Design-time Analysis of Time-Critical and Fault-Tolerance Constraints in Cloud Services

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

***Abstract*—This work presents a model for designing and deploying time-critical, cloud-native applications under fault conditions. Our model considers the interactions and interferences among service components, as well as the possible occurrence of faults. Given a set of to-be-deployed applications with precise temporal constraints and a pre-defined configuration of the service components, we devised an optimizer to verify at design time if the cloud services guarantee compliance with the timing constraints while minimizing the resources needed to achieve fault tolerance.**

***Index Terms*—Time-Critical Cloud, Fault Tolerance, Optimization**

I. Introduction

The latest developments towards performance predictability in virtualized environments [3], [5], [9], coupled with the recent advancements in 5G technologies, have exponentially increased the interest in hosting innovative time-critical applications on cloud-based infrastructures [6], [8]. Time-criticality requires that strict temporal requirements must be met regardless of the other workloads deployed within the cloud system, as well as the possible occurrence of hardware/software/network failures [2]. The inability to respect temporal constraints may cause undefined behaviors in the system, which ultimately can lead to unexpected if not catastrophic, consequences. Simply dedicating a single-tenant infrastructure for a time-critical system results in a huge waste of resources for the cloud provider, as it does not adhere to the traditional cloud principle of consolidation [1]. There have been multiple works [4], [7], [10] focusing on resource management with fault-tolerance in a cloud-based infrastructure. However most of them do not focus on worst-case response time guarantees, neither they take into account interferences of co-deployed, time-critical applications. Moreover, in a time-critical context, a deadline miss is considered a failure too; this is not explored in related works.

This paper models a time-critical, cloud-native application as a composition of microservices in a Directed Acyclic Graphs (DAG) topology, with associated precise end-to-end temporal constraints. The individual microservices may be shared among a multitude of applications, causing resource contentions at runtime. Fault tolerance is implemented through replication and resubmission. To determine if the to-be-deployed, time-critical applications can be admitted to the cloud infrastructure without violating the temporal and fault-tolerance constraints, we devised an analysis tool formulated as a Mixed-Integer Linear Programming (MILP) program. Additionally, our optimizer minimizes the use of replication, since it may result in a waste of computational resources.

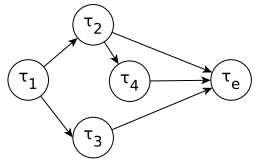

Fig. 1: Parallel real-time application modeled as a DAG of multiple tasks (the g superscripts have been removed to simplify the figure).

II. Problem Formulation

This section formally presents the application, the reference cloud infrastructure, and the fault model for time-criticality.

1) Application Model: A cloud platform hosts a collection of $n_{\mathcal{A}}$ cloud-native applications $\mathcal{A} = \{\mathcal{A}^g\}_{g=1}^{n_{\mathcal{A}}}$ which are built by composing independent and loosely coupled tasks. A cloud-native application $\mathcal{A}^g$ can be represented as a Directed Acyclic Graph (DAG) characterized by a set of tasks $\Gamma^g = \{\tau_i^g\}_{i=1}^{n_g}$, and by a set of directed edges $\mathcal{E}^g \subseteq \Gamma^g \times \Gamma^g$. A task $\tau_i^g \in \Gamma^g$ is a sequential activity that receives some input data, processes it, and then generates output data. A directed edge $(\tau_\mu^g, \tau_i^g) \in \mathcal{E}^g$ represents a logical communication link between two tasks, such that τ_μ^g sends its output data to τ_i^g. More specifically, τ_μ^g is a predecessor of task τ_i^g, and τ_i^g is a successor of task τ_μ^g. Figure 1 depicts a sample application. Tasks are synchronously "activated", meaning that each task τ_i^g waits for all its predecessors $Prev_i^g = \{\tau_\mu^g \in \Gamma^g : (\tau_\mu^g, \tau_i^g) \in \mathcal{E}^g\}$ to generate their outputs, before processing them as input. Once the computations are finished, the resulting output is then propagated to all the successors $Next_i^g = \{\tau_\lambda^g \in \Gamma^g : (\tau_i^g, \tau_\lambda^g) \in \mathcal{E}^g\}$, concluding the task activation. Therefore, set $\mathcal{E}^g$ defines *explicit dependencies* between contiguous tasks in $\mathcal{A}^g$. It is also possible to infer *implicit dependencies*: two non-contiguous tasks $\tau_i^g, \tau_j^g \in \Gamma^g$ have an implicit dependency if there is at least a directed path, inferred from $\mathcal{E}^g$, connecting the two. Each application $\mathcal{A}^g$ has exactly one task with no predecessors, called the *input task*, which coincides with $\tau_1^g \in \Gamma^g$. Analogously, there is exactly one task with

