RTilience: Fault-Tolerant Time-Critical Kubernetes

Harald Gustafsson*, Fredrik Svensson*, Raquel Mini*, Luca Abeni†, Remo Andreoli†, Tommaso Cucinotta†
*Ericsson Research, Lund, Sweden, *e-mail*: {first.last}@ericsson.com
†Scuola Superiore Sant'Anna, Pisa, Italy, *e-mail*: {first.last}@santannapisa.it

***Abstract*—This paper tackles the problem of optimal configuration and deployment of fault-tolerant time-critical service chains with arbitrary DAG-alike topologies. We propose RTilience, designed according to a scalable cloud microservice paradigm, and prototyped on top of the well-known Kubernetes cloud orchestrator. It features real-time reservation scheduling of containers to guarantee temporal isolation of time-critical tasks, leading to fine-grained control of compute latencies, while allowing for sharing physical CPUs among containers. A distributed routing library, ReqRoute, is configured with a timeout and primary and secondary routes, enabling autonomous and decentralized handling of failing requests. The routes are configured by a centralized controller that performs admission control, resource management of microservice instances, task placement, and fault detection and recovery, extending the features available in Kubernetes. Admission control is based on a theoretical framework enclosing a worst-case performance model for the experienced end-to-end response-time under various fault handling options, and an optimization framework that computes the optimum resource allocation for admitted services. Extensive experimentation of the proposed solution has been performed with synthetic examples, and an autonomous transport robot use-case, verifying that end-to-end deadlines are effectively respected, even in presence of high fault rates of individual microservice instances, according to the theoretical expectations. RTilience is made available as open-source software, released under a MIT license.**

***Index Terms*—Fault-Tolerance, Real-Time Systems, Resource Management and Optimization, Cloud Computing, Network Function Virtualization, Containers, Kubernetes**

I. Introduction

Cloud computing has emerged as a key technology for the growth and development of modern distributed applications and services, thanks to the tremendous benefits it offers to a wide spectrum of application domains [1]. Cloud computing principles are also being adopted in the Telco industry, where we are witnessing an increasing adoption of Network Function Virtualization (NFV) [2]. With this paradigm, network operators are shifting away from physical appliances sized for peak-hour operations. Instead, network functions are deployed as elastic software components (Virtual Network Functions – VNFs) able to dynamically scale out and back-in as needed, tracking the instantaneous workload variability over time. These are deployed as service chains composed of a multitude of virtual machines (VMs) or containers onto a flexible NFV infrastructure managed using a Virtual Infrastructure Manager (VIM), often implemented through a cloud orchestrator.

In distributed cloud and NFV infrastructures, fault tolerance is paramount, as complex computing networks are prone to failures given the amount of interconnected components [3]. Indeed, with millions of geo-distributed virtual resources hosted on commodity hardware, even the slightest probability of failure becomes realistic. Moreover, the dynamic and multi-tenant nature of cloud services reduces control on the hosted workloads, making it hard to identify the root cause of failures.

In this context, cloud services have evolved towards fully native, microservice-based architectures, where end-to-end response times are becoming increasingly important. Therefore, more and more attention of research and industry is focusing on the ability to host *fault-tolerant time-critical services* in geo-distributed cloud/edge infrastructures [4], [5], [6]. Affected application domains include, for example, time-critical VNFs in NFV, e.g., vRAN, factory automation use-cases within Industry 4.0, or self-driving/autonomous vehicles scenarios where cloud-enhanced functionality need to be provided within tight and predictable response times. These applications all need non-trivial solutions for fault-tolerance and timeliness, so that tight end-to-end response times can be guaranteed despite failures that may occur throughout the infrastructure.

Providing reliable real-time cloud services through a networked infrastructure is not easy. Traditional QoS-aware networking techniques [7], [8], as well as novel technologies such as 5G and Time-Sensitive Networking [9], are crucial to facilitate deterministic and low-latency communications. The physical hosts themselves are another source of unpredictability due to the contention of shared resources, which is typical of multi-tenant architectures: cores, caches, memory controllers, and buses are often contended in unpredictable ways. These problems may be mitigated by using mechanisms at the operating system (OS) kernel level to provide temporal isolation among workloads co-located on shared servers, thus reducing their interferences, as proposed by various studies [10], [11], [12].

Contrary to conventional web and time-sensitive applications, time-critical ones require a high level of availability and *reliability*, as well as precise latency guarantees, sometimes in the order of milliseconds [13]. Such strict requirements must be met regardless of the other workloads deployed within the cloud infrastructure or the possible occurrence of hardware/software/network failures [14]. Typical interactive multimedia cloud services, such as video-call platforms, are not seriously impacted in the event of temporary slowdowns. On the other hand, emerging time-critical use cases like a vision system for intersection safety in smart cities, or a cloud-enhanced controller for robotic actuators, require cloud/edge-based services to be constantly up, and replying within precise response times. The inability to respect temporal constraints may have critical, if not catastrophic, consequences.

A cloud provider may host time-critical services on dedicated, single-tenant, physical resources, through appropriate interfaces [15], [16]. However, this most likely results in a

waste of resources and money for both the provider and the customer, if not fully utilized, especially considering the recent efforts towards reducing energy consumption in cloud datacenters [17]. Hence, there is a need to take full advantage of multi-tenancy and shared resources to minimize costs and resource waste while ensuring strict performance guarantees.

A. Paper Contributions

In this paper, we tackle the challenge of *optimum deployment of fault-tolerant time-critical services* in a cloud infrastructure. We propose RTilience, an open-source cloud orchestrator for fault-tolerant, distributed service topologies with time-critical end-to-end deadline constraints. The discussion focuses on the design, implementation, and experimental evaluation of RTilience, as prototyped within the well-known Kubernetes open-source cloud orchestrator. The proposed solution includes a number of key components: i) a theoretical optimization model, previously described and evaluated through simulations in [18], to ensure predictable end-to-end response times for the execution of a DAG-alike distributed computation in presence of a maximum number of faults; ii) a hierarchical real-time scheduler for Linux, used to provide precise and fine-grained temporal scheduling guarantees to individual Kubernetes containers [19], guaranteeing temporal isolation among containers while allowing them to share physical CPUs; iii) a Kubernetes admission control component using the above optimization model to compute the best configuration for deploying time-critical services, or updating the configuration of existing ones; and iv) ReqRoute, a middleware to deploy our services which supports fault detection and handling via double forwarding of requests to primary and secondary nodes or request retransmissions upon timeout, as configured by the admission control component.

B. Paper Structure

This paper is structured as follows: Section II provides a comparison with related works found in the literature, then Section III outlines the motivations and a high-level description of RTilience. Section IV describes its core design and provides implementation details. Section V presents extensive results from the experimental evaluation, performed using synthetic scenarios, and a realistic autonomous transport robot use-case. Finally, conclusions are drawn in Section VI, along with a brief discussion of possible future research on the topic.

II. Related Works

The problem of designing real-time and fault-tolerant cloud services received considerable attention in the research literature [4]. Generally, we can distinguish among *reactive* and *proactive* fault tolerance approaches [20], [21], [22], [23]. The former approaches deal with a failure after it occurs, typically by replaying or retrying the task execution on another instance after a fault occurred (i.e., checkpointing and resubmission), or by concurrently running it on multiple instances (i.e., replication). On the other hand, a proactive approach tries to prevent faults to happen, by predicting failures in advance, and taking action beforehand. Several recent contributions leverage machine-learning and artificial intelligence for predicting and classifying system-level metrics from cloud instances [24], by applying preemptive migration [25], reboots or other actions [26]. For the sake of brevity, in the following we focus on reactive approaches, in which our proposed framework falls.

Several authors propose fault tolerance approaches based on replication and task re-submission due to the relatively low latency recovery time and simplicity. While a subset of these contributions consider also latency-sensitive applications, they are rarely effective for time-critical services, because faults manifest not only as conventional infrastructural problems (i.e., transient network issues or persistent hardware failures) but also as deadline misses. To the best of our knowledge, no other related work incorporates all the following features of RTilience: i) support for time-criticality through mechanisms at the bottommost level of the software stack (i.e., OS, Network protocols, I/O subsystem), and specifically real-time reservation-based CPU scheduling; ii) support for complex distributed applications with arbitrary DAG-alike topologies; iii) fault tolerance through a combination of retry and replicated executions, while guaranteeing end-to-end deadlines despite failures, thanks to a sound formal analysis and optimization framework used at admission time; and iv) a practical implementation integrated within an industrial-grade cloud orchestrator like Kubernetes, evaluated using a realistic workload.

Table I summarizes the major characteristics of the works discussed below, highlighting: their suitability for time-critical scenarios; the main proposed fault-tolerance mechanism; their applicability to complex, distributed end-to-end topologies; and whether they were evaluated through a practical implementation and real experimentation, or using simulations only.

FTCloud [27] introduces a component ranking framework to identify significant components (in terms of invocation frequency) in a cloud application and determine their optimal fault tolerance strategy. The proposal considers three variations of the replication technique: recovery block, N-version programming and parallel. The final decision is influenced also by a user-based preference in terms of response time and cost trade-off. However, the component ranking and strategy selection algorithms are simple and do not take into account temporal interferences. Furthermore, there is no information regarding the practical implementation. The same authors propose a variation of the framework to support Byzantine faults [28], a class of failures that appear inconsistently failed and functioning to different observers.

Javed et al. [29] propose a fault-tolerant architecture for the edge-cloud continuum. A federated management system is provided serving as a unified interface to data management and processing in both edge and centralized clouds. The performance of the architecture is extensively evaluated in terms of latency and throughput using a smart building use case. However, faults are handled solely in terms of data replication, with no attempts to react to faulty devices and without considering state changes in the IoT environment.

Mudassar et al. [30] present an adaptive fault-tolerance methodology for latency-oriented Internet-of-Things (IoT) applications. As in our model, an application is represented as a

TABLE I
COMPARISON WITH RELATED WORKS.

