Technology Guide

Telemetry Aware Scheduler - Enhancement Utilization with Platform Aware Scheduling

Authors

Madalina Lazar

Denisio Togashi

Marlow Weston

Ukri Niemimuukko

1 Introduction

In today's cloud technology, Kubernetes (K8s) is the gold standard for containers orchestration in all kinds of environments and areas that need workload management and deployment. K8s includes Network Function Virtualization (NFV), Internet of Things (IoT), 5G, and Artificial Intelligence (AI). These diverse areas have different optimizations, and K8s must be managed to handle all of them. One of the more challenging areas of optimization is resource management, especially with the increasing numbers of machines with the increasing cost of energy resources. Valuable solutions will both optimize the efficiency and functionality of workloads while also allowing for a reduction in both cost and environmental impact.

Platform Aware Scheduling (PAS) was developed at Intel to address these concerns. It is comprised of a group of related projects that expose platform specific attributes to the K8s scheduler and make this data available for scheduling and descheduling decisions in Kubernetes. Current extenders include Telemetry Aware Scheduling (TAS) and GPU Aware Scheduling (GAS). TAS helps Kubernetes and the community to improve resource optimization based on the telemetry data generated from the workload(s) resources. It is based on written policies, which provide full automation of workloads deployment and management in a closed loop. GAS allows the usage of GPU resources, such as memory amount, for scheduling decisions in Kubernetes). This allows users to handle fractional GPU workload requests in multi-card GPU nodes.

This guide is a refresher on the previous article on Telemetry Aware Scheduling; it presents the latest additions to PAS (TAS & GAS) and an example of TAS implementation that uses customer or platform generated telemetry data to assist the K8s scheduler in determining the best placement of a workload in the cluster. Not only does this paper appeal to a more general audience, it will also address K8s cluster operators, developers, and architects.

This document is part of the Network Transformation Experience Kits.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	September 2021	Initial release.
002	October 2022	New additions to features and improvements since the previous revision. Updated the installation steps to account for newer versions of Kubernetes and Kubernetes Descheduler.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
GAS	GPU Aware Scheduling
GB	Gigabits
GPU	Graphics Processing Unit
Intel® RDT	Intel® Resource Director Technology
IoT	Internet of Things
JSON	JavaScript Object Notation
K8s	Kubernetes
LLC	Last Level Cache
NFV	Network Function Virtualization
OPNFV	Open Platform for NFV
PAS	Platform Aware Scheduling
PMU	Performance Monitoring Unit
RAM	Random Access Memory
RAS	Reliability, Availability, and Serviceability
TAS	Telemetry Aware Scheduling
TLS	Transport Layer Security
VPU	Vision Processing Units

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
Barometer Home	https://wiki.opnfv.org/display/fastpath/Barometer+Home
collectd	https://github.com/collectd/collectd
Closed Loop Automation - Telemetry Aware Scheduler for Service Healing and Platform Resilience Demo	https://networkbuilders.intel.com/closed-loop-automation-telemetry-aware-scheduler-for-service-healing-and-platform-resilience-demo
Closed Loop Automation - Telemetry Aware Scheduler for Service Healing and Platform Resilience White Paper	https://builders.intel.com/docs/networkbuilders/closed-loop-platform-automation-service-healing-and-platform-resilience.pdf
Energy Savings & Resiliency with Closed Loop Platform Automation	https://tma.roc.cnam.fr/Proceedings/TMA_Demo_5.pdf
GPU Aware Scheduling	https://github.com/intel/platform-aware-scheduling/tree/master/gpu-aware-scheduling
Intel RDT	https://wiki.opnfv.org/display/fastpath/Intel_RDT
Kubernetes Descheduler User Guide	https://github.com/kubernetes-sigs/descheduler/blob/master/docs/user-guide.md
Kubernetes Prometheus Adapter	https://github.com/DirectXMan12/k8s-prometheus-adapter
Node Exporter	https://github.com/prometheus/node_exporter
One Click Install of Barometer Containers	https://wiki.opnfv.org/display/fastpath/One+Click+Install+of+Barometer+Containers
Prometheus	https://github.com/prometheus/prometheus
Platform Aware Scheduling	https://github.com/intel/platform-aware-scheduling
Telemetry Aware Scheduling	https://github.com/intel/platform-aware-scheduling/tree/master/telemetry-aware-scheduling