no successors, called the *output task*, which coincides with $\tau_e^g \in \Gamma^g$. An application activates when τ_1^g receives input data from an external source (i.e., a user request) and concludes after τ_e^g generates an output. Every application corresponds to a completely connected DAG so that for every node $\tau_i^g \in \Gamma^g$ there is always at least a sequence of tasks (i.e a directed path) between input task and output task which includes τ_i^g. A single application activation may trigger multiple concurrent sequences of task activations. In the context of a time-critical use case, each application $\mathcal{A}^g$ is characterized by a minimum inter-arrival period P^g, which specifies the minimum time interval between two consecutive activations of application $\mathcal{A}^g$, and an end-to-end deadline D^g (*relative deadline*), assumed to be $D^g \leq P^g$. An application activation must complete within D^g time units to be considered successful.

2) Cloud Model: A cloud platform offers a set of microservices $\mathcal{S} = \{S_l\}_{l=1}^{n_S}$, each dedicated to the execution of a certain activity. A microservice S_l is implemented by a pool of M_l workers, normally cloud instances such as containers or virtual machines, and a load balancer to distribute the workload among them. The execution of a given task τ_i^g by a worker in microservice S_l takes at most a worst-case execution time (WCET) c_l. More specifically, it takes into account the maximum amount of time required to process the input data and generate an output, regardless of the type of request, the application submitting it, and the data size. Then, the produced output data is sent to subsequent microservice(s), as instructed by the application topology to which τ_i^g belongs.

A microservice may be shared between applications: two tasks belonging to different applications may be implemented by the same microservice. Therefore, the execution of a task invocation may experience a *queuing delay*. The worst-case delay Q depends on the maximum number of tasks simultaneously submitting requests, the load balancing discipline, and the number of workers dedicated to the microservice. A cloud provider interested in providing temporal guarantees for co-located time-critical applications should plan the capacity of its infrastructure taking into account, for every microservice S_l, the overall worst-case response-time (WCRT) $C_l = c_l + Q$ for an invocation.

3) Fault Model: A faulty worker is detected if a task execution exceeds a maximum task deadline D_i^g, called a *relative partial deadline*. Therefore, every τ_i^g invocation must finish within D_i^g time units to guarantee a successful activation of the task. If all task activations are successful, the overall application activation is considered successful too. We model two possible ways to handle a fault: 1) *static replication*: send two task invocation requests in parallel to two different workers; 2) *task invocation re-submission*: re-submit the failed invocation for execution and let the load balancer choose another worker. Each method implies a different WCRT, namely C^{repl} for static replication, and $C_l'' = C_l + C_l'$ for re-submission. The latter is due to the fact that the *total* WCRT for a task activation must take into account both the *first* execution (i.e., C_l) and the re-execution (i.e., C_l'). A naïve fault model would assume that *all* task activations of the application may experience a fault condition, but this is highly unlikely, if not impossible, to happen in practice. Therefore, we assume that at most F failures occur during an application activation. Under these circumstances, it is convenient to also consider the case where a task activation did not fail because the expected F failures happened previously and affected the sequences of task activations reaching up to task τ_i^g. Finally, our model assumes that a second fault, located in the same microservice, is very unlikely to happen in the period of time prior to the recovery of the first faulty worker. This indirectly implies that when a task activation fails due to a transient fault, its re-execution will not fail again.

