Kubernetes - Understanding Resources

By Ben Visser on Dec 28, 2016

If you don't already know, Kubernetes is a container scheduling system for Docker and rkt (at the moment). In addition to other nice features like easy deployments, configuration management, service discovery and more, it allows us to manage compute resources in a much more effective manner. In this blog post I will explain how the Kubernetes resource model works, why you should **ALWAYS** set resource limits on your containers, and then show you how you can do exactly that.

THE NEED FOR RESOURCE MANAGEMENT

Before Kubernetes came along, a common method of running containers would be to punt your application container onto an instance and hopefully also set up a monitoring system to automatically restart your container in case it exits. The problem with this model is that your application perhaps only uses 10% of total CPU available on the instance. You're effectively wasting 90% of your CPU resources. With Kubernetes, discrete instances are combined into a pool of compute resources where multiple applications can scheduled to run on one physical instance. It's like "taking a massive pile of wooden blocks (containers/tasks) - blocks of all different shapes and sizes — and finding a way to pack all those blocks into buckets (servers)"[1]. If you can arrange these blocks (tasks) very carefully then you can have fewer buckets (servers).

However, there is a new risk of resource exhaustion that appears when running many containers on one instance. If your container suddenly tries to use 100% of CPU, there is nothing to stop it from exhausting all other containers of their CPU. Well, there is, this is where the Kubernetes resource model comes in. Now that I hooked you with a financial incentive and the risk of resource exhuastion, let me explain how it all works.

THE RESOURCE MODEL

What exactly is a resource in Kubernetes?

A resource is something that can be "requested by", "allocated to", "or consumed by" a pod or container. Examples: CPU, RAM, network bandwidth.

They can be either *compressible* (easily throttled) or *incompressible* (not easily throttled). Memory is incompressible, while CPU and network are compressible because they are easily throttled.

These resources can be categorized into two different states: desired state (specification) and current state (status). Resource requirements and resource capacity can be thought of as specifications (desired state) and resource usage can be thought of as status (current state). The Kubernetes scheduler can use these two states to reason about capabilities of nodes, resource requirements etc.

We can describe the specification for a resource in terms of "limit" and "request":

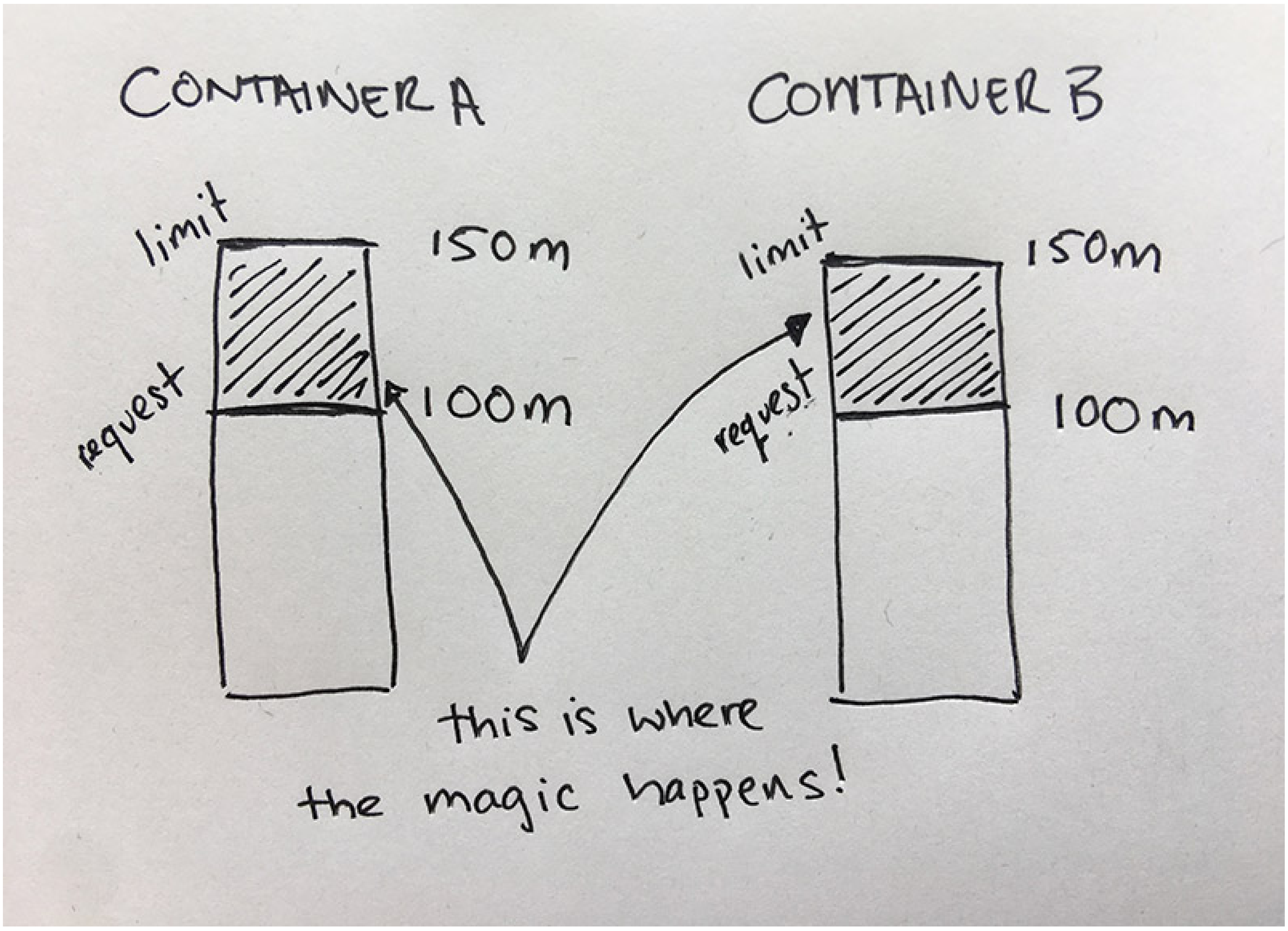
- **request**: the amount of resources being requested for a container. If a container exceeds its request for resources, it may be throttled back down to it's request.
- **limit**: an upper cap on the resources a container is able to use. If it tries to exceed this limit it may be terminated if Kubernetes decides that another container need the resources. Usually it makes sense to have the sum of all container resource limits equal the total resource capacity for your cluster (but this is a little hard to do in practice with incompressible resources like memory).

