Bachelor thesis
Computing Science

Radboud University

Optimizing Kubernetes Cluster Down-Scaling for Resource Utilization

Author:
Joshua Steinmann
s1015908

First supervisor/assessor:
prof. dr. Frits Vaandrager
`F.Vaandrager@cs.ru.nl`

Second assessor:
prof. dr. Sven-Bodo Scholz
`svenbodo.scholz@ru.nl`

March 31, 2022

Abstract

Kubernetes is a widely adopted technology today. One of its main advantages is automatic deployment and even cluster scaling. The default procedure for scaling down a cluster has some shortcomings and does not allow for predictable behavior while decreasing the cluster size as much as possible. While the default implementation works with per node resource utilization thresholds to determine which nodes can be removed from the cluster, the procedure proposed in this thesis makes use of thresholds set for the whole cluster. These thresholds describe the maximal resource utilization that should be present in the cluster after a scale-down. The notion of usable resources is introduced to express the ratio of free resources in a cluster state more precisely. The proposed down-scaler works towards decreasing the cluster size as much as possible without violating any thresholds. While a similar behavior can be reproduced using the default implementation and overprovisioning, it does not work in all scenarios and involves complex additional deployments.

Contents

1 Introduction **3**

2 Kubernetes Concepts **4**
2.1 Microservices . . . 4
2.2 Container Orchestration . . . 4
2.3 Kubernetes Concepts . . . 5
2.3.1 Pod . . . 5
2.3.2 Replica Set . . . 5
2.3.3 Pod Disruption Budget . . . 5
2.4 Resources and labels . . . 6
2.4.1 Requests and limits . . . 6
2.4.2 Units . . . 6
2.4.3 Node Labels . . . 7
2.5 Scheduler . . . 7
2.6 Autoscaling . . . 7
2.6.1 Horizontal Pod Autoscaler . . . 8
2.6.2 Cluster Autoscaler (CA) . . . 8

3 Shortcomings of the default scale-down procedure **10**

4 An alternative down-scaling procedure **12**
4.1 The concept . . . 12
4.2 Utility functions . . . 12
4.2.1 Cluster capacity . . . 12
4.2.2 Node and cluster requests . . . 13
4.2.3 Daemon set requests . . . 14
4.3 Usable resources and usable capacity . . . 15
4.4 Candidate check . . . 17
4.5 Evaluating candidates . . . 18
4.6 The complete procedure . . . 19
4.7 Functions to be implemented . . . 20
4.7.1 Bin-packing . . . 20
4.7.2 Removing Node from Cluster . . . 20

4.8 The order of checks 20

5 Discussion **22**

5.1 Differences to the default implementation 22

5.1.1 Thresholds 22

5.1.2 General approach 22

5.2 Advantages of the proposed down-scaler 23

5.3 Advantages of the default down-scaler 23

5.4 Recreating the proposed behavior 23

5.4.1 Overprovisioning using the default CA 23

5.4.2 Recreating proposed behavior 24

5.4.3 Use in practice 24

6 Conclusions and Future Work **25**

Chapter 1

Introduction

Kubernetes and containerization are widely adopted technologies. Kubernetes is a platform for managing and distributing containerized workloads across multiple nodes. 31% of backend developers have used Kubernetes in the past 12 months (based on data from December of 2021 [11]). Only 11% have never heard of it. The adoption in the industry is accompanied by research efforts dedicated to improving the technology. One of the biggest advantages of using Kubernetes is the automatic scaling of application deployments or even the whole cluster to meet the computational power required at any moment.

Related Work

Qiang Wu et al. [13] present a cluster auto-scaler that determines the optimal cluster size based on the notion of Quality of Service (QoS). The QoS is defined by the response time of user requests. Given a maximal desired response time, the presented auto-scaler scales the cluster to not exceed the desired response time, while keeping the cluster as small as possible. The auto-scaler consists of four modules. The first one monitors the cluster state while the second one decides the QoS threshold. Based on that the third module computes the optimal cluster size and the fourth module then adjusts the cluster size to match the optimal size.

Related work by László Toka et al. [12] presents an auto-scaler that uses machine learning to predict the computational load in the near future. Multiple forecast methods compete, and based on the current request dynamics, the best method is given the lead. Based on the result, deployments are horizontally scaled (explained in section 2.6.1) to meet the predicted load. The influence on the cluster size is indirect as scaling deployments might result in cluster up or down-scaling, but the cluster size is not adjusted directly.

Chapter 2

Kubernetes Concepts

2.1 Microservices

The concept of microservices is a software architecture design pattern.
The naive way of designing an application is to build all required features into one monolithic service. Designing an application using the microservices pattern enforces the separation of feature groups into distinct services. These services mostly use network communication to provide the desired functionality. More information on microservices can be found in the book "Building Microservices" [9].
The relevant aspect of microservices for this thesis is their horizontal scalability. It describes the process of increasing the processing capability of a service by increasing the number of concurrent instances. The typical way to deploy microservices is to use containerization. By using containers the dependencies and service binaries are bundled in one image that can be deployed using a container runtime like docker or containerd.

