14 Managing pods' computational resources

This chapter covers

- Requesting CPU, memory, and other computational resources for containers
- Setting a hard limit for CPU and memory
- Understanding Quality of Service guarantees for pods
- Setting default, min, and max resources for pods in a namespace
- Limiting the total amount of resources available in a namespace

Up to now you've created pods without caring about how much CPU and memory they're allowed to consume. But as you'll see in this chapter, setting both how much a pod is expected to consume and the maximum amount it's allowed to consume is a vital part of any pod definition. Setting these two sets of parameters makes sure that a pod takes only its fair share of the resources provided by the Kubernetes cluster and also affects how pods are scheduled across the cluster.

14.1 Requesting resources for a pod's containers

When creating a pod, you can specify the amount of CPU and memory that a container needs (these are called *requests*) and a hard limit on what it may consume (known as *limits*). They're specified for each container individually, not for the pod as a whole. The pod's resource requests and limits are the sum of the requests and limits of all its containers.

14.1.1 Creating pods with resource requests

Let's look at an example pod manifest, which has the CPU and memory requests specified for its single container, as shown in the following listing.

Listing 14.1 A pod with resource requests: requests-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: requests-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:
      requests:
        cpu: 200m
        memory: 10Mi
```

You're specifying resource requests for the main container.

The container requests 200 millicores (that is, 1/5 of a single CPU core's time).

The container also requests 10 mebibytes of memory.

In the pod manifest, your single container requires one-fifth of a CPU core (200 millicores) to run properly. Five such pods/containers can run sufficiently fast on a single CPU core.

When you don't specify a request for CPU, you're saying you don't care how much CPU time the process running in your container is allotted. In the worst case, it may not get any CPU time at all (this happens when a heavy demand by other processes exists on the CPU). Although this may be fine for low-priority batch jobs, which aren't time-critical, it obviously isn't appropriate for containers handling user requests.

In the pod spec, you're also requesting 10 mebibytes of memory for the container. By doing that, you're saying that you expect the processes running inside the container to use at most 10 mebibytes of RAM. They might use less, but you're not expecting them to use more than that in normal circumstances. Later in this chapter you'll see what happens if they do.

Now you'll run the pod. When the pod starts, you can take a quick look at the process' CPU consumption by running the `top` command inside the container, as shown in the following listing.

Listing 14.2 Examining CPU and memory usage from within a container

```
$ kubectl exec -it requests-pod top
Mem: 1288116K used, 760368K free, 9196K shrd, 25748K buff, 814840K cached
CPU:  9.1% usr 42.1% sys  0.0% nic 48.4% idle  0.0% io  0.0% irq  0.2% sirq
Load average: 0.79 0.52 0.29 2/481 10
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     R     1192  0.0   1 50.2 dd if /dev/zero of /dev/null
    7     0 root     R     1200  0.0   0  0.0 top
```

The `dd` command you're running in the container consumes as much CPU as it can, but it only runs a single thread so it can only use a single core. The Minikube VM, which is where this example is running, has two CPU cores allotted to it. That's why the process is shown consuming 50% of the whole CPU.

Fifty percent of two cores is obviously one whole core, which means the container is using more than the 200 millicores you requested in the pod specification. This is expected, because requests don't limit the amount of CPU a container can use. You'd need to specify a CPU limit to do that. You'll try that later, but first, let's see how specifying resource requests in a pod affects the scheduling of the pod.

14.1.2 Understanding how resource requests affect scheduling

By specifying resource requests, you're specifying the minimum amount of resources your pod needs. This information is what the Scheduler uses when scheduling the pod to a node. Each node has a certain amount of CPU and memory it can allocate to pods. When scheduling a pod, the Scheduler will only consider nodes with enough unallocated resources to meet the pod's resource requirements. If the amount of unallocated CPU or memory is less than what the pod requests, Kubernetes will not schedule the pod to that node, because the node can't provide the minimum amount required by the pod.

UNDERSTANDING HOW THE SCHEDULER DETERMINES IF A POD CAN FIT ON A NODE

What's important and somewhat surprising here is that the Scheduler doesn't look at how much of each individual resource is being used at the exact time of scheduling but at the sum of resources requested by the existing pods deployed on the node. Even though existing pods may be using less than what they've requested, scheduling another pod based on actual resource consumption would break the guarantee given to the already deployed pods.

This is visualized in figure 14.1. Three pods are deployed on the node. Together, they've requested 80% of the node's CPU and 60% of the node's memory. Pod D, shown at the bottom right of the figure, cannot be scheduled onto the node because it requests 25% of the CPU, which is more than the 20% of unallocated CPU. The fact that the three pods are currently using only 70% of the CPU makes no difference.

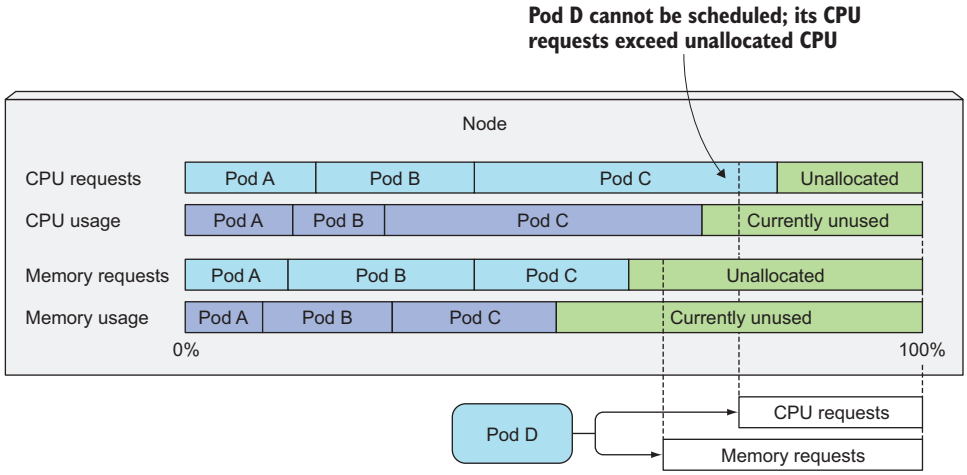

Figure 14.1 The Scheduler only cares about requests, not actual usage.

Understanding how the Scheduler uses pods' requests when selecting the best node for a pod

You may remember from chapter 11 that the Scheduler first filters the list of nodes to exclude those that the pod can't fit on and then prioritizes the remaining nodes per the configured prioritization functions. Among others, two prioritization functions rank nodes based on the amount of resources requested: `LeastRequestedPriority` and `MostRequestedPriority`. The first one prefers nodes with fewer requested resources (with a greater amount of unallocated resources), whereas the second one is the exact opposite—it prefers nodes that have the most requested resources (a smaller amount of unallocated CPU and memory). But, as we've discussed, they both consider the amount of requested resources, not the amount of resources actually consumed.

The Scheduler is configured to use only one of those functions. You may wonder why anyone would want to use the `MostRequestedPriority` function. After all, if you have a set of nodes, you usually want to spread CPU load evenly across them. However, that's not the case when running on cloud infrastructure, where you can add and remove nodes whenever necessary. By configuring the Scheduler to use the `MostRequestedPriority` function, you guarantee that Kubernetes will use the smallest possible number of nodes while still providing each pod with the amount of CPU/memory it requests. By keeping pods tightly packed, certain nodes are left vacant and can be removed. Because you're paying for individual nodes, this saves you money.

Inspecting a node's capacity

Let's see the Scheduler in action. You'll deploy another pod with four times the amount of requested resources as before. But before you do that, let's see your node's capacity. Because the Scheduler needs to know how much CPU and memory each node has, the Kubelet reports this data to the API server, making it available through

the Node resource. You can see it by using the `kubectl describe` command as in the following listing.

Listing 14.3 A node's capacity and allocatable resources

```
$ kubectl describe nodes
Name:       minikube
...
Capacity:                          <- The overall capacity of the node
  cpu:          2
  memory:       2048484Ki
  pods:         110
Allocatable:                       <- The resources allocatable to pods
  cpu:          2
  memory:       1946084Ki
  pods:         110
...
```

The output shows two sets of amounts related to the available resources on the node: the node's *capacity* and *allocatable* resources. The capacity represents the total resources of a node, which may not all be available to pods. Certain resources may be reserved for Kubernetes and/or system components. The Scheduler bases its decisions only on the allocatable resource amounts.

In the previous example, the node called `minikube` runs in a VM with two cores and has no CPU reserved, making the whole CPU allocatable to pods. Therefore, the Scheduler should have no problem scheduling another pod requesting 800 millicores.

Run the pod now. You can use the YAML file in the code archive, or run the pod with the `kubectl run` command like this:

```
$ kubectl run requests-pod-2 --image=busybox --restart Never
➥ --requests='cpu=800m,memory=20Mi' -- dd if=/dev/zero of=/dev/null
pod "requests-pod-2" created
```

Let's see if it was scheduled:

