KUBERNETES – CPU AND MEMORY MANAGEMENT

JOYDEEP Banerjee
Joydeep@Banerjee1@ibm.com

Table of Contents

Introduction

If you look at a Kubernetes ecosystem and how it manages the underlying infrastructure in detail, you will soon realize that Kubernetes manages complexity inherently. Therefore, instead of getting lost in a sea of metrics, we need to look at a few key ones to understand the state in which Kubernetes is and figure out if any external intervention is needed. If you are developing on Kubernetes or managing a Kubernetes based system, read the following sections.

Key Dynamics

Let us now try to understand some of the key dynamics in the Kubernetes system and thereby uncover the important metrics that we want to focus on. We will focus on two of the core components of Kubernetes. The ***Scheduler*** which runs on the Master Node and the ***kubelet*** which runs on each of the Worker Nodes. We will follow the journey of a Pod and try to understand how these 2 critical pieces work. A user creates a Pod (Kubernetes controller may do it as well – but that is for another day). At this stage the Pod is not assigned to a node as yet. The ***scheduler*** is watching for Pods which are not assigned to any node yet. The ***scheduler*** will assign the Pod to a Node following certain guidelines. ***Kubelet*** gets into action after Pod is assigned. It is watching for Pods which has been assigned to a node but is not running. ***Kubelet*** will now execute the Pod or in other words – start the Pod.

Before we can go further, we need to understand a few terms. CPU and memory are each a *resource type*. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. CPU and memory are collectively referred to as *compute resources*, or just *resources*. *Resources* are measurable quantities that can be requested, allocated, and consumed.

Request and Limits

Specifically, for each *resource*, containers specify a ***request***, which is the amount of that resource that the system will guarantee to the container, and a ***limit*** which is the maximum amount that the system will allow the container to use. How the ***request*** and ***limit*** are enforced depends on whether the *resource* is compressible (CPU) or incompressible (Memory or Disk).

The system computes pod level ***requests*** and ***limits*** by summing up per-resource ***requests*** and ***limits*** across all containers. When ***request == limit***, the resources are guaranteed, and when ***request < limit***, the pod is guaranteed the ***request*** but can opportunistically scavenge the difference between ***request*** and ***limit*** if they are not being used by other containers. This allows Kubernetes to oversubscribe nodes, which increases utilization, while at the same time maintaining resource guarantees for the containers that need guarantees. How the ***request*** and ***limit*** are enforced depends on whether the *resource* is compressible (CPU) or incompressible (Memory or Disk). We will be talking more about this soon.

Scheduler

Next, let us try to understand the working of the ***scheduler*** in more details. Among the many guidelines that the scheduler follows in assigning a Pod to a Node, one of the checks is `PodFitsResources`. This checks if the Node has free resources (CPU and Memory) to meet the requirement of the Pod.

PodFitsResources determines a fit based on resource availability. Fitness is based on requested resources rather than usage. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for the pods. The scheduler ensures that, for each *resource type*, the sum of the ***resource requests*** of the scheduled containers is less than the capacity of the node. To state differently, Kubernetes scheduler does not base its calculation on actual usage; it does its calculation based on declared ***CPU Capacity or Memory Capacity of the Node*** and ***Request Memory or Request CPU*** – therefore it is possible that the actual memory or the CPU usage on nodes is low, but the scheduler refuses to place a pod on a node if the capacity check fails. Concretely:

- A **Sum of** ***CPU Request*** of all the containers on the node is evaluated against the ***CPU capacity*** of the node and then the pods are scheduled. If there is not enough room, the pod remains unscheduled.
- The **Sum of** ***Request Memory*** of all the containers on the node is evaluated against the ***Memory Capacity*** of Node and then the pods are scheduled. If there is not enough memory room, the pod remains unscheduled.

A failed scheduling event is produced each time the ***scheduler*** fails to find a place for the Pod. It is important to check this event as scheduling may fail for a number of other reasons too.

Kubelet

Now let us try to focus on the ***kubelet***. The ***kubelet*** as explained earlier, does watch for Pod states not matching what it is supposed to be and acts to restore it – for example if Pod needs to be running but is in a stopped state, it will start the Pod. The ***kubelet*** also needs to preserve node stability when available compute resources are low. This is especially important when dealing with incompressible compute resources, such as memory or disk space. If such resources are exhausted, nodes become unstable.

