Self-adaptive K8S Cloud Controller for Time-sensitive Applications

Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Danylo Khalyeyev
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
Email: {bulej, bures, hnetynka, khalyeyev}@d3s.mff.cuni.cz

***Abstract*—The paper presents a self-adaptive Kubernetes cloud controller for scheduling time-sensitive applications. The controller allows services to specify timing requirements (response time or throughput) and schedules services on shared cloud resources so as to meet the requirements. The controller builds and continuously updates an internal performance model of each service and uses it to determine the kind of resources needed by a service, as well as predict potential contention on shared resources, and (re-)deploys services accordingly. The controller is integrated with our highly-customizable data processing and visualization platform IVIS, which provides a web-based front-end for service deployment and visualization of results. The controller implementation is open-source and is intended to provide an easy-to-use testbed for experiments focusing on various aspects of adaptive scheduling and deployment in the cloud.**

***Index Terms*—Self-adaptation; cloud; QoS; Kubernetes; visualizations**

I. Introduction

A system performing smart scheduling of workloads in a cloud environment is a classic example of a self-adaptive system. Even though frameworks such as OpenStack and Kubernetes (K8S) provide a basis for implementing such systems, the options for adaptive cloud scheduling remain limited. Despite all the advances in cloud technology (be it IaaS, CaaS, or PaaS), the options mainly include the ability to setup triggers for creating new service instances if utilization (or some other factor) exceeds a certain threshold. More elaborate scenarios, in which the system would observe and learn how a service performs over time and how it competes over shared resources with other services, still remain in the realm of research. One of the reasons is that such systems do not come in the form of turn-key solutions that could be easily used to carry out experimental evaluation, or to demonstrate the value of self-adaptation techniques going beyond event-action rules to practitioners. We argue that this seriously impairs the perceived utility of such self-adaptive systems (and related research), as well as the ability of researchers to evaluate advanced self-adaptation algorithms for the cloud.

In this paper, we aim to address this issue through a framework providing a Kubernetes-based cloud controller for time-sensitive services (with probabilistic requirements on response time or throughput over a given time period). The purpose of the controller is to manage service execution so as to meet their timing requirements while ensuring efficient utilization of cloud resources.

In contrast to other approaches, our cloud controller does not require an a priori service performance model—it builds such a model automatically. When a service is first deployed, the controller evaluates service performance in different deployment settings to make an initial guess on how CPU-, memory-, and I/O-intensive the service is. After deployment, the controller keeps updating the performance model using data collected at runtime, when the service competes for resources. The performance models are then used by the cloud controller to allocate resources and to deploy (and re-deploy) services in the K8S cluster.

We provide our framework in form of a pre-configured and ready-to-use K8S cluster (with the controller and a sample service), which is further integrated with our IVIS data-processing and visualization framework [1]. IVIS provides the management UI for service definition (along with timing requirements) and service deployment, and allows viewing the results of service execution (including statistics documenting how successful the controller was in satisfying the timing requirements). This provides a system which allows experimenting with different aspects of self-adaptation while enabling an end-to-end evaluation of the effects. We have deployed and successfully used a similar system in several international projects (within the EU ECSEL and EUREKA frameworks).

II. Motivation

In our research projects, we often need to run services in the (edge-)cloud. The services typically have some real-time requirements, but the end devices are often resource-constrained and therefore unsuitable for extensive computations (which may require access to a huge dataset), within the given time constraints. However, we also realized that an important aspect is that some devices simply do not provide a convenient programming platform (e.g., they cannot run a full-fledged Linux), which hinders the service development.

Naturally, we cannot guarantee hard real-time response when the cloud is involved. However, we realized that a system that coordinates multiple devices typically performs adaptation using a hierarchy of (two or more) control loops.

The first-level (innermost) loop corresponds to a well-known hard real-time control loop executing on an embed-

This is the authors' version of the paper: *L. Bulej, T. Bureš, P. Hnětynka, D. Khalyeyev. Self-adaptive K8S Cloud Controller for Time-sensitive Applications, in Proceedings of SEAA 2021, Palermo, Italy, pp. 166-169, 2021.*
The final authenticated publication is available online at `https://doi.org/10.1109/SEAA53835.2021.00029`

ded device [2]. The higher-level (outer) loops are relatively new and deal with strategic decisions and device coordination, providing the "smartness". Examples include early data analysis to adjust a drone's or a tractor's trajectory when operating on a field, coordination of a group of drones when mapping a field, architectural changes in distributed video pipelines to maintain a tradeoff between accuracy and energy consumption in ambient assisted living, etc.