2.2 Container Orchestration

Deploying an application as a collection of microservices implies that a multitude of containers needs to be deployed and managed. Container orchestrators, like the most prominent one Kubernetes, take care of this task. Google describes it as follows: "Google Cloud Platform provides a homogenous set of raw resources via virtual machines (VMs) to Kubernetes, and in turn, Kubernetes schedules containers to use those resources. This decoupling simplifies application development since users only ask for abstract resources like cores and memory, and it also simplifies data center operations." [8] Google continues to explain that Kubernetes also takes care of (inter-container) networking and persistent storage. For this thesis, the most relevant feature of Kubernetes is automatic scaling. The important terminology and the different concepts of autoscaling will be described in the following paragraphs.

2.3 Kubernetes Concepts

The following sections in this chapter are largely based on and frequently cite the official Kubernetes documentation [5].

2.3.1 Pod

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A pod's contents are always co-located and co-scheduled, and run in a shared context. A pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host. While pods provide more functionality in theory they are mostly used as a wrapper around a single container.

2.3.2 Replica Set

A replica set's purpose is to maintain a stable set of replica pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical pods. To do so, a replica set has a replication count and a template for the pod. Pods can be created based on the template or removed to match the replication count. Each pod can only be part of one replica set. Pods running on a node that fails get the state "Unknown" and are not counted as running replications. As a result, the replica set creates new pods on healthy nodes to reach the desired count of replications.

2.3.3 Pod Disruption Budget

A pod disruption budget specifies the desired minimum number of running instances of a pod in the ready state. This feature is useful for highly available applications as the pure amount of pods running the service is not enough to ensure availability. Pods in the creation or startup stage are counted by a replica set, but cannot provide their service yet. Pod disruption budgets ensure that the number of ready replications does not fall below the specified threshold by voluntary disruptions. A voluntary disruption is every operation on the cluster that is performed on purpose and causes a pod to be rescheduled. Host node failure obviously is not a voluntary disruption. Pod disruption budgets can be defined in terms of relative and absolute values for the minimum of available or the maximum of unavailable pods.

2.4 Resources and labels

The following sections explain the notion and units of resources as well as the concept of node labels.

2.4.1 Requests and limits

For each container requests, as well as limits can be specified for resource types. The most common resource types are CPU and RAM. For scheduling only the resource requests and limits of pods are relevant. Those are specified as the sum of requests and limits of all containers in the pod for each individual resource type. (Remember that pods are the smallest deployable unit of Kubernetes.) A request ensures that the particular resource is available for use by the pod. Hence the scheduler will not schedule another pod on a node the resources of which are already requested by running pods even though the current resource utilization might be low. The term ”request” is misleading in the way that the request is nothing that is granted or rejected, but more like a constraint for scheduling. If a request is set, the resources are always reserved if they are available and if the resources are not available the pod cannot be scheduled. A limit makes sure that the pod does not use more resources than the specified amount. If a container exceeds its memory limit, it might be terminated. If it is restartable, it will be restarted, as with any other type of runtime failure. If a container exceeds its memory request, its pod will likely be evicted whenever the node runs out of memory. A container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage. Setting resource requests and limits is a best practice to ensure that each pod has sufficient resources available, but also does not use more resources than it should. Misconfiguration or bugs might lead to excessive resource usage by a single pod otherwise [6]. This thesis assumes that all pods have resource requests specified for CPU and RAM.

2.4.2 Units

Limits and requests for CPU resources are measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. Fractional requests like 0.5 are allowed, but the unit ”millicpu” (m) is preferred as requests smaller than $1m$ are not allowed. A CPU request of 0.2 equals a request of $200m$. Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.

2.4.3 Node Labels

In Kubernetes, every object can be labeled. In the context of this thesis only node labels are relevant. A label is defined in terms of a key/value pair and can be used to express relevant differences in hardware for example. A cluster containing different types of nodes might have labels to express the differences in accelerators or drive types. Node type A might have especially fast SSDs, while node type B has a focus on the size of RAM. The according labels might look like this:
Node type A: { disk-type: nvme, memory-size: medium }
Node type B: { disk-type: sata-ssd, memory-size: large }
To ensure that pods housing workloads that benefit from special hardware run on the appropriate nodes there are required and preferred node (anti-)affinities as well as node selectors. A node selector requires the scheduler to place the pod on a node that has a certain label. Required node (anti-)affinity is essentially the same thing, while preferred node (anti-)affinity expresses a preference only. It does not prevent the pod to be scheduled on a node having or not having a certain label. Preferred node (anti-)affinities can also have a weight to balance multiple of the kind. A pod running an in-memory database might have a required node affinity for { memory-size: large } to not block all RAM on a node of a different kind. On the other hand, a pod running an application that needs a fast scratch disk might have a preferred node affinity for { disk-type: nvme } as the application can also run with slower disks without harming the cluster. The faster disks are preferred however to allow the application to run faster.