III. Approach

For each application $\mathcal{A} \in \mathcal{A}^g$: i) we introduce a relative partial deadline D_i^g for each task $\tau_i^g \in \Gamma^g$; ii) we ensure that the sum of partial deadlines over all end-to-end paths in the application topology Γ^g does not exceed D^g; iii) we ensure that the individual relative partial deadlines D_i^g cannot be missed for each task τ_i, regardless of possible microservice-level interferences and fault conditions at run-time. The partial deadline D_i^g directly depends of the partial deadlines on the predecessors of τ_i^g in the topology $\mathcal{E}^g$, but also on the fault tolerance method chosen for each task, and the associated WCRT. A naïve approach to achieve fault tolerance is to use static replication for every task activation. However, this consumes a lot of resources and it may require an oversized infrastructure, depending on the number of overlapping tasks between applications. Using re-submission for every task activation may be unfeasible by default in case of a tight end-to-end deadline requirement that does not leave enough spare time. Therefore, we devised a MILP-based offline analysis tool to configure the optimum fault tolerance method for every task of a set of interfering applications, so that each application meets its given end-to-end deadline constraint, despite the occurrence of a number of faults F. Static replication is used only for the minimal set of tasks whose failure would otherwise violate the end-to-end deadline D^g. Our optimizer computes every possible chain of faults during application activation, and then assigns an optimal partial deadline for every task τ_i^g within which its invocations must finish to guarantee a successful activation (recall Section II-3). In this way, the optimizer ensures that an application activation does not violate the end-to-end deadline D, regardless of interferences and the number of transient faults F. Under the assumption that a given end-to-end deadline D^g is not ill-posed (i.e., application $\mathcal{A}^g$ should be able to fulfill D^g under a no-interference, no-failure condition), the optimizer cannot find a feasible solution only if a subset of microservices is unable to accommodate all the to-be-deployed applications (i.e. the WCRTs are too long due to the queuing delay).

IV. Experimental Results

This section is dedicated to the evaluation of the optimization approach described in Section III, under the assumption of an underlying infrastructure capable to accommodate all

	Given Parameters		Inferred Parameters		
	c_l	M_l	C_l^{repl}	C_l	$C_l + C_l'$
S_1	5	2	20	10	30
S_2	15	4	30	15	45
S_3	25	2	100	50	150
S_4	30	3	90	60	120
S_e	10	4	20	10	30

TABLE I: Characteristics of the microservices. Since we assume a one-to-one task-to-service mapping, tasks and microservices can share the same indexing.

the microservices. The experiment have been performed using the application depicted in Figure 1 with every task activation realized by a dedicated microservice. This means that task τ_i^g activations are executed by microservice S_i. Interferences are modeled by duplicating the very same application topology 4 times, instead of using applications with different topologies. Table I describes the given characteristics of each microservice, in terms of WCETs and number of workers, as well as the estimated WCRTs. Since the application topologies and requirements are all the same, we can focus the analysis on one of the 4 application deployments, while the other 3 create interference. The g superscripts k subscripts are omitted for readability. For our experiments, we used a Gurobi solver version 9.5.1. The validity of a given end-to-end deadline for an application activation depends on the sum of the WCRTs on the critical path, simply called makespan. In a failure-free (i.e., $F = 0$) and interference-free (i.e., $n_{\mathcal{A}} = 1$) infrastructure with the characteristics in Table I, the makespan is $c_1+c_2+c_4+c_e = 60$ time units. Removing the interference-free assumption (i.e., $n_{\mathcal{A}} = 4$), the makespan becomes $C_1+C_2+C_4+C_e = 95$ time units. Therefore, an application activation with $D < 95$ implies an inaccurate assessment of the application's behavior in terms of the estimated WCETs c_l: the application requirements are beyond the capabilities of the microservices. Notice that the presented example cannot guarantee a successful activation within $D = 95$ under fault conditions and with the microservice characteristics of Table I. The reason is that there is no spare time between the end-to-end deadline D and the partial deadline d_e of output task τ_e, thus re-submission is not possible. Moreover, setting all task activations as statically replicated implies a makespan of $C_1^{repl} + C_2^{repl} + C_4^{repl} + C_e^{repl} = 160$ time units, which violates $D = 95$ by 65 units of time. This is due to the fact that the number of workers M_l is not enough to ensure fully parallel static replication, which requires $2 \cdot n_{\mathcal{A}} = 8$ workers per microservice. Such considerations justify the need for an analysis tool to verify if the current infrastructure is able to guarantee temporal and fault-tolerance constraints.