In the case that request is omitted for a container, it defaults to limit. If limit is not set, then if defaults to 0 (unbounded). As you can see, "request" is a soft limit on resources, while "limit" is a hard limit on how many resources a container can use. It then makes sense in practice to set request as a fraction of the limit of a container.

RESOURCE SCHEDULING

When a container is getting ready to start, the Kubernetes scheduler selects a instance for the container to run on. The scheduler ensures that for each resource type, the sum of the resource requests never exceed the total resource capacity of the node. In other words, over-provisioning of resources is not allowed, but **it has been hinted** (https://github.com/kubernetes/kubernetes/blame/master/docs/design/resources.md#L33-L34) that it may be included in the future. If the capacity check fails, then the scheduler will not put a container on the instance.

For example, please see the following chart.



In the very simplistic example above, container A and B have identical CPU request and CPU limit of 100m and 150m respectively. The space in between the request and limit for each container is a space where Kubernetes resource distributing algorithms live and work to ensure each container receives the resources it needs.

SUPPORTED RESOURCES

There a currently two supported resources to limit:

name	units	compressible
CPU	millicores	yes
Memory	bytes	no

Some resources yet to be implemented are storage time, storage space, storage operations, network bandwidth, network operations.

One thing to note here is that CPU is always an absolute quantity, not a relative quantity (like 40% of all CPU), but rather 0.5 of one CPU. The unit of CPU resources is millicores, which is 1/1000 of one core. With one core being equal to 1 vCPU on a supported cloud provider.

SETTING RESOURCE LIMITS

There’s exactly two good reasons you should be setting resource requests and resource limits for each container.

You should set a resource request so that Kubernetes can do a good job of scheduling containers on different instances to use as much potential capacity as possible. You should set a resource limit so that in the case you have a rogue container, it can’t eat up all the resources on your instance and affect other applications running on that same instance.

This is why you should **ALWAYS** set both resource requests and resource limits.

CONTAINER RESOURCE LIMITS

Unfortunately, Kubernetes has not yet implemented dynamic resource management, which is why we have to set resource limits for our containers. I imagine that at some point Kubernetes will start implementing a less manual way to manage resources, but this is all we have for now.

It's a common case that you're trying to deploy a new application and you don't exactly know how many resources it will use. In this case, just try a high estimate and you can always dial it back to a lower limit if need be.

This is an example of a spec setting resource limits on containers in a pod. It has a pod limit of 1000m and 256MiB of memory. It has a pod request of 500m CPU and 128MiB of memory. The request and limit of a pod is always the sum of request and limit of the containers contained within it.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

You can save this YAML to a file and set this pod

```
kubectl apply -f pod.yaml --namespace=development
```

NAMESPACE RESOURCE LIMITS

You can also set resource limits on namespaces if you wish. For example, this may be useful if you have a development and production namespace, and developers are testing containers on the development namespace without any kind of resource limits on their containers. Setting a resource limit on the development namespace would allow you to ensure that if a developer accidentally uses too many resources in development, it doesn't affect your production applications.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
spec:
  hard:
    cpu: "20"
    memory: 1Gi
    pods: "10"
    replicationcontrollers: "20"
    resourcequotas: "1"
    services: "5"
```

You can save this YAML and then apply the resource quota to the namespace

```
kubectl create -f resource-quota.yaml --namespace=development
```

As you might notice you can also set limits on Kubernetes objects, like services and replicationControllers. The full list of resources/objects you can limit in a namespace are listed **here** (http://kubernetes.io/docs/admin/resourcequota/). And a more advanced walk-through of namespace quotas can be found **here** (http://kubernetes.io/docs/admin/resourcequota/walkthrough/).

¹ John Wilkes - https://www.wired.com/2013/03/google-borg-twitter-mesos/

Updated Saturday, Jun 24, 2017

1 Comment **noqcks.io** 1 **Login**

Recommend Share **Sort by Best**

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Timo Reimann • 9 months ago

Excellent read, thanks for the post.

I'm wondering though if the "overcommitting is not supported" remark is still true: The resource QoS design doc at https://github.com/kubernet... states that overcommitting occurs when the sum of all limits is greater than the system capacity, which AFAICS may be true as soon as one pod defines a resource request smaller than its limit. This motivates the three QoS classes and defines the priorities by which different types of pods are evicted if the system capacity is exceeded (even though it only works for memory right now).

• Reply • Share ›

ALSO ON **NOQCKS.IO**

Dead Simple GitHub Releases for Golang Projects

1 comment • a year ago

Etienne Rocheleau — Very nice! Thank you so much it was very helpful in getting my project automatically built/released with Travis.I had to change a few things:First, I changed the VERSION setup to use Travis tag, …

Kubernetes SSL for AWS ELBs · Ben Visser - Toronto - DevOps

1 comment • 10 months ago

Brent Dorsey — Awesome article Ben! You saved me a ton of time, I found your article in the results of my first "aws-load-balancer-ssl" Google search.This functionality also requires an IAM user or instance role with …

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Privacy**