	Time-Critical	Fault-Tolerance	Distributed App. Model	Practical Evaluation
[27]	×	Repl. variants	✓	×
[28]	×	Replication	✓	×
[29]	×	Replication	×	✓
[30]	×	Replication, Checkpointing	×	×
[31]	×	Replication	×	✓
[32]	×	Re-submission, Circuit breaker	✓	✓
[33], [34]	∼	Replication	×	✓
[35]	✓	Repl. variant	×	×
[36]	✓	Replication	✓	×
RTilience	✓	Replication, Re-submission	✓	✓

Acronyms: Application (App.), Replication (Repl.)

DAG of tasks. The proposal leverages smart checkpointing and replication to provide a reliable distributed edge network. In particular, the task states are periodically stored as incremental snapshots and replicated on selected edge nodes (based on vicinity). However, the queuing delay is oversimplified and does not take into account end-to-end latencies. Moreover, the approach is evaluated through simulations only.

Zenpeng et al. [36] propose a greedy fault-tolerant scheduling algorithm for microservice-based, deadline-constrained applications. Their heuristic strategy optimizes the execution cost (in terms of billing) while satisfying the deadline and reliability requirements for each task. Fault tolerance is realized through task replication on the most suitable virtual resources. The approach is evaluated only by simulations.

ReLIEF [35] proposes a primary-backup strategy for the placement of real-time tasks on a fog network. The selection process is determined by a Reinforcement Learning (RL) model. Failures are tolerated on communication links and processing units by establishing a balance between communication delay and workload distribution on each fog device. In case of fault, the backup tasks reside on fog nodes where they have enough time to complete within the deadline. ReLIEF is also evaluated through simulations only.

Toka [31] extends Kubernetes to support latency-sensitive pod scheduling in a large-scale edge computing topology. The major contribution is an online scheduler for deploying on-the-fly Pods with latency requirements. Nodes that are unreachable, or whose network delay do not satisfy the latency requirements of the Pod, are filtered out during placement. Reliability is guaranteed by migrating the pods to healthy nodes. However, the work does not explicitly consider real-time applications and deadline constraints, nor does it employ a mechanism to ensure that the end-to-end latency is not disrupted by transient problems nor faults. Moreover, the paper lacks a proper evaluation of their contribution.

In the public cloud industry, fault-tolerance is supported mostly through the use of majority-based data replication, as available to cloud-native applications through a variety of data store services, from traditional relational DBMS to NoSQL ones. A few of these offer response time guarantees, like DynamoDB [33], promising "single-digit millisecond" latency. The guarantee only covers the 99th percentile of response times with small, simple requests (i.e. get or put operation), and when invoked from within the EC2 cloud. Similar guarantees are provided by BigTable [34] for clients within the GCP platform. The quorum-based design in these systems, often using 3 replicas, makes them naturally tolerating single failures in each replication group, whilst predictability guarantees are achieved using SSD drives, coupled with capacity-aware placement techniques. However, these systems: provide relatively weak end-to-end guarantees, with a non-negligible probability of client requests still exhibiting unacceptably high response times; focus on the data storage service only, which is just one brick in the construction of fault-tolerant distributed applications/services; and do not provide native support for DAG-alike applications, leaving developers with a bunch of good hints [37], [38], as to how to realize robust and well-performing cloud-native applications.

Service meshes are popular for enabling fault-tolerance in cloud-native applications, with studies (e.g., Sedghpour et al. [32]) showing that circuit breakers improve latency by failing fast, while retries can increase end-to-end delays. Similarly, tools like the open-source load balancer HAProxy[1] support high availability through connection timeouts, retry limits, and health checks. These tools provide best-effort performance, without the timing and fault-tolerance guarantees provided by RTilience. Hence, these are not directly comparable with our proposed solution.

III. THE FAULT-TOLERANT REAL-TIME CLOUD

The system model considered in this paper and the general architecture of RTilience have been introduced and evaluated by simulation in previous papers [39], [40], [18]. These are quickly recalled and summarized in this section.

The goal of this architecture is to execute a set $\mathcal{A}$ of $n_{\mathcal{A}}$ time-critical applications. Each application $\mathcal{A}^g \in \mathcal{A}$ (with $1 \leq g \leq n_{\mathcal{A}}$) is formally modeled as a Directed Acyclic Graph (DAG) [41], [42] represented as $\mathcal{A}^g \equiv (\Gamma^g, \mathcal{E}^g)$. Γ^g is a set of n_g tasks τ_i^g (with $1 \leq i \leq n_g$), and $\mathcal{E}^g \subseteq \Gamma^g \times \Gamma^g$ is a set of directed edges connecting the tasks to create an acyclic topology. Each task $\tau_i^g \in \Gamma^g$ is a sequential activity that processes some input data to generate output data after, at most, a Worst Case Execution Time (WCET) c_i^g. Each edge $(\tau_i^g, \tau_j^g) \in \mathcal{E}^g$ represents a direct communication between the two tasks: at the end of its computations, τ_i^g sends a message to τ_j^g. Each application has exactly one task without incoming edges (the *input task* τ_1^g) and one task without outgoing edges (the *output task* $\tau_{n_g}^g$, simply referred to as τ_e^g for simplicity).

An application *activates* when its input task τ_1^g receives some data from a client, and the activation finishes when the output task τ_e^g emits its output. A time-critical application $\mathcal{A}^g$ is also characterized by a minimum inter-arrival time P^g (the minimum time between two consecutive DAG activations) and an end-to-end deadline D^g: if τ_1^g is activated at time t, then

[1]More information is available at: https://www.haproxy.org/.

τ_e^g must generate its output before time $t + D^g$, or the DAG activation is considered failed (i.e., a deadline is missed).

For a cloud-native application $\mathcal{A}^g$, every task $\tau_i^g \in \Gamma^g$ is implemented by a microservice $S_l \in \mathcal{S}$ hosted on the cloud infrastructure. Tasks from different applications might share the same microservice, therefore function $\varphi^g : \Gamma^g \to \mathcal{S}$ maps each application task $\tau_i^g \in \Gamma^g$ to the microservice $S_l \in \mathcal{S}$ implementing it. In turn, a microservice S_l is composed of m_l workers/instances (e.g., typically containers or VMs) over which requests for the microservice are distributed. Notice that the WCET c_i^g of task τ_i^g corresponds to the WCET c_l of the microservice $S_l = \varphi^g(\tau_i^g)$ implementing the task.

The goal of RTilience is to be resilient to transient faults of the containers implementing the microservices. For this reason, the system must be able to detect transient faults timely and react to them so that the served time-critical application can tolerate the faults. The model achieves fault-tolerance through *replication* and *re-execution* by assigning *partial deadlines* to the tasks. Our analytical approach determines whether there is sufficient time to re-execute a task on a different instance of the same microservice, in the event of a partial deadline miss. If this is not possible, meaning that the re-execution results in an end-to-end deadline miss for the DAG activation, then the task is said to be *critical*, and it is simultaneously replicated on two different instances. This way, if one of them fails, the other one still completes on time. The model is able to guarantee that the end-to-end deadline D^g is respected in spite of F faults occurring across different microservices involved in a single DAG activation. The guarantee holds assuming that, for each microservice, one of the two invocations, primary or secondary, is successful, i.e., we neglect the probability that both the primary and secondary instances within the *same* microservice fail at the same time (this is achievable deploying them in fault-independent locations, e.g., different availability zones).

Partial deadlines can be computed and assigned in various ways. Our algorithm [18] computes a series of *Worst Case Response Time*s (WCRTs) for each task τ_i^g, obtained by considering the task's WCET as a baseline, and factoring in: i) the worst-case number of simultaneous activations requested by the admitted applications; ii) the possible number of faults (up to F) that may have occurred up until the task; and iii) the number of instances implementing each microservice. The third factor can be optionally left unspecified: in this case, our algorithm operates in *capacity planning mode*, suggesting the optimal number of instances per microservice. These WCRTs are then used to formulate an optimization problem (linear in standard mode, quadratic when in capacity planning mode) to compute the optimal partial deadlines. The objective is to minimize replication, i.e., the number of critical tasks. In practice, faults are detected via timeouts derived from these partial deadlines, and tolerated by static replication for critical tasks, or re-execution for non-critical ones. For the sake of brevity, we refer the readers to [18] for further details on the fault model, system analysis, WCRTs and partial deadlines computation, and the identification of critical tasks.

IV. Architecture and Implementation Details

RTilience is designed around the Fault-Tolerant Real-Time Cloud concept envisioned in [39], and has been implemented using RT-Kubernetes, our modified version of Kubernetes [19] supporting real-time containers. The main components of RTilience are illustrated in Figure 1, and described below. All the components of RTilience are released as open-source software components[2], made available through a MIT license (modifications to the Linux kernel are released under the kernel own GPL license).

A. RTilience Overview

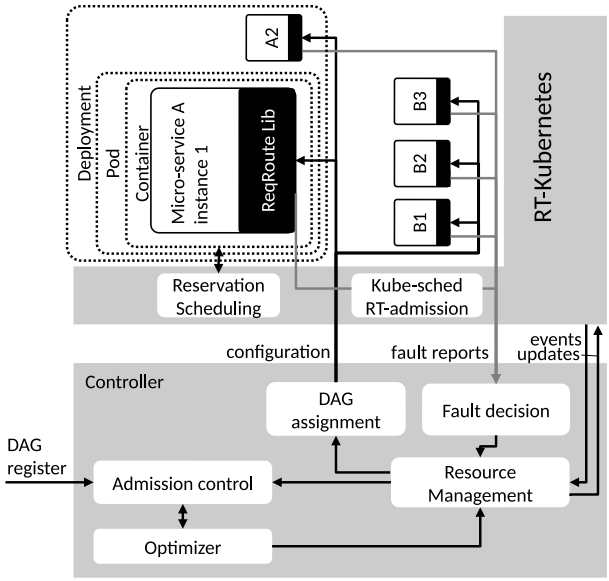

Fig. 1. RTilience overview with the Controller, ReqRoute library and RT-Kubernetes.

