Fault Tolerance in Real-Time Cloud Computing

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

***Abstract*—This paper presents the Fault-Tolerant Real-Time Cloud (FTRTC) project that aims to design cloud computing infrastructures capable of hosting highly reliable and real-time applications. These applications are characterized by strict timing and reliability constraints, as well as critical failure scenarios. For instance, such requirements are commonly found in the context of Industry 4.0. We present a formalization of the problem of designing real-time cloud applications supporting an adjustable level of fault tolerance throughout their distributed execution in a cloud infrastructure. The contributions presented in this paper indicate important research directions when building cloud infrastructures able to supporting ultra-reliable real-time applications.**

***Index Terms*—Cloud Computing, Real-Time Computing, Fault Tolerance**

I. Introduction

Cloud computing has become an increasingly pervasive technology, and it has transformed a large part of the IT industry, as foreseen in the seminal paper by Armbrust and others [5]. Today, cloud technologies are mature enough to handle applications with precise performance requirements, but the evolution of hardware, software, and networks is making cloud computing appealing also for a new class of real-time workloads [9], [15], [25]. Specifically, applications characterized by strict timing and reliability constraints that must be respected under a multitude of conditions and failure scenarios cannot be deployed in contemporary cloud infrastructures [17]. These include a number of computing scenarios that are becoming increasingly interesting, for example, in the context of Industry 4.0, with cloud/edge-assisted operation of robots in automated factories, or for deploying complex AI-based features needed to augment the real-time control capabilities of future autonomous vehicles.

Traditional time-critical applications generally run on bare hardware and are designed as a set of periodic or sporadic sequential real-time tasks [7], [12], [23]. It is relatively simple to schedule such a straightforward task model and provide appropriate guarantees in regard to task performance requirements, even in the presence of interactions among tasks due to locking [27]. More complex task models can also be considered, like, for example, parallel tasks [10], [20], [26].

However, cloud-native applications [8] are characterized by a quite more complex and often distributed architecture. Differently from traditional monolithic applications, cloud-native ones are often composed of a number of different services or microservices [21] that may be deployed on the same or different machines. Hence, the first step in designing an infrastructure providing performance guarantees to cloud-native real-time applications is to build an accurate performance model for distributed applications composed of a number of components, which we refer to as *microservices* from here onwards. Since microservices are generally distributed over multiple machines, possibly spread over different availability zones, the failure of a single machine does not completely compromise the functionality of a microservice that remains available with a degraded performance level until the fault is fixed. In particular, the architecture is designed and provisioned so that some failures can be tolerated without affecting the application's operation. For real-time applications, this means that the system can be designed and dimensioned so that a controlled number of failures do not result in missed deadlines. Of course, this requires designing the microservice architecture appropriately so that it is possible to both analyze and control its temporal behavior.

This paper introduces the Fault-Tolerant Real-Time Cloud (FTRTC) project, aiming at providing an adjustable level of fault tolerance in a cloud environment designed to support real-time applications. The main challenges in implementing fault tolerance in cloud-native real-time applications are identified, alongside the most important building blocks that can be leveraged, to build a software solution suitable for FTRTC.

II. Problem and Definitions

To properly design solutions able to meet the need of FTRTC, it is important to formally define the problem it addresses, which is serving and scheduling multiple cloud-native applications with a desired degree of fault tolerance. This section provides some important definitions. From a practical standpoint, the project will provide implementations based on the popular Kubernetes framework.

1) Application Model: A FTRTC platform hosts a set of cloud-native applications, built by users as composition of independent, loosely coupled tasks. Such distributed applications can be represented in the form of Directed Acyclic Graph (DAG) topologies. More formally, a FTRTC platform hosts a set of $\mathcal{A} = \{\mathcal{A}^g\}_{g=1}^{n_\mathcal{A}}$ cloud-native applications, where $\mathcal{A}^g$ is a tuple $(\Gamma^g, \mathcal{E}^g, P^g, D^g)$. The DAG topology is specified by a subset of tasks $\Gamma^g \subseteq \Gamma$ and by a set of directed edges $\mathcal{E}^g \subseteq \Gamma^g \times \Gamma^g$. A task $\tau_i^g \in \Gamma^g$ represents a sequential activity that receives some data in input, processes it, and

979-8-3503-3902-4/23/$31.00

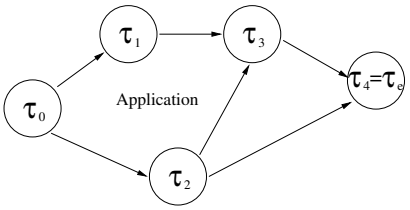

Fig. 1. Parallel real-time application modeled as a DAG of multiple tasks (the g have been removed to simplify the figure).

