RuntimeClass Scheduling Brainstorm

Tim Allclair (tallclair)
2018-10-05

This document is probably incomplete. Please share your ideas!

Overview

RuntimeClass is a new feature for supporting multiple different container runtimes in a cluster. In the 1.12 alpha, we assumed a cluster of homogeneous nodes (i.e. the same runtimes supported on all nodes), and said that any deviations were the cluster administrators responsibility to account for (through scheduling primitives like NodeAffinity and Taints/Tolerations).

Initial feedback has shown that heterogeneous clusters will be a common use case, which means that native scheduling support will be an important feature. We are kicking off the design discussions now, with the feature targeted at v1.14.

Goals

1. A pod that requests RuntimeClass “foo” is automatically scheduled to a node that supports RuntimeClass foo. If no supporting node is schedulable, the pod is left in a pending state with transparent error messages.

The following features do not exist yet, but our scheduling solution should be extendable if we pursue these in the future.

2. The scheduler should account for resources[1] associated with the runtime through PodOverhead.
3. A pod that requests a **set** of RuntimeClasses (“foo OR bar”) can be automatically scheduled to a node that supports one of those RuntimeClasses. If only one of classes is schedulable, then the Pod should be assigned that RuntimeClass and scheduled accordingly.

[1] Mixing PodOverhead with variable runtimes (requirement 3) introduces some difficult ordering problems. For instance, if resource requests depend on the scheduling decision, then the ResourceQuota controller needs to be invoked after scheduling. This is a much more complicated problem that we don't need to solve now, but should keep in mind.

Requirements

Our scheduling solution must solve these problems:

1. A scheduling mechanism to ensure pods are steered to appropriate nodes.
2. A discovery mechanism for nodes to determine which runtimes it supports AND/OR a registration mechanism, so that nodes can be assigned supported runtimes (by the cluster provisioner).
3. A reporting mechanism for supported runtimes, so the scheduler knows which nodes support which runtimes. *The solution for this requirement largely follows from the solution chosen for 1 & 2.*

1. Scheduling Mechanism

Option 1.A: Building on existing NodeAffinity primitives

A set of node labels or node affinity rules are configured as part of the RuntimeClass definition. When the pod is created, the RuntimeClass admission controller (NYI) automatically injects NodeAffinity rules based on labels or rules in the RuntimeClass definition.

This approach gets more complicated if multiple runtimes are selected (requirement 3), but NodeSelectorTerms are OR'd, so the RuntimeClassController could just add multiple terms for each possible runtime. If we add in PodOverhead as well, then this becomes much more complicated, and maybe infeasible.

Pros:

- No scheduler changes are required
- No node spec/status changes are required, supported runtimes are just represented through labels.
- Topology can be simplified by labeling for types of nodes, rather than each individual runtime.

Cons:

- Complexity quickly grows as the RuntimeClass feature set is expanded.
- Potentially difficult to reconcile with pods that set NodeAffinity for other use cases.
- The RuntimeClassController needs to modify the pod spec, which could potentially conflict with other modifications to the NodeAffinity.
- Debugging scheduling issues is potentially more complex, especially when other NodeAffinityTerms are provided by the user.

Option 1.B: Native scheduler support

Rather than building on existing primitives, we could extend the scheduler to understand RuntimeClasses natively. With this approach, nodes would need a way to express which RuntimeClasses were supported (e.g. a []RuntimeClassNames slice on the node status). A new scheduler predicate would compare the RuntimeClassName on a pod with the supported RuntimeClasses on a node to determine a fit. This approach could be more easily extended to support PodOverhead given multiple runtimes.

Pros:

- More "native" user experience feel, less indirection.
- More extensible for supporting new RuntimeClass features.

Cons:

- Larger change requires modifications to the scheduler & node API.

2. Discovery & Registration

No matter what scheduling approach we take, there needs to be a way of mapping runtime classes to the nodes that support them. At a high level, there are 2 different approaches here: nodes can automatically "discover" the supported runtimes and broadcast that information, or runtimes can be manually assigned to nodes (by the cluster admin or automated node provisioner), or we could also take a hybrid approach.

Option 2.A: RuntimeHandler discovery

Currently the only part of RuntimeClass that is tied to a node is the RuntimeHandler, which is configured through the CRI. At the moment, the CRI accepts the RuntimeHandler as part of the RunPodSandboxRequest, and rejects requests for an unsupported handler. A new CRI API could be implemented to report the list of accepted handlers. The Kubelet would then cross-reference this list with the registered RuntimeClasses to determine which ones had compatible runtime handlers, giving the list of supported classes. To support dynamic provisioning of runtimes, the Kubelet could requery the API whenever a new RuntimeClass is created, or periodically poll.

A variation on this is for the Kubelet to directly report the supported RuntimeHandlers, and resolve that to RuntimeClasses in another component (e.g. scheduler).

Pros:

- Automated runtime discovery makes configuration simpler.

Cons:

- CRI implementations must be updated to support the new API.

- Assumes that matched runtime handler implies support
  Ex: Suppose multiple versions of a runtime were represented in the cluster, or different nodes had the runtime configured to support a different subset of features.

Option 2.B: Manual configuration through nodes

List the supported RuntimeClasses in the nodes spec or status, and have the cluster administrator or automated node provisioner manually configure the nodes with the list of supported RuntimeClasses.

Pros:
- Simpler to implement on the Kubernetes side.
- More flexible to different deployment models (see 2.A cons)

Cons:
- Places more responsibility on the cluster admin to setup correctly.
- May be difficult to manage in clusters with a large number of nodes.

Option 2.C: Manual configuration through RuntimeClass

Rather than listing the supported RuntimeClasses on each node, select the supporting nodes through the RuntimeClass objects. Since nodes are typically configured in groups (e.g. GKE node pools), it may make more sense to configure RuntimeClass support for the whole group at once. Assuming node groups have a unique label set, one way to implement this is through a label selector on the RuntimeClass object. This approach is implied by scheduling option 1.A, but can also make sense for 1.B. The administrator still needs to manage node labels.

Pros:
- No API changes to the node object.
- Simpler to manage large numbers of nodes, assuming large groups (node pools)
- Flexible to different deployment models.

Cons:
- Cluster admin is still responsible to correctly configure the cluster.

3. Reporting API

These options will mostly be decided by the decisions in the previous 2 sections, but are listed here for the sake of completeness.

3.A: Rely on node labels
3.B: Nodes explicitly list supported RuntimeClasses (either in Spec or Status)
3.C: Nodes explicitly list supported RuntimeHandlers (either in Spec or Status)

Conclusion

Based on these options, I am favoring options 1.B (native scheduler support), 2.C, and 3.A (manual configuration through labels). That said, I'm looking forward to gathering feedback from **SIG Scheduling** and **SIG Node**.