RT-Kubernetes — Containerized Real-Time Cloud Computing

Stefano Fiori
University of Pisa
Pisa, Italy
s.fiori2@studenti.unipi.it

Luca Abeni
Scuola Superiore Sant'Anna
Pisa, Italy
luca.abeni@santannapisa.it

Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy
tommaso.cucinotta@santannapisa.it

ABSTRACT

This paper presents RT-Kubernetes, a software architecture with the ability to deploy real-time software components within containers in cloud infrastructures. The deployment of containers with guaranteed CPU scheduling is obtained by using a hierarchical real-time scheduler based on the Linux SCHED_DEADLINE policy. Preliminary experimental results provide evidence that this new framework succeeds in providing timeliness guarantees in the target responsiveness range, while achieving strong temporal isolation among containers co-located on the same physical hosts.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Real-time systems**; • **Software and its engineering → Cloud computing**; *Virtual machines.*

KEYWORDS

Real-Time Systems, Containers, Kubernetes, Cloud Robotics

ACM Reference Format:
Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. 2022. RT-Kubernetes — Containerized Real-Time Cloud Computing. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3477314.3507216

1 INTRODUCTION

In the last years, cloud computing and containerization technologies have become a standard and effective way to deploy applications over distributed and abundant hardware resources, that can easily scale to respect the applications timing requirements despite continuous fluctuations of the workload over time [9]. The recent advancements in hardware and software infrastructures are starting to make these technologies suitable for serving real-time applications with tight and precise timing constraints, as needed in novel cloud robotics and cloud-enhanced industrial automation scenarios.

However, it is not as easy to provide end-to-end responsiveness guarantees when enriching real-time applications with components deployed in remote cloud infrastructures. This happens because of many reasons, related to networking latency and remote servers' processing times.

SAC '22, April 25–29, 2022, Virtual Event

ACM ISBN 978-1-4503-8713-2/22/04.
https://doi.org/10.1145/3477314.3507216

The network latency issues can be mitigated by using general-purpose QoS control techniques over TCP/IP like DiffServ [8, 10], or can be more effectively addressed by recurring to mostly *private cloud infrastructures* of the industrial/robotic plant, so that the end-to-end networking path is completely under the control of the plant owner. As an alternative, fog/edge architectures [6] can be adopted to have time-sensitive components deployed onto nodes closer to the clients.

However, remote servers equipped with a standard cloud software stack cannot easily cope with the tight timing requirements and scheduling guarantees, turning out to provide unstable processing times that are highly fluctuating depending on what other workloads are deployed onto the same cloud server. In cloud computing and distributed service-oriented computing, this issue is generally addressed by employing elastic control loops [13] that dynamically adapt the number of instances of scalable cloud services. However, this approach cannot cope with the variability of the processing times of individual instances, and their fluctuations, due to virtualization overheads or interferences from other colocated instances on the same servers or physical CPUs.

This paper presents an orchestration platform for real-time multi core containers, based on Kubernetes, that addresses such an issue by allowing to schedule real-time containers providing them with guaranteed performance.

2 BACKGROUND

Although different definitions for "*container*" are possible, all of them generally present a container as an isolated execution environment encapsulating one or more processes or threads (*tasks*, in general). In this work, some of the tasks are characterised by real-time constraints. They can be a set of periodic or sporadic tasks characterised by deadlines, a set of interacting tasks characterised by end-to-end constraints [17], can form a Direct Acyclic Graphs (DAGs) [5], or can be organized in some other way. In any case, some specific real-time schedulability analysis allows designing appropriate scheduling parameters to respect the application's temporal constraints. For example, the so-called Compositional Scheduling Framework [12, 20] (CSF) or other application-dependent analysis techniques [1, 5] allow providing real-time guarantees to applications running inside VMs or containers scheduled through resource reservations [3, 18]. In this setup, each virtual CPU is reserved for execution an amount of physical CPU time Q every period P. An example of reservation-based scheduler is the Linux `SCHED_DEADLINE` policy [15].