```
$ kubectl get po requests-pod-2
NAME             READY     STATUS    RESTARTS   AGE
requests-pod-2   1/1       Running   0          3m
```

Okay, the pod has been scheduled and is running.

CREATING A POD THAT DOESN'T FIT ON ANY NODE

You now have two pods deployed, which together have requested a total of 1,000 millicores or exactly 1 core. You should therefore have another 1,000 millicores available for additional pods, right? You can deploy another pod with a resource request of 1,000 millicores. Use a similar command as before:

```
$ kubectl run requests-pod-3 --image=busybox --restart Never
➥ --requests='cpu=1,memory=20Mi' -- dd if=/dev/zero of=/dev/null
pod "requests-pod-2" created
```

NOTE This time you're specifying the CPU request in whole cores (`cpu=1`) instead of millicores (`cpu=1000m`).

So far, so good. The pod has been accepted by the API server (you'll remember from the previous chapter that the API server can reject pods if they're invalid in any way). Now, check if the pod is running:

```
$ kubectl get po requests-pod-3
NAME             READY     STATUS    RESTARTS   AGE
requests-pod-3   0/1       Pending   0          4m
```

Even if you wait a while, the pod is still stuck at Pending. You can see more information on why that's the case by using the `kubectl describe` command, as shown in the following listing.

Listing 14.4 Examining why a pod is stuck at Pending with `kubectl describe pod`

```
$ kubectl describe po requests-pod-3
Name:       requests-pod-3
Namespace:  default
Node:       /
...
Conditions:
  Type           Status
  PodScheduled   False
...
Events:
... Warning  FailedScheduling    No nodes are available
                                 that match all of the
                                 following predicates::
                                 Insufficient cpu (1).
```

No node is associated with the pod.

The pod hasn't been scheduled.

Scheduling has failed because of insufficient CPU.

The output shows that the pod hasn't been scheduled because it can't fit on any node due to insufficient CPU on your single node. But why is that? The sum of the CPU requests of all three pods equals 2,000 millicores or exactly two cores, which is exactly what your node can provide. What's wrong?

Determining why a pod isn't being scheduled

You can figure out why the pod isn't being scheduled by inspecting the node resource. Use the `kubectl describe node` command again and examine the output more closely in the following listing.

Listing 14.5 Inspecting allocated resources on a node with `kubectl describe node`

```
$ kubectl describe node
Name:                   minikube
...
Non-terminated Pods:    (7 in total)
  Namespace    Name             CPU Requ.    CPU Lim.   Mem Req.    Mem Lim.
  ---------    ----             ----------   --------   ---------   --------
  default      requests-pod     200m (10%)   0 (0%)     10Mi (0%)   0 (0%)
```

```
  default      requests-pod-2  800m (40%)  0 (0%)     20Mi (1%)   0 (0%)
  kube-system  dflt-http-b...  10m (0%)    10m (0%)   20Mi (1%)   20Mi (1%)
  kube-system  kube-addon-...  5m (0%)     0 (0%)     50Mi (2%)   0 (0%)
  kube-system  kube-dns-26...  260m (13%)  0 (0%)     110Mi (5%)  170Mi (8%)
  kube-system  kubernetes-...  0 (0%)      0 (0%)     0 (0%)      0 (0%)
  kube-system  nginx-ingre...  0 (0%)      0 (0%)     0 (0%)      0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits      Memory Requests Memory Limits
  ------------  ----------      --------------- -------------
  1275m (63%)   10m (0%)        210Mi (11%)     190Mi (9%)
```

If you look at the bottom left of the listing, you'll see a total of 1,275 millicores have been requested by the running pods, which is 275 millicores more than what you requested for the first two pods you deployed. Something is eating up additional CPU resources.

You can find the culprit in the list of pods in the previous listing. Three pods in the `kube-system` namespace have explicitly requested CPU resources. Those pods plus your two pods leave only 725 millicores available for additional pods. Because your third pod requested 1,000 millicores, the Scheduler won't schedule it to this node, as that would make the node overcommitted.

FREEING RESOURCES TO GET THE POD SCHEDULED

The pod will only be scheduled when an adequate amount of CPU is freed (when one of the first two pods is deleted, for example). If you delete your second pod, the Scheduler will be notified of the deletion (through the watch mechanism described in chapter 11) and will schedule your third pod as soon as the second pod terminates. This is shown in the following listing.

Listing 14.6 Pod is scheduled after deleting another pod

```
$ kubectl delete po requests-pod-2
pod "requests-pod-2" deleted

$ kubectl get po
NAME             READY     STATUS        RESTARTS   AGE
requests-pod     1/1       Running       0          2h
requests-pod-2   1/1       Terminating   0          1h
requests-pod-3   0/1       Pending       0          1h

$ kubectl get po
NAME             READY     STATUS    RESTARTS   AGE
requests-pod     1/1       Running   0          2h
requests-pod-3   1/1       Running   0          1h
```

In all these examples, you've specified a request for memory, but it hasn't played any role in the scheduling because your node has more than enough allocatable memory to accommodate all your pods' requests. Both CPU and memory requests are treated the same way by the Scheduler, but in contrast to memory requests, a pod's CPU requests also play a role elsewhere—while the pod is running. You'll learn about this next.

14.1.3 Understanding how CPU requests affect CPU time sharing

You now have two pods running in your cluster (you can disregard the system pods right now, because they're mostly idle). One has requested 200 millicores and the other one five times as much. At the beginning of the chapter, we said Kubernetes distinguishes between resource requests and limits. You haven't defined any limits yet, so the two pods are in no way limited when it comes to how much CPU they can each consume. If the process inside each pod consumes as much CPU time as it can, how much CPU time does each pod get?

The CPU requests don't only affect scheduling—they also determine how the remaining (unused) CPU time is distributed between pods. Because your first pod requested 200 millicores of CPU and the other one 1,000 millicores, any unused CPU will be split among the two pods in a 1 to 5 ratio, as shown in figure 14.2. If both pods consume as much CPU as they can, the first pod will get one sixth or 16.7% of the CPU time and the other one the remaining five sixths or 83.3%.

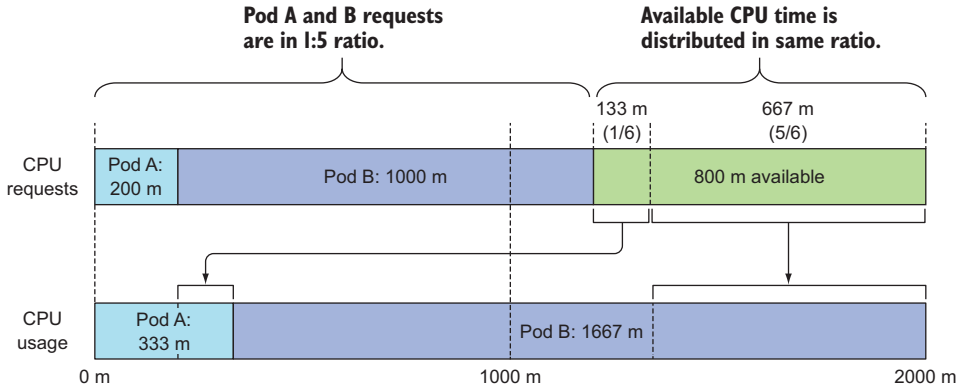

Figure 14.2 Unused CPU time is distributed to containers based on their CPU requests.

But if one container wants to use up as much CPU as it can, while the other one is sitting idle at a given moment, the first container will be allowed to use the whole CPU time (minus the small amount of time used by the second container, if any). After all, it makes sense to use all the available CPU if no one else is using it, right? As soon as the second container needs CPU time, it will get it and the first container will be throttled back.

14.1.4 Defining and requesting custom resources

Kubernetes also allows you to add your own custom resources to a node and request them in the pod's resource requests. Initially these were known as Opaque Integer Resources, but were replaced with Extended Resources in Kubernetes version 1.8.

First, you obviously need to make Kubernetes aware of your custom resource by adding it to the Node object's `capacity` field. This can be done by performing a `PATCH` HTTP request. The resource name can be anything, such as `example.org/my-resource`, as long as it doesn't start with the `kubernetes.io` domain. The quantity must be an integer (for example, you can't set it to 100 millis, because 0.1 isn't an integer; but you can set it to 1000m or 2000m or, simply, 1 or 2). The value will be copied from the `capacity` to the `allocatable` field automatically.

Then, when creating pods, you specify the same resource name and the requested quantity under the `resources.requests` field in the container spec or with `--requests` when using `kubectl run` like you did in previous examples. The Scheduler will make sure the pod is only deployed to a node that has the requested amount of the custom resource available. Every deployed pod obviously reduces the number of allocatable units of the resource.

An example of a custom resource could be the number of GPU units available on the node. Pods requiring the use of a GPU specify that in their requests. The Scheduler then makes sure the pod is only scheduled to nodes with at least one GPU still unallocated.

14.2 Limiting resources available to a container

Setting resource requests for containers in a pod ensures each container gets the minimum amount of resources it needs. Now let's see the other side of the coin—the maximum amount the container will be allowed to consume.

14.2.1 Setting a hard limit for the amount of resources a container can use

We've seen how containers are allowed to use up all the CPU if all the other processes are sitting idle. But you may want to prevent certain containers from using up more than a specific amount of CPU. And you'll always want to limit the amount of memory a container can consume.

CPU is a compressible resource, which means the amount used by a container can be throttled without affecting the process running in the container in an adverse way. Memory is obviously different—it's incompressible. Once a process is given a chunk of memory, that memory can't be taken away from it until it's released by the process itself. That's why you need to limit the maximum amount of memory a container can be given.

Without limiting memory, a container (or a pod) running on a worker node may eat up all the available memory and affect all other pods on the node and any new pods scheduled to the node (remember that new pods are scheduled to the node based on the memory requests and not actual memory usage). A single malfunctioning or malicious pod can practically make the whole node unusable.

CREATING A POD WITH RESOURCE LIMITS

To prevent this from happening, Kubernetes allows you to specify resource limits for every container (along with, and virtually in the same way as, resource requests). The following listing shows an example pod manifest with resource limits.

Listing 14.7 A pod with a hard limit on CPU and memory: limited-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: limited-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:
      limits:
        cpu: 1
        memory: 20Mi
```

Specifying resource limits for the container

This container will be allowed to use at most 1 CPU core.