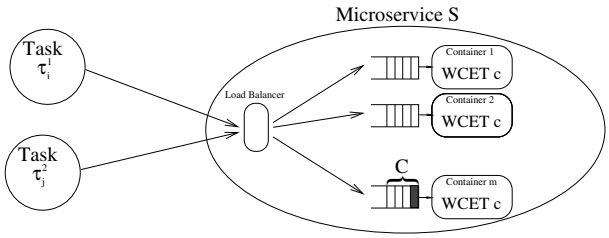

Fig. 2. Structure of a microservice S composed of m containers serving tasks τ_i^1 and τ_j^2 from two different applications, $\mathcal{A}^1$ and $\mathcal{A}^2$. Note that they are application-specific instances of the same functionality.

then generates some output. Each task τ_i^g is characterized by a Worst Case Execution Time (WCET), the maximum time for which a task needs to execute to generate an output after receiving a request.

Task-to-task communications in application $\mathcal{A}^g$ are represented by directed edges $(\tau_i^g, \tau_j^g) \in \mathcal{E}^g$, meaning that τ_i^g sends a message to τ_j^g at the end of its computations. More specifically, τ_i^g is said to be a *predecessor* of task τ_j^g, and τ_j^g is said to be a *successor* of τ_i^g. Therefore, for each activation of application $\mathcal{A}^g$, τ_j^g can be executed only after τ_i^g is terminated. Note that, since $\mathcal{A}^g$ is an acyclic graph, the transitive closure of $\mathcal{E}^g$ must be a partial order relation. Each task can have any number of predecessors and any number of successors (0, 1, or more). According to these precedence constraints, multiple tasks may be executed in parallel. For each application $\mathcal{A}^g$, there is exactly one task τ_0^g with no predecessors, called *input task*, and another task τ_e^g with no successors, called *exit task*. Finally, we assume each DAG to be completely connected, therefore for each node $\tau_i^g \in \Gamma^g$ exists a directed sequence of tasks (i.e. a path) from τ_0^g to τ_e^g which includes τ_i^g. Figure 1 shows an example of an application modeled as a DAG. Each application is activated when some data is available for processing at the input task τ_0^g. After the exit task τ_e^g finishes processing its input data, the application activation is concluded. Given a generic task τ_i^g, it becomes ready for execution after receiving data from all its predecessors $\{\tau_h^g \mid (\tau_h^g, \tau_i^g) \in \mathcal{E}^g\}$. Task τ_i^g then processes its input data and generate some output to be sent to all its successor tasks $\{\tau_j^g \mid (\tau_i^g, \tau_j^g) \in \mathcal{E}^g\}$. Similarly, each τ_j^g becomes ready for execution after receiving data from all its predecessors.

Real-time applications can be *periodic* or *sporadic*, depending on their activation patterns. An application is periodic if the time interval between two consecutive activations is constant and equal to the application period P^g, and is sporadic if the time interval between two consecutive activations is larger or equal than a minimum inter-arrival time P^g. Since the input task τ_0^g is the first task of the DAG, when it receives a request (and hence becomes ready for execution) the application is activated. For example, in the case of a periodic application, task τ_0^g receives its k^{th} request for execution at time $t_k^g = t_0^g + k \cdot P^g$, where t_0^g corresponds to the time of the first activation of the application. A real-time application $\mathcal{A}^g$ is also characterized by an end-to-end deadline D^g, so that if the application is activated (i.e., its input task τ_0^g receives a request and becomes ready for execution) at time $t_k^g = t_0^g + k * P^g$, then the application must finish (i.e. its output task τ_e^g must generate an output) before time $d_k^g = t_k^g + D^g = t_0^g + k * P^g + D^g$.

