Kubernetes cluster optimization using hybrid shared-state scheduling framework

Oana-Mihaela Ungureanu
University Politehnica of Bucharest
Bucharest, Romania
oana.ungureanu30@gmail.com

Călin Vlădeanu
University Politehnica of Bucharest
Bucharest, Romania
calin@comm.pub.ro

Robert Kooij
Delft University of Technology
Delft, The Netherlands
Singapore University of Technology and Design
Singapore, Singapore
r.e.kooij@tudelft.nl

ABSTRACT

This paper presents a novel approach for scheduling the workloads in a Kubernetes cluster, which are sometimes unequally distributed across the environment or deal with fluctuations in terms of resources utilization. Our proposal looks at a hybrid shared-state scheduling framework model that delegates most of the tasks to the distributed scheduling agents and has a scheduling correction function that mainly processes the unscheduled and unprioritized tasks. The scheduling decisions are made based on the entire cluster state which is synchronized and periodically updated by a master-state agent. By preserving the Kubernetes objects and concepts, we analyzed the proposed scheduler behavior under different scenarios, for instance we tested the failover/recovery behavior in a deployed Kubernetes cluster. Moreover, our findings show that in situations like collocation interference or priority preemption, other centralized scheduling frameworks integrated with the Kubernetes system might not perform accordingly due to high latency derived from the calculation of load spreading. In a wider context of the existing scheduling frameworks for container clusters, the distributed models lack of visibility at an upper-level scheduler might generate conflicting job placements. Therefore, our proposed scheduler encompasses the functionality of both centralized and distributed frameworks. By employing a synchronized cluster state, we ensure an optimal scheduling mechanism for the resources utilization.

ICFNDS, July 2019, Paris, France
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7163-6...$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Fault-tolerant network topologies**; • **Computing methodologies** → *Shared memory algorithms.*

KEYWORDS

Kubernetes cluster, pod, service, taints, tolerations, affinity, anti-affinity, optimistically concurrent transaction, opportunistic scheduling, node feasibility, priority preemption, collocation interference, inherent rescheduling

ACM Reference Format:
Oana-Mihaela Ungureanu, Călin Vlădeanu, and Robert Kooij. 2019. Kubernetes cluster optimization using hybrid shared-state scheduling framework. In *Proceedings of ACM International Conference on Future Networks and Distributed Systems (ICFNDS).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Nowadays server virtualization is widely used in data centers and its main aim is to decouple applications from the underlying infrastructure. Either used as hardware virtualization (virtual machines) or operating system level virtualization (containers), virtualization technologies are shaping the world of application deployment. The difference between the two types of virtualization mainly consists in the way the operating system is running: virtual machines are running their own operating system on top of a hypervisor which virtualizes the server's resources, while in containers, resources are virtualized at the operating system level as they encapsulate the application's processes and dependencies [31].

Containers and, in particular, Docker containers [8] have gained much popularity in the last couple of years due to the increasing number of container-based applications and the early adoption of DevOps technologies by large companies such as Google and Netflix. The manner in which an application is build, packaged and deployed has been simplified by container management frameworks such as Kubernetes [18] developed by Google, Docker Swarm [32] or Apache Mesos [26].

Nevertheless, in a container cluster (consisting of at least one master and multiple worker nodes) the resources utilization fluctuates for various reasons: users do not submit their jobs at the same time, jobs are different in terms of resource requirements, input data size differs for daily jobs, thus jobs pass during their life time through many stages and types of parallelism with different resource requirements. An important role in container management frameworks is played by the *scheduler's* components which are directly responsible for the distribution of workload across available hosts. The load spreading should be done according to cluster size as long as capacity constraints are not violated and it depends on the updated state of the cluster. From a deployment perspective, the scheduler manages the life cycle of containers and their distribution across nodes [9].

In a production environment running high-performance management clusters, the workload is mixed thus a single scheduler is not able to manage accordingly the long-running service jobs and batch analytics. Each application has different requirements which will increase the complexity and implementation of a scheduler. The scheduler's processing logic becomes an issue: queuing and job prioritization need to be carefully addressed [30].

The **centralized (monolithic)** schedulers (e.g., Borg [3], Apache Hadoop YARN [37], Docker Swarm [32], Kubernetes [18] and Firmament [13]) usually run complex algorithms for quality of Service (QoS) and high quality task placements with higher latencies, whereas the **distributed** schedulers (e.g., Sparrow [28]) use less complex algorithms that minimize the latency at task placements, but the resources utilization is not visible at an upper-level scheduler. The third category called **hybrid** or **semi-distributed** (e.g., Tarcil [7]) utilizes elaborated algorithms for the long-running tasks and rely on simple algorithms for short running tasks, but since it offers no visibility over the cluster state, it will sometimes end with job placement conflicts.

Apart from the listed schedulers categories, we have identified in the most recent literature two additional types of schedulers: **two-level** and **shared-state** schedulers. In contrast to the monolithic schedulers that use a centralized scheduling algorithm for all jobs, the **two-level** schedulers have a decentralized scheduling model which delegates control over scheduling to the frameworks (e.g., Hadoop-on-Demand [1] and Mesos, using the framework Marathon [26]). In a **shared-state** scheduling mechanism, an application-level scheduler updates multiple replicas of the cluster state. Every change triggers the scheduler to issue an *optimistically concurrent transaction* to update local changes to the shared cluster state (e.g., Omega [24], Apollo [2], Nomad [15] and Tarcil [7]).

