TECHNOLOGY GUIDE

Intel Corporation

Facilitating Consumption of Kubernetes Extenders with Telemetry Aware Scheduler

Authors

Denisio Togashi

Tejas Shah

Marlow Weston

1 Introduction

In today's Cloud technology, Kubernetes (K8s) is the leading edge solution 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 all have different optimizations, and K8s is tasked with directing those elements. One of the more challenging areas of optimization is resource management, especially with the increase in the number of machines due to the fast growth of new technologies. Valuable solutions optimize the efficiency and functionality of the current workloads while also enabling a reduction in both cost and environmental impact. At Intel, we have developed Telemetry Aware Scheduling (TAS) to address these needs.

TAS helps Kubernetes and the community to improve resource utilization based on the telemetry data generated from workload resources and allows data-driven workload scheduling for supported resources. It is based on defined policies that provide full automation of workloads deployment and management in a closed loop via predictive and reactive actions fed by the metrics from the resources. To address the difficulty of consumer consumption, TAS has been restructured to facilitate the use of its core libraries and functionalities by other technologies. With this concept, Platform Aware Scheduling (PAS) was created to enhance TAS utilization across other scheduler extenders, including GPU Aware Scheduling (GAS). GAS is the part of Platform Aware Scheduling that handles fractional GPU workload requests in multi-card GPU nodes.

This guide expands the previous article on Telemetry Aware Scheduling. It revisits how TAS paved the way to enhance K8s scheduling capacity, and it shows an example of TAS implementation using data metrics to assist the K8s scheduler to schedule the best workload placement in the cluster. This paper also appeals to K8s cluster operators, developers, and architects in introducing the changes made in TAS to allow other extenders to join Platform Aware Scheduling, the ultimate scheduler extender host platform.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	September 2021	Initial release.

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

REFERENCE	SOURCE
Telemetry Aware Scheduling	https://github.com/intel/platform-aware-scheduling/tree/master/telemetry-aware-scheduling
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

The Platform Aware Scheduling (PAS) repository hosts Telemetry Aware Scheduling, which is the main engine that drives the scheduler decisions for a workload deployment. TAS is thus an add-on component that extends the work of the Kubernetes Scheduler. Any other scheduler extender hosted by PAS, such as GAS, follows the same pattern, i.e., to be an extension for Kubernetes Scheduler by using the TAS share library, using platform metrics, and through the TAS policies gets the best decision for the workload deployment. Figure 1 shows the relationship between PAS, TAS, GAS, and other scheduler extenders.

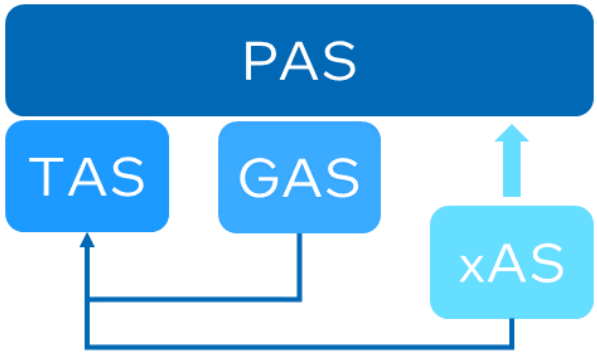

Figure 1. Platform Aware Scheduling: Scheduler Extender Relationship

Based on this concept, TAS, via Platform Aware Scheduling, can deliver solutions that the default Kubernetes Scheduler cannot. Solutions include noisy neighbor protection where co-located workloads can compete for specialized resources (e.g., last level cache) and visual processing at the edge, such as the Intel® Movidius™ Myriad™ X[1] Vision Processing Unit (VPU) applied in edge AI image processing workloads. TAS can ensure that any new VPU workload is scheduled on a node where there is more VPU capacity available, enabling high performance in use cases like traffic monitoring, video security systems, and facial recognition.

