K8s Proposal: Pod Ready++

freehan@ thockin@ bgrant0607@ erictune@ dchen1107@

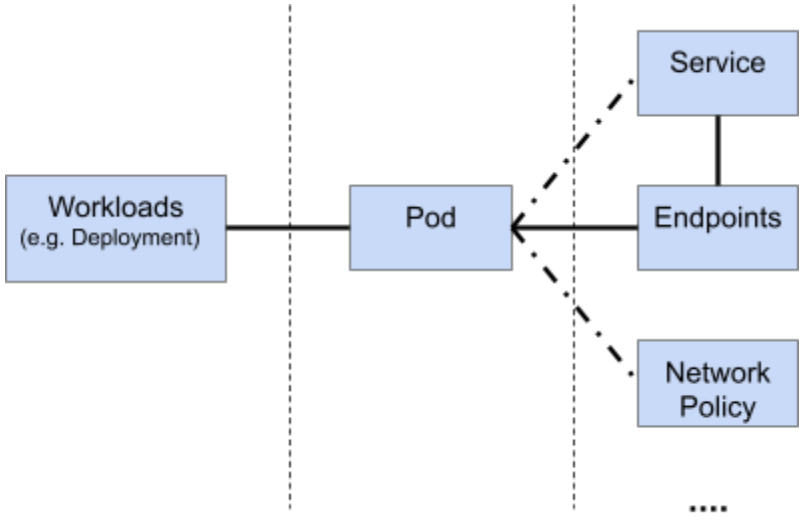
Public. Anyone in the world with a link can comment on this doc.

GKE internal note is here.
KEP is here.

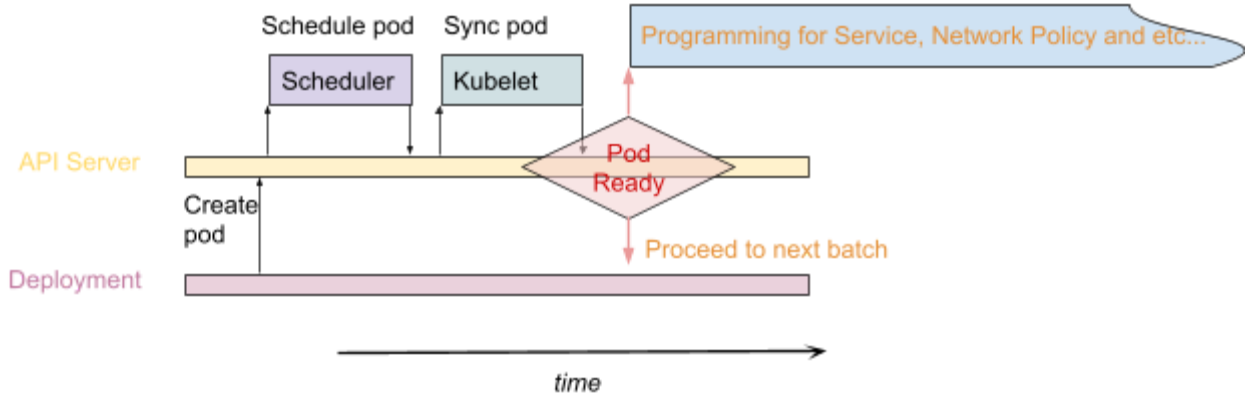
Problem Statement

tl;dr: pod life cycle ignores services, network policy and etc

Pod readiness indicates whether the pod is ready to serve traffic. Pod readiness is dictated by kubelet with user specified readiness probe. On the other hand, pod readiness determines whether pod address shows up on the address list on related endpoints object. K8s primitives that manage pods, such as Deployment, only takes pod status into account for decision making, such as advancement during rolling update.



For example, during deployment rolling update, a new pod becomes ready. On the other hand, service, network policy and load-balancer are not yet ready for the new pod due to whatever reason (e.g. slowness in api machinery, endpoints controller, kube-proxy, iptables or infrastructure programming). This may cause service disruption or lost of backend capacity. In extreme cases, if rolling update completes before any new replacement pod actually start serving traffic, this will cause service outage.



Requirements

- Solution must be backward compatible with pod specification and kubelet control flow.
- Solution must be backward compatible with core workload controllers (e.g. Deployment, StatefulSet).
- Solution must work with user-defined controllers (Operators).
- Solution preferably should be transparent to k8s users. User should not be required to change its existing manifests.

- Solution preferably does not create a dependency from a lower-level concept, (Pod), onto a higher level concept(Ingress). Violates layering principles.
- Solution preferably does not create a dependency from workloads onto networking concepts (e.g. Service, Ingress). This requires mass changes to all workload controllers.
- Solution preferably does not introduce new core API objects.
- Solution preferably does not introduces new backend selection mechanism (label selector) to Service. This fundamentally changed the control flow and feedback loop within k8s. Hence breaks all other systems that rely on this assumption.