The container will be allowed to use up to 20 mebibytes of memory.

This pod's container has resource limits configured for both CPU and memory. The process or processes running inside the container will not be allowed to consume more than 1 CPU core and 20 mebibytes of memory.

NOTE Because you haven't specified any resource requests, they'll be set to the same values as the resource limits.

Overcommitting limits

Unlike resource requests, resource limits aren't constrained by the node's allocatable resource amounts. The sum of all limits of all the pods on a node is allowed to exceed 100% of the node's capacity (figure 14.3). Restated, resource limits can be overcommitted. This has an important consequence—when 100% of the node's resources are used up, certain containers will need to be killed.

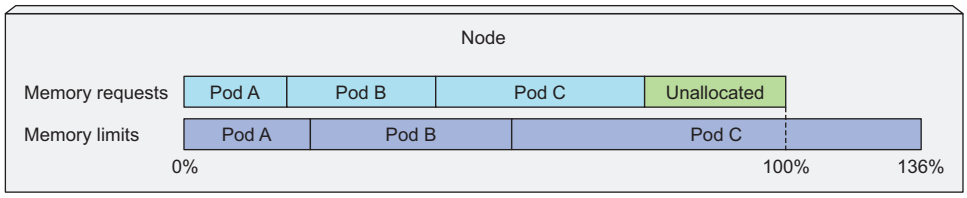

Figure 14.3 The sum of resource limits of all pods on a node can exceed 100% of the node's capacity.

You'll see how Kubernetes decides which containers to kill in section 14.3, but individual containers can be killed even if they try to use more than their resource limits specify. You'll learn more about this next.

14.2.2 Exceeding the limits

What happens when a process running in a container tries to use a greater amount of resources than it's allowed to?

You've already learned that CPU is a compressible resource, and it's only natural for a process to want to consume all of the CPU time when not waiting for an I/O operation. As you've learned, a process' CPU usage is throttled, so when a CPU limit is set for a container, the process isn't given more CPU time than the configured limit.

With memory, it's different. When a process tries to allocate memory over its limit, the process is killed (it's said the container is `OOMKilled`, where OOM stands for Out Of Memory). If the pod's restart policy is set to `Always` or `OnFailure`, the process is restarted immediately, so you may not even notice it getting killed. But if it keeps going over the memory limit and getting killed, Kubernetes will begin restarting it with increasing delays between restarts. You'll see a `CrashLoopBackOff` status in that case:

```
$ kubectl get po
NAME         READY     STATUS             RESTARTS   AGE
memoryhog    0/1       CrashLoopBackOff   3          1m
```

The `CrashLoopBackOff` status doesn't mean the Kubelet has given up. It means that after each crash, the Kubelet is increasing the time period before restarting the container. After the first crash, it restarts the container immediately and then, if it crashes again, waits for 10 seconds before restarting it again. On subsequent crashes, this delay is then increased exponentially to 20, 40, 80, and 160 seconds, and finally limited to 300 seconds. Once the interval hits the 300-second limit, the Kubelet keeps restarting the container indefinitely every five minutes until the pod either stops crashing or is deleted.

To examine why the container crashed, you can check the pod's log and/or use the `kubectl describe pod` command, as shown in the following listing.

Listing 14.8 Inspecting why a container terminated with `kubectl describe pod`

```
$ kubectl describe pod
Name:       memoryhog
...
Containers:
  main:
    ...
    State:          Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Tue, 27 Dec 2016 14:55:53 +0100
      Finished:     Tue, 27 Dec 2016 14:55:58 +0100
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    137
```

The current container was killed because it was out of memory (OOM).

The previous container was also killed because it was OOM

```
      Started:      Tue, 27 Dec 2016 14:55:37 +0100
      Finished:     Tue, 27 Dec 2016 14:55:50 +0100
    Ready:          False
...
```

The `OOMKilled` status tells you that the container was killed because it was out of memory. In the previous listing, the container went over its memory limit and was killed immediately.

It's important not to set memory limits too low if you don't want your container to be killed. But containers can get `OOMKilled` even if they aren't over their limit. You'll see why in section 14.3.2, but first, let's discuss something that catches most users off-guard the first time they start specifying limits for their containers.

14.2.3 Understanding how apps in containers see limits

If you haven't deployed the pod from listing 14.7, deploy it now:

```
$ kubectl create -f limited-pod.yaml
pod "limited-pod" created
```

Now, run the `top` command in the container, the way you did at the beginning of the chapter. The command's output is shown in the following listing.

Listing 14.9 Running the `top` command in a CPU- and memory-limited container

```
$ kubectl exec -it limited-pod top
Mem: 1450980K used, 597504K free, 22012K shrd, 65876K buff, 857552K cached
CPU: 10.0% usr 40.0% sys  0.0% nic 50.0% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.17 1.19 2.47 4/503 10
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     R     1192  0.0   1 49.9 dd if /dev/zero of /dev/null
    5     0 root     R     1196  0.0   0  0.0 top
```

First, let me remind you that the pod's CPU limit is set to 1 core and its memory limit is set to 20 MiB. Now, examine the output of the `top` command closely. Is there anything that strikes you as odd?

Look at the amount of used and free memory. Those numbers are nowhere near the 20 MiB you set as the limit for the container. Similarly, you set the CPU limit to one core and it seems like the main process is using only 50% of the available CPU time, even though the `dd` command, when used like you're using it, usually uses all the CPU it has available. What's going on?

Understanding that containers always see the node's memory, not the container's

The `top` command shows the memory amounts of the whole node the container is running on. Even though you set a limit on how much memory is available to a container, the container will not be aware of this limit.

This has an unfortunate effect on any application that looks up the amount of memory available on the system and uses that information to decide how much memory it wants to reserve.

The problem is visible when running Java apps, especially if you don't specify the maximum heap size for the Java Virtual Machine with the `-Xmx` option. In that case, the JVM will set the maximum heap size based on the host's total memory instead of the memory available to the container. When you run your containerized Java apps in a Kubernetes cluster on your laptop, the problem doesn't manifest itself, because the difference between the memory limits you set for the pod and the total memory available on your laptop is not that great.

But when you deploy your pod onto a production system, where nodes have much more physical memory, the JVM may go over the container's memory limit you configured and will be `OOMKilled`.

And if you think setting the `-Xmx` option properly solves the issue, you're wrong, unfortunately. The `-Xmx` option only constrains the heap size, but does nothing about the JVM's off-heap memory. Luckily, new versions of Java alleviate that problem by taking the configured container limits into account.

UNDERSTANDING THAT CONTAINERS ALSO SEE ALL THE NODE'S CPU CORES

Exactly like with memory, containers will also see all the node's CPUs, regardless of the CPU limits configured for the container. Setting a CPU limit to one core doesn't magically only expose only one CPU core to the container. All the CPU limit does is constrain the amount of CPU time the container can use.

A container with a one-core CPU limit running on a 64-core CPU will get 1/64th of the overall CPU time. And even though its limit is set to one core, the container's processes will not run on only one core. At different points in time, its code may be executed on different cores.

Nothing is wrong with this, right? While that's generally the case, at least one scenario exists where this situation is catastrophic.

Certain applications look up the number of CPUs on the system to decide how many worker threads they should run. Again, such an app will run fine on a development laptop, but when deployed on a node with a much bigger number of cores, it's going to spin up too many threads, all competing for the (possibly) limited CPU time. Also, each thread requires additional memory, causing the apps memory usage to skyrocket.

You may want to use the Downward API to pass the CPU limit to the container and use it instead of relying on the number of CPUs your app can see on the system. You can also tap into the cgroups system directly to get the configured CPU limit by reading the following files:

- /sys/fs/cgroup/cpu/cpu.cfs_quota_us
- /sys/fs/cgroup/cpu/cpu.cfs_period_us

14.3 Understanding pod QoS classes

We've already mentioned that resource limits can be overcommitted and that a node can't necessarily provide all its pods the amount of resources specified in their resource limits.