TAS complements the platform resilience use case, where health indicators from the platform enable TAS to deliver the best placement decision based on most-updated information regarding the health of the nodes to the K8s Scheduler. At the same time TAS indicates a possible faulty host node to the K8s Scheduler to avoid scheduling critical workloads on them. For more information on using TAS with platform resilience, refer to the Closed Loop Automation white paper.

TAS also helps to solve GPU resource fractionalization, which can happen when the K8s Scheduler is not able to distinguish, at a granular level, the amount of GPU resources available for a workload. GPU Aware Scheduling (GAS), which utilizes TAS in Platform Aware Scheduling, provides visibility and awareness to the K8s Scheduler. For more information about GAS, refer to the PAS repository.

2.1 Challenges Addressed

In Kubernetes, the term scheduling means assigning the pod to a node, and the component responsible for this is Kube-Scheduler (or simply, scheduler). When a pod with the workload is found by the scheduler, it tries to schedule it on the best possible available nodes that can provide the minimum computer resources 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 first-class resources (i.e., CPU and memory) can cause the scheduler to make less than optimal decisions and affect workload performance. A critical example would be the schedule of a workload to a node with a high CPU usage instead of a node with a low CPU usage.

[1] Intel, the Intel logo, Movidius, and Myriad are trademarks of Intel Corporation or its subsidiaries.

Other resources like GPUs, FPGAs, and network cards are seen as atomic devices and cannot be looked at as having available resources at a granular level. For example, workloads that need intensive usage of GPUs are 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. For more details, see the GAS repository.

Kubernetes' lack of insight to the current usage level of the available resources can lead the scheduler to make non-ideal scheduling decisions and cause an imbalanced distribution and unexpected resource usage by the workload across the cluster. TAS assists K8s in these cases and uses platform telemetry to support the scheduler so that it knows the current resource level usage and considers the node history usage. In this way, it can predict the best node for the workload and avoid scheduling on predicted faulty devices.

2.2 Technology Description

Upon deployment of a workload, i.e., placement of workload's pod on a feasible node in a cluster, the 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 the K8s default scheduler by providing a scheduling policy to the default Kubernetes scheduler. The following shows an example of such a policy.

2.2.1 scheduler-extender-policy Example

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scheduler-extender-policy
  namespace: kube-system
data:
  policy.cfg: |
    {
        "kind" : "Policy",
        "apiVersion" : "v1",
        "extenders" : [
            {
              "urlPrefix": "https://tas-service.default.svc.cluster.local:9001",
              "apiVersion": "v1",
              "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 the PAS repository. Some of these fields, such as setting the `urlPrefix`, `filterVerb`, and `prioritizeVerb`, are necessary to point the Kubernetes scheduler to the TAS scheduling service running at `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 the descheduler, which 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 `scheduleonmetric`, `dontschedule`, and `deschedule`. Each of these strategies is governed by rules that relate the platform metrics data values to a desired state defined by the user or operator.

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

2.2.2 demo-policy Example

```
apiVersion: telemetry.intel.com/v1alpha1
kind: TASPolicy
metadata:
  name: demo-policy
  namespace: default
spec:
  strategies:
    scheduleonmetric:
      rules:
      - metricname: health_metric
        operator: LessThan
    dontschedule:
      rules:
      - metricname: health_metric
        operator: Equals
        target: 1
      - metricname: health_metric
        operator: Equals
        target: 2
    deschedule:
      rules:
      - metricname: health_metric
        operator: Equals
        target: 2
```

The policy named `demo-policy` contains three separate rulesets defined under the requested strategies. Each of these rules uses a metric named `health_metric` to build a set of recommendations based on the comparison of values defined in the policy field, i.e., target, and the metrics collected by platform metrics.

The `health_metric` is the result of input from multiple sources, including Intel® Resource Director Technology (Intel® RDT), processor performance monitoring units (PMUs), and Intel® Xeon® reliability, availability, and serviceability (RAS) metrics. If the `health_metric` displays a value of 1, this means that there is some issue on the system and deployments of critical workloads should be blocked. Whereas a value of 2 means that there is a serious fault and critical workloads should be evicted and rescheduled on a healthier node. To learn more about using TAS for workload resilience and service healing, see the Closed Loop Automation white paper.

