**Chapter 5. Scheduler**

One of the primary jobs of the Kubernetes API is to schedule containers to worker nodes in the cluster of machines. This task is accomplished by a dedicated binary in the Kubernetes cluster called the Kubernetes scheduler. This chapter describes how the scheduler operates, how it can be extended, and how it can even be replaced or augmented by additional schedulers. Kubernetes can handle a wide variety of workloads, from stateless web serving to stateful applications, big data batch jobs, or machine learning on GPUs. The key to ensuring that all of these very different applications can operate in harmony on the same cluster lies in the application of *job scheduling*, which ensures that each container is placed onto the worker node best suited to it.

**An Overview of Scheduling**

When a Pod is first created, it generally doesn’t have a nodeName field. The nodeName indicates the node on which the Pod should execute. The Kubernetes scheduler is constantly scanning the API server (via a watch request) for Pods that don’t have a nodeName; these are Pods that are eligible for scheduling. The scheduler then selects an appropriate node for the Pod and updates the Pod definition with the nodeName that the scheduler selected. After the nodeName is set, the kubelet running on that node is notified about the Pod’s existence (again, via a watch request) and it begins to actually execute that Pod on that node.

**NOTE**

If you want to skip the scheduler, you can always set the nodeName yourself on a Pod. This direct schedules a Pod onto a specific node. This is, in fact, how the DaemonSetcontroller schedules a single Pod onto each node in the cluster. In general, however, direct scheduling should be avoided, since it tends to make your application more brittle and your cluster less efficient. In the general use case, you should trust the scheduler to make the right decision, just as you trust the operating system to find a core to execute your program when you launch it on a single machine.

**Scheduling Process**

When the scheduler discovers a Pod that hasn’t been assigned to a node, it needs to determine which node to schedule the Pod onto. The correct node for a Pod is determined by a number of different factors, some of which are supplied by the user and some of which are calculated by the scheduler. In general, the scheduler is trying to optimize a variety of different criteria to find the node that is best for the particular Pod.

**Predicates**

When making the decision about how to schedule a Pod, the scheduler uses two generic concepts to make its decision. The first is *predicates*. Simply stated, a predicate indicates whether a Pod fits onto a particular node. Predicates are hard constraints, which, if violated, lead to a Pod not operating correctly (or at all) on that node. An example of a such a constraint is the amount of memory requested by the Pod. If that memory is unavailable on the node, the Pod cannot get all of the memory that it needs and the constraint is violated—it is false. Another example of a predicate is a node-selector label query specified by the user. In this case, the user has requested that a Pod only run on certain machines as indicated by the node labels. The predicate is false if a node does not have the required label.

**Priorities**

Predicates indicate situations that are either true or false—the Pod either fits or it doesn’t—but there is an additional generic interface used by the scheduler to determine preference for one node over another. These preferences are expressed as *priorities* or *priority functions*. The role of a priority function is to score the relative value of scheduling a Pod onto a particular node. In contrast to predicates, the priority function does not indicate whether or not the Pod being scheduled onto the node is viable—it is assumed that the Pod can successfully execute on the node—but instead, the predicate function attempts to judge the relative value of scheduling the Pod onto that particular node.

As an example, a priority function would weight nodes where the image has already been pulled. Therefore, the container would start faster than nodes where the image is not present and would have to be pulled, delaying Pod startup.

One important priority function is the spreading function. This function is responsible for prioritizing nodes where Pods that are members of the same Kubernetes Service are not present. It is used to ensure reliability, since it reduces the chances that a machine failure will disable all of the containers in a particular Service.

Ultimately, all of the various predicate values are mixed together to achieve a final priority score for the node, and this score is used to determine where the Pod is scheduled.

**High-Level Algorithm**

For every Pod that needs scheduling, the scheduling algorithm is run. At a high level, the algorithm looks like this:

schedule(pod): string

nodes := getAllHealthyNodes()

viableNodes := []

for node in nodes:

for predicate in predicates:

if predicate(node, pod):

viableNodes.append(node)

scoredNodes := PriorityQueue<score, Node[]>

priorities := GetPriorityFunctions()

for node in viableNodes:

score = CalculateCombinedPriority(node, pod, priorities)

scoredNodes[score].push(node)

bestScore := scoredNodes.top().score

selectedNodes := []

while scoredNodes.top().score == bestScore:

selectedNodes.append(scoredNodes.pop())

node := selectAtRandom(selectedNodes)

return node.Name

You can find the actual code on the [Kubernetes GitHub page](http://bit.ly/2Or3Y5Z).