Imagine having two pods, where pod A is using, let's say, 90% of the node's memory and then pod B suddenly requires more memory than what it had been using up to that point and the node can't provide the required amount of memory. Which container should be killed? Should it be pod B, because its request for memory can't be satisfied, or should pod A be killed to free up memory, so it can be provided to pod B?

Obviously, it depends. Kubernetes can't make a proper decision on its own. You need a way to specify which pods have priority in such cases. Kubernetes does this by categorizing pods into three Quality of Service (QoS) classes:

- `BestEffort` (the lowest priority)
- `Burstable`
- `Guaranteed` (the highest)

14.3.1 Defining the QoS class for a pod

You might expect these classes to be assignable to pods through a separate field in the manifest, but they aren't. The QoS class is derived from the combination of resource requests and limits for the pod's containers. Here's how.

Assigning a pod to the BestEffort class

The lowest priority QoS class is the `BestEffort` class. It's assigned to pods that don't have any requests or limits set at all (in any of their containers). This is the QoS class that has been assigned to all the pods you created in previous chapters. Containers running in these pods have had no resource guarantees whatsoever. In the worst case, they may get almost no CPU time at all and will be the first ones killed when memory needs to be freed for other pods. But because a `BestEffort` pod has no memory limits set, its containers may use as much memory as they want, if enough memory is available.

Assigning a pod to the Guaranteed class

On the other end of the spectrum is the `Guaranteed` QoS class. This class is given to pods whose containers' requests are equal to the limits for all resources. For a pod's class to be `Guaranteed`, three things need to be true:

- Requests and limits need to be set for both CPU and memory.
- They need to be set for each container.
- They need to be equal (the limit needs to match the request for each resource in each container).

Because a container's resource requests, if not set explicitly, default to the limits, specifying the limits for all resources (for each container in the pod) is enough for

the pod to be `Guaranteed`. Containers in those pods get the requested amount of resources, but cannot consume additional ones (because their limits are no higher than their requests).

ASSIGNING THE BURSTABLE QOS CLASS TO A POD

In between `BestEffort` and `Guaranteed` is the `Burstable` QoS class. All other pods fall into this class. This includes single-container pods where the container's limits don't match its requests and all pods where at least one container has a resource request specified, but not the limit. It also includes pods where one container's requests match their limits, but another container has no requests or limits specified. `Burstable` pods get the amount of resources they request, but are allowed to use additional resources (up to the limit) if needed.

UNDERSTANDING HOW THE RELATIONSHIP BETWEEN REQUESTS AND LIMITS DEFINES THE QOS CLASS

All three QoS classes and their relationships with requests and limits are shown in figure 14.4.

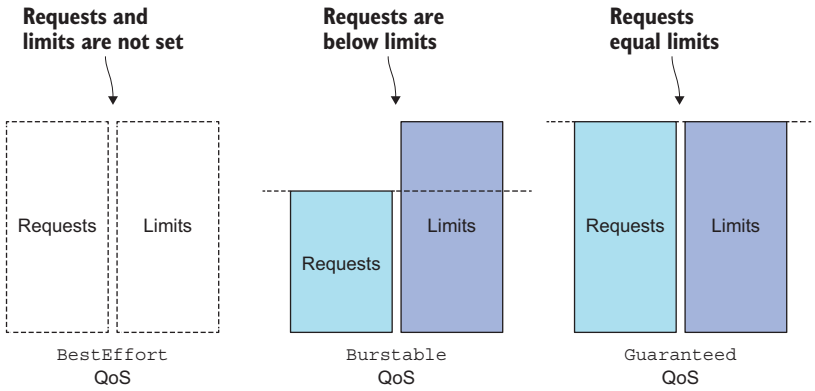

Figure 14.4 Resource requests, limits and QoS classes

Thinking about what QoS class a pod has can make your head spin, because it involves multiple containers, multiple resources, and all the possible relationships between requests and limits. It's easier if you start by thinking about QoS at the container level (although QoS classes are a property of pods, not containers) and then derive the pod's QoS class from the QoS classes of containers.

FIGURING OUT A CONTAINER'S QOS CLASS

Table 14.1 shows the QoS class based on how resource requests and limits are defined on a single container. For single-container pods, the QoS class applies to the pod as well.

Table 14.1 The QoS class of a single-container pod based on resource requests and limits

CPU requests vs. limits	Memory requests vs. limits	Container QoS class
None set	None set	`BestEffort`
None set	Requests < Limits	`Burstable`
None set	Requests = Limits	`Burstable`
Requests < Limits	None set	`Burstable`
Requests < Limits	Requests < Limits	`Burstable`
Requests < Limits	Requests = Limits	`Burstable`
Requests = Limits	Requests = Limits	`Guaranteed`

NOTE If only requests are set, but not limits, refer to the table rows where requests are less than the limits. If only limits are set, requests default to the limits, so refer to the rows where requests equal limits.

FIGURING OUT THE QOS CLASS OF A POD WITH MULTIPLE CONTAINERS

For multi-container pods, if all the containers have the same QoS class, that's also the pod's QoS class. If at least one container has a different class, the pod's QoS class is `Burstable`, regardless of what the container classes are. Table 14.2 shows how a two-container pod's QoS class relates to the classes of its two containers. You can easily extend this to pods with more than two containers.

Table 14.2 A Pod's QoS class derived from the classes of its containers

Container 1 QoS class	Container 2 QoS class	Pod's QoS class
`BestEffort`	`BestEffort`	`BestEffort`
`BestEffort`	`Burstable`	`Burstable`
`BestEffort`	`Guaranteed`	`Burstable`
`Burstable`	`Burstable`	`Burstable`
`Burstable`	`Guaranteed`	`Burstable`
`Guaranteed`	`Guaranteed`	`Guaranteed`

NOTE A pod's QoS class is shown when running `kubectl describe pod` and in the pod's YAML/JSON manifest in the `status.qosClass` field.

We've explained how QoS classes are determined, but we still need to look at how they determine which container gets killed in an overcommitted system.

14.3.2 Understanding which process gets killed when memory is low

When the system is overcommitted, the QoS classes determine which container gets killed first so the freed resources can be given to higher priority pods. First in line to get killed are pods in the `BestEffort` class, followed by `Burstable` pods, and finally `Guaranteed` pods, which only get killed if system processes need memory.

UNDERSTANDING HOW QOS CLASSES LINE UP

Let's look at the example shown in figure 14.5. Imagine having two single-container pods, where the first one has the `BestEffort` QoS class, and the second one's is `Burstable`. When the node's whole memory is already maxed out and one of the processes on the node tries to allocate more memory, the system will need to kill one of the processes (perhaps even the process trying to allocate additional memory) to honor the allocation request. In this case, the process running in the `BestEffort` pod will always be killed before the one in the `Burstable` pod.

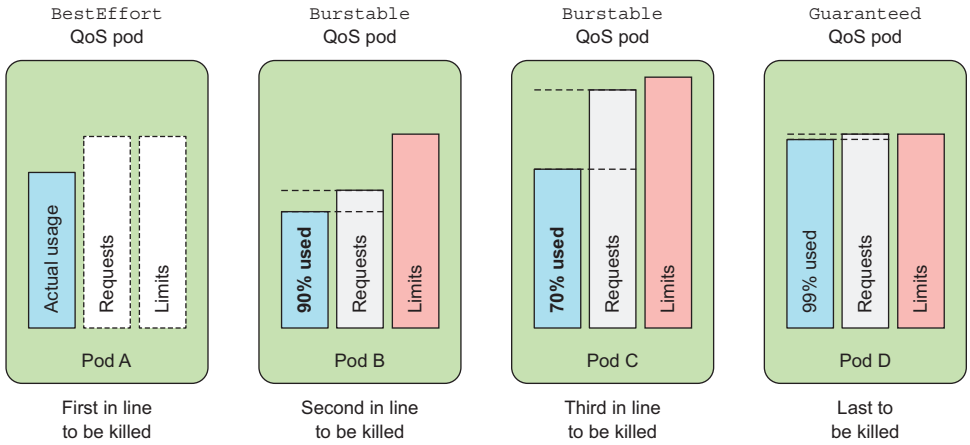

Figure 14.5 Which pods get killed first

Obviously, a `BestEffort` pod's process will also be killed before any `Guaranteed` pods' processes are killed. Likewise, a `Burstable` pod's process will also be killed before that of a `Guaranteed` pod. But what happens if there are only two `Burstable` pods? Clearly, the selection process needs to prefer one over the other.

UNDERSTANDING HOW CONTAINERS WITH THE SAME QOS CLASS ARE HANDLED

Each running process has an OutOfMemory (OOM) score. The system selects the process to kill by comparing OOM scores of all the running processes. When memory needs to be freed, the process with the highest score gets killed.