Containers are generally implemented by using specific kernel functionalities or virtualization mechanisms, controlled by a user-space *containerization software stack*, such as Kubernetes. Kubernetes manages the execution of containerized applications over a *cluster* composed of one or more physical machines that can be

master nodes (also called control nodes) or worker nodes (sometimes called simply nodes). The master nodes host the Kubernetes control plane, while the worker nodes host the containers running the applications.

The basic execution unit of Kubernetes is a "Pod", composed of one or more containers, network connections, and storage. The Kubernetes architecture is composed of multiple microservices distributed on the different nodes of the cluster. The most important ones for this work are the *Kubernetes Scheduler* (running on a master node), responsible for selecting the worker node on which a Pod is executed and some *Kubelets* (one per worker node), responsible for controlling the execution of Pods on the node.

The Kubelet starts the containers composing the Pods by invoking a *container runtime*; various container runtimes are supported, including the well-known Docker. The container runtime can implement containers by using hypervisor-based VMs (as in kata containers[1]), or by using OS-level virtualization based on Linux control groups and namespaces.

If a hypervisor is used, then real-time guarantees can be provided by scheduling its virtual CPUs with resource reservations. For bare-metal hypervisors such as Xen, this means that a reservation-based scheduler must be implemented in the hypervisor (as an example, Xen implements the Real-Time Deferrable Server — RTDS — algorithm [14]); for hosted hypervisors such as KVM, instead, the `SCHED_DEADLINE` scheduling policy can be used to serve their virtual CPU threads [2].

If Linux control groups and namespaces are used to implement the containers, then the mainline Linux scheduler has to be modified to provide real-time guarantees. For example, the Hierarchical CBS (HCBS) scheduler [1] uses the `SCHED_DEADLINE` policy to schedule groups of tasks (cgroups, in Linux jargon) and can hence be applied to containers (guaranteeing that each CPU of the container can execute for an amount of time Q every period P). In this work, Kubernetes has been extended to support the HCBS scheduler enabling it to provide real-time guarantees to the containerized applications.

Finally, using an appropriate scheduling algorithm is not enough to properly containerize real-time applications, because the scheduling algorithm must also be properly implemented, reducing the kernel latencies [4] to acceptable values. Hence, the Linux Preempt-RT patchset [19] is neeeded on the host.

3 DESIGN AND IMPLEMENTATION

According to what has been described in Section 2, our real-time containerization platform is based on an appropriate CPU scheduling algorithm, a low-latency host kernel (or bare-metal hypervisor), and a container management software that allows to correctly use the mentioned CPU scheduling algorithm.

For what concerns the CPU scheduling algorithm, it is possible to use existing schedulers such as the Xen RTDS, the Linux kernel's `SCHED_DEADLINE` policy, or the HCBS scheduling patch (allowing to use `SCHED_DEADLINE` for groups of tasks instead of single processes or threads). The container management software, instead, requires some modifications to existing open-source projects. In this paper, this feature has been implemented in *RT-Kubernetes*, our modification to the Kubernetes software constituting the main contribution of this work.

[1]https://katacontainers.io

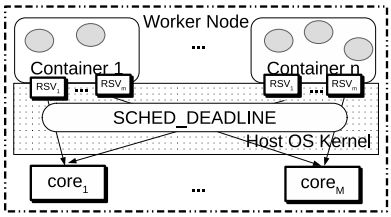

Figure 1: Real-Time Containers scheduling architecture.

Figure 1 displays the scheduling architecture: every worker node (having M CPU cores) can host multiple containers, and the i^{th} container can run real-time applications on m_i CPU cores. Each one of such cores is scheduled through a CPU reservation, reserving Q_i time units every period P_i to the real-time processes or threads of the container. The container management software (RT-Kubernetes, in this case) is responsible for selecting the worker node on which each container is started, and for assigning the correct m_i, Q_i and P_i scheduling parameters to the container, based on information about the timing requirements of the hosted applications, provided by the user at container instantiation time. For example, this can be done exploiting the multi-processor resource model (MPR) in [12]. Based on the previous discussion, RT-Kubernetes must allow describing the temporal requirements (or the real-time constraints) of the application to be containerized and use this description to compute the container's scheduling parameters (real-time control group's runtime and period, or runtimes and periods of the virtual CPU threads if a hypervisor is used). Then, it must select the node where the needed containers or VMs must be started, create the needed containers or VMs and properly configure them (this includes configuring the host and guest schedulers).