Reference	Source
Telemetry Aware Scheduling – Automated Workload Optimization with Kubernetes (K8s) Training Video	https://networkbuilders.intel.com/telemetry-aware-scheduling
Telemetry Aware Scheduling (TAS) - Automated Workload Optimization with Kubernetes (K8s) Technology Guide	https://builders.intel.com/docs/networkbuilders/telemetry-aware-scheduling-automated-workload-optimization-with-kubernetes-k8s-technology-guide.pdf

2 Overview

Telemetry Aware Scheduling extends the Kubernetes Scheduler and the main engine to drive the scheduler's decisions for a workload deployment. Based on this concept, TAS via the Platform Aware Scheduling can deliver solutions that the default Kubernetes Scheduler cannot solve, such as noisy neighbor protection where co-located workloads can compete for specialized resources (for example, Last Level Cache); Visual Processing at the edge (VPU), like the Intel® Movidius™ Myriad™ X VPU applied in edge AI image processing workloads. TAS can help to ensure that any new VPU workload is scheduled on a node where there is more VPU capacity available to enable high performance in use cases like traffic monitoring, video security systems, and facial recognition. TAS complements the platform resilience use case by delivering up-to-date node-health based placement suggestions and by indicating possible faulty nodes to the default K8s scheduler. For more information on using TAS with platform resilience, refer to the Closed Loop Automation - Telemetry Aware Scheduler for Service Healing and Platform Resilience White Paper.

Platform Aware Scheduling is also home to GPU Aware Scheduling (GAS); GAS allows the usage of GPU resources for scheduling decisions in Kubernetes. It is used to optimize scheduling decisions when POD resource requirements include the use of several or fragments of GPUs on a node, instead of traditionally mapping a GPU to a pod. It is designed to be a Kubernetes Scheduler Extender, and in doing so, will leverage Telemetry Aware Scheduling, through platform metrics and TAS policies, to get the best decision for the workload deployment.

In general, TAS under the Platform Aware Scheduling, can find solutions for many different areas like automation, service assurance, and resource management.

2.1 Challenges Addressed

In Kubernetes, scheduling occurs where a pod is assigned to a node. The component responsible for this is Kube-Scheduler (or simply, scheduler). When a pod with the workload is found by the scheduler, it will try to schedule it on the best available node(s) that can provide the minimum computer resource types (CPU and memory) that are requested to run the workload. This is the general approach that is based on the static state of those resources rather than the current state or any previous level of utilization of those resources.

The lack of knowledge of the current state of the resources, such as CPU and memory, can cause the scheduler to take less than optimal decisions and affect workload performance. A critical example of a poor placement would be the scheduling of a workload to a node with a high CPU usage instead of a node with a low CPU usage.

Other resources such as GPUs, FPGAs, and network cards are seen as mere atomic devices that is, either in usage or not. The scheduler does not have a direct way to access their utilization at a granular level. For example, workloads that need intensive usage of GPUs will be scheduled to nodes that have GPUs. However, the default scheduler does not prevent the scheduling of workloads in a situation that can only be achieved by fractioning over several GPUs in the node. More details are available in the GAS repository.

The inability of Kubernetes to consider the current usage level of the available resources can mislead the scheduler to take non-optimum scheduling decisions and cause an imbalanced distribution and unexpected resource usage from the workload across the cluster. TAS is the mechanism that assists K8s in such cases and involves the platform telemetry that becomes crucial in helping the scheduler not only to know the current resource level usage but also to consider the node history usage and predict the best node for the workload or avoid scheduling on predicted faulty devices.

2.2 Technology Description

On the deployment of a workload, that is, placement of workload's pod on a feasible node in a cluster, K8s default scheduler executes a set of operations such as Filtering and Scoring. Filtering determines the feasible nodes and Scoring assigns scores between the feasible nodes. The node that is ranked with the highest score is then selected to receive the pod deployment. The filtering and scoring behavior of the default scheduler can be configured by a scheduling policy configuration file.