In the above `demo-policy` example, the rule in the `scheduleonmetric` strategy advertises the default scheduler to place workloads toward nodes with the lowest reading, i.e., the most healthy `health_metric`. The rules in the second strategy, `dontschedule`, tell the scheduler to filter out nodes with readings of 1 or 2. The rule in the third strategy, `deschedule`, tells TAS that the node for which the metric `health_metric` reading is equal to 2 is not healthy enough to keep running workloads. Workloads in this node should be removed and rescheduled.

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 enables smart scheduling recommendations and 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.

2.3 Architecture

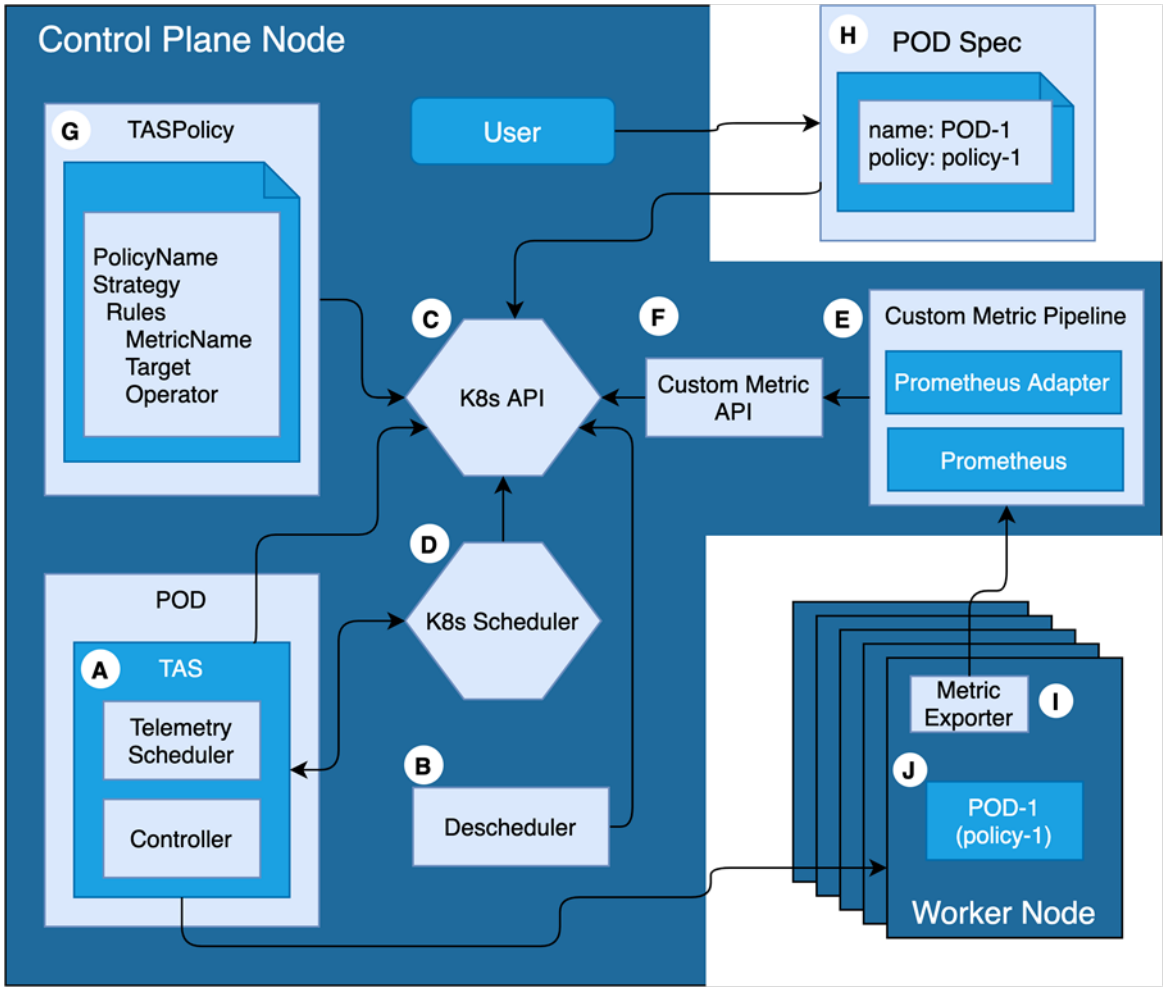

Figure 2. Architecture Diagram

Figure 2 shows a diagram of a Kubernetes cluster with a separate control plane node and worker nodes. The key components are as follows.

A. Telemetry Aware Scheduling: TAS extender is made up of two main elements in the code base deployed as single container.
   - Telemetry Scheduler recommends nodes for pod placement based on metrics rules.
   - Controller watches the state of critical metrics and marks violating nodes as such.

B. Kubernetes Descheduler picks up the pods due to be evicted, runs safety checks, and deschedules.
C. Kubernetes API server validates and configures data for the K8s API objects.
D. Kubernetes Scheduler performs scheduling operations and looks for advice from TAS.
E. Custom Metric Pipeline gets metrics from the node metric exporter agents and makes them available to K8s API via Custom Metric API.
F. Custom Metric API is a K8s API aggregation layer that receives metrics from Custom Metric Pipeline and makes them available to TAS.
G. TAS Policy is a set of strategies and rules made available as a policy custom resource in K8s API.
H. Pod Spec specifically references the policy that should be used to influence the scheduling of this workload.
I. Node Metric Exporter agents “Collectd” and/or "Node Exporter”.
J. POD-1 linked to policy-1 deployed in a feasible node.

The architecture changes made (related to previous guide) to provide the necessary structure for the creation of a Platform Aware Scheduler are as follows.

- Refactor of TAS Scheduler package of the previous code into generic and specific library packages. This allows improved usage of the code for other extenders within the Platform Aware Scheduler, e.g., GAS.