RTilience consists of RT-Kubernetes, a request routing library, and a controller enclosing the optimization algorithms from [18] recalled in Section III. The request routing library (ReqRoute) is used to interconnect the various microservices. Each microservice is implemented by multiple containers, and requests are distributed on them using a formally analyzed method based on the provided parameters. Containers act as both primary and secondary replicas for the different DAG's tasks, and the ReqRoute library distributes messages to the appropriate replica as configured by the controller. The microservices are implemented using a socket-like API exposed by ReqRoute, and the library implements all the RTilience routing features transparently. In Kubernetes, the basic execution unit is a "Pod", that can be composed of multiple containers. In RTilience, each Pod comprises exactly one container, which runs a microservice instance. Hence, in the following the terms "Pod", "container", and "microservice instance" (or simply "instance") will be used interchangeably[3].

[2]See: https://retis.santannapisa.it/~tommaso/papers/ieeetsc25-rtilience.php.

[3]We use different terms because Kubernetes manages Pods, the CPU scheduler in the OS kernel schedules processes within containers, and ReqRoute forwards requests among microservice instances.

From a practical standpoint, in order to use RTilience, one needs: 1) Kubernetes with our modified real-time scheduler [43] for the Linux kernel installed on all worker nodes; 2) the Controller described below, deployed in Kubernetes as a pod; 3) DAG applications (deployed as pods) and developed using our ReqRoute library, described below, enabling seamless interactions among individual tasks of the DAG and the controller. The latter performs an optimization of the deployment enabling a balanced distribution of primary and secondary routes among all microservice instances. The resulting routes and timeouts are configured within the ReqRoute library used by every microservice. This allows for the seamlesss use of the real-time scheduling features available in the underlying modified kernel, resulting in a DAG deployment with strong end-to-end latency guarantees, as modelled in the controller.

In the original model [39], a generic load balancer was introduced to distribute the requests on the various containers implementing a microservice, and in the previous analysis [18], a Round-Robin load balancer was assumed to compute the WCRT for each service. In this work, we simplified the computation of the queuing delay for a request submitted to a microservice (Equations (6), (7) and (8) in [18]), by statically configuring a primary and secondary route for each ReqRoute instance. This facilitates the computation of the worst-case queueing delay of all containers and their WCRTs, because the controller knows a-priori which ReqRoute instance submits its requests to which container.

B. RT-Kubernetes

RT-Kubernetes [19] is an extension of the popular Kubernetes container management software, adding integration with a real-time control group scheduler [43] we implemented in the Linux kernel. These modifications to Kubernetes allowed real-time containers to use an extension of the `SCHED_DEADLINE` reservation-based scheduler that can be found in the mainline Linux kernel, namely the HCBS (Hierarchical Constant Bandwidth Server) patchset [43], which is based on the Earliest Deadline First (EDF) algorithm. It allows for scheduling a container i (more precisely, the set of real-time tasks running in a container) through a (Q_i, P_i, m_i) tuple, meaning that the real-time tasks run for Q_i time units every P_i time units on m_i CPU cores. In the time-slices of the physical CPUs assigned by the kernel to each container, its real-time tasks are scheduled in turn using the standard Linux priority-based real-time scheduler, realizing effectively a hierarchical arrangement of schedulers (a priority-based scheduler nested within an outer EDF-based one). The benefit is that unprivileged real-time containers can share worker nodes with other containers without starving them and still be guaranteed their specified time-slices of the CPUs. In order for this to work, the worker node must not be overloaded, meaning that the utilization of each physical CPU core must be smaller than a maximum (the utilization of a CPU core is given by the sum of $\frac{Q_i}{P_i}$ for all the containers allocated to the core), normally below 0.95.

RT-Kubernetes [19] was based on modifications to the key Kubernetes components `kubelet` and `kube-scheduler`. The former controls the execution of Pods on each worker node, by invoking services exposed through the so-called Container Runtime Interface (CRI). A few CRI implementations are available, including `docker` and `containerd`, that leverage features available in the Linux kernel. On the other hand, `kube-scheduler` is the component running on the Kubernets controller node responsible for allocating Pods to worker nodes. The original RT-Kubernetes included invasive changes to these components, requiring significant effort to update the implementation to newer Kubernetes versions.

In the architecture presented in this paper, exemplified in Figure 2, the RT-Kubernetes functionalities have been re-implemented without modifying any upstream Kubernetes component. This has been done on the controller node by introducing the RT Admission Service, exploiting an extension mechanism provided by `kube-scheduler`, and on worker nodes by introducing a proxy between the `kubelet` and the container runtime. The RT Admission Service implements webhooks that are invoked by `kube-scheduler` every time it needs to take an allocation decision, to prioritize and filter worker nodes. In our implementation, we filtered worker nodes ensuring the hosted Pods do not over-subscribe them.

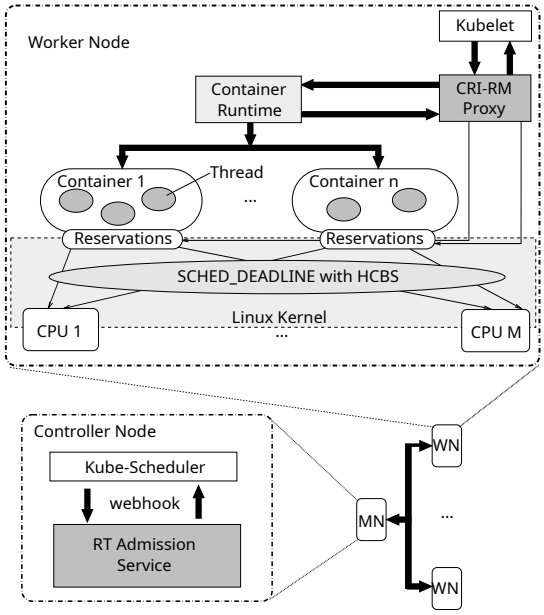

Fig. 2. A Real-Time Kubernetes cluster.

Once a worker node is selected by the Kubernetes scheduler and the container is created by the node's `kubelet`, the container's (Q_i, P_i, m_i) scheduling parameters are set using the `cgroup` interface[4]. Instead of modifying the `kubelet` to set the container's scheduling parameters, we inserted a proxy between `kubelet` and the container runtime, realized as a modified version of the Intel's CRI Resource Manager[5] (CRI-RM). The proxy relays requests and responses back and forth between the two components, altering the CRI requests so to apply the scheduling parameters. The extension server and the

[4]See: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.
[5]See: https://github.com/intel/cri-resource-manager

CRI-RM proxy are only used when starting a container, so a small overhead is added only at containers' startup time.

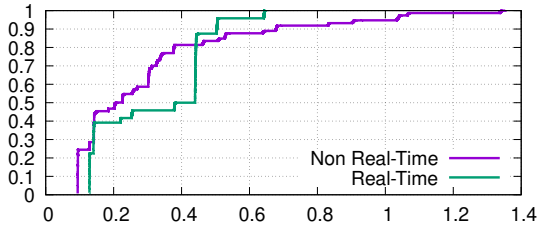

Fig. 3. CDF of the normalized response times of a set of real-time tasks.

The effectiveness of the implementation can be appreciated by executing some simple real-time applications inside RT-Kubernetes containers. For example, a real-time application is often modeled as a set of periodic real-time tasks τ_i with activation period T_i and execution time C_i. In this case, if such tasks are independent and executed with real-time priorities (for example, using the `SCHED_FIFO` scheduling policy) inside a container scheduled with parameters (Q, P, m), then the theoretical analysis of the so-called Compositional Scheduling Framework (CSF) [44], [45] allows computing the worst-case response time of each task. To verify this, such an application, similar to `rt-app`[6] has been implemented and executed inside a Kubernetes container, using the standard Linux scheduler or fixed priorities with RT-Kubernetes.

Figure 3 plots the experimental Cumulative Distribution Function (CDF) of the normalized response times (response time divided by the task period) for a set of real-time tasks with (C_i, T_i) parameters $(8ms, 60ms)$, $(13ms, 80ms)$, and $(22ms, 90ms)$. According to the theoretical CSF, this taskset is schedulable by a $(Q = 7.5ms, P = 11ms)$ reservation. As can be appreciated in the figure, when the standard CPU scheduler is used (green "Non Real-Time" line), the CDF has a long tail finishing near 1.4, with a 10% probability of normalized responses larger than 1, i.e., deadline misses. On the other hand, the "Real-Time" line (corresponding to fixed priorities scheduled by the $(Q = 7.5ms, P = 11ms)$ reservation computed according to CSF analysis) exhibits a 100% probability of normalized response times smaller than 0.7, i.e., all tasks finish well before their next activation.

C. ReqRoute

ReqRoute is a messaging library exposing a socket-like API providing routing, fault detection, and fault handling features required by the DAG instances. ReqRoute instances are managed by the controller that starts and stops Pods enclosing them. For each ReqRoute instance, the controller delivers a configuration that includes a list of the DAG instances to handle, including, for each DAG instance, input and output routes. *Input routes* specify what instances to receive messages from. If multiple routes are configured, the library will wait for all routes to deliver a message and all the received messages will be passed as input to the instance computing function. *Output routes* specify what peers a message should be sent to, at the end of the microservice computations. If multiple routes are stated, the message will be sent in a fan-out fashion to all of them. Each route can be critical or non-critical, depending on the criticality of the associated task, and has a configured primary and secondary peer to send messages to. If an output route is critical, the message is replicated and sent to both peers. Otherwise, it is sent to the primary first, then, if no ack is received within the configured timeout (computed based on the WCRT), the same message is sent to the secondary peer.

Passive fault monitoring is done locally by ReqRoute by capturing acknowledgment messages, leading to lower overheads compared to the active liveness probing in Kubernetes. Then, ReqRoute reports the DAG progress every 10 seconds and successor instance faults when needed to the controller. A fault report may trigger a Pod replacement for the affected ReqRoute's instance, and the reconfiguration of message routing. The overhead of the report mechanism is comparable to the one in the standard Kubernetes `kubelet`, which reports faults to the resource controller via the API and Etcd database.