Telemetry Aware Scheduling is then enabled as an extender for K8s default Schedule by providing a scheduling policy to the default Kubernetes scheduler. An example of a policy looks like:

```
apiVersion: kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
```

```
clientConnection:
  kubeconfig: /etc/kubernetes/scheduler.conf
extenders:
  - urlPrefix: "https://tas-service.default.svc.cluster.local:9001"
    prioritizeVerb: "scheduler/prioritize"
    filterVerb: "scheduler/filter"
    weight: 1
    enableHTTPS: true
    managedResources:
      - name: "telemetry/scheduling"
        ignoredByScheduler: true
    ignorable: true
    tlsConfig:
      insecure: false
      certFile: "/host/certs/client.crt"
      keyFile: "/host/certs/client.key"
```

There are a number of options available to customize the "extenders" configuration object. For more detail, see PAS repository. Some of these fields - such as `urlPrefix`, `filterVerb` and `prioritizeVerb` are necessary as they help point the Kubernetes scheduler to the TAS scheduling service URL (in the example above this will be: https://tas-service.default.svc.cluster.local:9001). With a policy like the above as part of the default scheduler configuration, the identified webhook then becomes part of the scheduling process. This allows TAS to modify the decisions made by the default scheduler for workloads that call for the use of TAS.

The Filtering and Prioritization operations in TAS are then aligned with the Filtering and Scoring of the default scheduler; and together with Descheduler - that removes a workload from its host causing it to be placed on a more suitable host - these three compose the main processes configured within TAS policies.

TAS policies, implemented as a K8s Custom Resource Definition, are then built on those processes Filtering, Prioritization, and Descheduling by defining respectively the scheduling Strategies: Scheduleonmetrics, Dontschedule, Deschedule, and Labeling.

Each one of these strategies is governed by rules that relate the platform metrics data values to a desired state defined by the user or operator.

The "Labeling" strategy is the latest addition to the PAS/TAS's available scheduling strategies. It is a multi-rule strategy meant for creating and attaching labels to nodes based on rule violations. Labels will be written to the namespace *telemetry.aware.scheduling.policyname*, where policyname will be replaced by the name of the policy. The labels can then be used with external components like the K8s Descheduler.

An example policy, with strategies and rules as below, is available on the open-source repository.

```
apiVersion: telemetry.intel.com/v1alpha1
kind: TASPolicy
metadata:
  name: demo-policy
  namespace: default
spec:
  strategies:
    scheduleonmetric:
      rules:
      - metricname: node_metric_1
        operator: LessThan
    dontschedule:
        logicalOperator: anyOf
        rules:
        - metricname: node_metric_1
           operator: GreaterThan
           target: 80
        - metricname: node_metric_2
           operator: LessThan
           target: 200
    deschedule:
```

```
      logicalOperator: allOf
      rules:
      - metricname: node_metric_1
         operator: GreaterThan
         target: 80
      - metricname: node_metric_2
         operator: LessThan
         target: 200
   labeling:
      rules:
      -  metricname: node_metric_1
         operator: GreaterThan
         target: 100
         labels: ["foo=1"]
```

The policy named demo-policy contains four separate rulesets defined under the requested strategies. Each one of these rules uses a series of metric names (that is, node_metric_1, node_metric_2) to build the set of recommendations based on the comparison of values defined in the policy field that is, target, and the metrics collected by platform metrics.

In the above "demo-policy" example, the rule in "scheduleonmetrics" strategy advertises the default scheduler to place workloads toward nodes with the lowest reading that is, lowest value of `node_metric_1`. The rules in the second strategy "dontschedule" tells the scheduler to filter out nodes where the `node_metric_1` metric value is greater than 80 or the node_metric_2 metric has a value smaller than 200. The rule in the third strategy "deschedule" tells TAS that the node for which node_metric_1 has a value greater than 80 and there is `node_metric_2` less than 200 is not healthy enough to keep running workloads. Workloads in this node should be removed and rescheduled. Labeling is a multi-rule strategy for creating node labels based on rule violations. In the example above if `node_metric_1` would be greater than 100, the telemetry.aware.scheduling.scheduling-policy/foo=1 node label will be created and attached to the respective node.

Labels should have different names in different rules. Labels are key-value pairs and only unique keys can exist in each label namespace. There can be situations where it is required to have similar label names across rules. If so, when using the *GreaterThan* operator the rule with the maximum metric value will be the only one honored. In case of the *LessThan* operator the rule with the minimum metric value will be the only one honored.

```
   labeling:
      rules:
      -  metricname: node_metric_1
         operator: GreaterThan
         target: 100
         labels: ["foo=1"]
      - metricname: node_metric_2
         operator: GreaterThan
         target: 100
         labels: ["foo=2"]
```

The above rules would create label `telemetry.aware.scheduling.scheduling-policy/foo=1 when node_metric_1` is greater than node_metric_2 and also greater than 100. If instead node_metric_2 would be greater than node_metric_1 and also greater than 100, the produced label would be `telemetry.aware.scheduling.scheduling-policy/foo=2`. If neither metric would be greater than 100, no label would be created. When there are multiple candidates with equal values, the resulting label is random among the equal candidates. Label cleanup happens automatically. An example of the labeling strategy can be found in here.