Due to the fact that Kubernetes is a centralized type of scheduler, we propose an optimization of the Kubernetes workload distribution using a **hybrid shared-state** scheduling framework. The problems identified in our deployed Kubernetes cluster workload presented in Section 4 show that a centralized model cannot solve problems like service high-availability, collocation interference, priority preemption or inherent rescheduling. Therefore, our approach is a request-model hybrid scheduler in which the application-level scheduler (the master-state agent) receives and updates the replicas of the entire cluster state. Compared to other current approaches, our proposal differs from a classical semi-distributed scheduling model (i.e., Tarcil [7]) where the short-tasks with low-priority are run by distributed agents, whereas the rest of the tasks are placed according to the central scheduler logic. We introduce the scheduling correction function that processes the unscheduled and unprioritized jobs of both categories. From the resources utilization perspective, if task workload tends to idle certain nodes (nodes are under-utilized) our approach recalls the "`opportunistic scheduling`" model in which the scheduling agents run tasks at lower priority so they can be evicted if nodes tend to become over-utilized. (i.e., Tarcil [7] and Apollo [2]).

This paper is structured as follows: firstly we present in Section 2 a taxonomy of the scheduling mechanisms in container management frameworks. This survey constitutes a novel contribution as we could not find in the literature a complete comparison of the existing schedulers. In order to analyze the cluster fluctuation behavior over time, we deployed a Kubernetes cluster and presented its architecture in Section 3. Next, Section 4 highlights the identified problems in the Kubernetes Default Scheduler also based on our tested scenarios and in Section 5 we presented our proposed hybrid shared-state scheduler architecture. In Section 6 we conducted an analysis of the proposed scheduler features compared to the existing integrated centralized schedulers in Kubernetes and in Section 7 we presented some of our conclusions and proposal of future work.

2 RELATED WORK AND A BRIEF TAXONOMY OF THE EXISTING SCHEDULING MECHANISMS

Since there is not a unique solution to solve every problem within the computing domain, multiple cluster schedulers have been developed by different large companies. For instance, Google (main contributor of Omega [24] and Kubernetes [18]) targets an architecture that gives control to the developers, assuming that they respect the rules concerning the priority of their jobs in the cluster, while Yahoo (main contributor for YARN [37]) targets an architecture that enforces capacity, fairness and deadlines [14]. Table 1

Table 1: Taxonomy of the different existing scheduling frameworks in the current literature

Scheduler	Centralized schedulers	Two-level schedulers	Shared-state	Hybrid	Fully distributed
`Borg`	✓				
`Omega`			✓		
`Apache Hadoop YARN`	✓				
`Apollo`			✓		
`Kubernetes`	✓				
`Hadoop-on-Demand`		✓			
`Apache Mesos`		✓			
`Docker Swarm`	✓				
`Firmament-Poseidon`	✓				
`Nomad`			✓		
`Sparrow`					✓
`Tarcil`			✓	✓	

summarizes our survey and classifies different schedulers identified in the current literature.

Kubernetes represents a widely used **centralized scheduling mechanism** and is the third container-management system developed at Google and the first open-source project, its predecessors being Borg [3] and Omega [24]. Unlike the first two, Kubernetes models its core as a shared persistent store, exposed through APIs. Building management APIs around containers is considered more developer-oriented and shifts the primary concern of data center provisioning from machine to the application layer.

Another monolithic scheduler is implemented in Docker Swarm [32]. The scheduler relies on filters to reduce the nodes' range based on container properties and strategies to decide on which node to schedule a container using three methods: *random, spread* and *binpack*. There have been new proposed scheduling strategies in Docker Swarm [5] based on classes (premium, advances and best effort) according to user needs. In this way, the CPU cores are distributed across containers based on the load and user economic model [25]. Apache Hadoop YARN (developed by Yahoo) [37] is a popular architecture for Hadoop that delegates many scheduling functions to per-application components. Its architecture also corresponds to a monolithic scheduler due to the fact that its resource master provides a global scheduler that processes the request from the application masters.

A more recent example of a centralized scheduler is Firmament [13], a flow-based scheduler. Firmament models the entire cluster as a flow network as it runs a *min-cost max-flow* (MCMF) optimization over the network to make scheduling decisions. Firmament has been integrated with Kubernetes under the Poseidon Project [29]. More key features of this scheduler will be presented in Section 6.

The Mesos' **two-level scheduler** is presented by Hindman et al. in [16]. This model is based on a resource manager that allocates computing resources to multiple parallel, independent ""`scheduler frameworks`". A resource offering may refer to a bundle of resources that a framework can delegate on a cluster node to run tasks. The scheduler decides how many resources to allocate per framework using an organizational policy such as fair sharing, while frameworks decide which resources to accept and which tasks to run on them. Hadoop-on-Demand (currently replaced by YARN) is another two-level scheduler that also uses a central coodinator to dynamically assign the allocation of resources to each scheduler. This strategy is called "`pessimistic`" and considered less error-prone since each resource is taken care at a time by a single scheduler.

Schwarzkopf et al. [24] implemented a new parallel **single shared-state scheduler** for Omega. There is no central resource allocator in Omega; all of the resource-allocation decisions take place in the schedulers. Each one of the schedulers receives an updated copy of the resource allocations called "`cell state`" used for scheduling decisions. In this manner, every scheduler has an overall view of the cluster state and can allocate any available resources with the right permissions and priorities. Every time a decision is made by the scheduler, it triggers a commit and updates the shared copy of the cell state. In case the commit fails due to mis-synchronization between the commit and update, the whole process is retaken and in the worst case scenario the scheduling algorithm is being re-run.