While still subject to real-time constraints, these control loops typically operate on more relaxed time scales (hundreds of milliseconds, seconds, or longer) and can tolerate infrequent transient deadline misses. This allows implementing such control loops as *time-sensitive* services with probabilistic real-time requirements, which could be offloaded to the cloud—if the timing requirements can be met on a platform specifically designed to achieve high throughput and resource utilization (through sharing) instead of guaranteed response time.

Interestingly, time-sensitive services are also useful for longer time frames—we have encountered many scenarios involving lengthy computations (minutes or hours) with upper real-time bounds on completion. For example, processing aerial images to detect high concentrations of weed needs to finish before the farmer sets off to apply a herbicide the next morning. In such cases, we need to provide services with the necessary resources and ensure that they meet the deadlines.

While this can be achieved through some trial and error by pre-allocating or reserving resources, it is impractical and inefficient. For example, the K8S cluster allows imposing limits on CPU and memory on individual containers. However, reserving resources for a particular container actually requires imposing resource limits on all the other containers, which at best results in the reserved resources being wasted most of the time. More importantly though, the limits imposed on the other containers may cause performance degradation due to momentary lack of resources, because service resource requirements change over time depending on service demand.

The situation is further complicated by the fact that service performance is influenced by contention on shared resources such as CPU caches and memory bandwidth which cannot be easily partitioned, and the impact depends on the character of (a priori unknown) collocated services sharing those resources. We therefore argue that decisions on resource allocation and service placement with the goal of satisfying particular timing requirements are best left with a cloud scheduler that is aware of the timing requirements and can reason about service performance.

To provide a realistic example of a time-sensitive service, we present a simplified scenario from one of the use-cases in the AFarCloud [3] project. In the scenario, a drone equipped with a multi-spectral imaging sensor captures images while flying over terrain and submits them to a cloud service performing feature detection analysis. The service is the first stage in an image processing pipeline that reconstructs 3D models from the captured images[1], and produces files with feature descriptors.

This is a computationally intensive task that may take hours or days for large datasets—here the dataset is a continuous stream of images coming from the drones. To process the stream in a timely fashion, the stakeholders may require a certain level of throughput, e.g., 1000 images per hour, which the system must maintain. When there is no need for a higher throughput (e.g., the images may not be available), the stakeholders require the computation to be energy efficient. This requires the system to avoid aggressive horizontal scaling and instead focus on minimizing energy consumption (e.g., by collocating tasks on the same node) while maintaining the required throughput.

Finally, while not required in the above scenario, the stakeholders may require the system to guarantee a particular service response time or end-to-end latency.

III. Framework Overview

The framework presented here implements the approach described in our previous paper [6], which relies on developers for providing the specification of timing requirements for services submitted to the K8S controller for deployment.

To this end, we extend the K8S service deployment descriptor with the concept of probes. A probe is defined as a command that can be executed in the service's container, and is meant to perform work that is characteristic to the service—to provide the cloud controller with a well-defined test point on which to measure service performance. Consequently, a probe is not supposed to change the state of the service to allow establishing statistical confidence over the measured performance through repeated execution (even in production). The descriptor in Listing 1 shows the definition of a service with a single probe, but a service may have multiple probes corresponding to different operating profiles or endpoints with different functionality.

```
application: featuredetection
complete: false
components:
- name: detector
  statefulness: none # Component is stateless
  cardinality: single # Component is not auto-scalable
  template: # K8S container template:
    name: container
    image: d3srepo/featuredetector
  probes:
  - name: detect # probe: command and standard input
    codefile: ./detect.py
    inputfile: ./input.json
```

Listing 1. Service descriptor defining a probe

Service timing requirements are then expressed over service probes. The example in Listing 2 states that it should be possible to execute the probe (sequentially) at least 5 times per minute. Note that this is a weaker statement than saying that each invocation should take less than 12 seconds, because it allows for larger variance in response time.

[1]More details on the whole process can be found on the OpenSfM [4] and OpenDroneMap [5] project websites.

This is the authors' version of the paper: *L. Bulej, T. Bureš, P. Hnětynka, D. Khalyeyev. Self-adaptive K8S Cloud Controller for Time-sensitive Applications, in Proceedings of SEAA 2021, Palermo, Italy, pp. 166-169, 2021.*
The final authenticated publication is available online at https://doi.org/10.1109/SEAA53835.2021.00029

```
application: featuredetection
components:
- name: detector
  QoSrequirements:
  - type: throughput
    probe: detect
    requests: 5
    per: minute # At least 5 requests per minute
```

Listing 2. Specification of timing requirements

The controller interprets these statements probabilistically and attempts to schedule the service so that its predicted performance satisfies the requirements. In this case, the requirement concerns throughput, and the service is scheduled so that its predicted mean throughput is above the given threshold. If response time (latency) is the concern, the timing requirement needs to specify a percentile (e.g., 90%) and a time period, and the service is scheduled so that the predicted percentile of its response time is below the given threshold.