With the addition of the *anyOf* and *allOf* logical operators, users can now create more complex rules in their TAS policies (that is, the "deschedule" rule above) which in term will allow them to tackle more complex scheduling scenarios. The operators work similar to "OR" and "AND" logical operators found in programming languages: that is, with "*anyOf*" the rule will be triggered if at least one of the rules is broken and with "*allOf*", the rule will be triggered only when all metric rules are broken.

TAS policy system enforces rule-based strategies to control pod distribution across the cluster and influence the scheduling and lifecycle placement process.

Telemetry Aware Scheduling via policy system allows smart scheduling recommendations and also allows automation of actions that would otherwise be done manually by the user/operator. As result, it increases the performance by improving the resource utilization across the whole cluster.

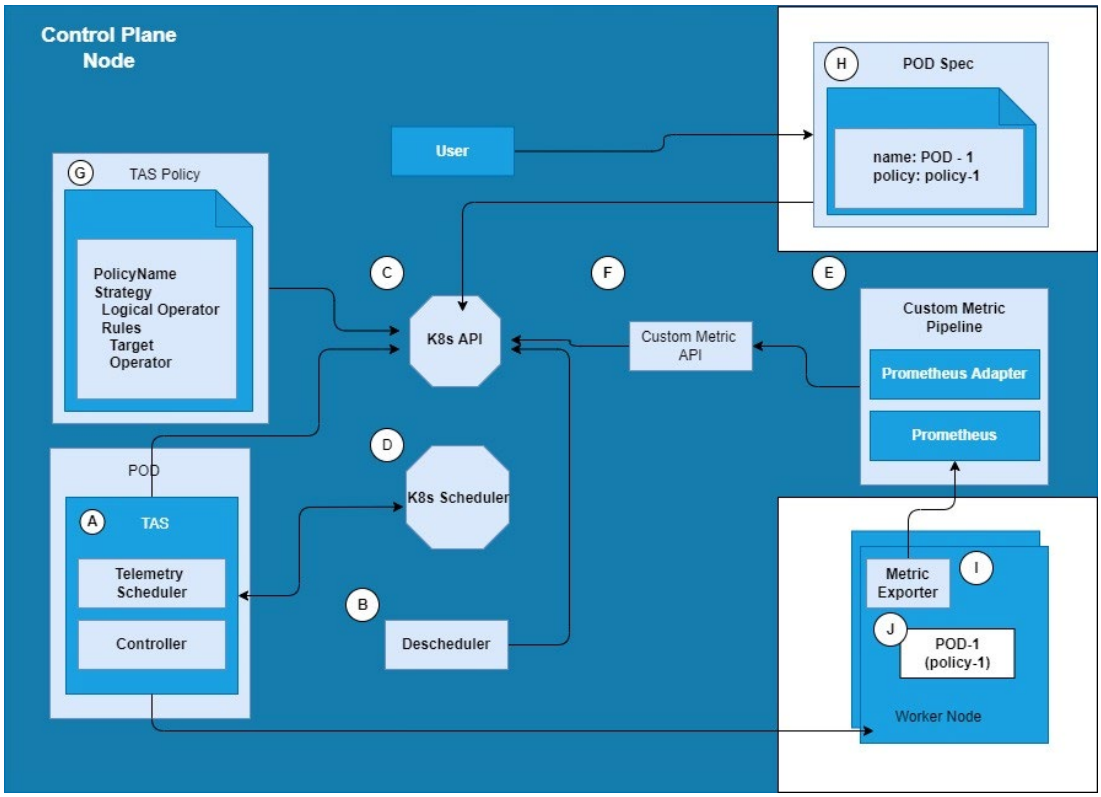

Figure 1. Architecture Diagram

2.3 Architecture

Figure 1 is a diagram of a Kubernetes cluster with a separate control plane node and worker nodes. The key components are:

Telemetry Aware Scheduling: TAS extender is made up of two main elements in the code base deployed as single container:

- *Telemetryscheduler*: Recommends nodes for pod placement based on metrics rules.
- *Controller*: Watches the state of critical metrics and marks violating nodes when deschedule or labeling strategies are specified in the policy.