As illustrated in Figure 4, an acknowledgment (`rr_ack`) of message processing is used to detect timeout violations. Consider a communication from instance X to a microservice with primary Y1 and secondary Y2. In the top half of the figure, X uses a critical route, so the message is sent to both Y1 and Y2 simultaneously. In the example, an acknowledgment from Y2 is not received within the configured critical timeout, so a fault is reported for that instance. On the bottom half of the figure, the route is non-critical, thus the message is first sent to Y1; since an acknowledgment is not received within the configured timeout, a fault is reported for Y1, and the same message is sent to the secondary instance Y2.

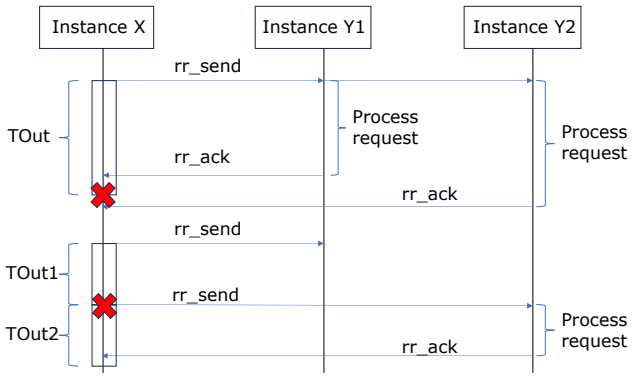

Fig. 4. ReqRoute detecting timeout violations.

Figure 5 shows a simplified overview of the internals of the ReqRoute library. On the left, there is the application facing socket-like functions exposed by the API:

- `rr_send`, used to send a message. To avoid blocking calls, a ReqRoute instance has one global transmit queue (TX) storing messages to be sent, and a condition variable is used to trigger the tx-thread to transmit the message. After a message has been sent, it is stored in a retransmit queue (RTX) used to detect timeouts and to perform retransmissions handled by the resend-thread (rtx-thread).

[6]https://github.com/scheduler-tools/rt-app

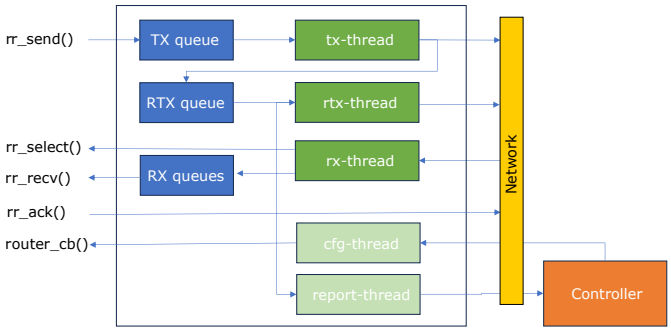

Fig. 5. ReqRoute internals.

- `rr_select`, used to wait until messages are received for a set of DAGs, using a condition variable that is signalled by the rx-thread when messages are available.
- `rr_recv`, used by the application thread to read messages for a given DAG, pulling from a per-DAG receive queue protected by a mutex, filled with messages by the rx-thread.
- `rr_ack`, used to acknowledge that a received message has been processed; this function triggers an acknowledgment packet to be sent to the sender.
- `router_cb`, a callback invoked when DAG route configurations are added, updated or removed.

The blue boxes in Figure 5 show the shared transmit (TX) queue, the retransmit (RTX) queue, and the per-DAG receive (RX) queues. The green boxes represent the ReqRoute threads:

- `tx-thread` waits on the send message condition variable to be signaled and (when woken up) reads the shared transmit queue to transmit messages. The transmission is done using UDP sockets with MessagePack encoding of metadata. After a message has been sent, it is moved to the retransmit queue to detect timeouts and to handle retransmissions.
- `rtx-thread` detects and reports timeouts, and handles retransmission for non-critical messages.
- `rx-thread` handles incoming messages, queued per DAG.
- `cfg-thread` handles configuration updates sent by the controller. A read-write lock is used to allow concurrent read access to the DAG route structures and to have protection when updating them.
- `report-thread` reports DAG progress and instance faults in mini-batches to the controller using an HTTP REST API exposed by the controller.

When the library is initialized in "real-time mode", `the tx-thread`, `rtx-thread`, and `rx-thread` are scheduled with the `SCHED_FIFO` policy using a configurable real-time priority (we use 33 in the experiments presented in this paper). Normally, this would require administration privileges or the `CAP_SYS_NICE` capability. However, when using RT-Kubernetes described in Section IV-B a containerized application can use `SCHED_FIFO` even without running with special privileges or capabilities, provided that a non-zero `rt_runtime_us` is allocated for the application's container.

D. The Controller

The RTilience controller is responsible for the control plane of the microservices and distributes the data plane configurations to every instance. It functions similarly to a service mesh control plane, allowing a central routing to control traffic within the cluster. The RTilience controller, in cooperation with the Kubernetes controller, works by keeping a declared desired state of the cluster: each Deployment has a specified desired replica count of Pods hosting each microservice instance; and each microservice is labelled with its WCET, microservice name, and real-time parameters to claim resources, as well as standard Kubernetes Pod specifications for prioritization. More in detail, the RTilience controller: i) derives the requirements from a new DAG registration for its admission, as illustrated in Figure 6; ii) monitors updates to the desired and actual state of Pods, Deployments and DAG requirements; iii) performs the corresponding DAG admission status and task-to-instance assignment updates, as depicted in Figure 8; and iv) collects fault and progress reports from the instances to determine their health status, as shown in Figure 9.

In what follows, we describe each of these activities in detail. The first activity is deploying a newly registered DAG to perform its tasks using the underlying microservices (see Figure 6). A user registers a DAG by issuing an HTTP POST request to the controller with the task DAG $\mathcal{A}^g$, the task-microservice mapping $\varphi^g : \Gamma^g \rightarrow \mathcal{S}$, minimum period P^g, max DAG faults F, and DAG deadline D^g. Then, the controller executes the optimizer [18] to derive the partial deadlines, the criticality mode, and the WCRTs for each DAG task. If the current number of microservice instances is insufficient to host the new DAG, the optimizer is executed again in *capacity planning mode* to derive microservice requirements on how many instances are needed for each microservice. While the new DAG is waiting in admitting status, the controller acts as a horizontal auto-scaler by adjusting the desired number of microservice instances in each Deployment using the Kubernetes API. The client registering a DAG can query the DAG's status to see when it is admitted. The client's data plane, e.g. in the same Pod, also makes use of the ReqRoute library to communicate with the first and last microservices. Hence, it also receives routing configuration updates from the controller, and the `router_cb` callback informs the client when the DAG is admitted, so it can start sending requests.

DAG registration incurs the highest overhead with respect to the original Kubernetes due to the optimization step. In [18] we conducted an extensive campaign of experiments to empirically evaluate the solve time overhead. We randomly generated 100 scenarios, each featuring either random or chain-like topologies, composed of $n_g \in [5, 20]$ tasks and $n_{\mathcal{A}} \in [1, 60]$ interfering DAGs with same topology. Figure 7 revisits the solving time plot of [18], showcasing the overhead introduced by operating the optimizer in capacity planning mode to compute the partial deadlines and determine the criticality of each task, for different scenarios. As expected, the solving time ramps up quickly as the problem size grows: 1-2 seconds for relatively small problems with up to 200 tasks; up to $\sim$ 10 seconds for larger problems.

The controller functionality for closing the gap between current and desired state is illustrated in Figure 8, and it is initiated when either the current or desired state is changed. The task to instance assignment procedure shown in Figure 8

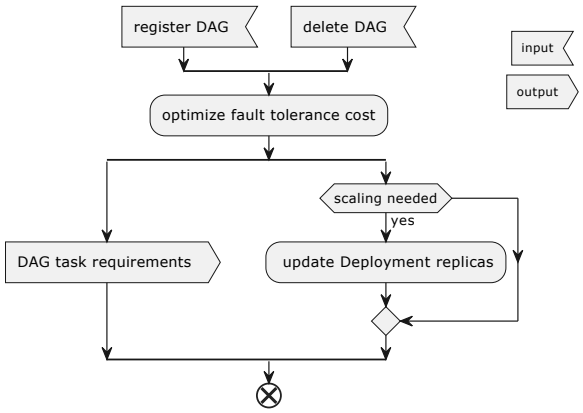

Fig. 6. Controller DAG register or delete activity flow diagram.

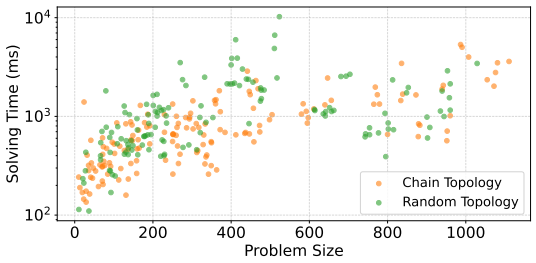

Fig. 7. Optimizer solving time vs problem size (i.e, $n_g \cdot n_{\mathcal{A}}$). From [18].

is carried out using a data plane configuration: for each task of each DAG, a primary and secondary instance are designated for request routing, along with the criticality mode and the timeout of the task. Additionally, the data plane configuration also contains information on which tasks an instance should receive messages from, so to handle joins before proceeding to the actual processing. This data plane configuration is then split into per-instance updates that are distributed in HTTP chunks to the ReqRoute within all affected instances. These updates contain information on when they should be applied (a client request sequence number), so that the update can be applied correctly in a fully distributed way across the

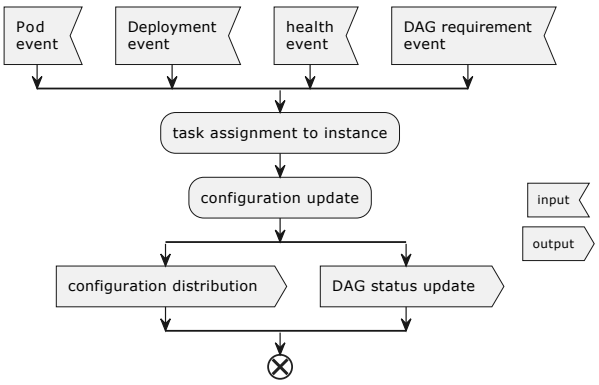

Fig. 8. Controller: tasks-to-instances assignment and configurations update.

instances. Thanks to the configuration of both primary and secondary routes for instances, the central controller is not part of the fast fault handling mechanism, so it has a more relaxed time window to apply re-configurations. The assignment uses a Worst-Fit policy, i.e. it picks the instance with the least number of tasks assigned, with the constraint of anti-affinity between primary and secondary instances. To prevent incomplete DAGs from locking resources for each other, only complete DAGs are assigned and status is updated to admitted. Priority is given to DAGs that already have some assignments, but lost a primary or secondary route. The overhead of updating assignments is moderate since the optimizer is not needed for this, but is larger than standard Kubernetes that would use a Service as a proxy for the Pods in the Deployments.