For compressible resource like CPU - Pods will not be killed by ***kubelet*** if CPU guarantees cannot be met, they will be only temporarily throttled (using Linux control groups under the covers)

The ***kubelet*** can proactively monitor for and prevent total starvation of a compute resource. In those cases, the ***kubelet*** can reclaim the starved resource by proactively failing one or more Pods. When the ***kubelet*** fails a Pod, it terminates all of its containers and transitions its *PodPhase* to *Failed*. If the evicted Pod is managed by a *Deployment* controller, the *Deployment* controller will create another Pod to be scheduled by Kubernetes.

The *kubelet* supports eviction decisions based on the free memory available at the node. If the free memory threshold is exceeded, or in other words, ***Node is under memory pressure***, the *kubelet* then:

1. Firstly, tries to reclaim node level memory resources prior to evicting pods
2. If it cannot reclaim enough memory without evicting pods, then it starts to evict pods. The **kubelet** ranks Pods for eviction first by whether or not their usage of the memory exceeds ***memory request***, then by *Priority* (term explained later in this document), and then by the consumption of the memory relative to the Pods' scheduling requests. There are lots of details under the hood. Look into *Further Readings* section below.

The node reports a condition when a compute resource is under pressure. The scheduler views that condition as a signal to dissuade placing additional pods on the node. Therefore, when the Node Memory Pressure is signaled, the Scheduler does not schedule any new *BestEffort* Pods (containers in the Pod does not have memory or CPU limits or requests specified)

There is an interesting nuance. The *kubelet* currently polls *cAdvisor* to collect memory usage stats at a regular interval. If memory usage increases between the collection window rapidly, the **kubelet** may not trigger *MemoryPressure* fast enough before a *system Out of Memory*. In that case, the *OOMKiller* will be invoked.

Unlike Pod eviction, if a container is OOM killed, it may be restarted by the *kubelet* based on its *RestartPolicy*. Recall if the evicted Pod is managed by a Deployment controller, the Deployment controller will create another Pod to be scheduled by Kubernetes.

There is another scenario. Without any resource issues at the node level, a container may simply be trying to exceed the ***memory limit***. In this case, the Linux control groups step in and kill the container. In this case, as mentioned previously, the kubelet will restart the container.

It is useful to compare the ***CPU capacity*** against the ***CPU usage*** at the node level. When these two values get closer, you know that your node may run out of actual capacity. When you compare the ***CPU capacity*** at the node level with the ***sum of the CPU requests for all the containers***, you are also warned of impending scheduling problems.

Tools for Control

You could set:

- The default CPU and Memory Request and Limits for each container in a namespace – if these values are not set explicitly at the container level, it inherits the defaults. This is done using the *LimitRange* object.
- The maximum and minimum CPU and Memory for each container in a namespace – if a container limit and request tries to get out of these bounds, it fails. This is done using the *LimitRange* object.
- The quota for total CPU and memory consumption allowed in a namespace – aggregated container CPU and Memory request and limit can never exceed the quota set. This sets the maximum allowed value for sum of all containers in the namespace on CPU requests and limit and Memory requests and limits. This is done using the *ResourceQuota* object.

Key Facts

	CPU	Memory
POD Scheduling	A Pod is scheduled to run on a Node only if the Node	A Pod is scheduled to run on a Node only if the Node

	has enough CPU resources available to satisfy the Pod *CPU request*. CPU Capacity of Node is compared with Σ(CPU Request of all containers in Node) and Scheduler aborts POD scheduling when this falls below a threshold. The POD will remain in Pending state.	has enough available memory to satisfy the Pod's ***memory request***. Memory Capacity of Node is compared with Σ(Memory Request of all containers in Node) and Scheduler aborts POD scheduling when this falls below a threshold. The POD will remain in Pending state.
POD Eviction	CPU is compressible resource. So, Pods do not get evicted for CPU usage – rather, CPU gets throttled.	Node Memory pressure signal may evict Pods; And this also sends signal to Scheduler to stop scheduling on that Node. If Node memory pressure is not contained, the System OOMKiller will trigger.
No Limits specified	If you do not specify a CPU limit for a Container, the Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available	If you do not specify a memory limit for a Container, the Container has no upper bound on the amount of memory it uses. The Container could use all of the memory available on

	on the Node where it is running.	the Node where it is running.
Exceeds request value	Pods are guaranteed to get the amount of CPU they request, they may or may not get additional CPU time (depending on the other jobs running). Excess CPU resources will be distributed based on the amount of CPU requested. For example, suppose container A requests for 600 milli CPUs, and container B requests for 300 milli CPUs. Suppose that both containers are trying to use as much CPU as they can. Then the extra 100 milli CPUs will be distributed to A and B in a 2:1 ratio (implementation discussed in later sections).	If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory (for example when another Pod needs the memory)
Exceeds limit value	A Container might or might not be allowed to exceed its CPU limit for extended periods of time.	If a Container exceeds its memory limit, it will be terminated. If it is restartable, the kubelet will

	However, it will not be killed for excessive CPU usage. Pods will be throttled if they exceed their limit. If limit is unspecified, then the pods can use excess CPU when available.	restart it, as with any other type of runtime failure.