Using Kubernetes, the application will run in a Pod, as described by a "manifest" file, written in YAML format. This manifest file is passed to the Kubernetes API server through a command line tool (`kubectl`); then, the API server communicates with the Kubernetes Scheduler to select a worker node for the Pod and the YAML description is communicated to the node's Kubelet. The Kubelet is then in charge of parsing it and creating the container.

Hence, supporting real-time applications in Kubernetes requires to implement some important features. First of all, the format of the manifest files needs to be extended so that they can provide the needed information about the application's real-time requirements. Then, the Kubernetes Scheduler has to be modified so that it selects a worker node for running a Pod only if such a node can host the containerized application without missing deadlines. Finally, Kubelet has to be modified to properly interact with the deadline scheduler (scheduling the Pod's containers with the appropriate algorithm and parameters).

When extending the application's description in the manifest file, it would be interesting to add information describing all the real-time tasks that compose the application, with their temporal parameters, constraints, and dependencies Based on this information, Kubernetes could compute an appropriate number of CPU cores for the container, and design their scheduling parameters

(the runtimes Q and periods P of the CPU reservations used to schedule the container's cores). However, this analysis and design is often application-specific [1], hence it is not possible to implement it in a *generic* container management software. As a result, the current version of the RT-Kubernetes manifest files directly contains the runtime and period for the container's reservations, as well as the the number of CPU cores used by the container (a similar solution is used by RT-OpenStack [21] too). Summing up, the format has been modified to allow specifying an "`rt_runtime`", an "`rt_period`", and an "`rt_cpu`" field (number of container's cores that can run real-time processes or threads). Notice that support for "`rt-runtime`" and "`rt-period`" is already provided by some container runtimes (since this interface is part of the standard RT control group) but not, for example, by Docker. Hence, in this work the Kubelet has been modified to use these parameters independently from the container runtime.

Another interesting thing to be noticed is the difference between the new "`rt_cpu`" attribute and the "`cpu`" attribute already supported by Kubernetes: the new attribute only affects the scheduling of real-time (`SCHED_FIFO` or `SCHED_RR`) processes or threads, when the HCBS scheduler is used. Hence, using the HCBS scheduler it is possible to define a container with a large number of CPU cores, allowing real-time tasks to run only on a small subset of such cores (this is not possible when a hypervisor is used).

The extended syntax allows specifying that a container is reserved an amount of time "`rt_runtime`" every "`rt_period`" on "`rt_cpu`" cores. This is implemented by instantiating on the physical host a multi-core CPU reservation spanning across $m =$ `rt_cpu` cores with runtime $Q =$ `rt_runtime` and period $P =$ `rt_period` (notice that the kernel's HCBS scheduler will not schedule real-time tasks on CPU cores with 0 runtime). Since the host CPU scheduler can properly serve a (Q, P) reservation (guaranteeing Q every P execution time units on *rt_cpu* CPUs) only if an admission test is passed, RT-Kubernetes must make sure that a Pod is started on a worker node only if the Pod's reservations pass the admission test on such a node. As previously mentioned, the Kubernetes Scheduler is responsible for selecting the worker node on which a Pod is started and is the component that needs to be modified to implement this feature.

When implementing an admission test in the Kubernetes Scheduler, different options are possible, depending on which kind of real-time guarantees should be provided (hard [7, 16] or soft [11]). The simplest possible admission tests compute the total utilization U as the sum of the utilizations $U_i = Q_i/P_i$ for all the reservations (Q_i, P_i) running on a node and compare it with a given threshold. These utilization-based admission tests are not the most efficient ones, but can be used to provide both hard and soft real-time guarantees; hence, the Kubernetes Scheduler has been modified to implement a utilization-based admission test. In particular, the RT-Kubernetes Scheduler makes sure that the total real-time utilization $\sum_i U_i$ of all the containers running in a worker node is not larger than a specified limit. Notice that this is similar to what the standard Kubernetes Scheduler does for the "Guaranteed" QoS class, with the difference that the RT-Kubernetes Scheduler can handle the requests of real-time tasks and is compatible with schedulability guarantees from real-time theory. In particular, according to previous works from real-time literature [16], if this limit is set to $(M + 1)/2$ (where M is the number of physical cores present on the node), then the reservations are guaranteed to be schedulable (assuming that RT-Kubelet uses appropriate algorithms to associate CPU reservations to physical CPU cores — see the next subsection).