Current Readiness Definitions

Container Readiness

A client determines whether a Container is "ready" by testing whether the Pod's `status.containerStatuses[]` list includes an element whose `name` == name of the container and whose `ready` == true. That condition is set according to the following rule.

Container is ready == Container is running AND readiness probe returns success

Pod Readiness

A client determines whether a Pod is "ready" by testing whether the Pod's `status.conditions[]` list includes an element whose `type` == "Ready" and whose `status` == "True". That condition is set according to the following rule.

Pod is ready == All containers are ready

Endpoint Readiness

A client determines whether an Endpoint is "ready" by testing whether the Endpoints' `subsets[].addresses[]` list includes an element whose `targetRef` points to corresponding pod. That condition is set according to the following rule.

Endpoint is ready == Pod is ready

Proposal

This proposal aims to add extensibility to pod readiness. Besides container readiness, external feedback can be injected into *PodStatus* and influence pod readiness. Thus, achieving pod "ready++".

Rationale

Why not fix the workloads?

There are a lot of workloads including core workloads such as deployment and 3rd party workloads such as spark operator. Most if not all of them take pod readiness as a critical signal for decision making, while ignoring higher level abstractions (e.g. service, network policy and ingress). To complicate the problem more, label selector makes membership relationship implicit and dynamic. Solving this problem in all workload controllers would require much bigger change than this proposal.

Why not extend container readiness?

Container readiness is tied to low level constructs such as runtime. This inherently implies that the kubelet and underlying system has full knowledge of container status. Injecting external feedback into container status would complicate the abstraction and control flow. Meanwhile, higher level abstractions (e.g. service) generally takes pod as the atom instead of container.

PodSpec

Introduce an extra field called *ReadinessGates* in *PodSpec.* The field stores a list of *ReadinessGate* structure as follows:

```
type PodReadinessGate struct {
     conditionType string
}
```

The *ReadinessGate* struct has only one string field called *ConditionType*. *ConditionType* refers to a condition in the *PodCondition* list in *PodStatus*. And the status of conditions specified in the *ReadinessGates* will be evaluated for pod readiness. If the condition does not exist in the *PodCondition* list, its status will be default to false.

Constraints

- *ReadinessGates* can only be specified at pod creation.
- No Update allowed on *ReadinessGates*.
- ConditionType must conform to the naming convention of custom pod condition.

Pod Readiness

Change the pod readiness definition to as follows:

Pod is ready == containers are ready AND conditions in ReadinessGates are True

Kubelet will evaluate conditions specified in *ReadinessGates* and update the pod "Ready" status. For example, in the following pod spec, two readinessGates are specified. The status of "www.example.com/feature-1" is false, hence the pod is not ready.

```
Kind: Pod
…
spec:
  readinessGates:
  - conditionType: www.example.com/feature-1
  - conditionType: www.example.com/feature-2
…
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2018-01-01T00:00:00Z
    status: "False"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2018-01-01T00:00:00Z
    status: "False"
    type: www.example.com/feature-1
  - lastProbeTime: null
    lastTransitionTime: 2018-01-01T00:00:00Z
    status: "True"
    type: www.example.com/feature-2
  containerStatuses:
  - containerID: docker://xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    ready : true
…
```

Another pod condition "*ContainerReady*" will be introduced to capture the old pod "*Ready*" condition.

ContainerReady is true == containers are ready

Custom Pod Condition

Custom pod condition can be injected thru PATCH action using KubeClient. Please be noted that “kubectl patch” does not support patching object status. Need to use client-go or other KubeClient implementations.

Naming Convention

The type of custom pod condition must comply with k8s label key format. For example, “www.example.com/feature-1”.

Kubelet PodStatus Control Flow

As kubelet no longer dictates every field in *PodStatus*, the following kubelet changes are needed:

- Use PATCH instead of PUT to update *PodStatus* fields that are dictated by kubelet.
- Only compare the fields that managed by kubelet for *PodStatus* reconciliation .
- Watch *PodStatus* changes and evaluate *ReadinessGates* for pod readiness.

Workloads

To conform with this proposals, workload controllers MUST take pod “Ready” condition as the final signal to proceed during transitions.

For the workloads that take pod readiness as a critical signal for its decision making, they will automatically comply with this proposal without any change. Majority, if not all, of the workloads satisfy this condition.

Feature Integration

In this section, we will discuss how to make *ReadinessGates* transparent to K8s API user. In order words, a K8s API user does not need to specify *ReadinessGates* to use specific features. This allows existing manifests to just work with features that require *ReadinessGate*.
Each feature will bear the burden of injecting *ReadinessGate* and keep its custom pod condition in sync. *ReadinessGate* can be injected using mutating webhook at pod creation time. After pod creation, each feature is responsible for keeping its custom pod condition in sync as long as its *ReadinessGate* exists in the *PodSpec*. This can be achieved by running k8s controller to sync conditions on relevant pods. This is to ensure that *PodStatus* is observable and recoverable even when catastrophic failure (e.g. loss of data) occurs at API server.