The basic operation of the scheduler is as follows: first, the scheduler gets the list of all currently known and healthy nodes. Then, for each predicate, the scheduler evaluates the predicate against the node and the Pod being scheduled. If the node is viable (the Pod could run on it), the node is added to the list of possible nodes for scheduling. Next, all of the priority functions are run against the combination of Pod and node. The results are pushed into a priority queue ordered by score, with the best-scoring nodes at the top of the queue. Then, all nodes that have the same score are popped off of the priority queue and placed into a final list. They are considered to be entirely identical, and one of them is chosen in a round-robin fashion and is then returned as the node where the Pod should be scheduled. Round robin is used instead of random choice to ensure an even distribution of Pods among identical nodes.

**Conflicts**

Because there is lag time between when a Pod is scheduled (time T\_1) and when the container actually executes (time T\_N), the scheduling decision may become invalid, due to other actions during the time interval between scheduling and execution.

In some cases, this may mean that a slightly less ideal node is chosen, when a better one could have been assigned. This could be caused by a Pod terminating after time T\_1 but before time T\_N or other changes to the cluster. In general, these sorts of soft-constraint conflicts aren’t that important and they normalize in the aggregate. These conflicts are thus ignored by Kubernetes. Scheduling decisions are only optimal for a single moment in time—they can always become worse as time passes and the cluster changes.

**NOTE**

There is some work going on in the Kubernetes community to improve this situation somewhat. A [Kubernetes-descheduler project](https://github.com/kubernetes-incubator/descheduler), which, if run in a Kubernetes cluster, scans it for Pods that are determined to be significantly suboptimal. If such Pods are found, the descheduler evicts the Pod from its current node. Consequently, the Pod is rescheduled by the Kubernetes scheduler, as if it had just been created.

A more significant kind of conflict occurs when a change to the cluster violates a hard constraint of the scheduler. Imagine, for example, that the scheduler decides to place Pod P on node N. Imagine that Pod P requires two cores to operate, and node N has exactly two cores of spare capacity. At time T\_1, the scheduler has determined that node N has sufficient capacity to run Pod P. However, after the scheduler makes its decision in code and before the decision is written back to the Pod, a new DaemonSet is created. This DaemonSet creates a different Pod that runs on every node, including node N, which consumes one core of capacity. Now Node N only has a single core free, and yet it has been asked to run Pod P, which requires two cores. This is not possible, given the new state of node N, but the scheduling decision has already been made.

When the node notices that it has been asked to run a Pod that no longer passes the predicates for the Pod and node, the Pod is marked as failed. If the Pod has been created by a ReplicaSet, this failed Pod doesn’t count as an active member of the ReplicaSet and, thus, a new Pod will be created and scheduled onto a different node where it fits. This failure behavior is important to understand because it means that Kubernetes cannot be counted on to reliably run standalone Pods. You should always run Pods (even singletons) via a ReplicaSet or Deployment.

**Controlling Scheduling with Labels, Affinity, Taints, and Tolerations**

Of course, there are times when you want more fine-grained control of the scheduling decisions that Kubernetes performs. You could have this by adding your own predicates and priorities, but that’s a fairly heavyweight task. Fortunately, Kubernetes provides you with a number of tools to customize scheduling—without having to implement anything in your own code.

**Node Selectors**

Remember that every object in Kubernetes has an associated set of *labels*. Labels provide identifying metadata for Kubernetes objects, and *label selectors* are often used to dynamically identify sets of API objects for various operations. For example, labels and label selectors are used to identify the sets of Pods that serve traffic behind a Kubernetes load balancers.

Label selectors can also be used to identify a subset of the nodes in a Kubernetes cluster that should be used for scheduling a particular Pod. By default, all nodes in the cluster are potential candidates for scheduling, but by filling in the spec.nodeSelector field in a Pod or PodTemplate, the initial set of nodes can be reduced to a subset.

As an example, consider the task of scheduling a workload to a machine that has high-performance storage, like NVMe-backed SSD. Such storage (at least at the time of this writing) is very expensive and therefore may not be present in every machine. Thus, every machine that has this storage will be given an extra label like:

kind: Node

metadata:

- labels:

nvme-ssd: true

...

To create a Pod that will always be scheduled onto a machine with an NVMe SSD, you then set the Pod’s nodeSelector to match the label on the node:

kind: Pod

spec:

nodeSelector:

nvme-ssd: true

...

Kubernetes has a default predicate that requires every node to match the nodeSelector label query, if it is present. Thus, every Pod with the nvme-ssd label will always be scheduled onto a node with the appropriate hardware.