After the Kubernetes Scheduler selects a worker node, Kubelet is responsible for starting the Pod's containers on such a node. Hence, Kubelet has been modified to support the real-time scheduling of the containers, obtaining what we call RT-Kubelet. While the standard Kubelet schedules the containers using the Linux CFS scheduler (corresponding to the POSIX `SCHED_OTHER` policy) and uses CFS quotas to enforce the resource usage limits, RT-Kubelet can set the real-time runtime and period of the container's control group according to the values specified in the manifest file, so that the HCBS scheduler can be used.

Since the original HCBS scheduler reserved the specified runtime on *all the CPU cores* of the physical system, it did not support the `rt_cpu` parameter (which allows reserving the runtime only on a subset of the CPU cores). To address this issue, the scheduler's interface has been changed to allow reserving different runtimes on different cores, so that RT-Kubelet can specify 0 runtime for the cores where the container's real-time tasks are not allowed to run. With this scheduler modification, when RT-Kubelet starts a container with `rt_cpu` $< M$ (where M is the number of physical cores) it is responsible for selecting the physical cores on which the container's real-time tasks will run. The Worst-Fit (WF) of First-Fit (FF) heuristics have been implemented and can be used to perform this selection. Worst-Fit has the advantage of spreading the real-time workload more uniformly over the available CPU cores, while First-Fit is compatible with the admission control mechanism used by the RT-Kubernetes Scheduler.

The "traditional way" to serve real-time containers is based on statically assigning full CPU cores to the container (using, for example, the Kubernetes "fixed" CPU management policy for the Guaranteed QoS policy). This solution allows providing predictable real-time performance to containerized applications, but forces to over-provision resources to the container. Moreover, all the CPU cores used by the container cannot be shared with other containers, even if only a fraction of the cores is needed by the application's real-time tasks. RT-Kubelet, instead, can use the HCBS scheduler to reserve only a fraction of the CPU cores to real-time tasks and to share CPU cores among containers.

4 EXPERIMENTAL VALIDATION

The proposed modifications have been implemented in the Kubernetes Scheduler and Kubelet and are available as open-source[2]. RT-Kubernetes has been tested on various machines, ranging from a 4-cores Intel NUC to a 40-cores Xeon-based server.

In the first set of tests, the ability of RT-Kubelet to correctly drive the HCBS scheduler has been verified, by running a CPU-hungry application inside containers with various scheduling parameters and verifying that the application receives the reserved amount of time. In a second set of experiments, multiple real-time

[2] https://github.com/stiflerGit/kubernetes

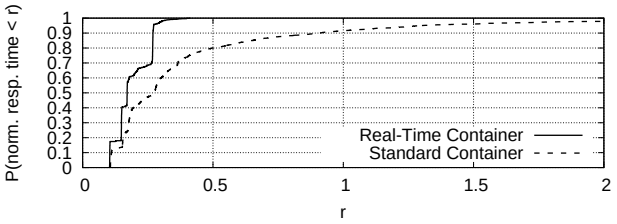

Figure 2: Experimental CDFs for the normalized response times in a real-time and in a standard container.

containers have been started in parallel, verifying that they are correctly scheduled on the available CPU cores (no core is overloaded and all the containers are able to execute for the reserved time).

Finally, the RT-Kubernetes ability to respect the applications' temporal constraints has been verified. This has been done by running a real-time application composed of multiple periodic real-time tasks inside a container and measuring the response time of each task. The container's scheduling parameters have been computed according to MPR [12] so that each task is guaranteed to have a response time shorter than its period. Different numbers of real-time tasks (ranging from 4 to 10), with different (randomly generated) execution times and periods (and a total utilization ranging from $U = 0.5$ to $U = 1.8$), have been tested, verifying that, when the container's scheduling parameters are assigned according to the MPR analysis, all the deadlines are respected.