The authors of [2] developed a **shared-state** scheduler named Apollo where the jobs' scheduling is done independently and a *Resource Monitor* (RM) collects load information from each *Process Node* (PN), providing an overall view of the cluster state. Based on this information the *Job Manager* (JM) makes scheduling decisions. The JM receives feedback also from the PN so it can perform task runtime estimation. When a task starts running, the PN provides information regarding the memory usage, CPU, execution time and I/O throughput. Moreover, the PN keeps an updated matrix of the estimated wait-times for future tasks based on the running and queued tasks. Apollo scheduler employs an ""`opportunistic mode`" as it runs tasks at a lower priority so they can be easily preempted or terminated if the server runs out of resources. The Apollo scheduler also employs correction mechanisms that periodically re-evaluate the scheduling decisions and make adjustments whenever necessary. For example, in case two tasks are simultaneously scheduled on the same server and they are competing for resources, in order to avoid such conflict, a small random number is added to each completion time estimation.

Nomad scheduler [15] developed by HashiCorp, represents another **shared-state scheduling** mechanism that uses a different architecture consisting of jobs, nodes, allocations and evaluations. The set of tasks that should run on a node is mapped to the jobs using allocations, while the scheduling mechanism of calculating the right allocations is called evaluation. Multiple schedulers are used in this architecture, for instance a service scheduler for long-lived services, a batch scheduler for optimized placement of batch jobs, a system scheduler to allocate tasks on the nodes and another core scheduler for internal processes.

In contrast to the shared-state scheduler, Sparrow [28] employs a **fully distributed scheduling** mechanism based on multiple schedulers that operate in parallel without maintaining a state of the entire cluster. On the contrary, the load information is retrieved from the worker machines. The tasks are also run by the worker machines so the jobs can be handled by schedulers.

Tarcil [7] is a **hybrid shared-state** without a central coordinator scheduling framework where the schedulers work in parallel and have an updated local copy of the shared server state as well as the status of the *Resource Units* (RU). Once a task is assigned to a server by a scheduling agent, the RU status is updated along with the master cluster state copy and the admission control is notified by the local monitor. The per-RU local monitor evaluates the performance and informs the agent whenever an allocation adjustment needs to be done with respect to CPU or memory saturation that ultimately triggers the auto-scaling mechanism. Every time a scheduling agent makes a decision, it also updates the master state. Conflicts can also occur since all schedulers have full access to the cluster state and this issue is resolved through the "`lock-free optimistic concurrency`" method. The role of the admission control is to preserve a queue of the jobs until resources become available and to estimate the application wait-time.

In our approach we employed a similar hybrid shared-state architecture where the collocated task interference is managed by a scheduling correction function. In contrast to the admission control component implemented in Tarcil, our scheduling correction function compares the wait-time task forwarded by the resource nodes with its estimate, calculated based on the number of concurrent transactions made over time by each of the scheduling agents.

Looking at the scheduling frameworks listed in Table 1, Firmament-Poseidon is the only scheduling framework that has been integrated so far with Kubernetes system. In order to optimize the scheduling in our deployed Kubernetes cluster, our proposal targets either a future integration/co-existence with the Kubernetes default scheduler or a written plugin to replace the default one. Therefore, in our analysis, we will further consider the Kubernetes defined concepts and components as well as the "`Binding-Object`" function that maps the pod requirements with the nodes capacity and resource allocation. The rest of the paper will focus on the integrated scheduling frameworks with Kubernetes as we will analyze the problems we encountered in a deployed Kubernetes cluster.

3 ARCHITECTURE OF OUR DEPLOYED KUBERNETES CLUSTER

In comparison to the other container management solutions, Kubernetes (K8S) uses the concepts of "`Labels`" and "`Pods`" for grouping containers that form a single deployed service.

A *Pod* has been defined as the smallest unit in the Kuberentes object model which represents a running process in a cluster. A *Service* (or micro-service) consists of a logical set of pods and a policy to access them. A *Label Selector* determines the set of Pods targeted by a service. A *Job* can consist of multiple tasks that will create one or more pods. The job completes when a specified number of successful "completions" (the pod life cycle ends) is reached [18].

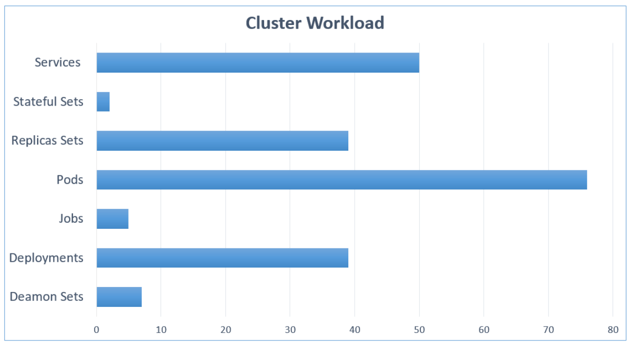


Figure 1: Workload distribution in our K8S deployed cluster

The *ReplicaSet* role is to ensure a set of Pods replicas running at certain time. On the other hand, a *Deployment* can create and update the ReplicaSet or the pod's state [17].