2.5 Scheduler

A scheduler watches for newly created pods that have no node assigned. For every pod that the scheduler discovers, the scheduler becomes responsible for finding the best node for that pod to run on. The default kube-scheduler selects a suitable node in two steps. The first step, the filtering step finds the set of nodes where it's feasible to schedule the pod. In the second step, the scoring step, the scheduler ranks the remaining nodes to choose the most suitable pod placement.

2.6 Autoscaling

Autoscaling is the automatic process of increasing or decreasing the computational capacity of a cluster or an application deployment.

2.6.1 Horizontal Pod Autoscaler

The horizontal pod auto-scaler adjusts the pod replication count of a replica set based on observed resource utilization. To allow scaling based on the desired resource metric a desired per pod utilization value needs to be set. The utilization value can be absolute or relative. The desired replication count is then computed using the following formula.

$$desiredReplicas = \lceil currentReplicas * (currentMetricValue/desiredMetricValue) \rceil$$

The current metric value is the average of all running pods. Only pods in the ready state are used for the computation.

2.6.2 Cluster Autoscaler (CA)

The CA's task is to adjust the size of the cluster in order to meet the current resource requirements. The assumption is that the cluster runs on a set of node groups. Within a node group, all nodes have identical hardware specifications.
In case a new pod cannot be scheduled because it for example requests more CPU or RAM than is available on any of the current cluster nodes, the scale-up routine is triggered. The routine checks if adding a node to any of the node groups allows the previously unschedulable pod to be scheduled. If one node group suits the needs a node is added to that group. If multiple node groups meet the requirements one is selected by instance price or more sophisticated criteria.
In case a current cluster node's CPU and RAM is utilized below a certain threshold the scale-down routine is triggered. The CA scans the cluster for underutilized nodes every 10 seconds (this interval can be changed). A node is considered underutilized if the sum of CPU and RAM requests of all pods running on the node is less than a certain percentage of the node's available resources. The default utilization threshold is 50%, but it can be adjusted. For every underutilized node, the CA checks whether all pods running on the node can be rescheduled on other nodes. The check uses the first fit decreasing bin-packing approximation algorithm to check whether the pods could be scheduled on other nodes. Pods, that do not need to be rescheduled in case the node is removed, are excluded from the check. The CA also checks whether the node has a scale-down disabled annotation which prevents the node from being removed. If the node's utilization stays below the threshold for 10 minutes (this time can be configured) and all its pods can be rescheduled the node is removed from the cluster. The procedure described is the default, but there are quite some cases that stop the removal of a node. The most relevant include pods that are part of the control plane of Kubernetes and do not have a pod disruption budget set or generally pods that have a pod disruption budget that currently does not allow the

pod to be terminated. There might also be other factors like pods requiring node labels that currently cannot be satisfied by any other node.

More information on Kubernetes can be found in the official documentation [5].

Chapter 3

Shortcomings of the default scale-down procedure

The following sections describe the shortcomings of the default CA's scale-down procedure and their implications for resource utilization in Kubernetes clusters.

Due to the fact that only nodes which are utilized below a specified threshold are considered for removal, the default scale-down procedure potentially misses many opportunities for scaling down the cluster. This fact can be illustrated using Figure 3.1. In the example, each node has 4 CPU cores and 8 gigabytes of RAM. Node 4 has a "green" label and pod F has a required node label affinity for the "green" label. This fact is indicated by the green color. The default scale-down procedure only considers node 4 to be removed as all other nodes have a resource utilized more than 50%. When trying to schedule pod F on a different node the procedure ends as no other node has the "green" label. If the utilization threshold is increased to 80%, then node 1 and node 2 are considered for removal. Node 1 can in fact be removed as pod A can be rescheduled on node 4. The same holds for node 2 as pod B can be rescheduled on node 3 and pod C can be rescheduled on node 4. There is no metric that determines which of the nodes should be removed first, but assuming that the pods are rescheduled according to the default first fit decreasing bin packing algorithm, the resulting cluster state would be like pictured in 3.2. When optimizing for resource utilization cluster state 2 seems optimal as the cluster size is reduced as much as possible. From

