Contract-Based Quality-of-Service Assurance in Dynamic Distributed Systems

Lea Schönberger*, Susanne Graf†, Selma Saidi*, Dirk Ziegenbein‡, Arne Hamann‡
*TU Dortmund University, Dortmund, Germany, {firstname.lastname}@tu-dortmund.de
†VERIMAG, Grenoble, France, susanne.graf@imag.fr
‡ Robert Bosch GmbH, Renningen, Germany, {firstname.lastname}@de.bosch.com

***Abstract*—To offer an infrastructure for autonomous systems offloading parts of their functionality, dynamic distributed systems must be able to satisfy non-functional quality-of-service (QoS) requirements. However, providing hard QoS guarantees without complex global verification that are satisfied even under uncertain conditions is very challenging. In this work, we propose a contract-based QoS assurance for centralized, hierarchical systems, which requires local verification only and has the potential to cope with dynamic changes and uncertainties.**

***Index Terms*—quality of service, autonomous systems, dynamic distributed systems, contracts, metric interval temporal logic**

I. Introduction

In various fields, a trend has been emerging towards highly distributed architectures, so called *dynamic distributed systems*, e.g., on-demand cloud platforms or road-side units on smart intersections. Such systems allow autonomous systems, e.g., (partially) autonomous vehicles or modern driving assistance systems, to offload time-critical and computation-intensive functionalities for remote execution. These functionalities typically come with a *quality of service (QoS) requirement*, which can be either functional, for instance regarding the computational correctness and quality of the result, or non-functional, e.g., the satisfaction of an end-to-end deadline. However, since the non-functional QoS serves as a basis for the functional QoS of sophisticated applications, this work aims to provide an assurance of the former, aiming to provide an infrastructure for autonomy.

In order to guarantee a QoS in terms of time, the system's resource management is crucial. Dynamic distributed systems are frequently designed in a decentralized way, e.g., as multi-agent systems, requiring a consensus between different system components. Recently, a decentralized architecture and consensus protocol was proposed by [1], a decentralized monitoring-based approach by [2]. While both approaches [1] and [2] are only applicable to provide weakly-hard QoS guarantees, [3] proposes a centralized, hierarchical architecture, which uses complex timing analyses to provide hard QoS guarantees for all applications passing an admission control.

Unlike existing works, we aim at providing hard QoS guarantees with low verification complexity, preferring a centralized, hierarchical architecture over a decentralized one. More precisely, we introduce *contracts*, i.e., formal agreements between an application and the system, similar to the concept of service level agreements in cloud computing (see e.g. [4], which are based on logic constraints on the behavior of different system components and are assured automatically by dynamic reconfiguration.

In the following, we introduce the considered system architecture in Sec. II and give a short introduction into the logic used in Sec. III, before we construct a core contract in Sec. IV. We conclude this work in Sec. V with a discussion about the benefits of our contract-based QoS assurance approach, its potential when dealing with uncertainties, and its evolution under dynamic changes.

II. System Model

We consider a centralized, hierarchical distributed system with an admission control and a global notion of time.

A. Architecture

The system is separated into three hierarchical layers with different responsibilities. On the top-layer, the *resource manager* is situated, which is in charge of the admission control, i.e., it decides if an application is accepted for execution or rejected. For this purpose, it transforms functional requirements of an application into system-specific constraints that are handed over to the *resource agents* situated on the medium-layer. Each resource agent administrates one of multiple heterogeneous computation and communication *resources* located on the base-layer and is responsible for enforcing the constraints imposed by the resource manager, e.g., by adjusting the scheduling. Depending on the actual system, the medium-layer may be implemented on the base-layer.

B. Resources

The system's base layer consists of a set $\mathcal{R}$ of heterogeneous and independent resources[1], each one being of a specific resource type, e.g., a particular computation platform or transmission technology. On each resource, a specific resource allocation mechanism suitable for the respective resource type is assumed to be implemented.

C. Applications

The set of executed applications is not fixed at design time. Applications can send *join requests* to the resource manager in order to be executed and, once their execution is completed, leave the system again. However, in the following, we restrict

[1]For the sake of simplicity, we assume resources to be independently assignable. This is realistic for many distributed systems, where communication is decoupled from computation via adequately dimensioned buffers.

our considerations to the case that applications stay in the system rather long-term, such that the rate of change does not have a direct impact on the processing of join requests.