As was mentioned earlier in the section on conflicts, Node selectors are only evaluated at the time of scheduling. If nodes are actively added and removed, by the time the container executes, its node selector may no longer match the node where it is running.

**Node Affinity**

Node selectors provide a simple way to guarantee that a Pod lands on a particular node, but they lack flexibility. In particular, they cannot represent more complex logical expressions (e.g., “Label foo is either A or B.”) nor can they represent *antiaffinity* (“Label foo is A but label bar is not C.”). Finally, node selectors are predicates—they specify a requirement, not a preference.

Starting with Kubernetes 1.2, the notion of *affinity* was added to node selection via the affinity structure in the Pod spec. Affinity is a more complicated structure to understand, but it is significantly more flexible if you want to express more complicated scheduling policies.

Consider the example just noted, in which a Pod should schedule onto a node that has either label foohas a value of either A or B. This is expressed as the following affinity policy:

kind: Pod

...

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

# foo == A or B

- key: foo

operator: In

values:

- A

- B

...

To show antiaffinity, consider the policy label foo has value A and label bar does not equal C. This is expressed in a similar, though slightly more complicated, specification:

kind: Pod

...

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

# foo == A

- key: foo

operator: In

values:

- A

# bar != C

- key: bar

operator: NotIn

values:

- C

...

**NOTE**

These two examples include the operators In and NotIn. Kubernetes also allows Exists, which only requires that a label key be present regardless of value, as well as NotExists, which requires that a label be absent. There are also Gt and Lt operators, which implement greater-than and less-than, respectively. If you use the Gt or Lt operators, the values array is expected to consist of a single integer and your node labels are expected to be integrals.

So far, we’ve seen node affinity provide a more sophisticated way to select nodes, but we have still only expressed a predicate. This is due to requiredDuringSchedulingIgnoredDuringExecution, which is a long-winded but accurate description of the node affinity behavior. The label expression must match when scheduling is performed but may not match when the Pod is executing.

If you want to express a priority for a node instead of a requirement (or in addition to a requirement), you can use preferredDuringSchedulingIgnoredDuringExecution. For example, using our earlier example, where we required that foo be either A or B, let’s also express a preference for scheduling onto nodes labeled A. The weight term in the preference struct allows us to tune how significant a preference it is, relative to other priorities.

kind: Pod

...

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

# foo == A or B

- key: foo

operator: In

values:

- A

- B

preferredDuringSchedulingIgnoredDuringExecution:

preference:

- weight: 1

matchExpressions:

# foo == A

- key: foo

operator: In

values:

- A

...

Node affinity is currently a beta feature. In Kubernetes 1.4 and beyond, Pod affinity was also introduced with similar syntax (substituting pod for node). Pod affinity allows you to express a requirement or preference for scheduling alongside—or away from—other Pods with particular labels.

**Taints and Tolerations**

Node and Pod affinity allow you to specify preferences for a Pod to schedule (or not) onto a specific set of nodes or near a specific set of Pods. However, that requires user action when creating containers to achieve the right scheduling behavior. Sometimes, as the administrator of a cluster, you might want to affect scheduling without requiring your users to change their behavior.

For example, consider a heterogenous Kubernetes cluster. You may have a mixture of hardware types—some with old 1 Ghz processors and some with new 3 Ghz processors. In general, you don’t want your users to have their work scheduled onto the older processors unless specifically requested. You can achieve this with node antiaffinity, since it requires that every user explicitly add antiaffinity to their Pods for the older machines.

It is this use case that motivated the development of *node taints*. A node taint is exactly what it sounds like. When a taint is applied to a node, the node is considered tainted and will be excluded by default from scheduling. Any tainted node will fail a predicate check at the time of scheduling.

However, consider a user who wants to access 1 Ghz machines. Their work isn’t time critical, and the 1 Ghz machines cost less, since there is far less demand. To achieve this, the user opts into the 1 Ghz machines by adding a *toleration* for the particular taint. This toleration enables the scheduling predicate to pass and thus allows for the node to schedule onto the tainted machine. It is important to note that, although a toleration for a taint enables a Pod to run on a tainted machine, it does not require that the Pod runs on the tainted machine. Indeed, all of the priorities run just as before and, thus, all of the machines in the cluster are available to execute on. Forcing a Pod onto a particular machine is a use case for nodeSelectors or affinity as described earlier.

**Summary**

One of the core features of Kubernetes is the ability to take a user’s request to execute a container and schedule that container onto an appropriate machine. For a cluster administrator, the operation of the scheduler—and teaching users how to use it well—can be critical to building a cluster that is reliable and that you can drive to high utilization and efficiency.