2) Cloud Model: The big difference between traditional real-time applications and the cloud-native real-time applications is that while the former are composed of tasks (threads or processes) that are statically deployed on physical machines using a Real-Time Operating System (RTOS), a task $\tau_i^g \in \Gamma^g$ of a cloud-native application $\mathcal{A}^g$ is executed by a *microservice* S_l. In a cloud scenario, the individual τ_i^g may be shared among multiple applications, therefore a set $\phi = \{\varphi_g\}_{g=1}^{n_\mathcal{A}}$ of DAG-to-microservice mapping functions $\varphi_g \colon \{0, ..., e\} \rightarrow \{1, ..., n_S\}$ is required to distinguish between the $\{\Gamma^g\}$ that may partially overlap. For instance, a task $\tau_i^g \in \Gamma^g$ is the i^{th} activity of application $\mathcal{A}^g$, and it is implemented by microservice S_l, where $\varphi_g(i) = l$. Every microservice S_l is formed by m_l containers and a load balancer distributing the input data to one of them. For example, Figure 2 shows a microservice S (the subscript l has been removed to simplify the figure) serving two tasks τ_i^1 and τ_j^2, where $\varphi_1(i) = \varphi_2(j) = l$. This means that the two tasks perform the same activity, but are part of two different applications, $\mathcal{A}^1$ and $\mathcal{A}^2$. When executing a task τ_i^g, a container of S_l takes at most a WCET c_l. Notice that the WCET characterizes a microservice S_l, therefore it is not application-dependent and can be associated with all the tasks executed by S_l. The various containers implementing a microservice S_l generally run on different machines, so that if one of them crashes or gets disconnected from the network, the containers running on different machines are still active and working. In this work, all the containers implementing S_l are assumed to be identical, so that c_l does not depend on the specific container that has been selected by the load balancer. Notice that this assumption applies to the amount of execution time needed by a container to serve a request, not to the real time after which the response is generated. Containers can suffer from interferences due to the other load running on the same machine, and can thus generate a response after different amounts of time — c_l only indicates the amount of time for which the container executes on a CPU core, without considering preemptions from the host OS or other containers running on the same CPU core, if any.

Figure 3 shows a cloud-native application $\mathcal{A}^g$ modeled as a DAG with its tasks served by microservices. A single S_l, which is composed of a load balancer and m_l containers

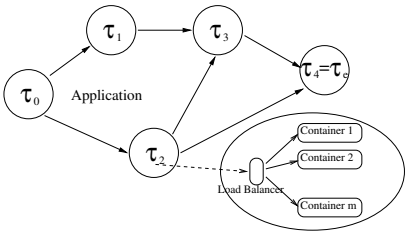

Fig. 3. Cloud-native real-time application composed of multiple parallel tasks executing on microservices (the g have been removed to simplify the figure).

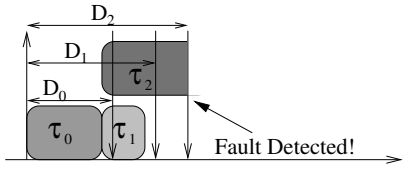

Fig. 4. Low-latency fault detection employed by FTRTC: τ_2 misses its partial deadline D_2, hence it is marked as faulty (the g have been removed to simplify the figure).

can execute tasks coming from different applications. In other words, microservices can be shared among multiple DAGs. Hence, they can simultaneously receive requests from different applications; as a result, the service of some request τ_i^g of an application $\mathcal{A}^g$ can be delayed by a queuing delay due to requests from other applications. Such delay can be estimated based on the maximum number of tasks concurrently submitting requests, the load balancing discipline and the model-of-computation of the containerized software.

3) Fault Model: The containers implementing a microservice can fail due to a number of reasons related to hardware or software problems (for example, the machine on which a container can crash or can be disconnected from the network, or the application running inside the container can crash). The goal of the fault-tolerance mechanism required in our envisioned FTRTC design is to ensure that even if one of the m_l containers implementing microservice S_l is down, all the applications using S_l get their requests serviced within the desired end-to-end deadlines. An easy way to accomplish this goal would be that when a task τ_i^g needs to be executed, two parallel instances of the task are actually started, on containers on different servers. However, such an approach would require to double the resource usage of all cloud-native applications, which is considered an excessively high price to pay, considering the negligible likelihood of the event that all said containers can be down at the same time.