OOM scores are calculated from two things: the percentage of the available memory the process is consuming and a fixed OOM score adjustment, which is based on the pod's QoS class and the container's requested memory. When two single-container pods exist, both in the `Burstable` class, the system will kill the one using more of its requested

memory than the other, percentage-wise. That's why in figure 14.5, pod B, using 90% of its requested memory, gets killed before pod C, which is only using 70%, even though it's using more megabytes of memory than pod B.

This shows you need to be mindful of not only the relationship between requests and limits, but also of requests and the expected actual memory consumption.

14.4 Setting default requests and limits for pods per namespace

We've looked at how resource requests and limits can be set for each individual container. If you don't set them, the container is at the mercy of all other containers that do specify resource requests and limits. It's a good idea to set requests and limits on every container.

14.4.1 Introducing the LimitRange resource

Instead of having to do this for every container, you can also do it by creating a LimitRange resource. It allows you to specify (for each namespace) not only the minimum and maximum limit you can set on a container for each resource, but also the default resource requests for containers that don't specify requests explicitly, as depicted in figure 14.6.

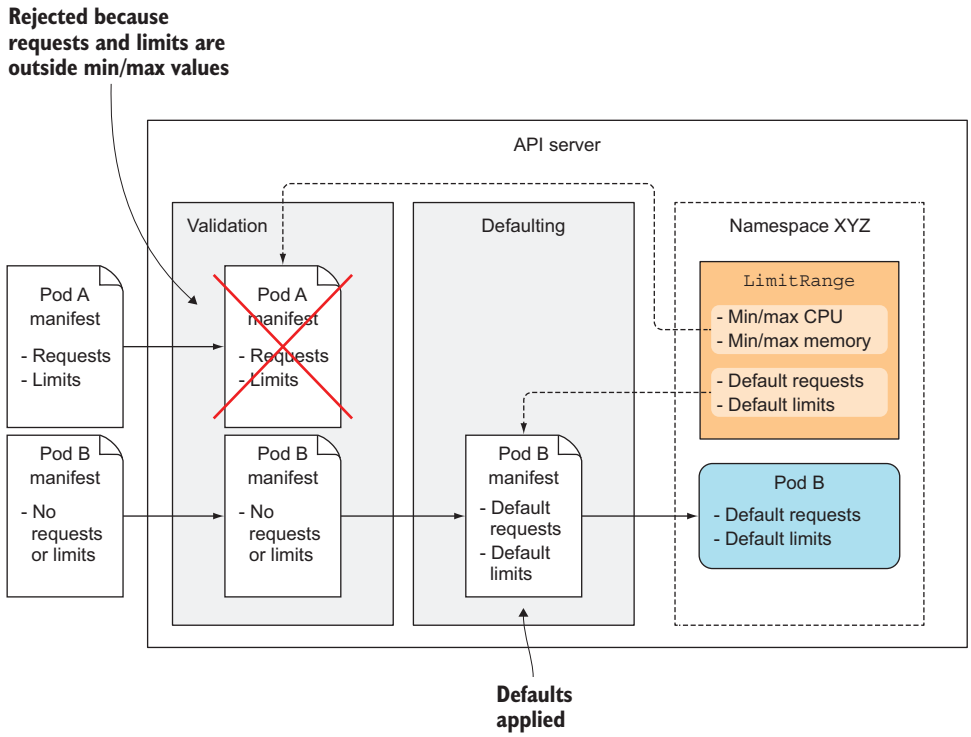

Figure 14.6 A LimitRange is used for validation and defaulting pods.

LimitRange resources are used by the LimitRanger Admission Control plugin (we explained what those plugins are in chapter 11). When a pod manifest is posted to the API server, the LimitRanger plugin validates the pod spec. If validation fails, the manifest is rejected immediately. Because of this, a great use-case for LimitRange objects is to prevent users from creating pods that are bigger than any node in the cluster. Without such a LimitRange, the API server will gladly accept the pod, but then never schedule it.

The limits specified in a LimitRange resource apply to each individual pod/container or other kind of object created in the same namespace as the LimitRange object. They don't limit the total amount of resources available across all the pods in the namespace. This is specified through ResourceQuota objects, which are explained in section 14.5.

14.4.2 Creating a LimitRange object

Let's look at a full example of a LimitRange and see what the individual properties do. The following listing shows the full definition of a LimitRange resource.

Listing 14.10 A LimitRange resource: limits.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: example
spec:
  limits:
  - type: Pod
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
  - type: Container
    defaultRequest:
      cpu: 100m
      memory: 10Mi
    default:
      cpu: 200m
      memory: 100Mi
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
    maxLimitRequestRatio:
      cpu: 4
      memory: 10
```

Specifies the limits for a pod as a whole

Minimum CPU and memory all the pod's containers can request in total

Maximum CPU and memory all the pod's containers can request (and limit)

The container limits are specified below this line.

Default requests for CPU and memory that will be applied to containers that don't specify them explicitly

Default limits for containers that don't specify them

Minimum and maximum requests/limits that a container can have

Maximum ratio between the limit and request for each resource

```
- type: PersistentVolumeClaim
  min:
    storage: 1Gi
  max:
    storage: 10Gi
```

A LimitRange can also set the minimum and maximum amount of storage a PVC can request.

As you can see from the previous example, the minimum and maximum limits for a whole pod can be configured. They apply to the sum of all the pod's containers' requests and limits.

Lower down, at the container level, you can set not only the minimum and maximum, but also default resource requests (`defaultRequest`) and default limits (`default`) that will be applied to each container that doesn't specify them explicitly.

Beside the min, max, and default values, you can even set the maximum ratio of limits vs. requests. The previous listing sets the CPU `maxLimitRequestRatio` to `4`, which means a container's CPU limits will not be allowed to be more than four times greater than its CPU requests. A container requesting 200 millicores will not be accepted if its CPU limit is set to 801 millicores or higher. For memory, the maximum ratio is set to 10.

In chapter 6 we looked at PersistentVolumeClaims (PVC), which allow you to claim a certain amount of persistent storage similarly to how a pod's containers claim CPU and memory. In the same way you're limiting the minimum and maximum amount of CPU a container can request, you should also limit the amount of storage a single PVC can request. A LimitRange object allows you to do that as well, as you can see at the bottom of the example.

The example shows a single LimitRange object containing limits for everything, but you could also split them into multiple objects if you prefer to have them organized per type (one for pod limits, another for container limits, and yet another for PVCs, for example). Limits from multiple LimitRange objects are all consolidated when validating a pod or PVC.

Because the validation (and defaults) configured in a LimitRange object is performed by the API server when it receives a new pod or PVC manifest, if you modify the limits afterwards, existing pods and PVCs will not be revalidated—the new limits will only apply to pods and PVCs created afterward.

14.4.3 Enforcing the limits

With your limits in place, you can now try creating a pod that requests more CPU than allowed by the LimitRange. You'll find the YAML for the pod in the code archive. The next listing only shows the part relevant to the discussion.

Listing 14.11 A pod with CPU requests greater than the limit: limits-pod-too-big.yaml

```
resources:
  requests:
    cpu: 2
```

The pod's single container is requesting two CPUs, which is more than the maximum you set in the LimitRange earlier. Creating the pod yields the following result:

```
$ kubectl create -f limits-pod-too-big.yaml
Error from server (Forbidden): error when creating "limits-pod-too-big.yaml":
pods "too-big" is forbidden: [
  maximum cpu usage per Pod is 1, but request is 2.,
  maximum cpu usage per Container is 1, but request is 2.]
```

I've modified the output slightly to make it more legible. The nice thing about the error message from the server is that it lists all the reasons why the pod was rejected, not only the first one it encountered. As you can see, the pod was rejected for two reasons: you requested two CPUs for the container, but the maximum CPU limit for a container is one. Likewise, the pod as a whole requested two CPUs, but the maximum is one CPU (if this was a multi-container pod, even if each individual container requested less than the maximum amount of CPU, together they'd still need to request less than two CPUs to pass the maximum CPU for pods).

14.4.4 Applying default resource requests and limits

Now let's also see how default resource requests and limits are set on containers that don't specify them. Deploy the `kubia-manual` pod from chapter 3 again:

```
$ kubectl create -f ../Chapter03/kubia-manual.yaml
pod "kubia-manual" created
```

Before you set up your LimitRange object, all your pods were created without any resource requests or limits, but now the defaults are applied automatically when creating the pod. You can confirm this by describing the `kubia-manual` pod, as shown in the following listing.

Listing 14.12 Inspecting limits that were applied to a pod automatically

