The Conductor of Kubernetes: The kube-scheduler

Learn about the kube-scheduler.

We'll cover the following

The `kube-scheduler`

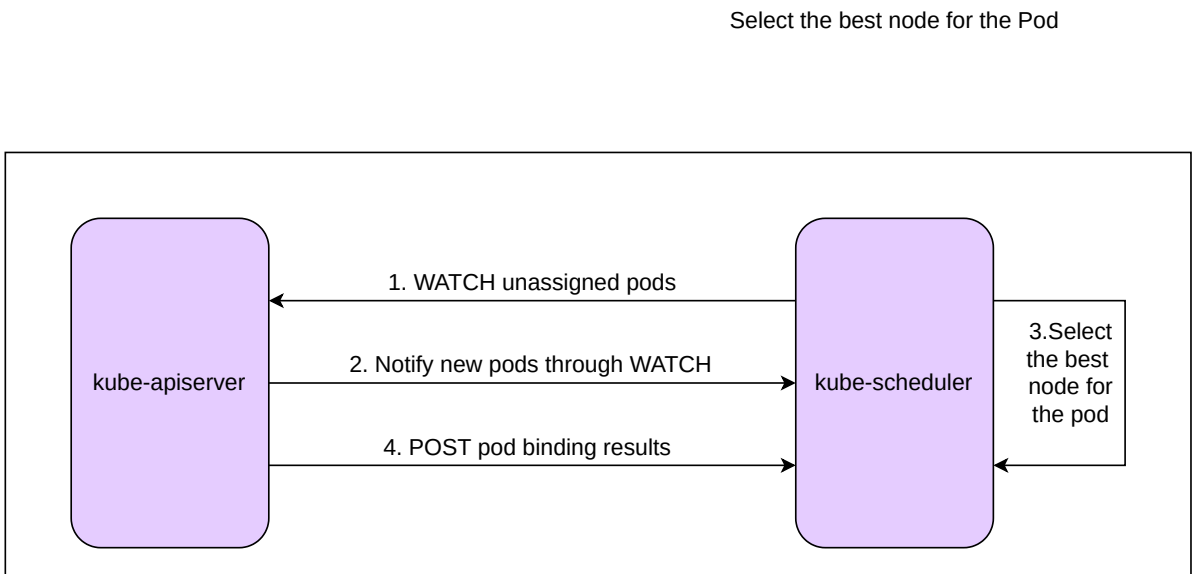
The `kube-scheduler` is a core component that helps assign `Pod` objects to the best node for them to run on with multiple scheduling strategies. It runs on the control plane, together with the `kube-apiserver` and `kube-controller-manager`. In this lesson, we present a brief introduction to the `kube-scheduler`.

What does the `kube-scheduler` do?

Generally speaking, the operation of `kube-scheduler` looks quite simple. All it does is wait for newly created `Pod` objects by watching the `kube-apiserver` and assigning a best-fitted node to those new `Pod` objects (also called “unassigned pods”) that don’t have a node assigned yet.

While the `kube-scheduler` neither manages the `Pod` directly nor operates the selected node to run the `Pod`, it updates the pod specification to the `kube-apiserver`. Then, the `kube-apiserver` will notify the `kubelet` running on the target node that the `Pod` has been scheduled.

The graph below shows the coarse-grained view of the scheduling process.



What the kube-scheduler does

On the startup of the `kube-scheduler`, it will send out a `WATCH` request on unassigned `Pod` objects to the `kube-apiserver`. Once a new `Pod` is created, the `kube-scheduler` will be notified. Inside of the `kube-schduler`, there's a queue to store all the unassigned pods, which will be scheduled one by one. The scheduling task to select a node that best fits the `Pod` isn't that simple. We aren't picking up a random node for the `Pod` to run, regardless of the `Pod` objects already running on that node.

In Kubernetes, every node running the `kubelet` will have some labels to indicate the node characteristics, such as region, disk type, operating system, and architecture. Here are some commonly used labels:

- `kubernetes.io/arch: amd64`
- `kubernetes.io/os: linux`
- `topology.kubernetes.io/zone: us-east-1c`

The `kube-scheduler` will first filter out those unmatched nodes by label selectors. Only those nodes with matching labels will get a chance to be selected.

Sometimes, we taint the nodes for exclusive use, such as testing, data protection, tenant isolation, etc. Normally, we'll taint master nodes with `node-role.kubernetes.io/control-plane` so that other `Pod` objects won't be scheduled. This taint is strongly suggested in any production environment, especially when running a very large Kubernetes cluster. This will keep the core Kubernetes components intact. It's a prerequisite for running applications in a stable and healthy manner. The taints could also be used when nodes undergo maintenance or crash, so that new `Pod` objects will be scheduled onto other healthy nodes. It's common to have nodes with hardware issues, such as memory, disk, and network outages. Only those `Pod` objects that tolerate the taints can be scheduled onto the nodes. For example, the `Pod` below can be scheduled to nodes with the taints `test-env=true:NoScheduler`:

```
 1  apiVersion: v1
 2  kind: Pod
 3  metadata:
 4    name: nginx-demo
 5    namespace: my-demo
 6  spec:
 7    containers:
 8    - name: nginx
 9      image: nginx:1.21
10    tolerations:
11    - key: "test-env"
12      operator: "Exists"
13      effect: "NoSchedule"
```

Pod with taints

Meanwhile, the `kube-scheduler` must know the exact resource utilization (including available capacity and total capacity) for each node across the cluster so that it can make the best decision with strategies. For example, if one application requires 2 CPU cores and 1 GB of memory, then the `Pod` objects will be scheduled onto a node with at least those available resources.