In FTRTC, we aim at tolerating *transient* faults: if a container fails, at time t, it can be able to successfully serve some tasks' requests in the future (either immediately or after a maximum recovery time δ). Hence, when a failed container is detected, a new container is not immediately started to preserve the capacity as long as the service has containers to uphold the deadline guarantees. During such a period, the robustness to further failures is limited.

III. Fault Detection and Tolerance

FTRTC needs to support the cloud-native parallel application model described in Section II, providing fault tolerance for real-time applications. In this system, the application's tasks are executed on microservices/functions composed of a load balancer and multiple containers, as is common for cloud deployments. The presence of multiple (appropriately dimensioned) containers and the possibility to distribute task executions to these containers is used to tolerate failures.

1) Fault Detection: It is important to identify a tight upper bound to the time between a fault happening and the cloud platform detecting it (the so-called detection latency) in order to provide fault tolerance for a real-time application characterized by temporal constraints. Cloud instances are often monitored for failures based on periodic health-checks (e.g., ping) to detect failures. This results in a minimum detection latency when no traffic is exchanged, often in the range of one or a few seconds [1], [2], before marking an instance unhealthy. When a connection to a server suddenly drops while serving traffic, load balancers can usually react by tolerating a maximum number of these faults, then they can readily mark the instance unhealthy. Additionally, a potentially lower latency fault detection mechanism may use a request timeout implemented by the requester application or using a service mesh [28].

These mechanisms may be inadequate to support typical FTRTC use-cases, where it is essential, for example, to consider faulty also a container that is not managing to deliver a response by the foreseen WCET c_l, or a maximum *partial deadline* elapsed. An offline analysis may be used to compute partial deadlines within which all real-time tasks τ_i^g should finish. If a task does not complete within its partial deadline, the cloud platform can assume that its underlying container failed. Figure 4 (g has been removed to simplify the figure) depicts an example of early fault detection applying this mechanism to the application of Figure 1. Each task τ_i is assigned a partial deadline D_i, and τ_2 misses its deadline. After a time D_2 from the DAG activation, it is possible to detect a failure of the container serving τ_2. A second fault detection mechanism considers that a container is supposed to serve a task executing for at most a WCET c_l. Therefore, the FTRTC platform has a second chance of temporal fault detection, by monitoring the container's task execution and assuming that it failed when no output is produced after executing for c_l. Such monitoring can be implemented by detecting overruns in the container schedule, and is then local to a host. These two fault detection mechanisms can be combined to improve the system resilience: monitoring c_l allows detecting applications' misbehavior, whereas monitoring the tasks' partial deadlines allows detecting faults in the containers (or any issue due to unexpected interference from other containers).

2) Reacting to Faults: The easiest way to tolerate task execution failures consists of starting two copies of each task (executing in parallel in two containers hosted by different

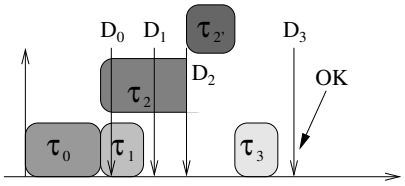

Fig. 5. Example of task re-execution: τ_2 re-executes after being marked as faulty (the g have been removed to simplify the figure).

physical machines) so that when one copy fails, the other one can still provide the correct result in time. This technique, however, consumes CPU time even when there are no failures (which is the most common situation). Therefore, replication should be used only when strictly needed to avoid wasting computational resources. As an alternative, it is possible to start a second copy of a task only when the task is not able to deliver its results in time, assuming the problem is that the container executing the task has failed or that it is delayed beyond what is accounted for. Of course, this is a good idea only when re-executing the task would not anyway lead to a deadline miss. That is, when there is enough time left before the deadline and one of the non-failed microservice's containers has enough spare time to re-execute the task.