As an example, Figure 2 reports the experimental Cumulative Distribution Function (CDF) of the normalized response times (response times divided by the task period) for a real-time application running in a container. This CDF indicates the fraction of tasks' activations (on the Y axis) that experienced a normalized response time smaller than r (on the X axis): if the plot arrives at 1 for a value of normalized response time smaller than 1, it means that all the real-time constraints have been respected. The figure contains 2 plots: "Real-Time Container" displays the results obtained using a container scheduled with parameters designed according to MPR analysis, while "Standard Container" displays the results obtained with a non-patched Kubernetes. As expected, the "Real-Time Container" plot shows that all temporal constraints are respected, while the other plot shows that about 10% of the tasks' activations finish after the end of the task period.

5 CONCLUSIONS

This paper presented RT-Kubernetes, supporting the deployment of multi-core real-time containers in a theoretically sound fashion. The proposed architecture is capable of efficiently hosting real-time software components with tight timing constraints in a container infrastructure. A set of experiments showed that the new RT-Kubernetes Scheduler is able to assign real-time containers to nodes that can properly serve them (respecting all the temporal constraints) and the new RT-Kubelet is able to configure the containers' scheduling parameters so that it is possible to guarantee that the temporal constraints of the containerized applications are respected.

REFERENCES

[1] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. 2019. Container-Based Real-Time Scheduling in the Linux Kernel. *SIGBED Review* 16, 3 (October 2019), 33–38.

[2] Luca Abeni, Alessandro Biondi, and Enrico Bini. 2019. Hierarchical scheduling of real-time tasks over Linux-based virtual machines. *Journal of Systems and Software* 149 (2019), 234 – 249.

[3] Luca Abeni and Giorgio Buttazzo. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. Madrid, Spain, 4–13.

[4] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. 2002. A Measurement-Based Analysis of the Real-Time Performance of Linux. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 133–142.

[5] Andoni Amurrio, Ekain Azketa, J. Javier Gutierrez, Mario Aldea, and Michael González Harbour. 2020. Response-Time Analysis of Multipath Flows in Hierarchically-Scheduled Time-Partitioned Distributed Real-Time Systems. *IEEE Access* 8 (2020), 196700–196711.

[6] Vasile-Daniel Balteanu, Alexandru Neculai, Catalin Negru, Florin Pop, and Adrian Stoica. 2020. Near Real-Time Scheduling in Cloud-Edge Platforms. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1264–1271.

[7] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. 2009. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems* 20, 4 (2009), 553–566.

[8] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. 1998. *An Architecture for Differentiated Services*. RFC 2475. RFC Editor.

[9] Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. 2013. Chapter 4 - Cloud Computing Architecture. In *Mastering Cloud Computing*, Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi (Eds.). Morgan Kaufmann, Boston, 111–140.

[10] B.E. Carpenter and K. Nichols. 2002. Differentiated services in the Internet. *Proc. IEEE* 90, 9 (2002), 1479–1494.

[11] UmaMaheswari C. Devi and James H. Anderson. 2008. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems* 38 (2008), 133–189. Issue 2.

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

[13] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. 2020. Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1576–1585.

[14] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. 2012. Realizing Compositional Scheduling through Virtualization. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. 13–22.

[15] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. 2016. Deadline Scheduling in the Linux Kernel. *Software: Practice and Experience* 46, 6 (2016), 821–839.

[16] Jose Maria López, Manuel García, José Luis Diaz, and Daniel F Garcia. 2000. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *12th Euromicro Conference on Real-Time Systems*. Stockholm, Sweden, 25–33.

[17] Riccardo Mancini., Tommaso Cucinotta., and Luca Abeni. 2020. Performance Modeling in Predictable Cloud Computing. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*. INSTICC, SciTePress, 69–78.

[18] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. 1994. Processor Capacity Reserves: Operating Systems Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. 90–99.

[19] Steven Rostedt. 2007. Internals of the RT Patch. In *Proceedings of the Linux Symposium*. Ottawa, Canada, 161–172.

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

[21] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. 2015. RT-OpenStack: CPU Resource Management for Real-Time Cloud Computing. In *2015 IEEE 8th International Conference on Cloud Computing*. 179–186.