The clients and instances that send requests using ReqRoute to the next task in the DAG wait for an acknowledgment that should come after the next task is processed and its results are sent. The ReqRoute library accumulates acknowledgment timeouts over mini-batches and sends fault reports with information on how many faults for primary and secondary instances have been observed. Therefore, each instance supervises its downstream instances. These faults correspond to the transient nature of faults in the model described in Section III. The controller then collects these fault reports and accumulates over all the reports for an instance, as shown in Figure 9.

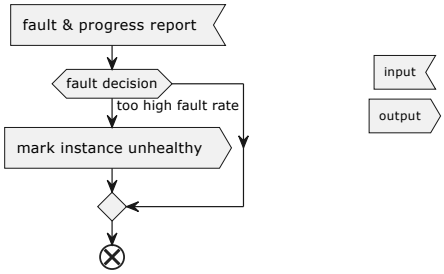

Fig. 9. Controller: fault reporting and marking of unhealthy instances.

Based on the fault rate (compared with the potential max rate of faults), a decision can be made to mark an instance as unhealthy, i.e. a change in current state and a trigger for the controller functionality shown in Figure 8. This first leads to a re-assignment of the tasks to other instances (when available), then to a replacement of an unhealthy instance. Re-assignment can be performed by the next DAG period if other instances have enough spare capacity. Otherwise, we need to mark an instance as unhealthy, going beyond a transient fault model, since on the next period the instance is not available and has a permanent fault. In this case, we need the time it takes for Kubernetes to start a new Pod and make it available (in our evaluation, that is around 4-10 seconds). This permanent fault decision is only activated in the evaluations in Section V-C. In addition to fault detection based on fault reports, standard Kubernetes methods using active health probes can be used, even if such monitoring has a higher overhead. On failure, Kubernetes will terminate unhealthy Pods. The potential tasks assigned to such terminating Pods will automatically be re-assigned by the controller to other Pods directly or when they

become available. The fault decision overhead in RTilience compared with standard Kubernetes is due to the accumulation of several fault reports instead of only one `kubelet` report on probe fault. The fault mitigation for permanent faults of replacing a Pod is using the same mechanism as in standard Kubernetes, and hence have the same overhead.

During periods of a permanently failed Pod, if the second replica also fails, a request may need to be dropped. As shown in Section V-C, our design is also robust in case of sporadic double faults, even though not strictly covered by the analysis. However, if a double fault happens in both replica Pods of the same task (either critical or non-critical), then the request experiencing the double fault is dropped, so this is a possibility that, albeit unlikely, needs attention at the application level.

V. Experimental Evaluation

To carry out an experimental validation of RTilience, we used the ReqRoute library presented in Section IV-C to develop a microservice-based application and a client to interact with it and measure response times. The application consists of multiple microservices, each replicated a number of time to assess RTilience's fault tolerance. The client registers a DAG with the controller described in Section IV-D, specifying the application topology and associated parameters, including execution times and real-time reservations. The controller then instructs the underlying microservices on how to route packets so to implement the desired DAG topology. Each microservice is configured by either the client or RTilience's controller to: i) receive packets from one or more input microservices, or the client; ii) execute for a specified amount of time c_i in a busy loop; and iii) produce outputs for one or more output microservices, or the client. Finally, the client periodically sends packets to the first task of the specified DAG, expecting to receive back a response from the last task, measuring the Round-Trip-Times (RTTs) experienced by each packet. At run-time, the client encodes in the packet which instances should inject a fault during task execution, based on the overall fault rate level for DAG activations. We consider two types of failures in the experimental evaluation: transient (i.e., the microservice instance does not send a response in time) and persistent (i.e., the underlying Pod crashes). Multiple client instances can be instantiated to deploy DAGs simultaneously, thus generating interference during their activations.

We used this setup to conduct experiments on a Kubernetes cluster with a controller node and two worker nodes. The controller, client and microservice instances all reside on dedicated Pods instantiated using Kubernetes' Deployment and Service objects (the recommended tools to deploy and manage Pod instances). The physical hosts are Dell machines based on an Intel Xeon E5-2650 CPU running at 2.60GHz, with 64 GB of RAM. Hyperthreading has been disabled, and only 16 CPU cores have been used; two of them have been isolated and are not available to Kubernetes. The hosts have a 10 Gb/s network interface and are connected through a 40 Gb/s switch. The ping time between two hosts (measured using the "`ping`" command) ranges from $32\mu s$ to $88\mu s$ (average $50\mu s$). The RTT between two containerized applications using ReqRoute ranges between $103\mu s$ and $256\mu s$ (average $116\mu s$) when the containers are deployed on the same host, and between $173\mu s$ and $347\mu s$ (average $190\mu s$) when the containers are deployed on different hosts. These times include the latencies introduced by the CNI plugin used to interconnect the containers.

As explained in Section IV-B, a scheduler extension is used to filter the worker nodes so that they are not overloaded. Except this, the standard Kubernetes scheduler is used. When a container is scheduled on a worker node, the modified CRI-RM proxy allocates it on the node's CPU cores using a First Fit policy so that no core is overloaded.

A. Simple 2-Tasks Pipeline

The first set of experiments evaluates RTilience using a simple 2-tasks pipeline, with τ_1 sending its output to τ_2, where the WCETs are $c_1 = 10ms$ and $c_2 = 30ms$, so the theoretical worst-case RTT is $c_1 + c_2 = 40ms$ (excluding network delays). The client periodically sends packets with a period $T = 300ms$, equal to the end-to-end DAG deadline $D = 300ms$. We assume only transient faults, thus the advanced healing functionalities of the controller are disabled.

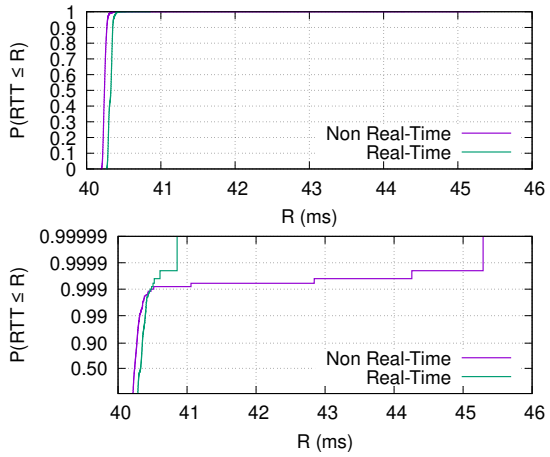

Fig. 10. CDF of the RTTs measured with 1 instance of the simple 2-tasks pipeline and no faults. The end-to-end deadline is $D = 300ms$.

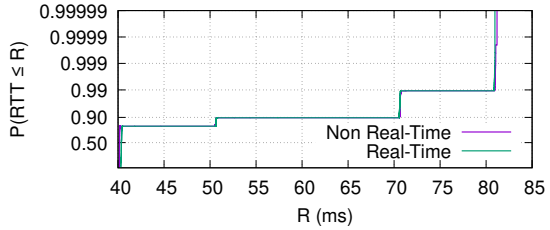

Fig. 11. CDF of the RTTs measured with the simple 2-tasks pipeline and a 10% failure probability on each node. End-to-end deadline: $D = 300ms$.

Figure 10 shows the experimental Cumulative Distribution Function (CDF) of the RTTs for a single DAG instance. The top subfigure depicts the experimental CDF with the Y axis in linear scale, and the bottom one with a non-linear scale

on the Y axis, to better highlight the details of the curve near probability 1. Due to the absence of additional real-time workloads, the obtained RTTs are close to the theoretical value of $40ms$. Due to the absence of faults in this scenario, the "Non Real-Time" curve corresponds to running this DAG on an unmodified Kubernetes cluster, which uses the Linux default scheduler. As visible in the plots, using hierarchical real-time scheduling features we embed in RTilience to schedule the applications' threads increases slightly the average RTT (the "Real-Time" plot is slightly on the right of the "Non Real-Time" plot), but reduces the worst-case RTT (the "Real-Time" plot has a shorter tail than the "Non Real-Time" plot). For these experiments, we used a reservation period $P = T$ equal to the DAG's period and a runtime $Q = 0.9P$ a bit smaller than the reservation period (so that every reservation is allocated on a dedicated CPU core, resulting in the scheduling model analyzed in previous papers).

Figure 11 presents the experimental CDF obtained when each one of the two tasks has a 10% probability of experiencing a transient failure. For space reasons, we just show the non-linear scale plot zoomed over high percentiles (similarly to the bottom plot of Figure 10), omitting the linear-scale plot. In this case, the "Real-Time" and "Non Real-Time" plots are almost coincident because the delays are dominated by the effects of tasks' failures. In any case, the figure shows that even with this high (even pretty unreasonable) failure rate, RTilience is able to respect all the deadlines. Figure 12

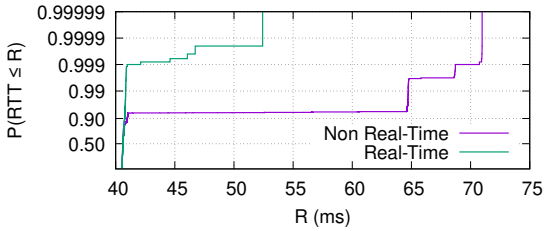

Fig. 12. CDF of the RTTs measured with the simple 2-tasks pipeline, no faults, and additional load on the worker nodes ($D = 300ms$).

shows the experimental CDF of the RTTs obtained when the worker nodes are disturbed by a background workload composed of `hackbench`[7] and a set of periodic tasks with a 40% utilization. The non-real-time containers experience some additional latency due to the "noisy neighbor" effect, and the tail of the "Non Real-Time" CDF is much longer.