The *DeamonSets* gives the option to run a particular pod on a set of nodes that match the expressed criteria. In this manner a database can be run on a defined set of nodes while ensuring the service availability without re-scheduling. On the other hand, *StatefulSets* allow pods to be rescheduled on other machines as long as a set of replicas maintains service availability [23].

Figure 1 shows the workload distribution in our deployed Kubernetes cluster. The testbed consists of a Kubernetes cluster with one master node, three workers and an NFS server running on the master node. The cluster has been deployed in a virtualized VMware environment. Each node has 32 GB of RAM, an 8 cores Processor and a disk space of 150 GB. As it concerns the workload distribution, it can be observed that the number of deployed Pods is almost twice the number of Deployments which is almost equal to the number of ReplicasSets.

At control plane level, the master's main responsibility is to take decisions like scheduling and detecting events (e.g., start of a new pod). The K8S master system components are:

- `kube-apiserver` : the front-end for the Kubernetes control plane that uses the master's API to retrieve current state information and also update the API with new information about the desired state (i.e., on which node a new pod should be scheduled, or which pod should be moved to another node) [18].
- `kube-scheduler` : the Kubernetes scheduler is one of the core components. Its main logic is to bind pods to nodes whenever a new node appears based on policies that contain a set of rules, called predicates and priorities [20]. The scheduler is constantly looking at the Kubernetes API for unscheduled pods and once pods are found, it makes a decision about the node where to place the pods based on the node's resources and applied filters (i.e., memory, disk space, ports). The node with the greatest eligibility that gets the best score will be elected to run the pod [6].
- `kube-controller-manager` : monitors the shared state of the cluster through the kube-apiserver in order to reach a desired state.

The K8S standard node components run on each of the nodes ensuring a runtime environment:

- `Kubelet` : is an agent that runs on each node in the cluster and ensures containers described in the `PodSpec` are running without errors.
- `kube-proxy` : maintains network rules on the host [18].

Figure 2 shows the architecture of the cluster we deployed that contains the standard K8S components for both master and worker nodes, the system components, the extra components and services we deployed on top. Apart from the K8S standard master system components, other components have been deployed to ensure cluster functionality:

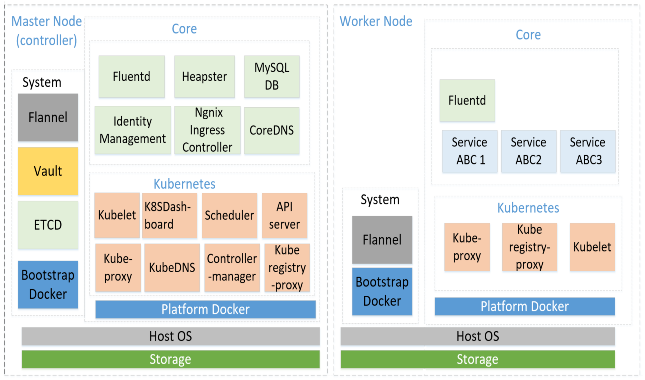


Figure 2: Our Kubernetes deployed cluster architecture, single master deployment with three worker nodes

- The `Flannel` daemon is responsible for setting up and managing the network that interconnects all of the Docker containers created by Kubernetes [18].
- `ETCD` is a distributed reliable key-value store [10].
- A `Docker Bootstrap` instance is used to start ETCD and Flannel, on which the Kubernetes components depend [19].
- `Vault` provides a token-based authentication method for Kubernetes pods [36].
- `Heapster` which provides cluster monitoring and performance analysis for Kubernetes [33].

Apart from the K8S components we have deployed on the master node a `MySQL` database, an identity management component for authentication purposes and an `Ngnix` ingress controller acting as a load balancer for the pods [27]. The `Fluentd` service runs on both master and worker nodes and provides log information, collects events and can be configured to send out alerts [11]. Note that in a production enviroment a high-availability (HA) deployment is required with at least three masters and three worker nodes, but in our case due to resource constraints (i.e., memory, CPU), we proceeded with a single master deployment.

4 IDENTIFIED PROBLEMS IN KUBERNETES SCHEDULER

As Kubernetes clusters are very dynamic and their state change over time, there may be a desire to move already running pods to some other nodes for various reasons:

- Nodes are under or over-utilized

- Taints or labels are added to or removed from nodes, pod/node affinity requirements are not satisfied anymore
- Nodes failed and their pods moved to other nodes
- New nodes are added to clusters

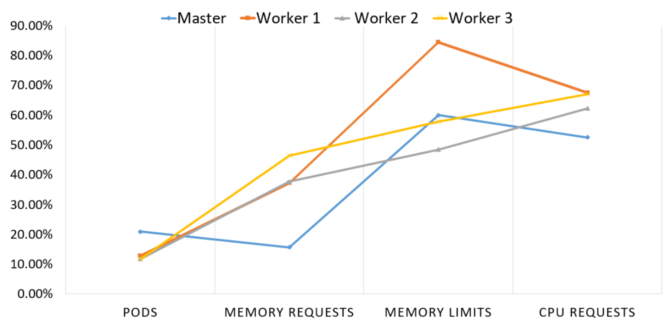
For instance, we assume three services A , B and C, each scaled to run three replicas on three different nodes. If Worker 1 becomes unrecheable, the scheduler running on the master node will notice that we only have two pods of each service. The Master Node will immediately reschedule one of each replicas on the remaining healthy nodes. The scheduler tries to avoid having duplicates on the same node, but in the case Worker 1 recovers in the meantime, it will be massively under-utilised compared to the other nodes. There is a high chance that all the new A_{v2} pod replicas will be scheduled on Worker 1 after its recovery. If Worker 1 were to fail again, the availability for service A is totally lost [35].