Hence, it is crucial to analyze the tasks' schedulability and to understand, for each task, if there is enough time to re-execute a task when a failure is detected. Then, it is possible to resort to starting two parallel copies of a task only when the analysis shows that the two copies are really needed. When possible, containers' failures should be handled by retrying the execution of the served task to a different container on a different host[1]. As a simple example, Figure 5 shows that it is possible to tolerate a fault of task τ_2 (in Figure 4) without missing the partial deadline of task τ_3 by re-executing τ_2 on a different container (and hence respecting the DAG's deadline). If the makespan of the application's DAG execution without faults is smaller than the application's deadline D^g, then the containers have some spare time that can be used to avoid end-to-end deadline misses when a container fails. When trying to re-execute a failed task[2] τ_i^g on a different container, two different situations are possible:

1) The microservice S_l has a dedicated backup container which has enough spare time to re-execute τ_i^g.
2) The microservice S_l is composed of multiple containers, and at least one of them is not failed and has enough spare time to re-execute τ_i^g.
3) If none of the available containers of microservice S_l has enough spare time to re-execute the failed task, a new container has to be initialized.

Therefore, the cost of re-executing τ_i^g depends on how the failure is handled and on some additional assumptions; in general, the WCET for the re-execution of the task is considered to be longer than a first execution. In the first case above, re-executing the task incurs a smaller overhead, which can be almost negligible under the assumption of keeping a dedicated backup container to be used for re-executions. In the second case, one of the $m_l - 1$ remaining containers is used for the re-execution, so the number of requests which can be found in the queue by the re-execution needs to be increased accordingly. For the third case, instead, a new container is needed, the additional time needed to boot and initialize a new container instance (often known as *start-up latency*) must be considered. Such start-up latency can be substantial compared to application deadlines; in those cases, this alternative is not usable.

Summing up, an offline analysis (performed when the system is designed or when a new application is accepted) is used to decide which tasks are statically replicated (tasks for which two concurrent instances are started when the task is activated). Then, when the failure of a container executing a non-replicated task is detected at runtime, the system can check if such tasks can be re-executed on a different container without missing the application's deadline. If such a re-execution causes a deadline miss, the application's activation should be dropped (avoiding to waste computation time with other tasks of the application if there is no possibility of finishing in time). If an application is activated while some containers are failed or are unavailable, and it is not possible to guarantee the respect of its deadline, then the activation can be immediately dropped without starting any of the tasks. So, it is only when the number of containers has decreased sufficiently below m_l that not even a fault-free execution can be performed that the task is dropped. During this time interval, the service S_l is executed with higher risk.

3) Tasks Partial Deadlines and Critical Tasks: To avoid the long fault-detection latencies introduced by "traditional" ping-based mechanisms, failures can be detected by monitoring the amount of execution time consumed by a container when serving a task, or the container's response time. In the second case, a failing node can be detected in a much shorter time by assigning partial deadlines to tasks and monitoring communications. In practice, if a task τ_i^g does not send data to its successor tasks $\{\tau_j^g \mid (\tau_i^g, \tau_j^g) \in \mathcal{E}^g\}$ before its partial deadline, then the container executing τ_i^g is considered to be failed. If application $\mathcal{A}^g$ is activated at time t_k^g (and must finish before time $d_k^g = t_k^g + D^g$), its end-to-end deadline D^g is split into partial deadlines D_i^g associated to the various tasks τ_i^g. Note that the partial deadline D_e^g of the output task τ_e^g must be equal to D^g. If task τ_i^g is "activated in time" (i.e. τ_i^g activation time is smaller than $t_k^g + \max\{D_h^g \mid h : (\tau_h^g, \tau_i^g) \in \mathcal{E}^g\}$) but does not finish within time $d_{k,i}^g = t_k^g + D_i^g$, then τ_i^g execution has failed. As previously mentioned, when analyzing the behavior of an application $\mathcal{A}^g$, it is necessary to check if a failed task can be re-executed without causing a deadline miss. This can be done both to provide a runtime check and to provide design-time properties:

1) At runtime, when a container's failure is detected, it is

[1]Here, "when possible" means "when it is possible to re-execute the task on a different container *without missing the application's deadline*".

[2]In this context, "failed task" means "tasks which was executed on a failed container".

necessary to check if there is time to re-execute the task executed by the failed container without violating the deadline. This check has to be performed considering available containers.

2) To ensure the desired fault tolerance properties, at design time, it is necessary to provide guarantees that the system can correctly serve $\mathcal{A}^g$ within the deadline in the presence of a given maximum number of failures (for example, tolerating a failing task execution per microservice). This must be checked through an offline analysis.

