opportunities for future internet. *Comp. Net.*, 75, 2014.