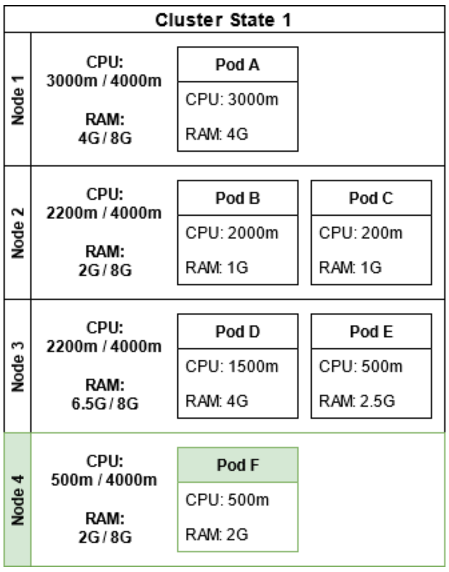

Figure 3.1: Cluster State 1

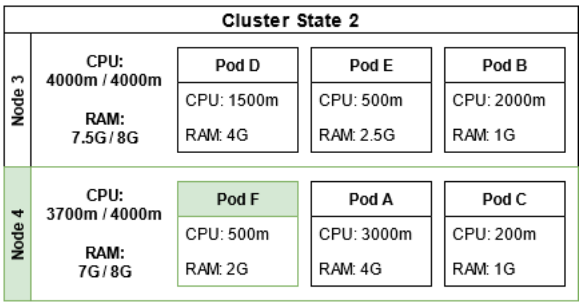

Figure 3.2: Cluster State 2

an operational perspective, the cluster state might not be desirable as it leaves no room for scaling any of the pods up horizontally except for pod C. In the case that any other pod would need to be replicated to deal with a peak load, a new node would have to be added to the cluster. Adding a node to the cluster can take multiple minutes, so the advantage of quick horizontal scaling would be lost. Hence picking a low utilization threshold potentially misses out on opportunities for scaling down, while picking a higher utilization threshold might remove too many nodes.

Chapter 4

An alternative down-scaling procedure

The following sections present a conceptually different down-scaling procedure that allows the CA to detect more opportunities for scaling the cluster down while keeping enough free resources to allow for quick horizontal deployment scaling.

4.1 The concept

The new down-scaler mainly relies on a total cluster utilization threshold. This threshold is defined in terms of both a CPU utilization and a RAM utilization percentage. The proposed down-scaler aims to remove the most expensive node possible from the cluster while guaranteeing that no resource is utilized above the set threshold. To give even stronger guarantees the notion of usable free resources is introduced to ensure that any resource unit considered free by the utilization percentage is actually available to be used by a pod. Examples and detailed explanations are given in the following sections.

4.2 Utility functions

This section describes some simple utility functions needed for the down-scaler.

4.2.1 Cluster capacity

The function *capacity* is used to obtain the resource capacity of the overall cluster. This capacity is just the sum of the individual capacities of all nodes in the cluster.

```
1: function CAPACITY($Cluster$)
Require: $Cluster = \{Node\}$
Ensure: Returned tuple has the sum of CPU and RAM capacity of all
    nodes in the cluster
2:     $c_{CPU} \leftarrow 0$
3:     $c_{RAM} \leftarrow 0$
4:     for all $Node(Id, Capacity(cpu, ram), Pods) \in Cluster$ do
5:         $c_{CPU} \leftarrow c_{CPU} + cpu$
6:         $c_{RAM} \leftarrow c_{RAM} + ram$
7:     end for
8:     return $(c_{CPU}, c_{RAM})$
9: end function
```

4.2.2 Node and cluster requests

The function *requests* is used to obtain the sum of resources requested by pods running on a particular node or on the whole cluster. The variant which takes a cluster as the argument gets the sum of requests for each node in the cluster and sums those to obtain the value for the whole cluster.

```
1: function REQUESTS($Node$)
Require: $Node = (Id, Capacity, Pods)$
Ensure: Returned tuple has the sum of CPU and RAM requests of all pods
    running on the node
2:     $r_{CPU} \leftarrow 0$
3:     $r_{RAM} \leftarrow 0$
4:     for all $Pod(Id, Requests(cpu, ram), Conditions) \in Pods$ do
5:         $r_{CPU} \leftarrow r_{CPU} + cpu$
6:         $r_{RAM} \leftarrow r_{RAM} + ram$
7:     end for
8:     return $(r_{CPU}, r_{RAM})$
9: end function

10: function REQUESTS($Cluster$)
Require: $Cluster = \{Node\}$
Ensure: Returned tuple has the sum of CPU and RAM requests of all pods
    running on the cluster.
11:     $r_{CPU} \leftarrow 0$
12:     $r_{RAM} \leftarrow 0$
13:     for all $Node \in Cluster$ do
14:         $(cpu, ram) \leftarrow requests(Node)$
15:         $r_{CPU} \leftarrow r_{CPU} + cpu$
16:         $r_{RAM} \leftarrow r_{RAM} + ram$
17:     end for
18:     return $(r_{CPU}, r_{RAM})$
19: end function
```

4.2.3 Daemon set requests

The function *daemon_set_requests* returns the sum of requests made by pods running on the passed node, which are part of a particular type of deployment. This type is the daemon set. It defines a pod and a type of nodes to schedule one instance of the pod on each node that has the specified type. Hence such pods are bound to the node they run on and they do not need to be rescheduled in case the node is removed from the cluster.

```
function DAEMON_SET_REQUESTS($Node$)
Require: $Node = (Id, Capacity, Pods)$
Ensure: Returned tuple has the sum of CPU and RAM requests of all pods
    which are part of a daemon set running on the node.
    $r_{CPU} \leftarrow 0$
    $r_{RAM} \leftarrow 0$
    for all $Pod(Id, Requests(cpu, ram), Conditions) \in Pods$ do
        if $part\_of\_daemon\_set(Pod)$ then
            $r_{CPU} \leftarrow r_{CPU} + cpu$
            $r_{RAM} \leftarrow r_{RAM} + ram$
        end if
    end for
    return $(r_{CPU}, r_{RAM})$
end function
```