Deep Dive Memory

In case you are wondering about the complexities of how Kubernetes monitors memory, it is useful to think that there are 2 parallel systems of monitoring memory usage at the Kubernetes Nodes:

1. Kubelet watches for memory usage based on metrics from *cadvisor*. Under the cover, kubelet uses docker and docker uses Linux control group to keep tab on the memory usage of a container. When Kubelet does eviction after grace period, **kubelet** kills the Pod immediately with no graceful termination.
2. Linux OOM Killer on the other hand watches for memory overreach at the system level.

It is desirable that the kubelet is able to contain the memory effectively without system (Kubernetes Node machine) getting so bad as to get a System OOM which would invoke the Linux OOM Killer. The kubelet has some support for influencing system behavior in response to a system OOM by having the system OOM killer see higher OOM score adjust scores for containers that have consumed the largest amount of memory relative to their request. System OOM events are very compute intensive and can stall the node until the OOM killing process has completed. In addition, the system is prone to return to an unstable state since the containers that are killed due to OOM are either restarted or a new pod is scheduled on to the node.

Deep Dive Priority and Pod QoS

Priority

Pods can have *priority*. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod.

Pod QoS

For a Pod to be given a ***QoS class of Guaranteed***:

- Every Container in the Pod must have a memory limit and a memory request, and they must be the same.
- Every Container in the Pod must have a CPU limit and a CPU request, and they must be the same.
- If a container specifies a limit but specifies no request, Kubernetes automatically assigns a request that matches the limit. This is true for both CPU and memory.

For a Pod to be given a ***QoS class of Burstable*** :

- The Pod does not meet the criteria for QoS class Guaranteed.
- At least one Container in the Pod has a memory or CPU request.

For a Pod to be given a ***QoS class of BestEffort***:

- the Containers in the Pod must not have any memory or CPU limits or requests.

Pod **Priority** and **QoS** are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler's preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by `PodDisruptionBudget` (more of that on another day).

The only component that considers both QoS and Pod priority is Kubelet out-of-resource eviction. The *kubelet* ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by Priority, and then by the consumption of the starved compute resource relative to the Pods' scheduling requests. As a result, kubelet ranks and evicts Pods in the following order:

- *BestEffort* or *Burstable* Pods whose usage of a starved resource exceeds its request. Such pods are ranked by Priority, and then usage above request.
- Guaranteed pods and *Burstable* pods whose usage is beneath requests are evicted last.
- *Guaranteed* Pods are guaranteed to never be evicted because of another Pod's resource consumption. If a system daemon (such as kubelet, docker, and journald) is consuming more resources than were reserved via system-reserved or kube-reserved allocations, and the node only has *Guaranteed* or *Burstable* Pods using less than requests remaining, then the node must choose to evict such a Pod in order to preserve node stability and to limit the impact of the unexpected consumption to other Pods. In this case, it will choose to evict pods of Lowest Priority first.

Recommendations

If you are a cluster administrator, you will want the system to be stable all the time and keep an eye when more capacity is needed. Therefore:

1. Monitor the Kubernetes Nodes machines for System OOM. System OOM is bad and should be prevented; it causes all sorts of stability issues.
2. Monitor Kubernetes Node machines for Capacity vs real Usage for CPU and Memory. These will signal when a new node may need to be added.
3. Check if any Pods are stuck in pending state for a long time and why. This also may signal capacity constraints.
4. Set resource quotas for CPU and Memory for each namespace. This may block Pods from getting created, but system stability is not compromised.

If you are a developer in charge of a microservice, you will want to make sure your code is behaving well and within limits:

1. Set Request and Limit values of the containers with utmost care.
2. If needed, watch out for containers which do not have Request and Limit not set.
3. Check if containers are repeatedly getting restarted and why.
4. Check if any Pods are stuck in pending state for a long time and why.
5. Check if containers are having their CPU throttled.

Further Readings:

1. https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/
2. https://kubernetes.io/docs/concepts/policy/resource-quotas/

3. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#resource-requests-and-limits-of-pod-and-container
4. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/#qos-classes
5. https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/#exceed-a-container-s-memory-limit
6. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/resource-qos.md
7. https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/
8. https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/
9. https://lwn.net/Articles/391222/
10. https://medium.com/@dominik.tornow/the-kubernetes-scheduler-cd429abac02f

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

The client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or