Figure 3a shows the initial resource consumption in our deployed K8S cluster, where Worker 1 has 84.45% utilization in terms of memory limits. Our findings show that in a long-time running cluster environment after Worker 1 fails, the workload is not equally balanced across the nodes, Worker 2 pod allocation will increase up to 20%, while memory limits requests achieve 97.6% and CPU 80.75%. A similar behavior happens for Worker 3 which reaches 99% CPU requests. Even after Woker 1 recovers (see Figure 3b), Workers 2 and 3 are over-utilized with the same memory consumption rate of 97.6% and 80.75%, respectively, while Worker 1 is completely under-utilized. As for the Master CPU and memory utilization, it remains the same and it does not react to the cluster state changes.

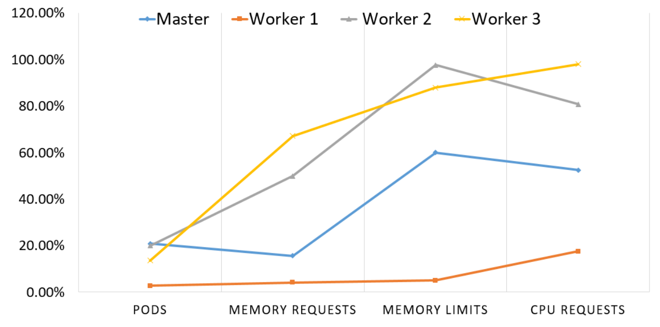
In a Kubernetes cluster, scheduling happens at pod creation only when pods are spread out intelligently but whatever change occur with the pods distribution across all nodes, is not proactively enforced. Kubernetes default scheduler is lacking of an optimal pod-by-pod scheduling mechanism as well as a rescheduling mechanism when the cluster state has been changed.

A possible solution to the high-availability issue is the **Kubernetes Descheduler** [20], a recent open-source project currently in beta release that has as a primary goal to re-establish balance within the cluster and avoid multiple pod replicas on the same node. Since Kubernetes clusters are very dynamic and their states change over time, the Descheduler includes several strategies:

- `RemoveDuplicates` – This strategy aims is to kill or reschedule identical pods on different nodes.
- `LowNodeUtilization` – The Descheduler searches for the nodes that are under-utilized and evicts pods in order to re-create the drained pods on those under-utilized nodes.



(a) Initial resource consumption in the deployed K8S cluster



(b) Resource consumption in the deployed K8S cluster after Node 1 recovers

Figure 3: Resource consumption in the deployed K8S cluster

- `RemovePodsViolatingInterPodAntiAffinity` – This strategy ensures that pods violating the interpod anti-affinity rule are removed from the nodes.
- `RemovePodsViolatingNodeAffinity` – The strategy enforces the pods which are violating the node affinity rule to be removed from the nodes.

Nevertheless, the current version of the Descheduler does not provide a strategy to consider the number of pending pods or the pod life time. Moreover, it does not consider the pod taints and toleration filters nor the Kubernetes scheduler's predicates. Currently, there is no integration with the cluster autoscaler or with metric providers to obtain the real load metrics. Another important aspect is that the Descheduler does not have a QoS mechanism implemented as *"Best efforts"* pods might be evicted before *"Burstable"* and *"Guaranteed"* pods.

5 THE PROPOSED HYBRID SHARED-STATE SCHEDULER

The proposed architecture of the hybrid shared-state scheduler (Figure 4) consists of several *Scheduling Agents* (SAs) configured as slaves, a master that has an overview of the cluster state, the *Master-State Agent* (MSA) and a *Scheduling*

Correction (SC) function that estimates the queuing time for the remaining unscheduled and unprioritized jobs. The *Resource Node* (RN) stores information regarding the resources utilization (i.e., memory usage, CPU, throughput and execution time) and shares this information with the SAs. The RNs manage the life time of each task, while the SAs run the life cyle of a job. Each SA processes this information along with the job priorities and *Node Feasibility* (NF), thus, after it takes a job placement decision, it sends the updated state to the MSA which further sends it to the SC and all the SAs.

Similar to the shared-state scheduler presented in Apollo [2], each RN has a local queue and sends a wait-time matrix with the resources availability (i.e.,memory usage, CPU time and I/O throughput) to the SA that predicts a task runtime. The SC function also calculates an average expected task runtime based on the information retrieved from the RNs. The main difference in our approach consists in the role that the SAs play in the decision's logic and the overall scheduling mechanism. In contrast to the Apollo scheduler where the process nodes send the load information to a central resource monitor, in our proposed scheduling mechanism the job placement decision is taken at the level of each SA based on the task runtime, pod priorities and task-wait time. More specifically, both batch-jobs and long running tasks are being processed locally be each of the SAs, while the unscheduled and unprioritized jobs are being taken care of by the SC.