Each application $a = (\mathcal{T}, P, Q)$ consists of a partially ordered set $\mathcal{T}$ of tasks, represented as a directed acyclic graph (DAG) termed *task graph*. The partial order specifies dependencies between tasks, i.e., if for any tasks τ_1, τ_2, it holds that $\tau_1 < \tau_2$, then τ_2 is dependent on τ_1 and requires the result (e.g., a computation result or a completed data transmission) of τ_1 as an input. Application instances are assumed to be released periodically, i.e., with an inter-arrival time P. The *high-level QoS requirement* Q of an application is an end-to-end deadline, i.e., a time interval, which is dimensioned in such a way that further constraints such as an end-to-end sensitivity margin, i.e., a percentage indicating by how much the end-to-end deadline may be overshot, etc. have already been factored in. Moreover, we assume that P can be shorter than the end-to-end deadline, i.e., multiple instances of the application may be executed simultaneously.

Each task $\tau \in \mathcal{T}$ is a tuple $\tau = (C, \varrho, q)$. C is a set of time estimates, where each time estimate c refers to the worst-case execution/transmission time of τ on a particular resource type, which is eligible for its execution/transmission, e.g., a specific CPU architecture or communication technology. These time estimates are assumed to be obtained in advance applying any method such as, e.g., profiling or benchmarking, and to be provided by the application designers. Each task is time-triggered, i.e., it is released according to a *release offset* ϱ with respect to the start of the application's period. The release offset is not known in advance, but is determined by the resource manager during the admission control process and remains unchanged, once an application has been admitted for execution. This corresponds to the logical execution time (LET) paradigm [5] for decoupling dependencies between tasks and implies that, if for two tasks τ_1, τ_2 it holds that $\tau_1 < \tau_2$, task τ_2 must not start before the execution of task τ_1 has been completed. Along with the release offset, a local deadline q, i.e., a time interval, termed *low-level QoS constraint*, is associated with each task during the admission control process.

D. Admission Control

In the admission control process, it is evaluated for each executable application[2], which QoS can be guaranteed by the system. This *QoS offer* $\bar{Q}$ is communicated to the application, which is free to accept or decline it depending on its requirements. To derive the QoS offer, three steps are followed, as illustrated in Fig. 1:

(i) *Decomposition:* The application's high-level QoS requirement Q is *decomposed* into per-task low-level QoS constraints by the resource manager using a *decomposition algorithm*, which can be chosen according to global optimization goals. Each low-level QoS constraint (together with a time estimate c and the period P) is forwarded to the resource agent maintaining the related resource. For the rest of this work, we consider an abstract decomposition function:

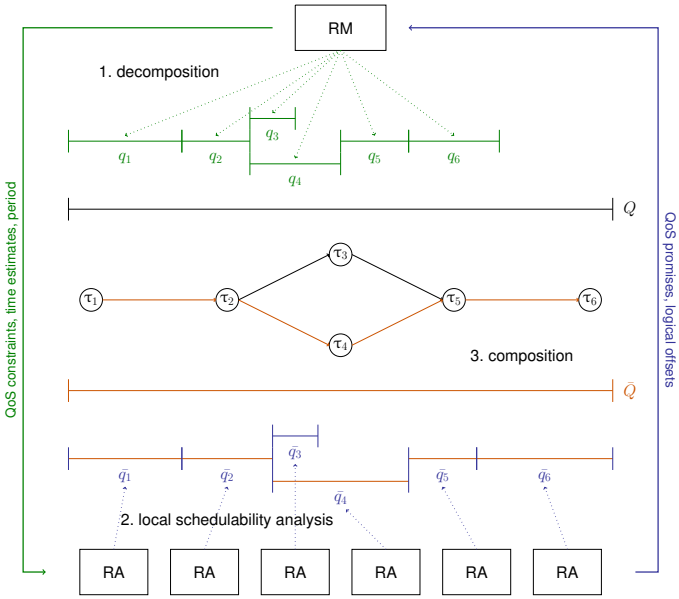

Fig. 1. A global picture of the admission control process.