The design problem in item 2 allows for identifying "critical" tasks for which it is not guaranteed in advance that there will be time for re-execution in case of failure. Critical tasks should be statically replicated. Otherwise, a task failure could only be handled by dropping the activation.

IV. Allocating Resources to Containers

The analysis described at the end of Section III is similar to the schedulability analysis typically performed to analyze and design real-time systems. However, traditional real-time analysis must be modified to account for microservices and containers execution/scheduling. Given an application $\mathcal{A}^g$, there are two schedulability questions:

- For which tasks a re-execution on a different container would result in a missed deadline, if their serving container fails?
- In which conditions a task can be re-executed without missing a deadline, if its serving container fails?

While the answer to the first question does not depend on how the containers are scheduled, to answer the second question it is necessary to know how resources are allocated to the various containers. In particular, the containers composing a microservice can execute on dedicated CPU cores (so that a container does not suffer preemptions by other containers) or can share the CPU with other containers (in this case, a CPU scheduler decides which containers are executed at a given time, and should use an appropriate scheduling algorithm to limit the interferences between containers). In any case, microservices are shared by multiple applications; hence, the temporal behavior of an application $\mathcal{A}^g$ also depends on the other applications using the same microservices. This fact can be accounted for in the analysis without having to consider all the details of every application running in the cloud, but only $\mathcal{A}^g$. It can be done by inflating the execution time c_l of each task τ_i^g to take into account the queue delay, defined as C_l, and the corresponding re-execution time C'_l.

Based on this, it is possible to check when there surely is no time to re-execute a task, independently on how the containers are scheduled or how resources are allocated to them. In particular, the function `critical(`τ, t`)` returns `true` if task τ cannot be re-executed in case of failure at time t, and returns `false` otherwise. By considering the latest possible time when a failure of task τ_i^g can be detected, it is possible to identify *critical* tasks, which are tasks for which it is not guaranteed that there is time for re-execution in case of failure. For example, assume that container is considered to have failed if it tries to execute for longer than its WCET. In this case, `critical(`τ_i^g, t`)` is `true` if $C_l + C'_l > D_i^g$. If all containers are statically assigned their CPU cores (and are hence not scheduled), then a non-critical task is guaranteed to always have time for being re-executed as soon as there is at most one failed container per microservice.

If, instead, containers are not assigned whole CPU cores but are scheduled, then to guarantee that the re-execution of a task does not result in a missed deadline, it is necessary to use an appropriate container scheduling algorithm. FTRTC considers two possible scheduling strategies: one based on the *Hierarchical Constant Bandwidth Server* (HCBS) [3], [11] and one based on fixed priorities. The HCBS is a reservation-based scheduling algorithm that has the advantage of supporting the Compositional Scheduling Framework [13], [29]. Hence, it allows re-using a lot of results from real-time literature (which investigated at length hierarchical scheduling based on resource reservations). An implementation in the Linux kernel (based on a modification of the `SCHED_DEADLINE` policy [22]) is already available, and some prototype patches are available to support it in Kubernetes [16]. For applications needing very low latencies, such as a cloud Radio Access Network (cRAN) [4], it might make sense to schedule the containers according to fixed priorities and not use CPU reservations. In this case, the schedulability of an application can be checked without assigning partial deadlines to tasks but by using a different analysis based on self-suspending tasks [6]. Such analysis can be extended to account for failing containers, replicated tasks, and tasks' re-execution (when needed).

V. Related Work

Fault tolerance in real-time systems has been studied for a long time [18] and has been generally implemented through specialized CPU scheduling algorithms. Later works considered fault tolerance in real-time clouds [24], but did not consider cloud-native applications and the microservices model, which is supported by FTRTC.

Some works support fault tolerance in cloud environments by scheduling backup copies of failed tasks. For example, QAFT [32] schedules primary and backup copies of tasks on different cloud nodes, considering different QoS levels for the tasks and different speeds for the cloud nodes. However, it only considers independent aperiodic real-time tasks (and not parallel applications), and it does not consider a microservice model as FTRTC does. FESTAL [31] also considers sets of independent real-time tasks and implements fault tolerance by scheduling backup copies of failed tasks. Migratable Virtual Machines are used to move primary and backup copies between physical hosts so that faults are tolerated without overallocating resources. EFTR [19] is a similar algorithm that takes energy consumption into account. FTRTC does not use VMs or migrations of tasks between physical hosts because

they introduce overheads and delays that are not compatible with the deadlines to be respected.