The use of probes and the expression of timing requirements over probes instead of service endpoints makes the performance contract easy to comprehend for a developer, because it defines the desired outcome instead of the amount of resources to be provided (such as CPU cores and IOPS). From the cloud perspective, it absolves the controller from having to understand the service endpoints (or how requests on the endpoints change the service state) and provides a way to directly observe the impact of resource allocation and deployment decisions on service performance. It is, however, up to the developer to ensure that the probe workload characterizes a particular service endpoint as closely as possible. This can be achieved by having the probe internally call the service's endpoint with a suitable input, which is the approach we follow in our framework.

Once a service has been submitted to the cloud controller, it enters the *assessment phase* in which the controller builds an initial performance profile of the service. This is done by deploying the service in an assessment cluster (along with characteristic workloads) and measuring the service performance while varying the deployment configuration (in terms of available resources and collocated services). The profile is then used for prediction of the service performance in configurations that were not measured directly, using the performance prediction methodology described in [6]. This approach allows the controller to treat an incoming service as a black-box, instead of relying on an a priori service performance model (such as a queuing network, or a Palladio model [7]).

If the assessment phase determines that the specified timing requirements can be satisfied, the cloud controller admits the service to the production cluster. In production, the controller uses performance prediction to check whether the timing requirements of each service in different combinations are expected to be satisfied in order to determine a combination of services to be deployed on each node.

After deployment to production, the controller periodically executes service probes to monitor its performance. The data is used to improve the service performance profile and allows the controller to detect (and react to) failures to satisfy the timing requirements.

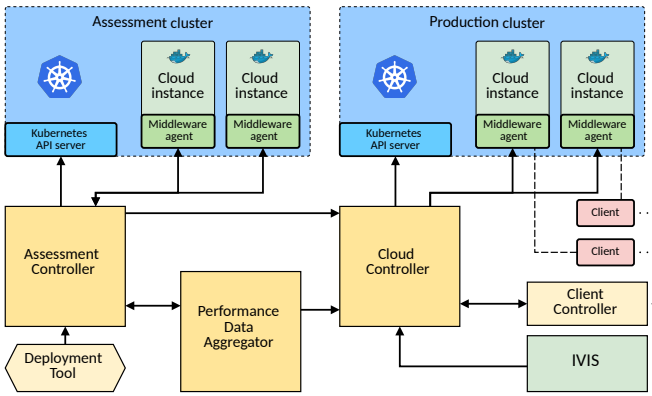

Fig. 1. High-level overview of the framework architecture.

The architecture of the framework is shown in Fig. 1. The framework spans two K8S clusters—one for the assessment phase and the other for production deployment. A web-based fronted integrated into IVIS [1] allows submitting services to the cloud and viewing statistical information about service execution.

IV. Framework Usage

As the architecture overview in Fig. 1 suggests, the framework is heavily distributed and requires launching several virtual machines. To aid users in getting started, the project repository provides detailed installation and usage instructions[2] as well as configuration files for Vagrant (a tool for automated setup of local VMs). Here we will just briefly review the basic usage of the framework.

To enable quick startup (to avoid assessment of the included sample service prior to deployment), the full-fledged performance predictor has been replaced with a simpler version that uses pre-trained performance model of the sample service. When deploying other services, the full-fledged predictor can be re-enabled.

Thanks to its modular architecture, the framework allows experimenting with different performance predictors, deployment schedulers, and cloud controllers. All yellow components of the architecture shown in Fig. 1, as well as their subcomponents, represent framework extension points and can be replaced with custom implementations.

From the user perspective, the framework can be used either via (i) a command line interface or, (ii) a web-based frontend integrated into IVIS. In the first case, the user needs to prepare the service descriptors (c.f. Listings 1 and 2) and submit them to the cloud controller using a command-line tool, which also allows viewing and managing the service state. When using the web-based frontend, the

[2]See README.md at https://github.com/smartarch/qoscloud.

This is the authors' version of the paper: *L. Bulej, T. Bureš, P. Hnětynka, D. Khalyeyev. Self-adaptive K8S Cloud Controller for Time-sensitive Applications, in Proceedings of SEAA 2021, Palermo, Italy, pp. 166-169, 2021.*
The final authenticated publication is available online at https://doi.org/10.1109/SEAA53835.2021.00029

user only needs to enter the service timing requirements—the UI handles the rest.

The framework can obviously deploy other services besides the pre-configured sample service implementing feature detection (c.f. Sect. II). Such a service can be provided either as a piece of code (executed within the default Docker image), or as a custom Docker image configured in a way that enables integration with the cloud controller. This can be done in two ways (the framework documentation provides a detailed description of both).