4.3 Usable resources and usable capacity

The notion of usable units of resources is introduced to have a metric that represents the possibility of scheduling more pods on the cluster better than the simple sum of free resources in the cluster. The example shown in figure 4.1 demonstrates that well. The sum of free resources would be CPU: $2600m$ and RAM: $6.5G$. This sum gives the impression that for example another instance of pod B could be scheduled to run on the cluster without a problem. Looking at the actual nodes it becomes clear quickly that this is not the case. Not even a single pod can be added to node 2, as there is no RAM available to any additional pod. Node 1 has some CPU units available, but no typical pod will be able to use $6.5G$ of RAM while only having $200m$ of CPU available. To handle the two examples of free, but unusable resources two kinds of usability thresholds are introduced. "Usability thresholds" is abbreviated with UT in the pseudo-code due to space constraints.

The first one is the minimal amount of CPU and RAM that needs to

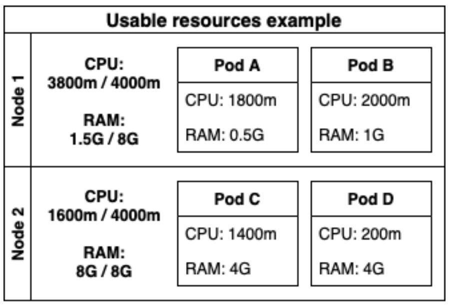

Figure 4.1: Usable Resources Example

be free on a node to allow any pod to be scheduled on the node. If there typically is no pod that uses less than $0.1G$ of RAM for example, then a good threshold could be $0.09G$ for RAM, as any node that has less RAM than this threshold defines cannot run any additional pods. If a node has less CPU or less RAM than defined, all of its free resources are considered unusable. The resources that are already in use are considered usable.

The second type of usability threshold is defined in terms of resource to resource ratios. In the example, pod A has the maximal CPU to RAM request ratio. It requests 3.6 times more CPU than RAM. Hence 3.6 would be a good maximal CPU to RAM ratio in this example. The pod with the maximal RAM to CPU request ratio is pod D with a ratio of 20.

To conclude suitable usability thresholds for the example given in figure 4.1 could be:

- $min_{CPU} = 100m$
- $min_{RAM} = 0.9G$
- $max_{CPU2RAM} = 3.6$
- $max_{RAM2CPU} = 20$

To get the free usable RAM for node 1 the minimum of $free_{RAM}$ and $free_{CPU} * max_{RAM2CPU}$ is taken. In this case, the RAM to CPU ratio would lower the usable free RAM to $4G$ even though there are $6.5G$ free on the node. The function *usable_capacity*, which takes a node as a parameter, performs the calculations described above for individual nodes based on provided usability thresholds. The function, which takes a cluster as an argument, gets and sums the usable capacity of all nodes in the cluster.

```
1: function USABLE_CAPACITY($Node, UsabilityThresholds$)
Require: $UsabilityThresholds = (min_{CPU}, min_{RAM}, max_{CPU2RAM}, max_{RAM2CPU})$
Ensure: Returned tuple has the sum of usable CPU and RAM units of the
    node.
2:     $(r_{CPU}, r_{RAM} \leftarrow requests(Node)$
3:     $(free_{CPU}, free_{RAM}) \leftarrow capacity(Node) - requests(Node)$
4:     if $free_{CPU} < min_{CPU}$ or $free_{RAM} < min_{RAM}$ then
5:         return $(0, 0)$
6:     end if
7:     $usable_{CPU} \leftarrow min(free_{CPU}, free_{RAM} * max_{CPU2RAM}) + r_{CPU})$
8:     $usable_{RAM} \leftarrow min(free_{RAM}, free_{CPU} * max_{RAM2CPU}) + r_{RAM})$
9:     return $(usable_{CPU}, usable_{RAM}$
10: end function

11: function USABLE_CAPACITY($Cluster, UT$)
Require: $Cluster = \{Node\}$
Ensure: Returned tuple has the sum of usable CPU and RAM units of the
    cluster
12:     $(u_{CPU}, u_{RAM}) \leftarrow (0, 0)$
13:     for all $Node \in Cluster$ do
14:         $(u_{CPU}, u_{RAM}) \leftarrow (u_{CPU}, u_{RAM}) + usable\_capacity(Node, UT)$
15:     end for
16:     return $(u_{CPU}, u_{RAM})$
17: end function
```

4.4 Candidate check

To determine which nodes are possibly suitable for removal from the cluster, a simple check is performed for each node. This check determines whether removing the node's resource capacity from the cluster, while preserving the requests of the pods running on the node, leads to a cluster state in which any of the utilization thresholds is violated. The requests made by pods that run on the node and are part of a daemon set are subtracted from the current sum of requests in the cluster. If the check fails for a particular node it is guaranteed that the node cannot be removed from the cluster. On the other hand that does not imply that the node can be removed from the cluster. The function *candidate_check* performs this check for a given node.

```
function CANDIDATE_CHECK($Cluster, Node, Thresholds$)
Require: $Thresholds = (t_{CPU}, t_{RAM})$
Ensure: Returns false if removing the node from the cluster is guaranteed
    to violate the utilization thresholds
    $(r_{CPU}, r_{RAM}) \leftarrow requests(Cluster) - daemon\_set\_requests(Node)$
    $(c_{CPU}, c_{RAM}) \leftarrow capacity(Cluster) - capacity(Node)$
    return $r_{CPU}/c_{CPU} < t_{CPU}$ and $r_{RAM}/c_{RAM} < t_{RAM}$
end function
```