Finally, Figure 13 shows the experimental CDF measured when the containers serve 3 instances of the DAG, still configured with the same end-to-end deadline equal to the activation period $D = T = 300ms$. Notice that, consistently with the theoretical analysis, RTilience is still able to respect all the deadlines; moreover, using real-time priorities allows again for reducing the length of the CDF tail.

B. More Complex DAGs

The next set of experiments has been performed on a more complex DAG, shown in Figure 14, with 5 tasks and a source

[7]https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/hackbench

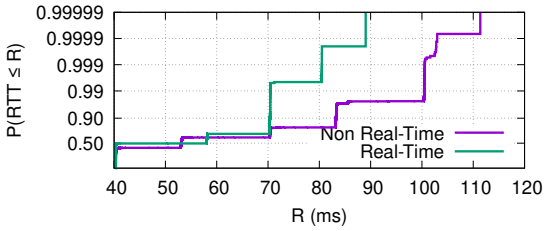

Fig. 13. CDF of the RTTs measured for 3 instances of the simple 2-tasks pipeline, and no faults. The end-to-end deadline is $D = 300ms$.

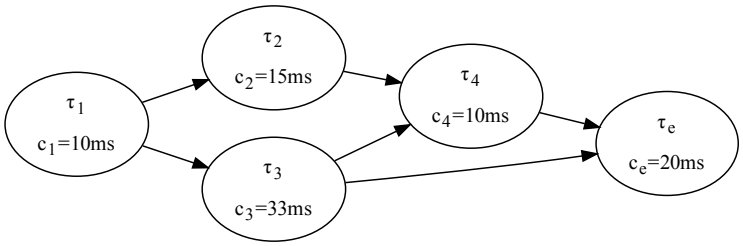

Fig. 14. A more complex DAG with 5 tasks.

period equal to the end-to-end deadline of $T = D = 250ms$. The experimental CDFs of the RTTs measured in case of

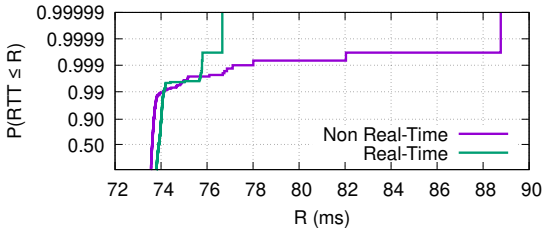

Fig. 15. CDF of the RTTs measured for 3 instances of the DAG in Figure 14, and no faults. The end-to-end deadline is $D = 250ms$.

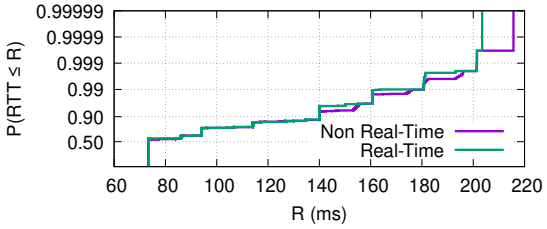

Fig. 16. CDF of the RTTs measured for 3 instances of the DAG in Figure 14, and a 10% failure probability on each microservice instance ($D = 250ms$).

no failures and with a 10% failure probability are reported in Figures 15 and 16, respectively. Again, the figures show that RTilience is able to respect the application's temporal constraints, and Figure 16 confirms that the deadlines are respected even in the presence of some transient faults. Moreover, Figure 15 again shows that using real-time priorities allows for reducing the curve's tail (at the cost of slightly increasing the average RTT).

Finally, a more realistic DAG, modelling the control loop of an Autonomous Transport Robot (ATR) offloaded to an on-

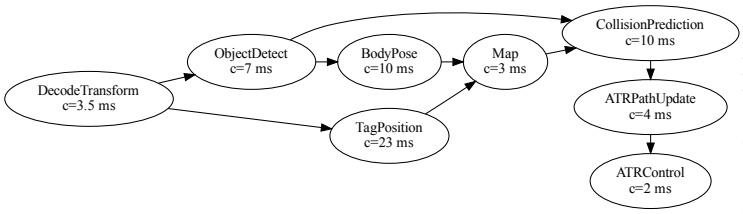

Fig. 17. DAG describing the cloud offloaded control loop of an Autonomous Transport Robot.

prem cloud has been analyzed [46]. Figure 17 shows the DAG modeling the behavior of the program used by the robot to move, based on video captured by a ceiling mounted camera.

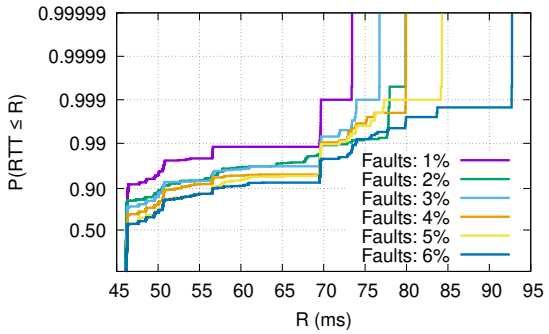

Fig. 18. CDF of the RTTs measured for 2 instances of the DAG in Figure 17, and a failure probability ranging from 1% to 6% ($D = 100ms$).

First, two instances of the ATR control DAG have been simultaneosuly created with period equal to $100ms$. With no faults, the measured RTTs resulted almost constant, and equal to about $46ms$, as expected. The experimental CDFs of the RTTs measured when each instance has a failure probability ranging to 1% to 6% are reported in Figure 18. The figure shows that increasing the instance failure probability decreases the probability to have a response time near to $46ms$, as expected. It can also be noticed that with failure rates up to 6% RTilience is able to respect the application's temporal constraints, even in presence of a non-negligible amount of transient faults (and even if the probability of not respecting the single-failure assumption is not negligible).

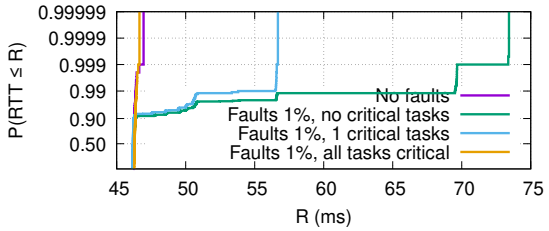

Fig. 19. CDF of the RTTs measured for 1 instance of the DAG in Figure 17 and different task criticalities ($D \in \{100ms, 60ms, 48ms\}$).

In all the experiments up to now, we tested the effect of the fault tolerance mechanism on the end-to-end response time using the DAGs as "black boxes". We empirically demonstrated how RTilience is able to respect the deadlines even in the presence of a 10% failure rate, but without checking how many failures were detected, how many re-transmissions happened, and/or which tasks were configured as critical; as a matter of fact, the controller did not introduce any critical task due to the deadlines being too lax.

In the next experiment, we decreased the period and relative deadline of the ATR DAG in Figure 17 so that some tasks resulted to be critical (experiments measuring the detected failures and re-transmissions will be presented in the next subsection). For example, with $D = T = 60ms$ the "TagPosition" task is critical, and all the others are not, while with $D = T = 48ms$ all the tasks are critical. Figure 19 compares the experimental CDFs of the RTTs when no instances fail (this is the same for all the DAGs) and with a failure probability equal to 1% for $D = T = 100ms$ (no node is critical), $D = T = 60ms$ (only "TagPosition" is critical), and $D = T = 48ms$ (all tasks are critical). It can be noticed that the CDF for $D = T = 48ms$ is almost identical to the "no failures" CDF, because each instance simultaneously sends its output to both the primary and secondary instance, so a failure in one of the two instances does not increase the RTT. In the $D = T = 60ms$ case, instead, failures of a "TagPosition" instance do not increase the RTT, but failures of the other nodes do. Finally, for $D = T = 100ms$, every node failure increases the RTT. Of course, the RTT reduction introduced by critical tasks has a cost: for every critical task, both the primary and secondary instances are active simultaneously, doubling the CPU consumption for the task.

C. Fault tolerance

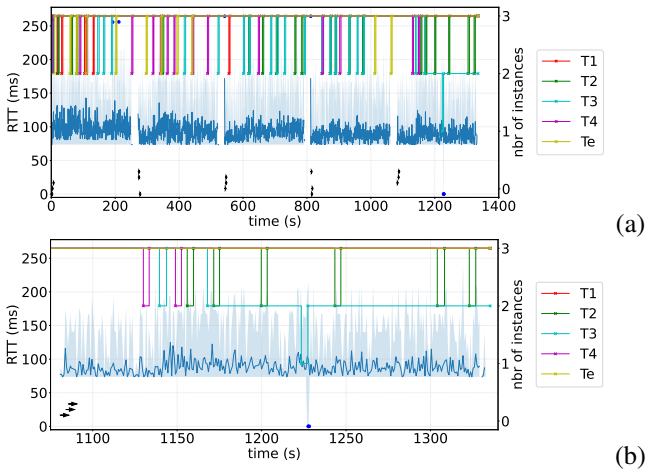

Fig. 20. RTTs from an experiment with 3 DAG instances, repeated 5 times, and a $250ms$ end-to-end deadline. (a) full series, (b) detail of the fifth run.

In the next set of experiments, failures are persistent. The controller's healing functionalities are enabled by activating the fault decision functionality (in Figure 9) leading to task re-assignment and replacement of permanently unhealthy microservice instances. As a reminder, this feature was disabled in previous experiments because the faults analyzed earlier were transient. The DAGs are executed with a high fault rate of 15%, but only one instance of *each* microservice is allowed

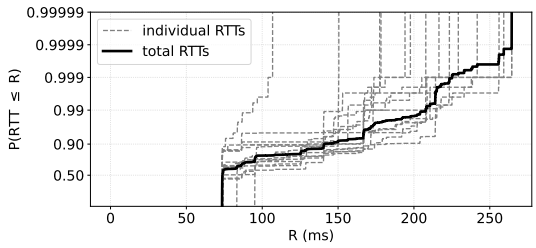

Fig. 21. CDF of RTTs of the experiment in Figure 20. The bold line is the total RTT CDF while the dashed are individual DAG's RTT CDF.