Definition 1 (Decomposition Functions)**.** *Decomposition functions are defined as functions* $\text{dec}(\mathcal{T})$ *deriving the task parameters* (q, ϱ) *such that the following properties are satisfied: For each task* τ_i *it holds that* $q_i \geq c_i$*. For any two tasks* τ_i, τ_j *with* $\tau_i < \tau_j$ *regarding the partial order of* $\mathcal{T}$*, it holds that* $\varrho_i + q_i \leq \varrho_j$*.*

(ii) *Local Schedulability Analysis:* Each resource agent performs a local schedulability analysis[3] to determine whether q can be satisfied. In the course of the analysis, it derives a *low-level QoS promise* $\bar{q}$, which is either q or the next best latency it can guarantee for the execution/transmission of τ.

(iii) *Composition:* Based on all low-level QoS promises, the resource manager uses a given *composition theory* to compute the high-level QoS offer $\bar{Q}$. The composition theory is imposed at system design time and therefore serves as a ground truth, while each implemented decomposition algorithm must be explicitly designed against the background of the composition theory in order to be compatible. In this work, we assume the composition to be performed by the following abstract composition function:

Definition 2 (Composition Function)**.** *For an application* a*, the composition function is defined as* $\text{comp}(\mathcal{T}) := maxpath(\mathcal{T})$*, where* $maxpath(\mathcal{T})$ *returns the length*

[2]An application is executable on the system if for each task τ a suitable resource (of a type for which a time estimate is provided by τ) exists. In this case, the task graph is mapped to the system, i.e., each task is assigned to one specific resource. Please note that the DAG as mapped to the system may differ from the original task graph in terms of parallelism, depending on the available resources. Due to space limitations, we henceforth assume a correct mapping of the task graph to be given, so that whenever we speak of $\mathcal{T}$, we refer to the DAG as mapped to the system.

[3]We make the assumption that for each resource, a suitable schedulability analysis method is chosen, which delivers correct results.

of the maximum path (in terms of the low-level QoS promises $\bar{q}_\tau$ and taking into account the release offsets) through the task graph or all $\tau \in \mathcal{T}$.

E. Execution Phase

To keep its low-level QoS promise regarding an application, each resource agent reserves a so-called *budget* b on the resource, i.e., a certain amount of resource service required to perform the execution or transmission of a task τ within an upper-bounded response time. This can be, e.g., execution time on a computation resource and a number of transmission slots or bandwidth on a communication resource. The budget is said to be *consumed* during the execution or transmission of a task and to be *replenished* after each period of the task. Accordingly, it is represented as a real-valued signal, for which must hold that $c \leq b \leq q$.

III. Metric Interval Temporal Logic

In order to assure that the considered system is able to provide the offered QoS to an executed application, we consider the result of a successful admission as a formal contract engaging both, the application and the system. To formally express the responsibilities of each contracting party, we formulate constraints using a Metric Interval Temporal Logic (MITL), from which monitors can be automatically derived, similar to [6].

The state of a system can be described by state variables $x_1, \ldots, x_n$, whose values evolve over a time domain $\mathbb{T}$. Accordingly, the state space of the system is defined as $\mathbb{S} = \mathbb{S}_1 \times \cdots \times \mathbb{S}_n$, where $\mathbb{S}_i$ is the value domain of variable x_i, which may be discrete, e.g. Boolean, or continuous, e.g. real-valued. A function $s : \mathbb{T} \to \mathbb{S}$ from the time domain to the state space represents a *behaviour* of the system and it is termed a *signal*. The value of a signal s at time t, i.e., the values of the state variables, is denoted $s[t] \in \mathbb{S}$. A *sub-signal* of s corresponding to the behavior of x_i is represented by $s_i : \mathbb{T} \to \mathbb{S}_i$. The set of all possible signals over $\mathbb{S}$ is denoted by $\mathbb{S}^*$. Moreover, a *property* φ defines a set of signals $\mathbb{L}_\varphi \subseteq \mathbb{S}^*$, i.e., the set of signals satisfying property φ.

The syntax of MITL over a set of predicates is expressed by the grammar

$$\varphi := \pi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$$

where π is a predicate, i. e., a Boolean expression over $x_1, \ldots, x_n$[4], $\mathcal{U}$ is the *until* operator, and $a, b \in \mathbb{T}$. Based on this grammar, further operators can be derived such as the *eventually* operator $\Diamond$ and the *always* operator $\Box$.