All nodes that pass this first check are added to the candidate list together with the current time. All nodes that do not pass the check are removed from the candidate list. This task is performed by the function *update_candidates*.

```
function UPDATE_CANDIDATES($Cluster, Candidates, Thresholds$)
Require: $\forall(node, time) \in Candidates, node \in Cluster$
Ensure: $Candidates$ contains only nodes which pass the candidate check
    for all $Node \in Cluster$ do
        if $candidate\_check(Cluster, Node, Thresholds)$ then
            if $(Node, *) \notin Candidates$ then
                $Candidates \leftarrow Candidates + (Node, current\_time)$
            end if
        else
            $Candidates \leftarrow Candidates - (Node, *)$
        end if
    end for
end function
```

4.5 Evaluating candidates

In this step, not all candidates that are on the candidate list are evaluated. To prevent momentary fluctuations of resource usage from triggering a scale-down of the cluster, a delay value is used. Only if a candidate is on the list for a longer time than the delay specifies, it is processed further. Now for all applicable candidates, an attempt is made to reschedule all pods, that are running on the node that is evaluated, on other nodes of the cluster. Pods that are part of a daemon set are not rescheduled. The resulting cluster states that are invalid, which means they contain pods that could not be rescheduled, are not processed any further. For all the valid resulting states the utilization ratio of usable resources is computed. This value is defined by dividing the sum of all requests in the cluster state by its usable capacity. Finally, all nodes that can be removed from the cluster without producing a cluster state that has a higher usable resource utilization ratio than specified by the thresholds are considered final candidates. At this point, it is guaranteed that any one of the final candidate nodes can be removed from the cluster.

```
1: function PROCESS_CANDIDATES($Cluster, Candidates, Thresholds, Delay, UT$)
Require: $Cluster = \{Node\}$, $Thresholds = (t_{CPU}, t_{RAM})$
Ensure: Any one of the nodes in the returned list of nodes can be removed
    from the cluster without violating any thresholds
2:     $ResultStates \leftarrow \{\}$
3:     for all $(Node, time) \in Candidates$ do
4:         if $current\_time - time > Delay$ then
5:             $ResultStates \leftarrow ResultStates + (Node, bin\_pack(Cluster, Node))$
6:         end if
7:     end for
8:     $final\_candid \leftarrow \{\}$
9:     for all $(Node, ClusterState) \in ResultStates$ do
10:        if $valid(ClusterState)$ then
11:            $(c_{CPU}, c_{RAM}) \leftarrow usable\_capacity(ClusterState, UT)$
12:            $(r_{CPU}, r_{RAM}) \leftarrow requests(ClusterState)$
13:            if $r_{CPU}/c_{CPU} < t_{CPU}$ and $r_{RAM}/c_{RAM} < t_{RAM}$ then
14:                $final\_candid \leftarrow final\_candid + Node$
15:            end if
16:        end if
17:    end for
18:    return $final\_candid$
19: end function
```

4.6 The complete procedure

The complete procedure first updates the candidate list (section 4.4) to then process the candidates that continuously are candidates longer than the delay specifies (section 4.5). If there are final candidate nodes, the most expensive is chosen to be removed from the cluster. The notion of cost is subject to the concrete implementation of the procedure. It could for example express the monetary cost or the environmental (CO^2) cost of running the node. After the removal of the chosen node is completed, the procedure is repeated.

```
1: function DOWN_SCALING_ALGORITHM($Cluster, Thresholds, Delay, UT$)
Require:
Ensure: Whenever nodes can be removed from the cluster without violating
    the thresholds, the most expensive node possible is removed.
2:     $Candidates \leftarrow \{\}$
3:     while $True$ do
4:         $update\_candidates(Cluster, Candidates, Thresholds)$
5:         $FinalCandidates \leftarrow process\_candidates(Cluster, Candidates, Thresholds, Delay, UT)$
6:         if $FinalCandidates \neq \{\}$ then
7:             $NodeToBeRemoved \leftarrow most\_expensive(FinalCandidates)$
8:             $remove\_node(NodeToBeRemoved)$
9:         end if
10:    end while
11: end function
```

4.7 Functions to be implemented

4.7.1 Bin-packing

The bin-packing algorithm can be implemented in different ways. The multi constraint bin-packing problem is a subject treated in scientific literature. It is NP-hard, but there are heuristics to find a good, but not the optimal solution quicker [1]. For this special application, there are not only the CPU and RAM dimensions but also additional constraints like pod disruption budgets and node labels which need to be respected. The *bin_pack* function is assumed to be present and to return the cluster state resulting from removing the given node from the given cluster. All pods that need to be rescheduled when the node is removed are placed on the remaining nodes. If pods cannot be rescheduled that is noted and the resulting cluster state is considered invalid. The algorithm can be implemented similar to the existing bin-packing implementation of the default cluster auto-scaler [2].