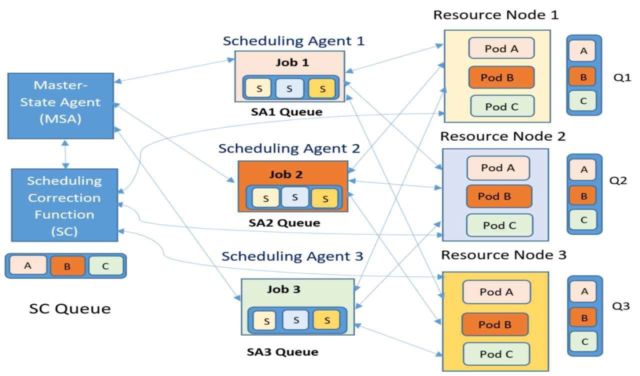


Figure 4: The proposed hybrid shared-state scheduler architecture

Scheduling Correction (SC) function

One of the main advantages of this kind of scheduling framework is that all the scheduling corrections are done outside the SAs by the SC function which has full control over the cluster state. The SC's role is to both maintain an updated state of the entire cluster and add in its queue the unscheduled and unprioritized jobs. The *unscheduled jobs* could be the result of slower processing tasks with high I/O latencies that have no pod priority specfications. On the other hand, the unscheduled tasks might constitute the reason for pod conflicts (pods that share the same node affinity/anti-affinity rule) when pods are competing for the same resources that generate *collocation interference* of the pods.

We consider the total job execution time of an incoming job in the SC queue $S = [S_1, S_2, \ldots, S_n]$. We define the total execution time of a task (S_i) as the sum of the task estimated wait-time (T_{w_i}) plus the estimated task runtime (R_i), where $i \in \{1, .., n\}$. The interference time for an incoming job is $I = [I_1, I_2, \ldots, I_n]$. In Tarcil [7], the interference is quantified using a set of microbenchmarks that determines the amount of pressure a job can tolerate and generate at the same time in a shared resource (i.e., caches, memory, and I/O channels). For the time being we consider in our approach the interference related to the response time/performance of the RN to the requested queries. Both parameters (S_i) and (I_i) are expressed in [ms] as the measure for job processing latency. In order to calculate the total execution time for an unscheduled task, S'_i, the SC function assigns a weight to the time derived from the interference latency (see Equation 1).

$$w_i = 1 - \frac{T_a}{T_{wi}} \tag{1}$$

The weight (w_i) (see Equation 2) is generated based on a list with the latest estimated task runtime calculated by the SC function taking into account the wait-time matrix provided by the RNs. In order to calculate (w_i), we propose to normalize the average estimated task runtime (T_a) to the retrieved estimated task wait-time (T_{w_i}). Then, the prioritized SC queue contains the new values for the total execution time of an unscheduled job $S' = [S'_1, S'_2, \ldots, S'_n]$ with the minimal collocation interference time (see Equation 2). The prioritization is then made based on the task with the highest total execution time so it can be executed on the chosen RN.

$$S'_i = S_i + w_i \times I_i \tag{2}$$

The *unprioritized jobs* category refers to the collocated jobs (e.g. *StatefulSets* used for running databases) that need to share the same node resources. We consider the collocation execution time $C = [\overline{S'_1}, \overline{S'_2}, \ldots, \overline{S'_n}]$ and the collocation function P_C which depends on the Node Feasibility (F_n) and the average of total execution time for an unscheduled task (T_{exec}). Until now in Kubernetes version 1.12, the scheduler worked on the principle to stop looking for more feasible nodes once it finds a certain number of them, hence this action improves the scheduler's performance in large clusters. The number is specified as a percentage of the cluster size and it is controlled by a configuration parameter called the `percentageOfNodesToScore`. The range should be between 1 and 100. Larger values are considered as 100%, default value of this option is 50% and zero is considered without the configuration option. A new mechanism has been recently introduced in Kubernetes 1.14 which scores the percentage of nodes based on the cluster size when there is no specification in the configuration. The minimum score will be 5% of the cluster size if the user does not provide a configuration option smaller than 5% [21]. Therefore, we consider the optimal values in the interval [0.2 - 0.5]. In addition, we recall the average estimated task runtime (T_a) normalized by the average of total execution time for an unscheduled job ($\overline{S'_n} = \frac{Ta}{T_{exec}}$). In this manner we determine the collocation cases: if P_C =1, the collocation of resources is optimal and the result is sent to the SAs, if $P_c < \overline{S'_n}$ we have a sub-optimal scheduling, otherwise the SC employs the entire queue rescheduling (Equation 3):

$$P_C = \begin{cases} 1 - F_n, & \text{if } F_n \geq \overline{S'_n}. \\ F_n & F_n < \overline{S'_n}. \end{cases} \tag{3}$$

Since the proposed scheduling is a distributed model, the tasks wait-time and runtime are not always accurate and this may generate competing decisions between the scheduler agents. Moreover, multiple scheduling agents can allocate tasks to the same resource node at the same time which again might generate conflicts.

As in Omega [24] and Tarcil [7], we employ the *lock-free optimistic concurrency* to resolve the conflicts between SAs. This functions as follows: the system forwards a local copy of the master state to each of the scheduler agents. When the agent makes a job placement decision, it will update the MSA using an "`atomic write`" operation. If there is a likelihood for the job commit to be successful, the resources are allocated to the corresponding agent and the other competing agents will have to reschedule the job placement.

6 ANALYSIS OF OUR PROPOSED HYBRID SHARE-STATE SCHEDULER IN COMPARISON TO OTHER EXISTING SCHEDULERS INTEGRATED WITH KUBERNETES

Both Kubernetes [18] and Firmament [13] are centralized schedulers that demand complex algorithms for task scheduling and prioritization. In Section 4 we identified several problems with the Kubernetes Default Scheduler including node over/under-utilization and node failures that lead to cluster instability and loss of high-availability when multiple pod replicas are scheduled on the same node. To solve several of these issues, a Kubernetes Descheduler [20] has been developed and already integrated with K8S system, therefore we will consider it as a legitimate candidate in our analysis.