Definition 3 (Satisfaction Relation)**.** *The satisfaction relation states whether a signal s satisfies a property φ at time t. It is defined as follows:*

$$\begin{array}{rcl}
(s,t) \vDash \pi & \Leftrightarrow & \pi(s[t]) = true \\
(s,t) \vDash \neg\varphi & \Leftrightarrow & (s,t) \nvDash \varphi \\
(s,t) \vDash \varphi_1 \vee \varphi_2 & \Leftrightarrow & (s,t) \vDash \varphi_1 \text{ or } (s,t) \vDash \varphi_2 \\
(s,t) \vDash \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 & \Leftrightarrow & \exists t' \in [t+a, t+b] \text{ s.t. } (s,t') \vDash \varphi_2 \\
 & & \text{and } \forall t'' \in [t,t'] : (s,t'') \vDash \varphi_1 \\
(s,t) \vDash \Diamond_{[a,b]} \varphi & \Leftrightarrow & \exists t' \in [t+a, t+b] \text{ s.t. } (s,t') \vDash \varphi \\
(s,t) \vDash \Box_{[a,b]} \varphi & \Leftrightarrow & \forall t' \in [t+a, t+b] : (s,t') \vDash \varphi
\end{array}$$

where $b > a \geq 0$ are time instants defined by events occurring in the system.

IV. Online Verification

To assure the QoS guaranteed to an application, it is necessary to conclude a contract between each application admitted to the system and the resource manager, which is verified online. Aiming to establish such a contract, we formulate constraints on the behavior of different system components to be evaluated at run-time using online monitoring. By means of the monitors, the admission control process can be verified and, moreover, sources of potential misbehavior, e.g., faulty components, can be determined.

A. Admission Control Correctness

In order to verify the correctness of the admission control process, two constraints must be verified whenever an application's high-level QoS requirement is decomposed. Please note that these are static constraints with respect to a single application, which are independent from all other applications.

1) Compatibility: The decomposition algorithm must be compatible with the composition theory, i.e.,

$$comp(dec(Q)) \leq Q. \tag{1}$$

2) Task Order: The decomposition algorithm must not violate the given task order, i.e.,

$$\varrho_2 \geq \varrho_1 + q_1 \text{ if } \tau_1 < \tau_2. \tag{2}$$

B. Execution Correctness

To ensure that the low-level QoS constraints retrieved during the admission control process are enforced by the resource agents in the execution phase, a number of local per-resource constraints must be satisfied[5].

Before we formulate the constraints, we state the following definitions: For any event ε, we denote $t(\varepsilon_i)$ its occurrence time. For an arbitrary but fixed task τ to be scheduled on the resource, we consider the events ε_{rel}, ε_{start}, ε_{term}, and ε_{period}, referring to the release, the start, and the termination of τ as well as to the recurring start of the period of the application a. We define two time intervals, namely, the period interval $I_{period} = [t(\varepsilon_{period}), t(\varepsilon_{period}) + P]$, and the scheduling interval of τ defined by $I_{sched} = [t(\varepsilon_{rel}), t(\varepsilon_{rel}) + q]$, which is the time interval in which τ can be scheduled on the resource. Moreover, we define the predicates π_{start} and π_{rel}, which are meant to hold as of the corresponding event until the end of the period, as well as the predicates π_{term} and π_{stop} in order to distinguish proper and abnormal termination. For each task τ, we also define the predicate $\pi_b = (b \geq c)$ with respect to its budget.

[4]Hence, for each $v \in \mathbb{S}$, $\pi\,[v/x_1, \ldots, x_n]$ (or simply $\pi(v)$) evaluates to a Boolean value.

[5]Due to space limitations, we provide the essential constraints only.

For a task τ, the following constraints on the time points and order of events as well as on the availability of budget must be satisfied:

1) No Budget Over-Consumption: We expect the resource agent to not allow any task to overdraw its budget. That means, whenever τ has exhausted its allocated budget but has not terminated, it is expected to be stopped (abnormal termination) within a short interval of time, i.e.,

$$\Box_{I_{sched}} \ (b = 0 \wedge \neg \pi_{term} \implies \Diamond_{[0,\epsilon]} \pi_{stop}) \tag{3}$$

for some $\epsilon > 0$. Please note that this property must hold in every scheduling interval.