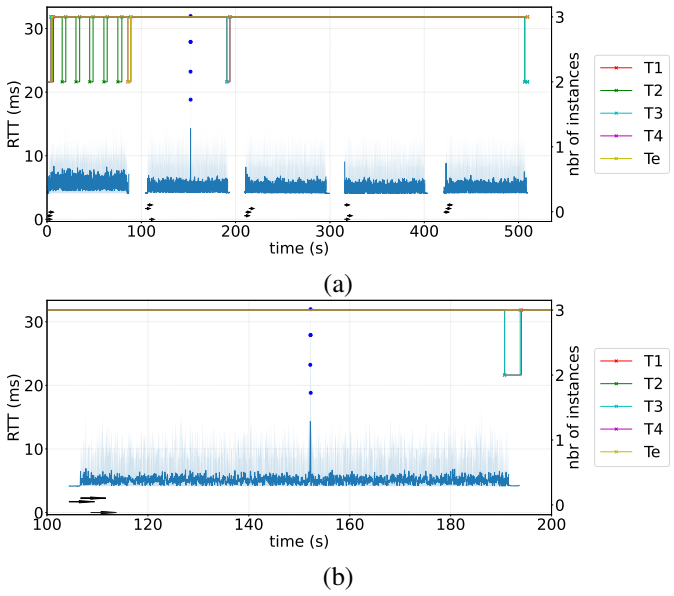

Fig. 22. A time-series of an experiment with 3 DAG instances, repeated 5 times, with a shorter end-to-end deadline of $17ms$. (a) full series, (b) detail of second run. The visual notation is the same as for Figure 20.

to fail at any time. Due to the high fault rate a constant churn of instances are replaced. Hence, this example operates outside the models assumptions to provide full guarantees, since more than F faults may occur for a DAG, and less than M instances may temporarily be available at DAG activation. For this first experiment, the more complex DAG of Figure 14 is used, with 3 DAGs executing in parallel and repeated 5 times. The same setup as in section V-B with WCETs $C_1 = 10ms$, $C_2 = 15ms$, $C_3 = 33ms$, $C_4 = 10ms$, $C_e = 20ms$ and the source period $T = 250ms$. To these execution times we add an extra $400\mu s$ for network round-trip latency. The execution model used by the controller's optimizer to analyze the DAG assumes that each container executes on a dedicated CPU core (hence, up to now real-time priorities have been used by assigning a container runtime almost equal to the reservation period, as previously mentioned). However, with this kind of setup it is not possible to fit these complex experiments in the 2 worker nodes of the cluster. Hence, we tried using smaller values for the reserved runtimes Q_i (the reservation periods P_i have been set equal to the DAG's period T). Obviously, the reserved runtime Q_i affects the amount of timeout violations, especially for tasks with multiple inputs. After some tests[8], the reserved budgets have been set to $Q_1 = 33ms$, $Q_2 = 48ms$, $Q_3 = 99ms$, $Q_4 = 50ms$, and $Q_e = 63ms$ with a period $P_i = 100ms$ for all the tasks.

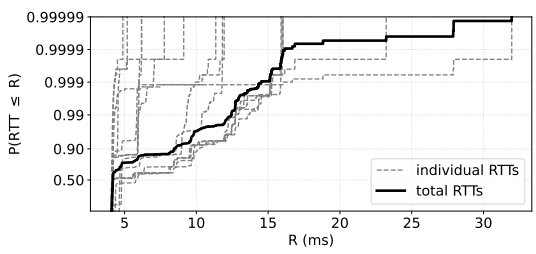

Fig. 23. CDF of RTTs of the experiment in Figure 22. The heavy line is the total RTT CDF while the dashed are individual DAG's RTT CDF.

The RTTs and number of healthy instances over time are shown in Figure 20, where: the left vertical-axis shows the RTT and the right one shows the number of instances in each microservice; the blue dots at zero RTT indicate dropped requests and above end-to-end deadline is overrun RTT; the blue line is the average RTT and the lightblue fill is the min and max RTT; the arrows in lower part of the graph indicate the start time of a DAG. As it can be seen, the number of dropped requests is quite low even when we have an unrealistic high fault rate. Also, sometimes the number of instances for a microservice goes down. During those periods, no healthy instances exist to take new tasks, but tasks already assigned continue to use an unhealthy instance until a new one becomes available. This is possible since instances can use the secondary route if the primary fails. In these experiments, the new instance becomes available in about 4 seconds, but that depends on the start-up time of a Pod. For this time-series, a CDF of the RTTs is shown in Figure 21. It shows that a few RTTs are breaking the end-to-end deadline at $T = 250ms$, in the probability interval between 3 and 4 nines. The average RTT is close to the minimum seen in the experiments with no replacement of unhealthy instances (shown in Figure 16).

A second experiment with a modified complex DAG with about 20 times shorter execution times and end-to-end deadline has been performed to illustrate that this configuration also works. The execution times are $C_1 = 500\mu s$, $C_2 = 750\mu s$, $C_3 = 1600\mu s$, $C_4 = 500\mu s$, $C_e = 1000\mu s$ and the source period $T = 17ms$. To these execution times we still add an extra $400\mu s$ for network round-trip latency. The reserved budgets are $Q_1 = 1700\mu s$, $Q_2 = 2600\mu s$, $Q_3 = 4500\mu s$, $Q_4 = 2700\mu s$, and $Q_e = 3300\mu s$ and a period $P_i = 5000\mu s$ $\forall i$.

The RTTs and number of healthy instances over time are shown in Figure 22. As can be seen, there is a burst of overrunning messages, but besides that the system runs without issues. In Figure 23 it can be seen that the CDF crosses the end-to-end deadline at a probability just above 4 nines.

VI. Conclusions and Future Work

This paper presented RTilience, an implementation of the Fault-Tolerant Real-Time Cloud architecture [39], [18] based

[8]A theoretical model for formally analyzing the performance of tasks with $Q_i << P_i$ is currently under development.

on Kubernetes and a real-time control group scheduler [43]. The proposed architecture uses a centralized controller and a request routing library (ReqRoute) that implements the partial deadlines and re-execution mechanisms proposed by RTilience in a distributed way. We performed an extensive experimentation of the proposed solution over a dedicated testbed, using synthetic applications, and an autonomous transport robot use-case. The obtained results are consistent with the theoretical expectations, confirming that RTilience is able to provide end-to-end real-time guarantees for distributed DAG workloads, tolerating a specified amount of faults.

Regarding future works on the topic, we plan to investigate on how to overcome some limitations of our proposal. In our model, we estimate end-to-end response times under worst-case conditions, which relies heavily on prior knowledge of worst-case execution times for all deployed tasks. This can be cumbersome in real-world, multi-tenant, Cloud infrastructures, and may lead to overestimations of the execution times, resulting in relatively low resource usage levels (notice that if a WCET is overestimated, the system still provides the expected guarantees; the issue is that it might require too many resources). Therefore, in order to decrease the pessimism in our current methodology, we plan to consider probabilistic approaches for modeling the components' behavior [47], [48], [49], [50]. Also, we plan to consider more dynamic scenarios where the worst-case bounds may need to be re-assessed at run-time, so that a re-run of our optimizer may ensure that the system is configured optimally according to dynamically changing conditions. This may be implemented by expanding the information in the messages already exchanged between the controller and the Pods, re-running the optimizer on a need-by basis, and finally applying the new configuration with a mode-change protocol [51], [52]. For example, this might be useful to deal with workload changes over time, or a wider set of faults, including performance degradation problems. The MILP-based optimizer exhibits a solving time that grows exponentially with the problem size (i.e., number of total tasks to be deployed), making our approach impractical for large deployments. A heuristic-based approach will be investigated to find suboptimal task criticalities and partial deadlines, investigating useful trade-offs between optimality of the solution and solving time. Last, our architecture relies on our non-standard extensions to the Linux kernel process scheduler, which are hard to maintain across future kernel releases. We have on-going discussions [53] with core kernel developers, about how to possibly ease this task in upcoming versions of the mainline kernel, exploiting the "deadline servers" feature currently under discussion [54].

REFERENCES

[1] R. Buyya *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, nov 2018.

[2] M. Chiosi *et al.*, "Network Functions Virtualisation – An Introduction, Benefits, Enablers, Challenges & Call for Action," SDN and Openflow World Congress, Darmstadt, Germany, White Paper 1, 2012.

[3] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comp. Surveys*, vol. 31, no. 1, 1999.

[4] R. Andreoli, R. Mini, P. Skarin, H. Gustafsson, J. Harmatos, L. Abeni, and T. Cucinotta, "A multi-domain survey on time-criticality in cloud computing," *IEEE Transactions on Services Computing*, p. 1–19, 2025.

[5] M. Szalay, P. Mátray, and L. Toka, "Real-time task scheduling in a FaaS cloud," in *IEEE 14th International Conference on Cloud Computing*, Sep. 2021, pp. 497–507.

[6] P. O'Donovan, C. Gallagher, K. Leahy, and D. T. O'Sullivan, "A comparison of fog and cloud computing cyber-physical interfaces for industry 4.0 real-time embedded machine learning engineering applications," *Computers in Industry*, vol. 110, pp. 12–35, 2019.

[7] I. Mahadevan and K. Sivalingam, "Quality of service architectures for wireless networks: Intserv and diffserv models," in *Fourth Intern. Symp. on Parallel Architectures, Algorithms, and Networks*, 1999, pp. 420–425.

[8] X. Xiao, A. Hannan, B. Bailey, and L. M. Ni, "Traffic engineering with mpls in the internet," *IEEE network*, vol. 14, no. 2, pp. 28–33, 2000.

[9] N. Finn, "Introduction to time-sensitive networking," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 22–28, 2018.