The Firmament scheduler is already integrated with Kubernetes under the Poseidon project [29] and runs the *min-cost max-flow* (MCMF) optimization algorithm. Even though its main concern is to solve the pod-by-pod scheduling and rescheduling by considering a globally optimal scheduling environment and real-time metrics, it cannot solve problems such as collocation interference avoidance.

Our new proposed scheduler is partially distributed as it incorporates the scheduling logic across the scheduling agents, but it also maintains an updated copy of the cluster state in the MSA and a central scheduler for the unscheduled jobs which makes it a hybrid scheduler. In addition to that, our approach preserves the Kubernetes *object pod* specifications in order to simplify the future integration with the Kubernetes architectural framework.

At code level, the Kubernetes scheduler creates a *Binding Object* function which makes the connection between the pod specifications and the nodes' utilization. Upon creation of a Binding Object, the Kubernetes API will update the Pod's `Spec.NodeName` [34] (see Figure 5). The `nodeSelector` is a field of `PodSpec` and it includes a map of the key-value pairs. In order for a pod to be considered eligible to run on a node, that particular node should have as labels each of the indicated key-value pairs.

Consequently, we will perform a comparison of the common features as well as the differences between the three scheduling candidates integrated with the Kubernetes scheduling framework (Descheduler, Firmament-Poseidon and the Hybrid shared-state proposed scheduler), see Table 2. We further analyze how our proposed scheduler intends to solve some of the encountered issues in our K8S deployed cluster by considering as a reference the Kubernetes Default Scheduler.

(A) **The node Affinity/Anti-Affinity** defines the manner in which nodes are selected by the scheduler using custom labels on the nodes and selectors on the pods. Normally, these rules must be met by a pod to be scheduled on a particular node and in case the node runs out of resources, the pod will not be scheduled. The rule applies by setting the parameter called "*requiredDuringSchedulingIgnoredDuringExecution*" of the `nodeAffinity` field.
   i. *Descheduler* – The node affinity/anti-affinity represents a strategy to remove from the nodes the pods which are not compliant with the node affinity. For instance, if Pod A initially scheduled on Node A, met the node affinity rule at the time of scheduling (admitting that parameter *requiredDuringSchedulingIgnoredDuringExecution* was initially set), in the likely event that over time another Node B becomes available and meets the node affinity rule requirement, then pod A will be evicted from Node A.
   ii. *Firmament-Poseidon scheduler* – Satisfies the node affinity/anti-affinity rules since it can co-exist and

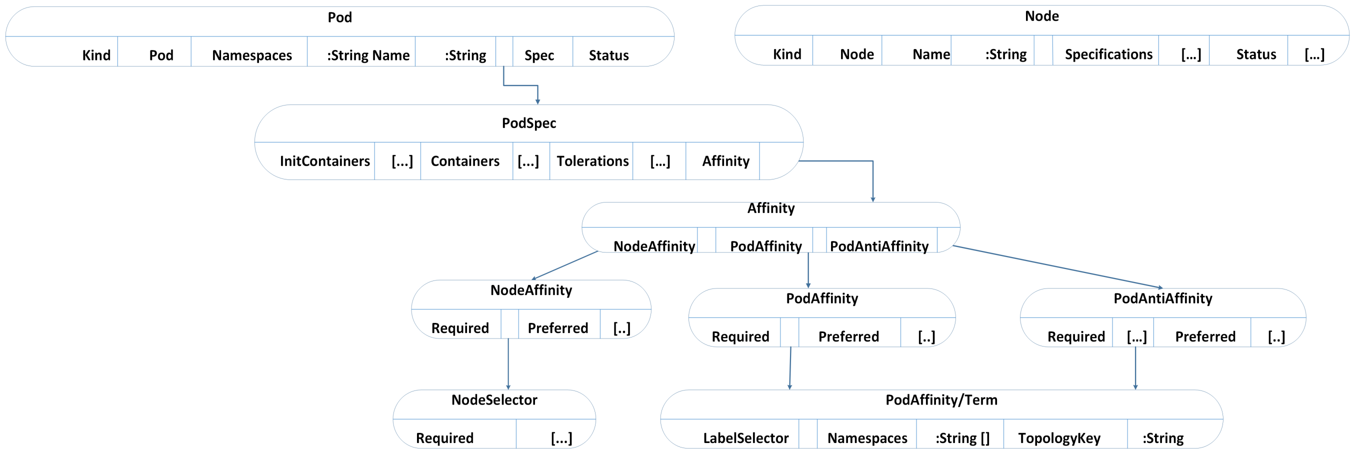


Figure 5: K8S Binding Object function - Relationship between Node Selector and Affinity Requirements, extracted from [34]

run as an alternate scheduler to the Kubernetes Default Scheduler by preserving the Kubernetes Binding Object function.

iii. *Hybrid shared-state proposed scheduler* – Includes support for the node affinity/anti-affinity as the scheduling decisions and job placements are processed by the distributed scheduling agents. This feature is particularly useful especially in situations when the K8S system components only need to be allocated to the master node, or when it creates a dedicated set of nodes to a particular group of users or assigns pods with special hardware requirements to the right resource nodes.