Kubernetes Descheduler: Picks up the pods due to be evicted, runs safety checks, and deschedules.

Kubernetes API server: Validates and configures data for the K8s API objects.

Scheduler: Performs scheduling operations and looks for advice from TAS.

Custom Metric Pipeline: Gets metrics from the node metric exporter agents and make them available to K8s API via Custom Metric API.

Custom Metric API: A K8s API aggregation layer that receives metrics from the Custom Metric Pipeline and makes them available to TAS.

TAS Policy: A set of strategies and rules made available as a policy custom resource in K8s API.

Pod Spec: Specifically references the policy that should be used to influence the scheduling of this workload.

Node metric exporter agents: "Collectd" and/or "Node Exporter".

POD-1 linked to policy-1 deployed in a feasible node.

The changes made (related to previous guide) to the Platform Aware Scheduler, are:

- The addition of the "labeling" scheduling strategy
- The support for multiple rules inside scheduling strategies via the "anyOf", "allOf" logical operators
- All modules inside the PAS project are now tagged in a similar way as the Docker* container image. This release tags the extender, TAS, and GAS go modules via git tags in the format described in https://go.dev/doc/modules/version-numbers
- The tests and documentation have been updated to illustrate the newly added features
- TAS has been updated to work with K8s 24
- GAS has added support for doing GPU-tile allocations as a new resource type

- GAS has added support for the "gas-same-gpu" Pod annotation – the annotation value is a list of container names. This tells GAS which containers should only be given the same GPU. This allows different containers of a POD to e.g., share the framebuffer of a GPU, as the allocated GPU can be guaranteed to be the same.

3 Deployment

TAS is an open-source community project hosted by the Platform Aware Scheduling available at https://github.com/intel/platform-aware-scheduling/tree/master/telemetry-aware-scheduling Deployment is done directly on K8s, using Helm* charts provided at the same repository.

3.1 Installation of TAS and Dependencies

1. Install programming languages and packages to deploy and build TAS.
   - Go* installation 1.18 or higher
   - K8s v23 or higher
   - K8s installed on a multi-node cluster using Kubeadm, Helm installed on the K8s cluster
2. Install custom metrics pipeline
   - Prometheus - https://github.com/prometheus/prometheus
   - Prometheus Adapter - https://github.com/DirectXMan12/k8s-prometheus-adapter
   - Node exporter - https://github.com/prometheus/node_exporter
   - Collectd - https://github.com/collectd/collectd
3. Install TAS - Build, deploy, and configure TAS
   - Configure K8s scheduler extender - https://github.com/intel/platform-aware-scheduling/tree/v0.9/telemetry-aware-scheduling#extender-configuration
   - Deploy the TAS system - https://github.com/intel/platform-aware-scheduling/tree/v0.9/telemetry-aware-scheduling/deploy
4. Install the Kubernetes Descheduler - https://github.com/kubernetes-sigs/descheduler/tree/v0.23.1

Notes:

- The custom metric pipeline components can be installed by using Helm charts in the quick install at https://github.com/intel/platform-aware-scheduling/tree/v0.9/telemetry-aware-scheduling#custom-metrics-pipeline
- Collectd is an additional complementary node metrics provider, which makes many additional platform metrics available.

3.2 TAS in Operation

Once all the dependency components are installed, configured and up and running, TAS is ready to provide schedule decisions to the default scheduler. Following the diagram in Figure 1, the steps below demonstrate how TAS works during the workload deployment that is associated to the TAS policy. Note that the sequence of events is not necessarily in the presented order since some of the processes can run concurrently.

1. Node metric agents (I) send the monitoring metrics (for example, health_metric) to custom-metrics API (F) via custom-metric pipeline (E) and therefore to K8s API (C).
2. TAS policies (G) deployed in the cluster are then advertised in K8s API (C).
3. TAS in the cluster (A) starts to watch continuously on the K8s API (C) for TAS policies and updates of policy changes.
   - TAS checks the requested strategies and rules from TAS policies in the cluster.
   - Based on the policy specs, TAS pulls the metric values advertised in Custom-metric API through K8s API.
   - If the "deschedule" strategy's rules are requested and applicable, TAS labels the nodes that complied with the "deschedule" strategy's rules as "violating". The "violating" label is advertised on K8s API.
   - If any "labeling" strategy rules are defined in the TAS policy and they are matched, TAS will label the nodes for which the policy is applicable with the requested key – value pairs. For example, looking at the labeling strategy below, if the value "node_metric_1" is greater than 100 then the node in question will get the following labels: *telemetry.aware.scheduling.scheduling-policy/label1=foo* and *telemetry.aware.scheduling.scheduling-policy/label2=bar*

```
labeling:
rules:
metricname: node_metric_1
operator: GreaterThan
target: 100
```

```
labels: ["label1=foo", "label2=bar"]
```