The `kube-scheduler` needs to make sure that the instances of the same application are scheduled and running on different groups of nodes when possible. This helps keep applications running with high availability. Even if a node crashes or one of the instances gets an exception, we have other replicas on other nodes running healthily.

Sometimes, some affinity rules are set so that applications can be scheduled to the same group of nodes. This could help us improve service-level agreements (SLA) and reduce the latency. For example, we're running two correlated applications `frontend` and `backend` in a cluster. If instances `frontend` and `backend` are scheduled and running on a same node, the network latency will be greatly reduced. This helps improve the throughput of the whole service.

Sometimes, we don't want application A running on the same nodes as application B. In this case, some anti-affinity rules are set. For example, we have two unrelated core applications running in a cluster. We don't want both of them to crash. If the instances are running on the same group of nodes, both of them will potentially crash at the same time.

The `kube-scheduler` knows that it's a bad idea to put all the containers into a single node or a small group of nodes. It needs to make sure all the nodes are comparatively balanced. It spreads all the containers across the matching nodes.

`Pod` objects do have priorities. The `kube-scheduler` orders all the pending `Pod` objects by their priority. `Pod` objects with higher priority will be scheduled ahead of those with lower priority.

We can see that the `kube-scheduler` strikes a balance between taints, spreading, affinity, priority, and efficiency during the scheduling cycle. Additionally, concurrency is another important metric to indicate if a scheduler is good or not.

Scheduling results

After picking up the best node for a `Pod`, the `kube-scheduler` will send the scheduling result to the `kube-apiserver`. Below is a code snippet of how the `kube-scheduler` binds a node to a `Pod`.

```
1  // Code from https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/defau
2  // Bind the pods to nodes using the k8s client.
3  func (b DefaultBinder) Bind(ctx context.Context, state *framework.CycleState, p *v1.Pod, nodeName string
4     klog.V(3).InfoS("Attempting to bind pod to node", "pod", klog.KObj(p), "node", nodeName)
5     binding := &v1.Binding{
6        ObjectMeta: metav1.ObjectMeta{Namespace: p.Namespace, Name: p.Name, UID: p.UID},
7        Target:     v1.ObjectReference{Kind: "Node", Name: nodeName},
8     }
```

```
 9     err := b.handle.ClientSet().CoreV1().Pods(binding.Namespace).Bind(ctx, binding, metav1.CreateOptions{
10     if err != nil {
11        return framework.AsStatus(err)
12     }
13     return nil
14  }
```

How to bind a node to a pod

From the code above, we can see that the `kube-scheduler` does not directly update to `podSpec.nodeName`. Instead, it sends the result with a `Binding` object **(lines 5–8)**, which has the same namespace and name as the `Pod` **(line 9)**. The target node name is set in the field `Target` **(line 7)**. This request will eventually go to the sub-resource handler `pods/binding` serving in the `kube-apiserver`. The `kube-apiserver` will extract the target node name from `Binding` and update to `podSpec.nodeName`. Then, the `kube-apiserver` will store this change to `etcd`.

This method has benefits:

- First, updating the pod specification by mistake can be avoided. The `kube-scheduler` only needs to focus on binding the result, which is a singleton that can only be assigned once. Changing the `nodeName` from one to another is forbidden. This adheres to the design principle of the immutable `Pod`.
- Second, this simplifies the binding process and provides a unified interface to all external schedulers. Any other schedulers with distinct scheduler names conform to this binding API.

Summary

In Kubernetes, the `kube-scheduler` only schedules the `Pod`. It's simple, but not that simple. Scheduling is a dynamic process. The scheduler needs to keep an eye on the whole cluster, such as on the:

- Real-time node capacity
- Node characteristics
- Node health conditions
- `Pod` creation
- `Pod` deletion

The `kube-scheduler` is a smart calculator that obtains an optimal result from a group of nodes.

In the `kube-scheduler`, various built-in plugins and strategies are provided, such as:

- Node affinity
- `Pod` affinity
- `Pod` anti-affinity
- `Pod` priority

- `Pod` topology spread

In addition to these, external schedulers with distinct scheduler names can be used as well. We can specify the desired scheduler to schedule our `Pod` objects by setting the `schedulerName` in `podSpec`.

There should be a component that can transform high-level abstraction objects, like from `Deployment` to `Pods`. This is handled by another key component of the control plane, the `kube-controller-manager`.

The `kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, and `etcd` comprise the whole control plane of Kubernetes. They control the cluster, manage the cluster state, and work in perfect harmony.

Back

The Heart of Kubernetes: The kube-apiserver

Next

The Manufactory of Kubernetes: The kube-controller-...