```
$ kubectl describe po kubia-manual
Name:           kubia-manual
...
Containers:
  kubia:
    Limits:
      cpu:      200m
      memory:   100Mi
    Requests:
      cpu:      100m
      memory:   10Mi
```

The container's requests and limits match the ones you specified in the LimitRange object. If you used a different LimitRange specification in another namespace, pods created in that namespace would obviously have different requests and limits. This allows admins to configure default, min, and max resources for pods per namespace.

If namespaces are used to separate different teams or to separate development, QA, staging, and production pods running in the same Kubernetes cluster, using a different LimitRange in each namespace ensures large pods can only be created in certain namespaces, whereas others are constrained to smaller pods.

But remember, the limits configured in a LimitRange only apply to each individual pod/container. It's still possible to create many pods and eat up all the resources available in the cluster. LimitRanges don't provide any protection from that. A ResourceQuota object, on the other hand, does. You'll learn about them next.

14.5 Limiting the total resources available in a namespace

As you've seen, LimitRanges only apply to individual pods, but cluster admins also need a way to limit the total amount of resources available in a namespace. This is achieved by creating a ResourceQuota object.

14.5.1 Introducing the ResourceQuota object

In chapter 10 we said that several Admission Control plugins running inside the API server verify whether the pod may be created or not. In the previous section, I said that the LimitRanger plugin enforces the policies configured in LimitRange resources. Similarly, the ResourceQuota Admission Control plugin checks whether the pod being created would cause the configured ResourceQuota to be exceeded. If that's the case, the pod's creation is rejected. Because resource quotas are enforced at pod creation time, a ResourceQuota object only affects pods created after the ResourceQuota object is created—creating it has no effect on existing pods.

A ResourceQuota limits the amount of computational resources the pods and the amount of storage PersistentVolumeClaims in a namespace can consume. It can also limit the number of pods, claims, and other API objects users are allowed to create inside the namespace. Because you've mostly dealt with CPU and memory so far, let's start by looking at how to specify quotas for them.

Creating a ResourceQuota for CPU and memory

The overall CPU and memory all the pods in a namespace are allowed to consume is defined by creating a ResourceQuota object as shown in the following listing.

Listing 14.13 A ResourceQuota resource for CPU and memory: quota-cpu-memory.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-and-mem
spec:
  hard:
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi
```

Instead of defining a single total for each resource, you define separate totals for requests and limits for both CPU and memory. You'll notice the structure is a bit different, compared to that of a LimitRange. Here, both the requests and the limits for all resources are defined in a single place.

This ResourceQuota sets the maximum amount of CPU pods in the namespace can request to 400 millicores. The maximum total CPU limits in the namespace are set to 600 millicores. For memory, the maximum total requests are set to 200 MiB, whereas the limits are set to 500 MiB.

A ResourceQuota object applies to the namespace it's created in, like a LimitRange, but it applies to all the pods' resource requests and limits in total and not to each individual pod or container separately, as shown in figure 14.7.

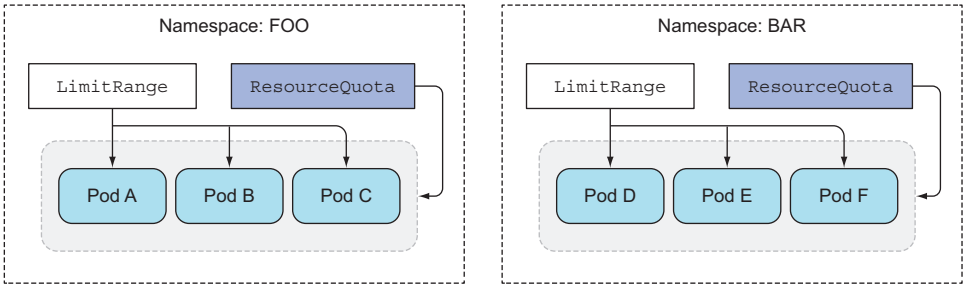

Figure 14.7 LimitRanges apply to individual pods; ResourceQuotas apply to all pods in the namespace.

INSPECTING THE QUOTA AND QUOTA USAGE

After you post the ResourceQuota object to the API server, you can use the `kubectl describe` command to see how much of the quota is already used up, as shown in the following listing.

Listing 14.14 Inspecting the ResourceQuota with `kubectl describe quota`

```
$ kubectl describe quota
Name:           cpu-and-mem
Namespace:      default
Resource        Used   Hard
--------        ----   ----
limits.cpu      200m   600m
limits.memory   100Mi  500Mi
requests.cpu    100m   400m
requests.memory 10Mi   200Mi
```

I only have the `kubia-manual` pod running, so the `Used` column matches its resource requests and limits. When I run additional pods, their requests and limits are added to the used amounts.

CREATING A LIMITRANGE ALONG WITH A RESOURCEQUOTA

One caveat when creating a ResourceQuota is that you will also want to create a LimitRange object alongside it. In your case, you have a LimitRange configured from the previous section, but if you didn't have one, you couldn't run the `kubia-manual` pod, because it doesn't specify any resource requests or limits. Here's what would happen in that case:

```
$ kubectl create -f ../Chapter03/kubia-manual.yaml
Error from server (Forbidden): error when creating "../Chapter03/kubia-
    manual.yaml": pods "kubia-manual" is forbidden: failed quota: cpu-and-
    mem: must specify limits.cpu,limits.memory,requests.cpu,requests.memory
```

When a quota for a specific resource (CPU or memory) is configured (request or limit), pods need to have the request or limit (respectively) set for that same resource; otherwise the API server will not accept the pod. That's why having a LimitRange with defaults for those resources can make life a bit easier for people creating pods.

14.5.2 Specifying a quota for persistent storage

A ResourceQuota object can also limit the amount of persistent storage that can be claimed in the namespace, as shown in the following listing.

Listing 14.15 A ResourceQuota for storage: quota-storage.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage
spec:
  hard:
    requests.storage: 500Gi
    ssd.storageclass.storage.k8s.io/requests.storage: 300Gi
    standard.storageclass.storage.k8s.io/requests.storage: 1Ti
```

The amount of storage claimable overall

The amount of claimable storage in StorageClass ssd

In this example, the amount of storage all PersistentVolumeClaims in a namespace can request is limited to 500 GiB (by the `requests.storage` entry in the ResourceQuota object). But as you'll remember from chapter 6, PersistentVolumeClaims can request a dynamically provisioned PersistentVolume of a specific StorageClass. That's why Kubernetes also makes it possible to define storage quotas for each StorageClass individually. The previous example limits the total amount of claimable SSD storage (designated by the `ssd` StorageClass) to 300 GiB. The less-performant HDD storage (StorageClass *standard*) is limited to 1 TiB.

14.5.3 Limiting the number of objects that can be created

A ResourceQuota can also be configured to limit the number of Pods, ReplicationControllers, Services, and other objects inside a single namespace. This allows the cluster admin to limit the number of objects users can create based on their payment

plan, for example, and can also limit the number of public IPs or node ports Services can use.

The following listing shows what a ResourceQuota object that limits the number of objects may look like.

Listing 14.16 A ResourceQuota for max number of resources: quota-object-count.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: objects
spec:
  hard:
    pods: 10
    replicationcontrollers: 5
    secrets: 10
    configmaps: 10
    persistentvolumeclaims: 4
    services: 5
    services.loadbalancers: 1
    services.nodeports: 2
    ssd.storageclass.storage.k8s.io/persistentvolumeclaims: 2
```

Only 10 Pods, 5 ReplicationControllers, 10 Secrets, 10 ConfigMaps, and 4 PersistentVolumeClaims can be created in the namespace.

Five Services overall can be created, of which at most one can be a LoadBalancer Service and at most two can be NodePort Services.

Only two PVCs can claim storage with the ssd StorageClass.

The ResourceQuota in this listing allows users to create at most 10 Pods in the namespace, regardless if they're created manually or by a ReplicationController, ReplicaSet, DaemonSet, Job, and so on. It also limits the number of ReplicationControllers to five. A maximum of five Services can be created, of which only one can be a `LoadBalancer`-type Service, and only two can be `NodePort` Services. Similar to how the maximum amount of requested storage can be specified per StorageClass, the number of PersistentVolumeClaims can also be limited per StorageClass.

Object count quotas can currently be set for the following objects:

- Pods
- ReplicationControllers
- Secrets
- ConfigMaps
- PersistentVolumeClaims
- Services (in general), and for two specific types of Services, such as `LoadBalancer` Services (`services.loadbalancers`) and `NodePort` Services (`services.nodeports`)

Finally, you can even set an object count quota for ResourceQuota objects themselves. The number of other objects, such as ReplicaSets, Jobs, Deployments, Ingresses, and so on, cannot be limited yet (but this may have changed since the book was published, so please check the documentation for up-to-date information).

14.5.4 Specifying quotas for specific pod states and/or QoS classes

The quotas you've created so far have applied to all pods, regardless of their current state and QoS class. But quotas can also be limited to a set of *quota scopes*. Four scopes are currently available: `BestEffort`, `NotBestEffort`, `Terminating`, and `NotTerminating`.

The `BestEffort` and `NotBestEffort` scopes determine whether the quota applies to pods with the `BestEffort` QoS class or with one of the other two classes (that is, `Burstable` and `Guaranteed`).