2) Timely Termination with Non-Negative Budget: In every scheduling interval, τ is expected to terminate (properly) before its local deadline, i.e., not violating its local QoS constraint, with non-negative budget. Aiming to distinguish two possible fault causes in case of violation, i.e., the budget is exhausted but τ has not terminated, we express this requirement by two distinct constraints:

$$\Box_{I_{sched}} \ (b = 0 \implies \pi_{term}) \tag{4}$$

$$\Diamond_{I_{sched}} \ (b = 0 \vee \pi_{term}) \tag{5}$$

If constraint 4 is violated, the low-level QoS constraint q imposed by the resource manager is not suitable, e.g., due to an unauthorized change of the application requirements. If, however, constraint 4 holds, but constraint 5 is violated, the resource agent did not give τ the opportunity to consume its budget within the scheduling interval, e.g., due to bad scheduling decisions.

3) Sufficiently Dimensioned Budget: At the beginning of every scheduling interval, i.e., at the release of task τ, the reserved budget is required to be at least as large as the task's time estimate. Moreover, the budget must not decrease before τ begins its execution, i.e.,

$$\Box_{I_{sched}} \ \pi_b \ \mathcal{U}_{I_{sched}} \ \pi_{start}. \tag{6}$$

C. QoS Assurance

The assurance of the high-level QoS of an application executed on the system relies on the satisfaction of the contract concluded at its system admission. This contract is a set of constraints as introduced above and requires all involved system components as well as the application to fulfill their responsibilities. De facto, if the admission control is not correct, the low-level QoS provided by the resource agents cannot suffice to satisfy the high-level QoS requirement. In turn, if the execution phase does not operate properly, e.g., due to faults or resource unavailability, the low-level QoS promises cannot be kept. However, if an application did not reveal realistic characteristics as an input to the admission control process, e.g., unrealistic time estimates or a wrong period, the resource agents may be unable to accommodate unexpected task behavior. We summarize this as follows:

Conjecture 1. *If for an application a admitted to the system, constraints 1 and 2 hold and, moreover, for each task τ executed on the system constraints 3, 4, 5, and 6 are satisfied, then the offered high-level QoS $\bar{Q}$ is provided by the system.*

Please note that Conjecture 1 can be extended by additional constraints. For the verification of each individual constraint, online monitors synthesized from the MITL formulation can be used. Since the fine-grained constraint formulation, i.e., the large number of monitors, allows to localize the cause of an observed constraint violation very precisely, the system is able to automatically perform a targeted reconfiguration in order to sustain the related contract. We will discuss the further potential of contract-based QoS assurance in the next section.

V. Discussion and Conclusion

Compared to previously mentioned approaches, our proposed contract-based QoS assurance has several advantages. Most significantly, all contracts in the system are independent of each other, i.e., no composition of contracts is necessary, but only a co-existence, which allows to modularly add and remove contracts and, moreover, reduces the verification complexity drastically. More precisely, no global verification is required, since global properties can be verified based on local properties.

To cope with the dynamicity of the system and thus also with uncertain conditions, it can be meaningful to introduce different types of contracts. On the one hand, the quality of contracts can be gradated: While an expensive contract may require static reservation on some resources, cheap contracts could be handled in a best-effort manner. On the other hand, the period of validity, can be varied, so that, for instance, long-term and short-term contracts can be distinguished, depending on the application requirements. In view of potentially changing application characteristics, e.g., strongly varying execution times, it may be sensible to combine both contract types and to adjust the respective parameters over time. Additionally, applications violating a contract by disrespecting budget constraints can be enforced to undergo a re-admission process.

Not least, our contract-based QoS assurance is also valid if the system topology changes, since only local reconfiguration, but no contract modification is necessary. Thus, it can also be beneficial in the context of fault-tolerance; discussing this, however, is beyond the scope of this work.

References

[1] L. Prenzel and S. Steinhorst, "Decentralized autonomous architecture for resilient cyber-physical production systems," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 2021, pp. 1300–1303.

[2] J. Peeck, J. Schlatow, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *IEEE Design Automation and Test in Europe (DATE)*, 2021.

[3] V. Millnert, J. Eker, and E. Bini, "End-To-End Deadlines over Dynamic Topologies," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 10:1–10:22.

[4] P. Patel, A. Ranabahu, and A. Sheth, "Service level agreement in cloud computing," 2009.

[5] R. Ernst, L. Ahrendts, and K. B. Gemlau, "System level LET: mastering cause-effect chains in distributed systems," in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society, Washington, DC, USA, October 21-23, 2018*. IEEE, 2018, pp. 4084–4089.

[6] O. Maler, D. Nickovic, and A. Pnueli, *Checking Temporal Properties of Discrete, Timed and Continuous Behaviors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 475–505.