[10] B. Ordozgoiti, A. Mozo, S. G. Canaval, U. Margolin, E. Rosensweig, and I. Segall, "Deep convolutional neural networks for detecting noisy neighbours in cloud infrastructure," *COSTAC 2017*, p. 59, 2017.

[11] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong temporal isolation among containers in OpenStack for NFV services," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.

[12] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Ninth ACM international conference on Embedded software*, 2011, pp. 39–48.

[13] F. Alriksson, L. Boström, J. Sachs, Y.-P. E. Wang, and A. Zaidi, "Critical IoT connectivity Ideal for Time-Critical Communications," *Ericsson technology review*, vol. 2020, no. 6, pp. 2–13, 2020.

[14] M. N. Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *Journal of Network and Computer Applications*, vol. 61, pp. 81–92, 2016.

[15] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[16] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 194–205.

[17] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos, "Cloud computing: Survey on energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, dec 2014.

[18] R. Andreoli, H. Gustafsson, L. Abeni, R. Mini, and T. Cucinotta, "Optimal deployment of cloud-native applications with fault-tolerance and time-critical end-to-end constraints," in *IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2023.

[19] S. Fiori, L. Abeni, and T. Cucinotta, "RT-Kubernetes: Containerized Real-time Cloud Computing," in *Proceedings of the 37th ACM/SIGAPP Symp. on Applied Computing*. ACM, 2022, pp. 36–39. [Online]. Available: https://dl.acm.org/doi/10.1145/3477314.3507216

[20] A. U. Rehman, R. L. Aguiar, and J. P. Barraca, "Fault-tolerance in the scope of cloud computing," *IEEE Access*, vol. 10, 2022.

[21] M. Hasan and M. S. Goraya, "Fault tolerance in cloud computing environment: A systematic survey," *Computers in Industry*, vol. 99, pp. 156–172, 2018.

[22] T. Welsh and E. Benkhelifa, "On resilience in cloud computing: A survey of techniques across the cloud domain," *ACM Comput. Surv.*, vol. 53, no. 3, may 2020. [Online]. Available: https://doi.org/10.1145/3388922

[23] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, March 2021.

[24] G. Lanciano, F. Galli, T. Cucinotta, D. Bacciu, and A. Passarella, "Predictive auto-scaling with OpenStack monasca," in *14th IEEE/ACM International Conference on Utility and Cloud Computing*, Dec. 2021.

[25] D. Saxena and A. K. Singh, "Ofp-tm: An online vm failure prediction and tolerance model towards high availability of cloud computing environments," *The Journal of Supercomputing*, vol. 78, no. 6, 2022.

[26] G. Lanciano, R. Andreoli, T. Cucinotta, D. Bacciu, and A. Passarella, "A 2-phase strategy for intelligent cloud operations," *IEEE Access*, 2023.

[27] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "Ftcloud: A component ranking framework for fault-tolerant cloud applications," in *IEEE 21st International Symposium on Software Reliability Engineering*, 2010.

[28] Y. Zhang, Z. Zheng, and M. R. Lyu, "Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing," in *IEEE 4th International Conference on Cloud Computing*, 2011, pp. 444–451.

[29] A. Javed, J. Robert, K. Heljanko, and K. Främling, "Iotef: A federated edge-cloud architecture for fault-tolerant iot applications," *Journal of Grid Computing*, vol. 18, pp. 57–80, 2020.

[30] M. Mudassar, Y. Zhai, and L. Lejian, "Adaptive fault-tolerant strategy for latency-aware iot application executing in edge computing environment," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 13 250–13 262, 2022.

[31] L. Toka, "Ultra-reliable and low-latency computing in the edge with kubernetes," *Journal of Grid Computing*, vol. 19, no. 3, p. 31, 2021.

[32] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson, "An empirical study of service mesh traffic management policies for microservices," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, p. 17–27.

[33] M. Elhemali, N. Gallagher, B. Tang, N. Gordon, H. Huang, H. Chen, J. Idziorek, M. Wang, R. Krog, Z. Zhu *et al.*, "Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service," in *USENIX Annual Technical Conference*, 2022.

[34] F. C. et al., "Bigtable: A distributed storage system for structured data," *ACM Transactions Computing Systems*, vol. 26, no. 2, jun 2008.

[35] R. Siyadatzadeh, F. Mehrafrooz, M. Ansari, B. Safaei, M. Shafique, J. Henkel, and A. Ejlali, "Relief: A reinforcement learning-based real-time task assignment strategy in emerging fault-tolerant fog computing," *IEEE Internet of Things Journal*, 2023.

[36] Z. Li, H. Yu, G. Fan, and J. Zhang, "Cost-efficient fault-tolerant workflow scheduling for deadline-constrained microservice-based applications in clouds," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3220–3232, 2023.

[37] "Reliability Pillar: AWS Well-Architected Framework," https://docs.aws.amazon.com/pdfs/wellarchitected/latest/reliability-pillar/wellarchitected-reliability-pillar.pdf, Dec 2023.

[38] "Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS: AWS Whitepaper," Nov 2021.

[39] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *IEEE 26th International Symposium on Real-Time Distributed Computing*, 2023, pp. 170–175.

[40] R. Andreoli, H. Gustafsson, L. Abeni, R. Mini, and T. Cucinotta, "Design-time analysis of time-critical and fault-tolerance constraints in cloud services," in *IEEE 16th International Conference on Cloud Computing*. IEEE, 2023, pp. 415–417.

[41] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *31st IEEE Real-Time Systems Symposium*, 2010, pp. 259–268.

[42] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of dags," *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3242–3252, 2014.

[43] L. Abeni, A. Balsini, and T. Cucinotta, "Container-Based Real-Time Scheduling in the Linux Kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019, publisher: ACM New York, NY, USA.

[44] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *25th IEEE International Real-Time Systems Symposium*, Dec 2004.

[45] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, and O. Sokolsky, "CARTS: A tool for compositional analysis of real-time systems," *SIGBED Review*, vol. 8, no. 1, Mar 2011.

[46] D. Wang, "Low latency object detection on the Edge-cloud AprilTag-assisted object detection and positioning – Dissertation," http://uu.diva-portal.org/smash/get/diva2:1569020/FULLTEXT01.pdf, Jun 2021.

[47] L. Abeni, N. Manica, and L. Palopoli, "Efficient and robust probabilistic guarantees for real-time tasks," *Journal of Systems and Software*, vol. 85, no. 5, pp. 1147–1156, 2012.

[48] K. Konstanteli, T. Cucinotta, K. Psychas, and T. A. Varvarigou, "Elastic admission control for federated cloud services," *IEEE Trans. on Cloud Computing*, vol. 2, no. 3, pp. 348–361, Jul. 2014.

[49] R. I. Davis, A. Burns, and D. Griffin, "On the meaning of pwcet distributions and their use in schedulability analysis," in *Real-Time Scheduling Open Problems Seminar*, 2017, pp. 1–4.

[50] B. V. Frías, L. Palopoli, L. Abeni, and D. Fontanelli, "Probabilistic real-time guarantees: There is life beyond the i.i.d. assumption," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 175–186.

[51] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-time systems*, vol. 26, no. 2, 2004.

[52] A. Burns, "System mode changes-general and criticality-based," in *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, 2014, pp. 3–8.

[53] J. Corbet, L. Abeni, and Y. Andriaccio, "Reports from OSPM 2025, day two – Hierarchical CBS with deadline servers," https://lwn.net/Articles/1021332/, May 2025.

[54] J. Corbet, "Deadline servers as a realtime throttling replacement," https://lwn.net/Articles/934415/, Jun 2023.

Harald Gustafsson is an Expert in Network-Compute Application Platforms at Ericsson Research, Ericsson (Sweden). He has a MSc in Electrical Engineering from Lund University (Sweden), and a PhD in Applied Signal Processing from the Blekinge Institute of Technology (Sweden). His research addresses multimedia systems, distributed systems, communication systems and cloud systems on aspects as time criticality, reliability, and energy efficiency.

Fredrik Svensson Fredrik Svensson is a researcher at Ericsson in the area of cloud and distributed systems. He has a MSc in Computer Science from the Blekinge Institute of Technology (Sweden). His research interest ranges from embedded to distributed systems and cloud technologies with focus on mission critical and real-time applications.

Raquel Mini is a researcher at Ericsson working in the Cloud Systems and Platforms area. She got her BSc, MSc, and PhD in Computer Science from Federal University of Minas Gerais (UFMG), Brazil. Before joining Ericsson in 2021, Raquel worked for more than 20 years as a Professor of Computer Science in Brazil. Her research addresses the area of sensor networks, IoT, ubiquitous computing, and cloud computing.

Luca Abeni is an associate professor at the Real-Time Systems Laboratory (ReTiS) of Scuola Superiore Sant'Anna (SSSA), Pisa since January 2017. He graduated in computer engineering from the University of Pisa 1998, and has been a PhD student at SSSA 1999-2002. From 2007 to 2016, he has been first researcher and then associate professor at University of Trento. Luca's main research interests are operating systems (real-time OSs, virtualisation technologies, networking stacks and architectures), scheduling algorithms (real-time CPU scheduling, chunk/peer scheduling in P2P systems), Quality of Service management, multimedia applications, and audio/video streaming.

Remo Andreoli has a MSc with honors in Computer Science from University of Pisa. He held a research scholarship from SSSA, during which he worked on differentiated performance mechanisms for NoSQL databases, earning a best student paper award at CLOSER 2021. He is currently a third-year PhD student at SSSA researching on performance optimization techniques for cloud infrastructures, and a software engineer at AWS Lambda.

Tommaso Cucinotta is Associate Professor at ReTiS of SSSA in Pisa. He has a MSc in Computer Engineering from University of Pisa, and a PhD in Computer Engineering from SSSA. His research interests include real-time scheduling for soft real-time applications, and predictability in infrastructures for cloud computing and NFV. He has been MTS in Bell Labs in Dublin (Ireland), investigating on security and real-time performance of cloud services. He has been a software engineer in AWS in Dublin (Ireland), where he worked on improving the performance and scalability of DynamoDB.