The other two scopes (`Terminating` and `NotTerminating`) don't apply to pods that are (or aren't) in the process of shutting down, as the name might lead you to believe. We haven't talked about this, but you can specify how long each pod is allowed to run before it's terminated and marked as `Failed`. This is done by setting the `activeDeadlineSeconds` field in the pod spec. This property defines the number of seconds a pod is allowed to be active on the node relative to its start time before it's marked as `Failed` and then terminated. The `Terminating` quota scope applies to pods that have the `activeDeadlineSeconds` set, whereas the `NotTerminating` applies to those that don't.

When creating a ResourceQuota, you can specify the scopes that it applies to. A pod must match all the specified scopes for the quota to apply to it. Additionally, what a quota can limit depends on the quota's scope. `BestEffort` scope can only limit the number of pods, whereas the other three scopes can limit the number of pods, CPU/memory requests, and CPU/memory limits.

If, for example, you want the quota to apply only to `BestEffort`, `NotTerminating` pods, you can create the ResourceQuota object shown in the following listing.

Listing 14.17 ResourceQuota for `BestEffort/NotTerminating` pods: quota-scoped.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort-notterminating-pods
spec:
  scopes:                          This quota only applies to pods
  - BestEffort                     that have the BestEffort QoS and
  - NotTerminating                 don't have an active deadline set.
  hard:
    pods: 4                        Only four such
                                   pods can exist.
```

This quota ensures that at most four pods exist with the `BestEffort` QoS class, which don't have an active deadline. If the quota was targeting `NotBestEffort` pods instead, you could also specify `requests.cpu`, `requests.memory`, `limits.cpu`, and `limits.memory`.

NOTE Before you move on to the next section of this chapter, please delete all the ResourceQuota and LimitRange resources you created. You won't

need them anymore and they may interfere with examples in the following chapters.

14.6 Monitoring pod resource usage

Properly setting resource requests and limits is crucial for getting the most out of your Kubernetes cluster. If requests are set too high, your cluster nodes will be underutilized and you'll be throwing money away. If you set them too low, your apps will be CPU-starved or even killed by the OOM Killer. How do you find the sweet spot for requests and limits?

You find it by monitoring the actual resource usage of your containers under the expected load levels. Once the application is exposed to the public, you should keep monitoring it and adjust the resource requests and limits if required.

14.6.1 Collecting and retrieving actual resource usages

How does one monitor apps running in Kubernetes? Luckily, the Kubelet itself already contains an agent called cAdvisor, which performs the basic collection of resource consumption data for both individual containers running on the node and the node as a whole. Gathering those statistics centrally for the whole cluster requires you to run an additional component called Heapster.

Heapster runs as a pod on one of the nodes and is exposed through a regular Kubernetes Service, making it accessible at a stable IP address. It collects the data from all cAdvisors in the cluster and exposes it in a single location. Figure 14.8 shows the flow of the metrics data from the pods, through cAdvisor and finally into Heapster.

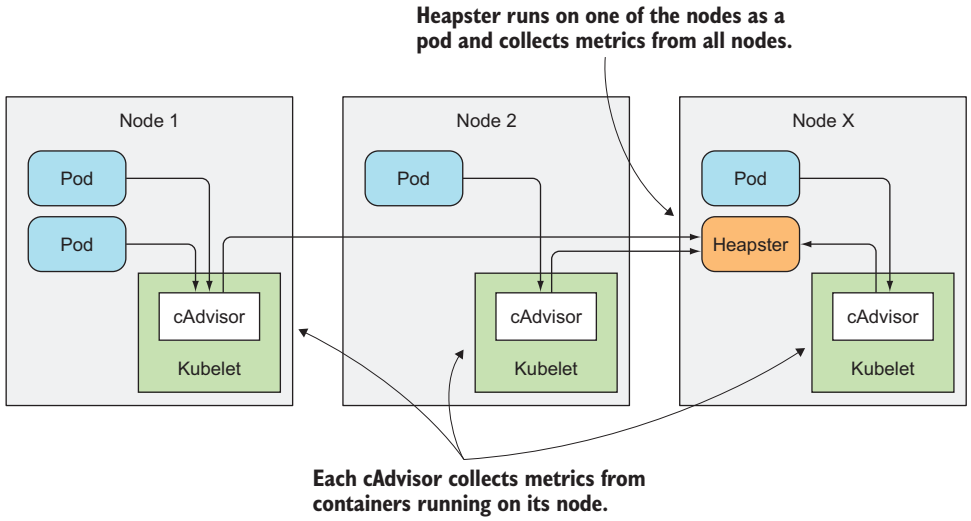

Figure 14.8 The flow of metrics data into Heapster

The arrows in the figure show how the metrics data flows. They don't show which component connects to which to get the data. The pods (or the containers running therein) don't know anything about cAdvisor, and cAdvisor doesn't know anything about Heapster. It's Heapster that connects to all the cAdvisors, and it's the cAdvisors that collect the container and node usage data without having to talk to the processes running inside the pods' containers.

Enabling Heapster

If you're running a cluster in Google Kubernetes Engine, Heapster is enabled by default. If you're using Minikube, it's available as an add-on and can be enabled with the following command:

```
$ minikube addons enable heapster
heapster was successfully enabled
```

To run Heapster manually in other types of Kubernetes clusters, you can refer to instructions located at https://github.com/kubernetes/heapster.

After enabling Heapster, you'll need to wait a few minutes for it to collect metrics before you can see resource usage statistics for your cluster, so be patient.

Displaying CPU and Memory usage for cluster nodes

Running Heapster in your cluster makes it possible to obtain resource usages for nodes and individual pods through the `kubectl top` command. To see how much CPU and memory is being used on your nodes, you can run the command shown in the following listing.

Listing 14.18 Actual CPU and memory usage of nodes

```
$ kubectl top node
NAME       CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
minikube   170m         8%        556Mi           27%
```

This shows the actual, current CPU and memory usage of all the pods running on the node, unlike the `kubectl describe node` command, which shows the amount of CPU and memory requests and limits instead of actual runtime usage data.

Displaying CPU and Memory usage for individual pods

To see how much each individual pod is using, you can use the `kubectl top pod` command, as shown in the following listing.

Listing 14.19 Actual CPU and memory usages of pods

```
$ kubectl top pod --all-namespaces
NAMESPACE      NAME                              CPU(cores)   MEMORY(bytes)
kube-system    influxdb-grafana-2r2w9            1m           32Mi
kube-system    heapster-40j6d                    0m           18Mi
```

```
default         kubia-3773182134-63bmb           0m           9Mi
kube-system     kube-dns-v20-z0hq6               1m           11Mi
kube-system     kubernetes-dashboard-r53mc       0m           14Mi
kube-system     kube-addon-manager-minikube      7m           33Mi
```

The outputs of both these commands are fairly simple, so you probably don't need me to explain them, but I do need to warn you about one thing. Sometimes the `top pod` command will refuse to show any metrics and instead print out an error like this:

```
$ kubectl top pod
W0312 22:12:58.021885   15126 top_pod.go:186] Metrics not available for pod
     default/kubia-3773182134-63bmb, age: 1h24m19.021873823s
error: Metrics not available for pod default/kubia-3773182134-63bmb, age:
     1h24m19.021873823s
```

If this happens, don't start looking for the cause of the error yet. Relax, wait a while, and rerun the command—it may take a few minutes, but the metrics should appear eventually. The `kubectl top` command gets the metrics from Heapster, which aggregates the data over a few minutes and doesn't expose it immediately.

TIP To see resource usages across individual containers instead of pods, you can use the `--containers` option.

14.6.2 Storing and analyzing historical resource consumption statistics

The `top` command only shows current resource usages—it doesn't show you how much CPU or memory your pods consumed throughout the last hour, yesterday, or a week ago, for example. In fact, both cAdvisor and Heapster only hold resource usage data for a short window of time. If you want to analyze your pods' resource consumption over longer time periods, you'll need to run additional tools.

When using Google Kubernetes Engine, you can monitor your cluster with Google Cloud Monitoring, but when you're running your own local Kubernetes cluster (either through Minikube or other means), people usually use InfluxDB for storing statistics data and Grafana for visualizing and analyzing them.

INTRODUCING INFLUXDB AND GRAFANA

InfluxDB is an open source time-series database ideal for storing application metrics and other monitoring data. Grafana, also open source, is an analytics and visualization suite with a nice-looking web console that allows you to visualize the data stored in InfluxDB and discover how your application's resource usage behaves over time (an example showing three Grafana charts is shown in figure 14.9).

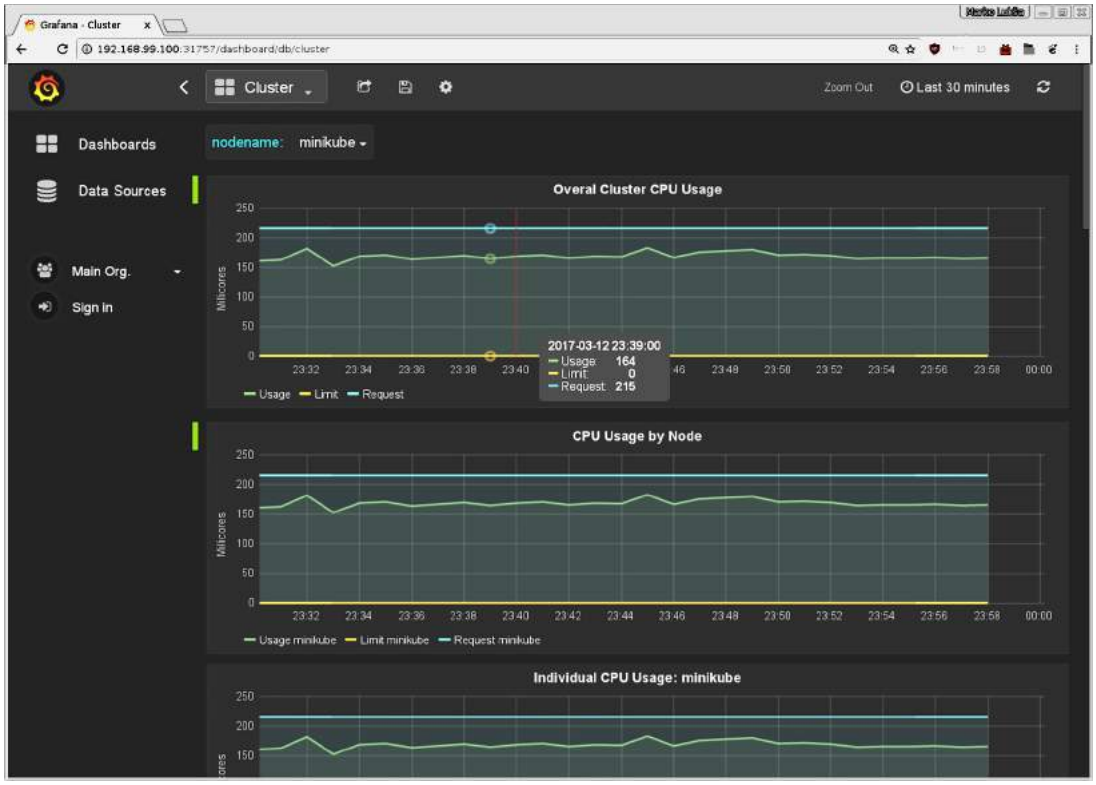

Figure 14.9 Grafana dashboard showing CPU usage across the cluster

Running InfluxDB and Grafana in your cluster

Both InfluxDB and Grafana can run as pods. Deploying them is straightforward. All the necessary manifests are available in the Heapster Git repository at http://github.com/kubernetes/heapster/tree/master/deploy/kube-config/influxdb.

When using Minikube, you don't even need to deploy them manually, because they're deployed along with Heapster when you enable the Heapster add-on.

Analyzing resource usage with Grafana

To discover how much of each resource your pod requires over time, open the Grafana web console and explore the predefined dashboards. Generally, you can find out the URL of Grafana's web console with `kubectl cluster-info`:

```
$ kubectl cluster-info
...
monitoring-grafana is running at
     https://192.168.99.100:8443/api/v1/proxy/namespaces/kube-
     system/services/monitoring-grafana
```

When using Minikube, Grafana's web console is exposed through a `NodePort` Service, so you can open it in your browser with the following command:

```
$ minikube service monitoring-grafana -n kube-system
Opening kubernetes service kube-system/monitoring-grafana in default
     browser...
```

A new browser window or tab will open and show the Grafana Home screen. On the right-hand side, you'll see a list of dashboards containing two entries:

- Cluster
- Pods

To see the resource usage statistics of the nodes, open the Cluster dashboard. There you'll see several charts showing the overall cluster usage, usage by node, and the individual usage for CPU, memory, network, and filesystem. The charts will not only show the actual usage, but also the requests and limits for those resources (where they apply).

If you then switch over to the Pods dashboard, you can examine the resource usages for each individual pod, again with both requests and limits shown alongside the actual usage.

Initially, the charts show the statistics for the last 30 minutes, but you can zoom out and see the data for much longer time periods: days, months, or even years.

USING THE INFORMATION SHOWN IN THE CHARTS

By looking at the charts, you can quickly see if the resource requests or limits you've set for your pods need to be raised or whether they can be lowered to allow more pods to fit on your nodes. Let's look at an example. Figure 14.10 shows the CPU and memory charts for a pod.

At the far right of the top chart, you can see the pod is using more CPU than was requested in the pod's manifest. Although this isn't problematic when this is the only pod running on the node, you should keep in mind that a pod is only guaranteed as much of a resource as it requests through resource requests. Your pod may be running fine now, but when other pods are deployed to the same node and start using the CPU, your pod's CPU time may be throttled. Because of this, to ensure the pod can use as much CPU as it needs to at any time, you should raise the CPU resource request for the pod's container.

The bottom chart shows the pod's memory usage and request. Here the situation is the exact opposite. The amount of memory the pod is using is well below what was requested in the pod's spec. The requested memory is reserved for the pod and won't be available to other pods. The unused memory is therefore wasted. You should decrease the pod's memory request to make the memory available to other pods running on the node.

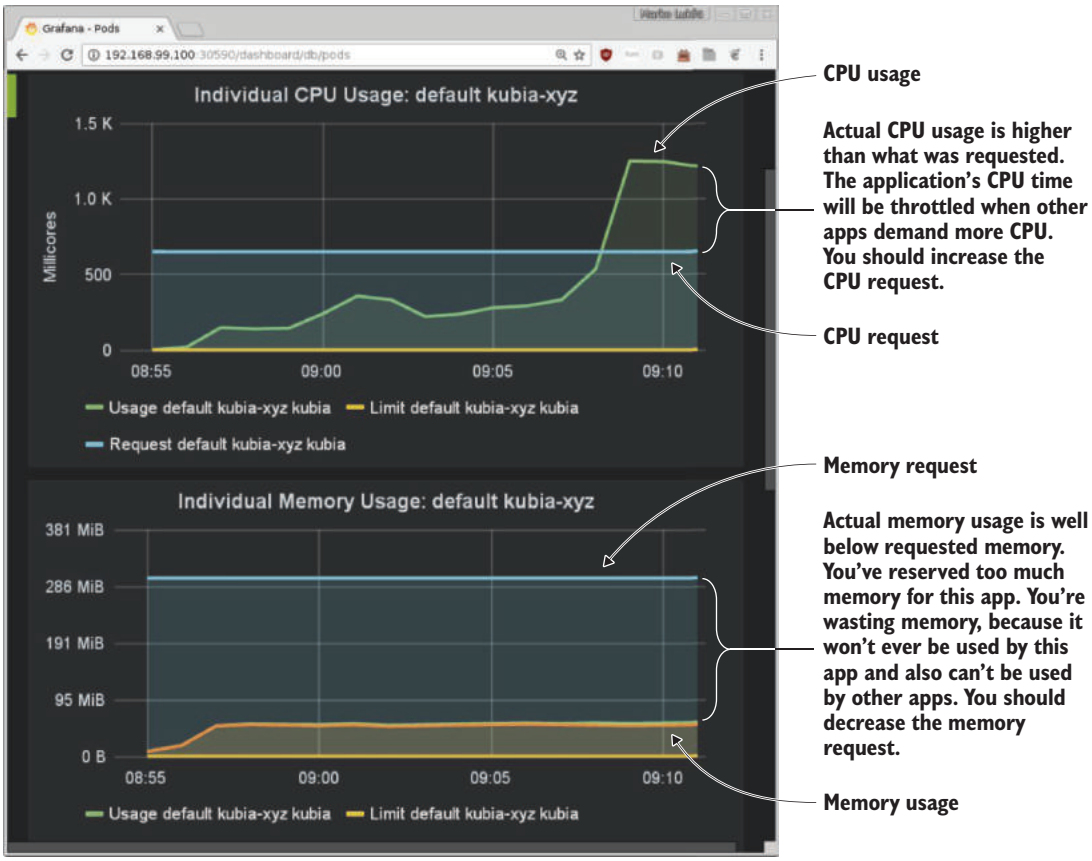

Figure 14.10 CPU and memory usage chart for a pod

14.7 Summary

This chapter has shown you that you need to consider your pod's resource usage and configure both the resource requests and the limits for your pod to keep everything running smoothly. The key takeaways from this chapter are

- Specifying resource requests helps Kubernetes schedule pods across the cluster.
- Specifying resource limits keeps pods from starving other pods of resources.
- Unused CPU time is allocated based on containers' CPU requests.
- Containers never get killed if they try to use too much CPU, but they are killed if they try to use too much memory.
- In an overcommitted system, containers also get killed to free memory for more important pods, based on the pods' QoS classes and actual memory usage.

- You can use LimitRange objects to define the minimum, maximum, and default resource requests and limits for individual pods.
- You can use ResourceQuota objects to limit the amount of resources available to all the pods in a namespace.
- To know how high to set a pod's resource requests and limits, you need to monitor how the pod uses resources over a long-enough time period.

In the next chapter, you'll see how these metrics can be used by Kubernetes to automatically scale your pods.