Comparison of Kubernetes with the proposed orchestration approach

Florian Hofer

November 7, 2018

1 Kubernetes

Kubernetes is a well-known container management platform developed by Google. It is able manage multiple containers also distributed on various hosts. The advantage is for sure the offered centralized control portal.

In Kubernetes the resource management is performed previous to container deployment. Similar to docker-compose, the resources such as memory and CPU-limit can be specified in a JSON based configuration file. The configuration file describes the resource requests and upper limits for a set of containers, also called a Pod. For these settings there are three combinations, fitting into three Quality of Service classes: (see Kubernetes QoS)

- Best-effort: limits and requests are not set, thus all the pods and containers are run in a best effort manner. These are also the lowest priority and the first pods to be killed when a system runs low of memory.
- Burst: for some containers, requests and optionally limits are set greater than 0 and not equal for one or more resources. Here a minimum of resources are guaranteed, which can top up until they reach the configuration or system limit. Once threre are no more best-effort pods available, these are the most likely pods to be killed in system memory distress.
- Guaranteed: limits and optionally requests are set greater than 0 and equal for all resources and containers, the pod is said to be guaranteed. These pods run top priority and are not to be killed until either they exceed their memory limits or there are no lower priority pods to be depleted.

In all cases, if CPU-time can not be met, the process will be throttled down and not killed. The memory distress behavior of the classes can be defined by a score system. The Out Of Memory score is again determined by the QoS class: a best effort pod has by default a score of 1000, while a guaranteed pod has a worst case score of 1. The limits of OOM score are 0 to 1000, thus in a OOM scenario, the best effort containers are the first to be killed by the linux OOM killer.

The required resources of a Pod are actually the request amount for the run of the whole container collection, thus the sum of resources. All resource requests of the pod must be fulfilled for the containers to be deployed.

The information about Kubernetes has been taken from the Kubernetes Documentation.

2 Orchestration engine

The orchestration engine, different from Kubernetes, is a management tool that works directly on the scheduling level. The planned application code will run in a container in the same engine as the other real-time applications. As such, the orchestrator runs in an ephemeral unit which is easily replaceable and upgradeable. An eventual additional tuning configuration can be deployed on the host and mounted as volume inside the container. A downside of the solution is that the engine can operate just on the local host.

In the previous example with Kubernetes, the configuration of the resources has been realized through requests and limits in a JSON file. The orchestration engine, on the other hand, will work with task data and their real-time properties. Thus, the input parameters are start-time, runtime, period and eventually deadline in addition to the regular task priority. All the tasks will be managed separately, giving the tool little more flexibility in local deployment of CPU-time. In addition, the continuous monitoring and interaction with the system scheduler will give the orchestrator the flexibility to reschedule containers during runtime.

3 Comparison

Table 1 gives a quick overview of the technologies in comparison. The advantage of Kubernetes is the centralized management of containers. The limits for each container can be set in a group configuration and then downloaded to the hosts. The disadvantage here are two: a) the limits must be set big enough to manage eventual overshoot and b) the settings must be defined offline before deployment. Thus, the CPU-time must be set to the maximum expected and/or measures or an overload caused by a overshoot will never be recovered again. Kubernetes can not manage such situations as it is not directly acting on the scheduler but considers only the offline configuration. An overshooting thread will be throttled down, an undesirable state in any realtime system. Thus, in the event of doubt, the CPU-time limits should be set on the higher end of a monitored maximum runtime.

The orchestration engine on the other hand, knows the tasks to be run, their properties and deadlines. In the dynamic version it will also be able to shift tasks to other computation units and relocate resources when needed. The advantage here is that a critical situation sourcing from a container's task overload might be avoided. The complexity of this approach is for sure higher and while Kubernetes might simply set control group limits on the container tasks, the orchestrator has to continuously monitor the execution of the tasks. On the other hand, this monitoring brings the advantage that the engine knows status and limits of the system. Hence, it is possible to manage resources in a more efficient manner. The computation resource bounds can be set more aggressively with task deadlines as upper bound and task runtimes as lower bound. The resulting resource allocation improvement might thus be manifold.

Target	Property	Kubernetes	Orchestrator	
			static	dynamic
System	Action range	Cluster	Local*	Local*
	Configuration	JSON	Task + file (opt)	Task + file (opt)
	Interface	CLI + WebUI	CLI	CLI
Resource management	Resources	CPU, memory	CPU, memory*	CPU, memory*
	Ctrl. type	static	static	dynamic
	Groups	Pods	CGroups	CGroups
	Realtime aware	no	yes	yes
CPU	Upper limit	expected max	Configuration	Deadline
	Overshoots	Critical	Critical/Contained	Managed
Memory	Upper limit	System	Configured	Configured
	Overshoots	Critical-OOM	Contained*	Contained*

Table 1: Comparison table of the different management approaches

Table 2 describes possible configurations for an extreme case, a sample task of 10m runtime, 20m deadline and an expected 300m overshoots and random intervals. The first row shows the maximum tasks allowable per CPU. Considering peaks of 300m and that we don't want task to be throttled, a maximum of three containers can run on one CPU for a Kubernetes configuration. The actual behavior has to be verified as Kubernetes is not aware of realtime priorities and scheduling and thus it might be possible that realtime-classified containers are not throttled down.

The static orchestration engine has the same setup, but is actively monitoring the realtime tasks. Depending on how aggressive we want to schedule our resource allocation, from as low as the safe three container configuration we can reach up to a maximum of 35 scheduled tasks. The monitoring of active tasks can also help to predict overshoot probability for each container and thus schedule (statically) on up-start of a new container depending on the assumed contemporaneity factor. Upto 35 scheduled tasks, for the example of Table 2, one overshoot per 1000m can be safely managed. The main difference to Kubernetes is that tasks are actively scheduled as realtime and are not throttled. Thus, we are not bound to the max value, but can instead refer to deadline values. Overshoots in a static setting are critical, which means that, exceeding 35 scheduled tasks, every overshoot will cause additional costs by delaying also other containers.

Thus, in the latter case, a dynamic rescheduling is needed to reallocate resources in order to meet container deadlines. A dynamic reallocation is more flexible and might exploit reference values instead, permitting a total 100 containers.

It is clear that the values are ideal. They do not consider the engines background operation, and we might need to allocate some extra CPU-time for system tasks. The values in Table 2 are an extreme example. However, a dynamic orchestration is able to work on request time instead of deadline or max for scheduling. Thus, overshoots can be manged using resource reallocation and thus sharing the requirements peak through all the available processing resources.

Property	Kubernetes	Orchestrator	
		static	dynamic
Tasks per CPU	3	3-35*	≤ 100
Config. limit	300m	300m	10-20m
Config. request	10m	10m	10m
Spare CPU-time	970m	970-300m*	500-0m

Table 2: Task grouping example and values. CPU-Time values in mills

But the advantages are not finished here: the two approaches are different and not concurrent and can exploit advantages on both sides. While it might be true that also the orchestrator could, with the help of CGroups, add also memory management including NUMA node management, the cooperation of the remote management tool and the integration of the orchestrator with the `kubelet` agent might be the most profitable solution.

4 Conclusions

The two solutions are different in kind. While a static orchestrator approach would get very similar results to Kubernetes, the dynamic version could reach higher resource economy by allocating way more tasks on the same resources. The feasibility of a combined approach and the actual performance of the orchestrator are still to be determined. Anyway, the data seems promising.