4.7.2 Removing Node from Cluster

The *remove* function makes the appropriate Kubernetes API calls do drain the node, which means removing all pods from the node. After that, the node is removed from the cluster by an additional API call. The same functionality can be found in the default cluster auto-scaler implementation.

4.8 The order of checks

The order of checks is chosen to minimize computational cost while giving as many guarantees about cluster states as possible. This section refers to the pseudo-code of the *down_scaling_algorithm*. The first check that is performed by the function call in line 4 has linear complexity. It serves as a filter to discover nodes that cannot be removed from the cluster because their removal leaves the cluster with too few resources no matter what bin-packing result is obtained. For each node found here with a linear check the following steps do not need to be conducted. Bin-packing is performed next within the function called in line 5. It is the computationally most expensive step. The complexity of the algorithm depends on the actual implementation. The high cost is accepted because it is necessary to find possible scale-down opportunities. Lastly, still within the function called in line 5, a linear check is performed to give additional guarantees about the resulting cluster states.

Chapter 5

Discussion

5.1 Differences to the default implementation

5.1.1 Thresholds

The first key difference between the default down-scaler and the proposed procedure is the resource utilization thresholds. The default implementation works with a threshold that determines the single node resource utilization ratio. If a node's resource utilization is below the threshold, it is considered for removal. The proposed procedure works with a threshold that determines the maximal desired cluster resource utilization ratio. While the default implementation uses the threshold to guard the entry of the procedure for individual nodes, the proposed procedure uses the threshold to give guarantees about the cluster state after a node is removed.

5.1.2 General approach

The default down-scaler performs bin-packing attempts similar to the proposed down-scaler if nodes are utilized below the threshold for the amount of time specified by the delay. If this attempt is successful for a particular node, the node is instantly treated as a final removal candidate. If there are multiple, only one is chosen, but one final candidate is instantly removed from the cluster without performing additional checks. The proposed down-scaler works towards not exceeding the cluster utilization threshold while decreasing the cluster size as much as possible. All nodes are considered for removal, but if an easy check can determine, that their removal will violate the threshold, no bin-packing attempt is made. For all other nodes, the attempt is made and the resulting cluster state is evaluated to find out whether it violates the threshold. Only nodes that can be removed without violating any threshold are considered final removal candidates.

5.2 Advantages of the proposed down-scaler

The main advantage of the proposed down-scaler is the fact that the threshold value has a direct influence on the cluster state after removing a node. Removing underutilized nodes, like the default implementation, seems reasonable from a naive view as they do not seem to be needed. However, there are no guarantees that there are enough free resources in the cluster to scale a deployment horizontally to cope with a peak load after a node is removed. As quick and automatic deployment scaling is one of the key advantages of Kubernetes, setting an aggressive down-scaling threshold is dangerous in the default down-scaler. There is guesswork involved to determine which down-scaling threshold leads to which resulting cluster states as shown in chapter 3. To ensure production without unpredictable performance fluctuations, very conservative down-scaling thresholds need to be chosen. This results in many missed down-scaling opportunities. The proposed down-scaler allows the definition of maximal resource utilization that is allowed to result from the removal of a node. That inherently also defines a minimal ratio of unused resources. The desired ratio of free resources needed can be chosen directly and is not required to include a big safety margin. All possible opportunities for scaling down the cluster can be found and executed.

The proposed down-scaler is also conceptually closer to the general approach of abstracting away from the single node used by Kubernetes. Pod deployments, networks, and storage for example are all defined on a cluster and not on a single node level.

5.3 Advantages of the default down-scaler

The default down-scaler takes a conceptually more simple approach to determine which node can be removed from the cluster. The simplicity has the advantage that the basic concept can be understood more easily and thereby lowers the barrier of meaningful use. Another positive aspect is that using the default utilization threshold of 50%, the down-scaler does not remove highly utilized nodes and thereby triggers fewer rescheduling events.

5.4 Recreating the proposed behavior

5.4.1 Overprovisioning using the default CA

The official documentation of Kubernetes [3] tackles the shortcomings of the Cluster Autoscaler 3 using a mechanism called overprovisioning. Pause pods with a low priority are used as dummy pods to make requests for resources. If a real workload needs to be scheduled, but not enough resources are available, a low priority dummy pod is evicted and the workload takes their resources. Now the pause pod needs to be rescheduled, but not enough resources are available. A cluster scale-up is triggered. In case of a cluster scale-down, the dummy pods guarantee that some free resources remain in the cluster as all the dummy pods, running on a node to be removed, need to be rescheduled.

The number of dummy pods and their resource requests can be defined using absolute values. It is also possible to manage these values using the so-called Horizontal Cluster Proportional Autoscaler pod [4], which allows the values to be adjusted relative to the cluster size and the number of CPU cores available in the cluster. The use of an additional pod for automatic scaling and the dummy pod deployment itself make overprovisioning a complex task.

