Substrate Deploys without Downtime

Document Level Classification

200

Overview

There are certain configuration steps that applications can take to eliminate downtime during deployments when pods are being rolled and updated. The primary goals are to ensure that enough pods remain during the deployment to support the load and that pods will only take traffic when they are ready to do so. All API references in this document are used in the context of a Kubernetes Deployment object, but most/all of these references will appear in other Kubernetes objects, such as DaemonSets and StatefulSets.

Rolling Update Strategy

A RollingUpdateStrategy defines the behavior of a deployment when a configuration change is made to the deployment. It is critically important to include a RollingUpdateStrategy as the default behavior without one is to restart all pods nearly simultaneously. Rolling updates are located at deployment.spec.strategy.rollingUpdate.

The following references are two knobs available under RollingUpdateStrategy:

```
maxSurge<string>
     The maximum number of pods that can be scheduled above the desired
     of pods. Value can be an absolute number (ex: 5) or a percentage o
     pods (ex: 10%). This can not be 0 if MaxUnavailable is 0. Absolute
     is calculated from percentage by rounding up. By default, a value
     used. Example: when this is set to 30%, the new RC can be scaled u
     immediately when the rolling update starts, such that the total nu
     old and new pods do not exceed 130% of desired pods. Once old pods
     been killed, new RC can be scaled up further, ensuring that total
     pods running at any time during the update is at most 130% of desi

   maxUnavailable       <string>
     The maximum number of pods that can be unavailable during the upda
     can be an absolute number (ex: 5) or a percentage of desired pods
     10%). Absolute number is calculated from percentage by rounding do
     can not be 0 if MaxSurge is 0. By default, a fixed value of 1 is u
     Example: when this is set to 30%, the old RC can be scaled down to
     desired pods immediately when the rolling update starts. Once new
     ready, old RC can be scaled down further, followed by scaling up t
     RC, ensuring that the total number of pods available at all times
     the update is at least 70% of desired pods.
```

The following strategy would allow +1 additional pod to be created in your deployment during the update, and 25% of your total deployment pods to be unavailable at any given time:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
     maxUnavailable: 25%
     maxSurge: 1
```

Liveness Probe vs Readiness Probe

A Liveness probe defines when a container is no longer healthy and needs to be restarted within the pod. A Readiness probe defines when your container is ready to start serving traffic after first starting up. It is important to include both of these lifecycle definitions when defining each of your containers. An undefined or misconfigured Liveness probe can allow a container to continue to attempt to serve traffic when it is no longer able to do so, and a missing Readiness probe can allow Kubernetes to send traffic to a container before it is able to handle it.

The official documentation for these two probes can be found here: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-probes

Readiness probes specifically are very important to ensure you pod is not put into service before it is capable of handling traffic.

The following example readiness probe would delay for 5 seconds before starting to run an http get against / on port 8080 and would continue to check every 5 seconds, the pod is considered ready after this check has successfully returned once:

```
        readinessProbe:
          httpGet:
             path: /
             port: 8080
             initialDelaySeconds: 5
             periodSeconds: 5
             successThreshold: 1
```

If a pod is becoming ready too soon even with a readiness check, increasing the initial delay and success thresholds are a good place to start tweaking.

In addition to an http get and a simple TCP socket check, readiness probes allow you to optionally exec an arbitrary script in your container that will pass if it returns 0 - this allows for more complicated readiness checks, such as simulating user traffic, to truely verify your service is ready.

Pod Disruption Budget

Pod disruption budgets or PDBs are similar to rolling update strategies, but instead of managing pod behavior during rolling updates, they mange pod behavior during less expected disruptions - read more about potential disruptions here: https://kubernetes.io/docs/concepts/workloads/pods/disruptions/.

PDBs are very simple Kubernetes objects - in this example are defining that only 1 zookeeper pod should be disrupted at a time:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
```

```
  selector:
    matchLabels:
      app: zookeeper
```

We can also define a percentage of pods unavailable at a time, or define the minimum pods that should exist. Please read more here:

https://kubernetes.io/docs/tasks/run-application/configure-pdb/#think-about-how-your-application-reacts-to-disruptions

ALB Target Type

In Substrate clusters, we use an application called ALB Ingress Controller to automatically create ALBs for Ingresses when they are created and many of the features available to ALBs are exposed via annotations to the Ingresses. Specifically, one of the annotations changes the pathing behavior from ALB to pod in a way that can affect uptime in some cases - the `**alb.ingress.kubernetes.io/target-type: ip**` annotation.