Other, more recent, works [14], [30] base the fault tolerance mechanisms on containers, using Kubernetes to manage them and improve the Kubernetes scheduler and fault-detection mechanism to react to failures in a shorter time. However, they do not explicitly consider real-time applications and deadline constraints.

VI. Conclusions and Future Work

This paper described the design of a fault-tolerant real-time cloud computing platform, starting by identifying the challenges that need to be addressed and the fundamental core requirements to be satisfied by possible mechanisms to be used in said context. The paper introduced formally the problem of fault-tolerant execution of real-time cloud applications, and discussed the difficulties in ensuring predictability in execution of distributed cloud-based applications modeled as DAGs. It also discussed the need for providing low-latency fault detection, e.g., by monitoring the respect of the tasks' partial deadlines, as well as the need for tolerating a controlled number of faults by using microservices composed of multiple containers and by re-executing failed tasks on different containers. As a future work, the presented architecture will be implemented based on Kubernetes, and its properties will be tested through an extensive set of experiments.

References

[1] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973, 2018.

[2] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019.

[3] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *SIGBED Review*, 16(3):33–38, November 2019.

[4] Luca Abeni, Tommaso Cucinotta, Balázs Pinczel, Péter Mátray, Murali Krishna Srinivasan, and Tobias Lindquist. On the use of linux real-time features for ran packet processing in cloud environments. In Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin, editors, *High Performance Computing. ISC High Performance 2022 International Workshops*, pages 371–382, Cham, 2022. Springer International Publishing.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Federico Aromolo, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Event-driven delay-induced tasks: Model, analysis, and applications. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 53–65, 2021.

[7] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[8] Sandro Brunner, Martin Blöchlinger, Giovanni Toffetti, Josef Spillner, and Thomas Michael Bohnert. Experimental evaluation of the cloud-native application design. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing*, pages 488–493, Dec 2015.

[9] Kun Cao, Shiyan Hu, Yang Shi, Armando Walter Colombo, Stamatis Karnouskos, and Xin Li. A survey on edge and edge-cloud computing assisted cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 17(11):7806–7819, Nov 2021.

[10] Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[11] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in OpenStack for NFV services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.

[12] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[13] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, September 2009.

[14] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. Latency-aware industrial fog application orchestration with kubernetes. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 164–171, 2020.

[15] Erik Ekudden. Five network trends: Towards the 6g era. *Ericsson Technology Review*, 2021(9):2–10, Sep. 2021.

[16] Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. Rt-kubernetes: Containerized real-time cloud computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 36–39, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014.

[18] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(3):272–284, 1997.

[19] Pengze Guo, Ming Liu, Jun Wu, Zhi Xue, and Xiangjian He. Energy-efficient fault-tolerant scheduling algorithm for real-time tasks in cloud-based 5g networks. *IEEE Access*, 6:53671–53683, 2018.

[20] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *2010 31st IEEE Real-Time Systems Symposium*, pages 259–268, 2010.

[21] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

[22] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[23] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[24] Sheheryar Malik and Fabrice Huet. Adaptive fault tolerance in real time cloud computing. In *2011 IEEE World Congress on Services*, pages 280–287, 2011.

[25] Peter O'Donovan, Colm Gallagher, Kevin Leahy, and Dominic T.J. 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*, 110:12–35, 2019.

[26] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Trans. on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[27] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[28] Rahul Sharma and Avinash Singh. *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes*. Apress, 2019.

[29] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004.

[30] László Toka. Ultra-reliable and low-latency computing in the edge with kubernetes. *Journal of Grid Computing*, 19(3):1–23, 2021.

[31] Ji Wang, Weidong Bao, Xiaomin Zhu, Laurence T. Yang, and Yang Xiang. Festal: Fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds. *IEEE Transactions on Computers*, 64(9):2545–2558, 2015.

[32] Xiaomin Zhu, Xiao Qin, and Meikang Qiu. Qos-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters. *IEEE Transactions on Computers*, 60(6):800–812, 2011.