4. Descheduler component (B) in the cluster watches K8s API (C)
   - If it perceives a node with "violating" label, it evicts pods associated to TAS policy on that node.
   - The K8s Schedule (D) with TAS will re-schedule the evicted pods to other available and feasible nodes.
   - TAS works with Descheduler up to and including v0.23.1. There are newer versions available, but there seem to be compatibility issues from v0.24.0 onwards. https://github.com/intel/platform-aware-scheduling/issues/90#issuecomment-1169012485 links to an issue cut to the Descheduler project team to try to find a solution to this problem.
5. User deploys a Pod associated to TAS policy (H) for example, set "policy-1" as a label in the pod specs.
   - The requested pod is advertised on K8s API (C) and picked up by TAS via K8s Scheduler (D).
   - TAS checks if the pod is linked to any TAS policy in the cluster by getting its name in the pod spec.
   - TAS internally applies Filter/Prioritize according to the associated strategies and rules in TAS policy linked to the deployed pod (G, H)
   - TAS sends the results to K8s Scheduler (D).
   - K8s scheduler (D) schedules the pod by considering TAS results and then advertises the result in K8s API (C).
   - Then Kubernetes internal components will make sure that the pod associated to TAS policy is ready in the feasible worker node (J).

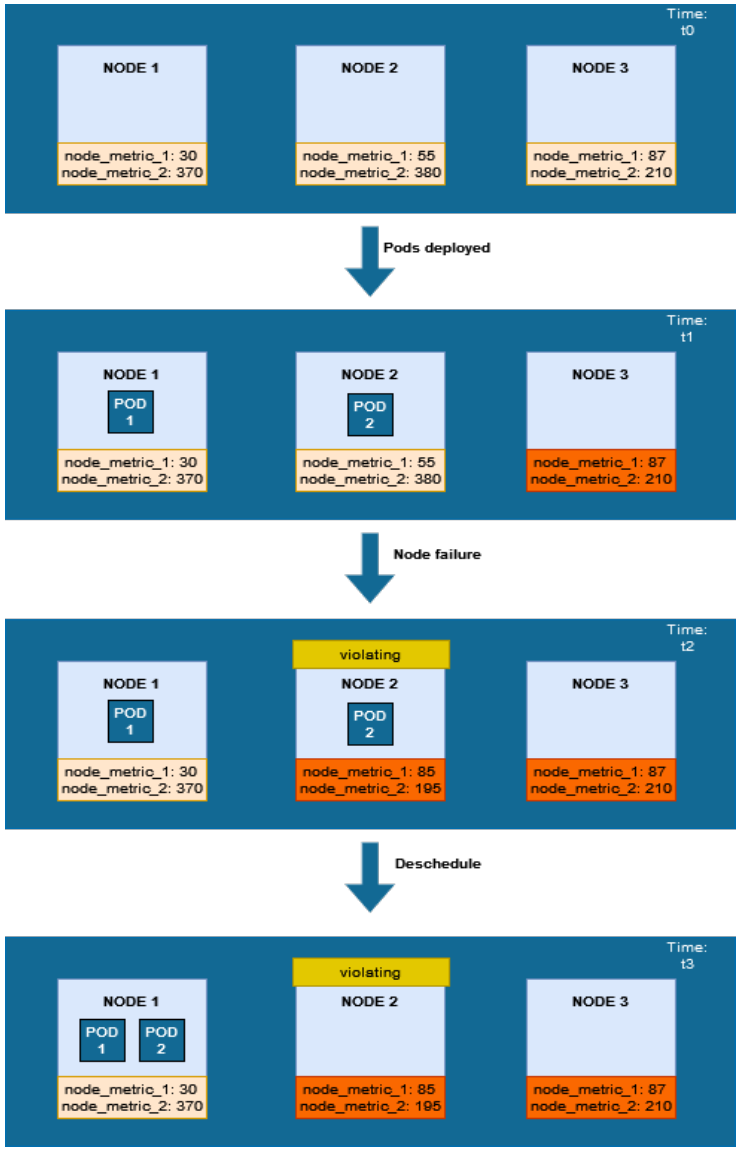

Figure 2. Deployment Diagram

Figure 2 illustrates an example of how pods are deployed with a "*demo-policy*" policy demonstrated in previous section:

- Time t0:
  - No pods with '*demo-policy'* label in the nodes.
  - Node 3 is filtered out by strategy *dontschedule* in TAS policy.
- Time t1:
  Pod1 and Pod2 are deployed with '*demo-policy'*.
  Since, the metric values don't violate the policy, Pod1 and Pod2 are deployed on Node1 and Node2 respectively.
- Time t2:
  - Node2 has node_metric_1 value = 85 and node_metric_2 = 195. This violates policy scheduling and follows strategy *deschedule*.
  - Node 2 now labelled as 'violating'.
  - Descheduler evicts Pod2.
- Time t3:
  - Node2 is now filtered out along with Node3
  - Node1 metric value allows strategy *scheduleonmetric*
  - Pod2 is now scheduled on Node

4 Implementation Example

By using either the previous guide or the repository, TAS and its dependencies are then up and running in the cluster with the "demo-policy" TAS policy. The implemented example uses two dummy metrics (*node_metric_1*, *node_metric_2*) whose values are manually changed in a text file that is scraped by the node metric agent (node exporter). The changes of the metrics' values will trigger specific actions from TAS. Two value thresholds are defined for *node_metric_1*, respectively *node_metric_2*, in the "demo-policy" TAS policy described in section 2.2. Therefore, where node_metric_1 has values less than 80 and node_metric_2 has values greater than 200, that will make the node in question feasible for the deployment of workloads that are associated to the "demo-policy". If there is more than one feasible node, TAS will advertise a priority list based on the magnitude of the metric values and the rule applied for the "scheduleonmetric" strategy defined in the "demo-policy". In the implemented example, the rule tells the scheduler to direct workloads toward nodes with the lowest value for *node_metric_1* metric.

If, for example, the value of the node_metric_1 metric would be 87 and node_metric_2 would be equal to 210, TAS will indicate to the K8s scheduler that nodes with that metric value are not feasible, and they should be filtered out of scheduling ("dontschedule"). Further on, if the node_metric_2 value would now decrease to 190 (so node_metric_1 = 87, node_metric_2 = 190) nodes where this happens will also be labeled by TAS as "violating, workloads in such nodes will be suitable for eviction by the Descheduler and rescheduled in another feasible node (Figure 2 summarizes a similar example).

5 Benefits

The addition of the labeling strategy in TAS allows the creation of custom node labels based on rule violations. The labels can then be used by other external components (that is, Descheduler, GAS) to allow for more complex node filtering queries. The introduction of the "anyOf" and "allOf" operators throughout all the TAS policy strategy types allows customers to describe more complex scheduling scenarios. Policy rules will now be able to reference more than one piece of telemetry.

GAS can now take into consideration more resource types as it will offer support for GPU-tiles and from this release it will allow containers to be scheduled on the same GPU.

6 Summary

Telemetry Aware Scheduling is introduced as an extender to help Kubernetes and the community to improve the resource optimization based on the telemetry data generated from the workload's resources. TAS overcomes the performance difficulties faced by the default scheduler by bridging data information from the platform telemetry to decisions taken at the scheduler. It does that by targeting specific metrics to the desired workloads specifics within customized policies - a set of customized strategies and rules - which allows TAS to forward the workloads to the right node in the working cluster. TAS also helps to indicate critical nodes that should not receive workloads or even nodes that should have workloads descheduled from one platform.

Platform Aware Scheduling that hosts TAS is ready to receive other extenders that utilize TAS code in order to take telemetry into account for scheduling workloads with the desired resources.

TAS and other extenders within the Platform can provide the automation actions on workloads scheduling and many opportunities to reduce the operational cost and at the same time reduce the overhead on the cluster management and make an overall improvement of the system. Those interested in further information on the benefits of PAS/TAS can access the open-source project repository at https://github.com/intel/platform-aware-scheduling where deployment scripts and instructions are readily available.

1022/DN/WIPRO/PDF 648029-002US