- TAS Extender and TAS Policy Controller combined in one binary component. Previously, the two components were deployed in different containers by building different images. To help improve security, the two components were combined into a single code and deployed in a single container.
- Implementation of mutual TLS. TAS Scheduler Extender listens on a TLS endpoint, which requires a certificate and a key to be supplied.
- Implementation of structured logs of klog to allow improved log description at different log levels.

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 1.13 or higher
   - K8s installed on a multi-node cluster using Kubeadm/Kubespray 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
   - Deploy the TAS system - https://github.com/intel/platform-aware-scheduling/blob/v0.4/telemetry-aware-scheduling/deploy/
   - Configure K8s scheduler by deploying the TAS extender policy - https://github.com/intel/platform-aware-scheduling/blob/v0.4/telemetry-aware-scheduling/deploy/extender-configuration/
4. Install the Kubernetes Descheduler - https://github.com/kubernetes-sigs/descheduler

Note: The deployment steps are based on TAS v0.1 described in the previous paper [https://builders.intel.com/docs/networkbuilders/telemetry-aware-scheduling-automated-workload-optimization-with-kubernetes-k8s-technology-guide.pdf] and the sequence is valid for TAS within PAS v0.4. Refer to the repository for the latest aspects of the deployment process.

Note: The custom metric pipeline components can be installed by using Helm charts in the quick installation at https://github.com/intel/platform-aware-scheduling/blob/v0.4/telemetry-aware-scheduling/docs/custom-metrics.md#quick-install

Note: Collectd is an additional complementary node metrics provider that makes many additional platform metrics available.

3.2 TAS in Operation

After 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 2, 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 (e.g., `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 complied, TAS labels the nodes that comply with the `deschedule` strategy's rules as "violating".
   - The "violating" label is advertised on K8s API.
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 reschedules the evicted pods to other available and feasible nodes.
5. User deploys a pod associated to TAS policy (H), e.g., 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).
- Kubernetes internal components make sure that the pod associated to TAS policy is ready in the feasible worker node (J).