Figure 2 depicts a series of experiments using our optimizer with different end-to-end deadlines. It depicts the minimum amount of statically replicated task executions required to deploy the application in Figure 1, alongside 3 other interfering

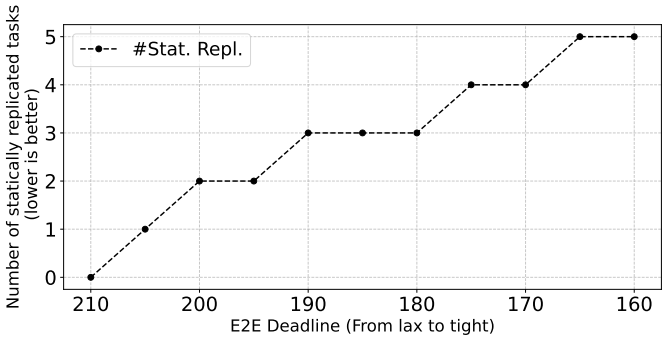

Fig. 2: Number of statically replicated tasks (black dashed line) as the end-to-end deadline shrinks. Number of tolerable transient faults is $F = 3$, and number of deployed applications is $n_{\mathcal{A}} = 4$.

instances of the same application, with no deadline violation. The number of transient faults that may happen during activation is $F = 3$. The x-axis shows the different end-to-end deadlines. As we can see, the number of statically replicated tasks increases as the end-to-end deadline shrinks, and the assignment problem becomes unfeasible for deadlines earlier than 160.

REFERENCES

[1] Rajkumar Buyya, Christian Vecchiola, and S Thamarai Selvi. *Mastering cloud computing: foundations and applications programming*. Newnes, 2013.

[2] Mehdi Nazari Cheraghlou, Ahmad Khadem-Zadeh, and Majid Haghparast. A survey of fault tolerance architecture in cloud computing. *Journal of Network and Computer Applications*, 61:81–92, 2016.

[3] 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.

[4] Zengpeng Li, Huiqun Yu, Guisheng Fan, and Jiayin Zhang. Cost-efficient fault-tolerant workflow scheduling for deadline-constrained microservice-based applications in clouds. *IEEE Transactions on Network and Service Management*, pages 1–1, 2023.

[5] Bruno Ordozgoiti, Alberto Mozo, Sandra Gómez Canaval, Udi Margolin, Elisha Rosensweig, and Itai Segall. Deep convolutional neural networks for detecting noisy neighbours in cloud infrastructure. *COSTAC 2017*, page 59, 2017.

[6] 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.

[7] Roozbeh Siyadatzadeh, Fatemeh Mehrafrooz, Mohsen Ansari, Bardia Safaei, Muhammad Shafique, Jörg Henkel, and Alireza Ejlali. Relief: A reinforcement learning-based real-time task assignment strategy in emerging fault-tolerant fog computing. *IEEE Internet of Things Journal*, pages 1–1, 2023.

[8] Márk Szalay, Péter Mátray, and László Toka. Real-time task scheduling in a FaaS cloud. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 497–507, Sep. 2021.

[9] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 39–48, 2011.

[10] Guangshun Yao, Qian Ren, Xiaoping Li, Shenghui Zhao, and Rubén Ruiz. A hybrid fault-tolerant scheduling for deadline-constrained tasks in cloud systems. *IEEE Transactions on Services Computing*, 15(3):1371–1384, 2022.