Generally when traffic is routed into Kubernetes, traffic travels from a load balancer to any of the Kubernetes nodes, from there to a node that has a pod of the desired service, and then into the pod itself. The mapping of which pods are on which nodes is handled by a kubernetes object called an endpoint which brings pod targets in and out as they become ready or go unready and update that list on each node via IP Tables rules managed by kube-proxy. Kubernetes endpoint updates are nearly instant.

In the case that **target-type: ip** is specified, traffic from the ALB is sent directly to a network interface for the pod which is attached to the node where it lives, bypassing Kubernetes endpoints and kube-proxy entirely.

Due to the more direct path and the backend behavior of ALBs behind the scenes we've seen using target-type: ip help in cases where traffic across k8s pods is consistently unevenly distributed.

Blake Stoddard noted:

> The devportal charts are missing a few things to work properly with `target-type:ip` (https://github.ol.epicgames.net/online-

web/devportal-src/blob/master/deploy/chart/values/values.yaml#L18):

- it's a delicate balance with timing the k8s shutdown procedure with the ALB deregistration process
- the flow needs to be (in increasing time order): ALB deregistration delay -> prestop sleep -> terminationGracePeriod Seconds, such that you are sleeping longer than the ALB deregistration delay (default dereg delay is 300 seconds, lower it to something like 45 via `alb.ingress.kubernetes.io/target-group-attributes: deregistration_delay.timeout_seconds=45`)
- if you set the deregistration delay to 45 seconds, you need a prestop sleep on the pods that is, say, 60 seconds because the target will remain in the target group (iirc) for the duration of the deregistration period, *and the ALB does not immediately stop sending it traffic once it enters the "deregistering state"* (last time I looked into this deeply, you'll keep receiving requests for 10-15 seconds
- finally you can get terminationGracePeriodSeconds to the prestop sleep time + like 10-15 seconds as one last backstop before k8s kills the pod for real.

since you're using nginx too, there's some extra stuff you have to do to make nginx and node shutdown in sync/nginx let running conns finish before node shuts down

See also:

- https://docs.google.com/presentation/d/16rPym7vHtrdsxPDXOiAvZ4yi6Jq7CvSUVqSmhQ2qYUU/edit#slide=id.g127dad295e2_0_5
- https://twitter.com/t3rabytes/status/1522192953606815745

Matt Armstrong noted:

https://github.ol.epicgames.net/online-security/ecosec-sandbox-service/blob/master/deploy/chart/values.yaml

The following are the noteable improvements:

- deregistration_delay
- pod-readiness-gate-inject
- lifecycle pre-stop hook

The following also needs to be done: https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.1/deploy/pod_readiness_gate/#configuration

Additional Reading

These are some good resources for further reading and understanding the kubernetes resources available and how they relate to pod life cycle.

- https://medium.com/platformer-blog/enable-rolling-updates-in-kubernetes-with-zero-downtime-31d7ec388c81
- https://blog.gruntwork.io/delaying-shutdown-to-wait-for-pod-deletion-propagation-445f779a8304
- https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-probes
- https://kubernetes.io/docs/tasks/run-application/configure-pdb/
- https://kubernetes.io/docs/concepts/workloads/pods/disruptions/
- https://github.com/kubernetes-sigs/aws-alb-ingress-controller
- https://engineering.rakuten.today/post/graceful-k8s-delpoyments/
- https://learnk8s.io/production-best-practices
- https://github.com/integrii/go-k8s-graceful-termination