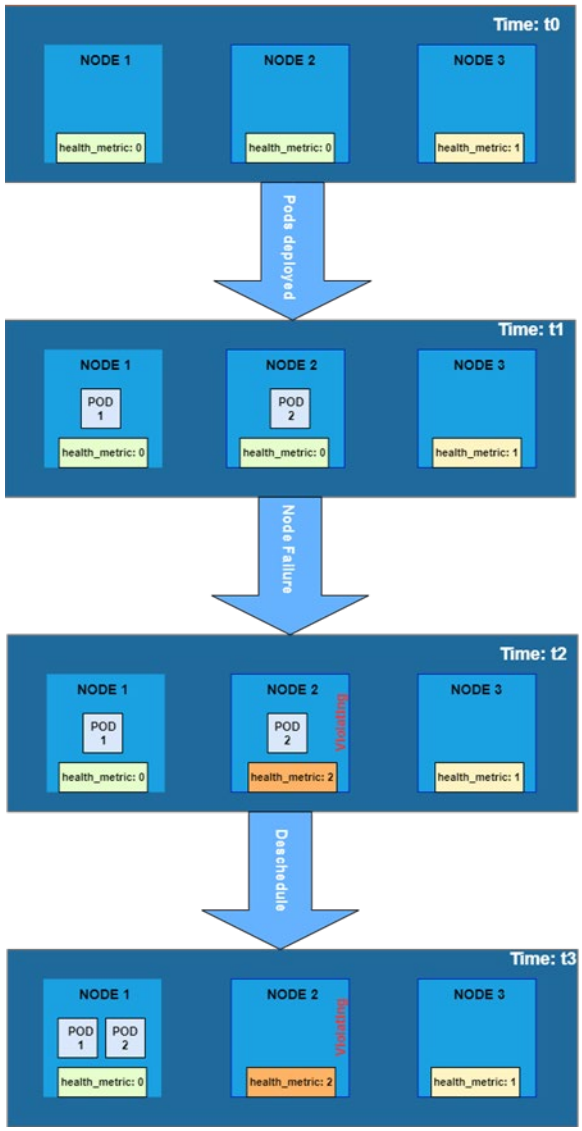

Figure 3. Deployment Diagram

The above figure illustrates an example of how pods are deployed with a `demo-health` policy demonstrated in Section 2.2, where:

- Time t0:
  - No pods with `demo-policy` label in the nodes.
  - Node3 is filtered out by strategy `dontschedule` in TAS policy.
- Time t1:
  - Pod1 and Pod2 are deployed with `demo-policy`.
  - Since the metric values do not violate the policy, Pod1 and Pod2 are deployed on Node1 and Node2.
- Time t2:
  - Node2 has metric value = 2. This violates policy scheduling and follows strategy `deschedule`.
  - Node2 now labeled 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 Node1.

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 a dummy metric (`health_metric`) with three different values that

represent the metrics for the health of a node. These values are manually changed in a text file that is scraped by the node metric agent (node exporter). The changes in the `health_metric` values trigger specific actions from TAS. Two values are defined in the `demo-policy` TAS policy described in Section 2.2. Therefore, any metric values different from 1 or 2 make the nodes feasible for the deployment of workloads that are associated to the `demo-policy`. If there is more than one feasible node, TAS advertises a priority list based on 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 reading, i.e., most healthy `health_metric` reading. For the value of 1 equivalent to `dontschedule` strategy, TAS indicates to the K8s scheduler that nodes with that metric value are not feasible and they should be filtered out of scheduling. Nodes with `health_metric` equal to 2 are labeled by TAS as "violating." Workloads in such nodes are suitable for eviction by the Descheduler and are rescheduled in another feasible node (see Figure 3 that summarizes the example implemented).

4 Summary

Telemetry Aware Scheduling is introduced as an extender to help Kubernetes and the community to improve 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 use 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 automation actions on workloads scheduling and many opportunities to reduce the operational cost, and at the same time reduce the overhead, on 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.

0921/DN/WIT/PDF 648029-001US