One way is to use a dedicated script provided by the framework as the entry point of the service container. The script launches the service part of the Middleware Agent (c.f. Fig. 1) which allows the framework to control service initialization and to manage service instances.

Alternatively, if a more fine-grained control over service instances is required, the service part of the Middleware Agent (provided as a class) needs to be instantiated within the service code. A service client (running outside the framework) needs to launch the Client Agent (i.e., the client side of the Middleware Agent) through which the client registers itself and communicates with the framework.

V. Related work

In this section we briefly review several related frameworks, i.e., frameworks that aim at providing certain quality-of-service (QoS) guarantees in cloud environment.

The CloudPick [8] targets QoS-aware service deployment in a multi-cloud environment. It manages QoS by choosing a cloud provider for every application service. However, it only consider services belonging to a single user, and does not take into account the impact that different applications running in the same cloud have on each other.

Cloudroid [9] is a framework that supports QoS-aware cloud application deployment for robotic systems and mainly targets timeliness. It offloads computational tasks from robots to cloud and reserves the resources required by those tasks. Our approach differs in that it manages Qos by controlling the deployment of instance combinations on nodes instead of relying on resource allocation.

DDS [10] proposes an approach to a deadline-aware deployment of time critical workloads in cloud. It uses the Earliest Deadline First algorithm to manage the priority of the workloads based on the explicitly stated deadlines. Similarly to CloudPick (and unlike our framework), it does not take into account the performance interference caused by collocated workloads.

Pythia [11] is similar to our approach by relying on pre-assessment of workloads. It measures the contention for system resources between different processes during pre-assessment, and uses the data to predict the contention for the collocated process combinations that were not measured directly. Our framework differs in that instead of measuring low-level resource contention, it measures end-to-end service performance on developer-defined probes, and provides guarantees on the individual measured operations.

In general, many related frameworks focus on low-level resource allocation and deal with the impact of resource availability on performance of cloud workloads. Our approach provides an easier-to-grasp alternative which avoids micromanagement of resources and instead uses performance measurement and performance prediction methods to control deployment of service instances to satisfy service timing requirements.

VI. Conclusion

We have presented an open-source framework for deploying and scheduling time-sensitive applications. The framework is modular and allows its parts to be easily replaced. This makes it an ideal test-bed for experimenting with different control and scheduling techniques for deployment of time-sensitive applications in the cloud, and for experimenting with self-adaptive systems in general. By taking care of complex issues such as orchestration of benchmarking, interaction with K8S cluster, user interface, etc., the framework allows researchers to focus on the development and experimental evaluation of novel self-adaptation algorithms. The framework is available at https://github.com/smartarch/qoscloud.

Acknowledgment

The research leading to these results has received funding from the ECSEL Joint Undertaking (JU) under grants agreement No 783162 and No 783221.

References

[1] L. Bulej, T. Bureš, P. Hnětynka, V. Čamra, P. Siegl, and M. Töpfer, "IVIS: Highly customizable framework for visualization and processing of IoT data," in *Proc. SEAA*. IEEE, 2020.

[2] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, 2013, vol. 7475.

[3] "Aggregate FARming in the Cloud," http://www.afarcloud.eu/.

[4] "OpenSfM project website," https://www.opensfm.org/.

[5] "Drone mapping software | opendronemap," https://www.opendronemap.org/.

[6] L. Bulej, T. Bureš, A. Filandr, P. Hnětynka, I. Hnětynková, J. Pacovský, G. Sandor, and I. Gerostathopoulos, "Managing latency in edge–cloud environment," *Journal of Systems and Software*, vol. 172, 2021.

[7] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, Oct. 2016.

[8] A. V. Dastjerdi, S. K. Garg, O. F. Rana, and R. Buyya, "CloudPick: a framework for QoS-aware and ontology-based service deployment across clouds," *Software: Practice and Experience*, vol. 45, no. 2, pp. 197–231, 2015.

[9] B. Hu, H. Wang, P. Zhang, B. Ding, and H. Che, "Cloudroid: A cloud framework for transparent and qos-aware robotic computation outsourcing," in *Proc. CLOUD*, 2017.

[10] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. de Laat, and Z. Zhao, "Deadline-aware deployment for time critical applications in clouds," in *Proc. Euro-Par*, 2017.

[11] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, and S. Bagchi, "Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads," in *Proc. Middleware*, 2018.

This is the authors' version of the paper: *L. Bulej, T. Bureš, P. Hnětynka, D. Khalyeyev. Self-adaptive K8S Cloud Controller for Time-sensitive Applications, in Proceedings of SEAA 2021, Palermo, Italy, pp. 166-169, 2021.*
The final authenticated publication is available online at `https://doi.org/10.1109/SEAA53835.2021.00029`