5.4.2 Recreating proposed behavior

The behavior of the proposed down-scaling procedure can be recreated using the default down-scaling implementation and dummy pods. The down-scaling threshold can be set to 100% so every node is considered for removal in each iteration of the down-scaler. Dummy pods are dynamically scaled to request a certain percentage of the available resources. Removal of a particular node can then only happen if all workload pods and all dummy pods running on it can be rescheduled on other nodes. The dummy pods need to be rescheduled in their current configuration even though their amount and value of requests might change after the cluster is scaled down. In a cluster that only contains one node type, this problem can be resolved by adjusting the parameters of the dummy pod deployment to reflect the required free resources for the cluster state after a single node is removed. If the cluster contains nodes of different types the mentioned solution does not work as the cluster resource capacity after the removal of a node depends on the type of the node that is removed. This makes clear, that this approach is not a replacement for the alternative down-scaling procedure proposed in this thesis.

5.4.3 Use in practice

Mentions of the overprovisioning method in publications [10] [7] indicate that the method is used in practice to tackle the shortcoming of the default down-scaler. In real-world applications, the method also has distinct advantages including the possibility to make requests for custom resources like IP addresses [10].

Chapter 6

Conclusions and Future Work

This thesis presents an alternative down-scaling procedure that can find more opportunities for scaling Kubernetes clusters down than the default implementation. While the default implementation is not able to give guarantees about the cluster state after issuing a scale-down operation, the proposed procedure ensures that user-defined resource utilization ratios are not exceeded. It even ensures that only resource units, which can be used to run pods, are considered as free resources for computing the utilization ratio. The workaround to overcome the shortcomings of the default implementation, which is presented in the official Kubernetes documentation, implies that there is awareness for the problems in the community. While the presented workaround has real-world advantages it does not resolve the shortcomings of the default implementation in all cluster scenarios. The simplicity of specifying thresholds and a delay value used by the proposed down-scaler makes the procedure accessible. The workaround, on the other hand, needs an additional complex deployment which results in a higher barrier of entry.

The following steps describe possible future work that builds on the ideas presented in this thesis. The notion of usable resources can be applied to an up-scaling procedure as well to form a complete cluster auto-scaler. A working implementation needs to be created for real-world testing. The theoretical advantages of the presented down-scaler need to be verified. This can be done by comparing real-world test results to the default implementation and related work like the previously mentioned QoS-based auto-scaler [13]. While CPU and RAM are the basic resource requests of concern in a Kubernetes cluster, the proposed cluster auto-scaler can be extended to also be aware of other types of resources like IP addresses or storage bandwidth. Different approaches like machine learning, which is already applied to horizontal pod auto-scaling [12], can be applied to cluster scaling as well. The resulting implementations can be tested and compared to find the optimal solution for the problem of cluster auto-scaling in Kubernetes.

Bibliography

[1] Roberto Aringhieri, Davide Duma, Andrea Grosso, and Pierre Hosteins. Simple but effective heuristics for the 2-constraint bin packing problem. *Journal of Heuristics*, 24:1–13, 06 2018.

[2] Project contributors. binpacking_estimator.go. `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/estimator/binpacking_estimator.go`. Accessed: 2022-03-27.

[3] Project contributors. Cluster autoscaler - frequently asked questions. `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md`. Accessed: 2022-03-21.

[4] Project contributors. Horizontal cluster-proportional-autoscaler container. `https://github.com/kubernetes-sigs/cluster-proportional-autoscaler/blob/master/README.md`. Accessed: 2022-03-21.

[5] Project contributors. Kubernetes concepts. `https://kubernetes.io/docs/concepts`. Accessed: 2022-03-21.

[6] Sandeep Dinesh. Kubernetes best practices: Resource requests and limits. `https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits`. Published: 2018-05-11, Accessed: 2022-03-21.

[7] Hasham Haider. Kubernetes autoscaling in production: Best practices for cluster autoscaler, hpa and vpa. `https://www.replex.io/blog/kubernetes-in-production-best-practices-for-cluster-autoscaler-hpa-and-vpa`. Published: 2019-12-05, Accessed: 2022-03-21.

[8] Craig Mcluckie. Containers, vms, kubernetes and vmware. `https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html`. Published: 2014-08-25, Accessed: 2022-03-21.

[9] Sam Newman. *Building microservices.* OReilly Media, 2015.

[10] Michael Seiwald. Cluster overprovisioning in kubernetes. `https://medium.com/scout24-engineering/cluster-overprovisiong-in-kubernetes-79433cb3ed0e`. Published: 2019-07-30, Accessed: 2022-03-21.

[11] SlashData. The state of cloud native development. `https://www.cncf.io/wp-content/uploads/2021/12/Q1-2021-State-of-Cloud-Native-development-FINAL.pdf`. Published: 2021-12, Accessed: 2022-03-21.

[12] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.

[13] Qiang Wu, Jiadi Yu, Li Lu, Shiyou Qian, and Guangtao Xue. Dynamically adjusting scale of a kubernetes cluster under qos guarantee. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 193–200, 2019.