(B) **Inter-pod Affinity/Anti-affinity** rule represents a constraint against pod labels rather than node labels. For instance, a stateful pod which runs a high I/O database might have a storage backend requirement on a preferred node. Moreover, due to latency constraints, a set of pods can be ensured to run on the same node [18]. The Kubernetes Default Scheduler provides two types of pod affinity and anti-affinity corresponding to "hard" vs. "soft" requirements. There are currently two types of node affinity rules, named *requiredDuringSchedulingIgnoredDuringExecution* and *preferredDuringSchedulingIgnoredDuringExecution.* The first rule collocates pods running service A and service B in the same zone since they communicate to each other, whereas the latter represents a softer requirement - if the condition to run the pods in the same zone is not satisfied, it will stop making another node selection.

(C) **Taints and Tolerations** are the opposite of the node affinity as they allow a node to reject a set of pods. A pod may be bound to a node such that the node's taints match the pod's tolerations. A pod must not be bound to a node if the node's taints do not match the pod's tolerations [22].

i. *Descheduler* – Does not include support for pod inter-affinity nor taints and tolerations. Nevertheless, it has a strategy to remove pods violating inter-pod anti-affinity. This strategy makes sure that pods violating inter-pod anti-affinity are removed from the nodes. For example, if pod A, pod B and pod C are running on the same node and all three pods have anti-affinity rules which prohibit them to run on the same node, then pod A will be evicted from the node so that pod B and pod C can continue to run. This issue might happen if the anti-affinity rules for pods B and C are created when they were already running on the node. Currently, the Descheduler does not present any parameters associated with this strategy.

ii. *Firmament-Poseidon* – Includes support for the pod affinity/anti-affinity as well for node taints and tolerations. Despite all these, its performance in terms of pod affinity/anti-affinity match is relatively slow in comparison to the Default Scheduler [29].

iii. *Hybrid shared-state proposed scheduler* – Ensures this feature in a similar way to the Kubernetes Default Scheduler with the difference that the distributed scheduling agents process the inter-pod affinity/anti-affinity/taints and tolerations rules. The "*requiredDuringSchedulingIgnoredDuringExecution*" affinity groups pods of service A and B in the same zone, as they have an intensive communication with each other, while the field "*preferredDuringSchedulingIgnoredDuringExecution* for anti-affinity will ensure the pods are spread out from each other. The service collocation is performed in our proposal by the scheduling correction (SC) function whose role

Table 2: Features Comparison in Kubernetes Integrated Schedulers

Feature	Default Scheduler	Descheduler	Poseidon-Firmament	Proposed scheduler
Node Affinity/Anti-Affinity	y	y	y	y
Inter-pod Affinity/Anti-Affinity	y	partially	y	y
Taints/Tolerations	y	n	y	y
Baseline Scheduling/Optimal Scheduling	n	y	partially	y
Collocation Interference Avoidance	n	n	partially	y
High-availability	n	y	n	y
Priority Preemption	y	n	partially	y
Inherent Rescheduling	n	n	y	y

is to find a node alternative for the *preferredDuringSchedulingIgnoredDuringExecution* rule.

(D) **Baseline Scheduling/Optimal Scheduling** – In order to make scheduling decisions in accordance to the resource requirements, the scheduler uses predicates and priorities. Predicates represent hard constraints in the sense that if they are violated, the pod will not be run properly (e.g., the amount of memory requested by a pod). The role of a priority function is to score a relative value for the pod to run on a targeted node and prioritize the nodes where pods members of the same running service are not present. The priority function is also an indicator of the pod reliability as it reduces the chances that a node failure will disable all the containers of a particular service [4]. Kubernetes scheduler processes one pod a time. This might result in assigning tasks to sub-optimal machines that will inevitably lead to sub-optimal scheduling.

i. *Descheduler* – The under-utilization of nodes is determined by a configurable threshold. This threshold can be configured for CPU, memory, and number of pods in terms of percentage. For example, if a node is below 20% CPU utilisation and below 20% memory utilisation and it has less than 20 pods, it is considered under-utilised. If a node is above 50% CPU utilisation and memory utilisation or has more than 50 pods, it will be considered over-utilised [20].

ii. *Firmament-Poseidon* – Implementation of the Kubernetes scheduler's predicates is not fully supported. This scheduler employs bulk scheduling which processes all the unscheduled tasks at the same time including their soft and hard constraints. For each task the scheduling algorithm typically performs a feasibility check to identify suitable nodes, then scores them according to a preference order and finally places the task on the best-scoring node [13].

iii. The *Hybrid shared-state proposed scheduler* – When calculating the node feasibility, the scheduling agents iterate over the array of nodes in a round-robin fashion. For a given pod, the scheduling agents starts from the start of the array and checks feasibility of the nodes until it finds enough nodes as specified by the `percentageOfNodesToScore`. For the next pod, the scheduler continues from the point where the node array stopped at, while checking feasibility of nodes for the previous pod. The SC function employs this capability for the collocated services using the node Feasability score. If the collocation result is less than the average estimated task runtime normalized by the average of total execution time of the unscheduled job, the RNs will send an updated wait-time to the SC function so it can start running again the scheduling algorithm.

(E) **Collocation Interference Avoidance** occurs between pods competing for the same node resources. Stateful applications (e.g., databases) might have more requirements (i.e., CPU, memory, etc.) than stateless services, thus the scheduler needs to designate a specific set of nodes to run the database, whereas for batch jobs the workload resources requirements are uniform across Replicasets/Deployments/Jobs.

i. *Descheduler* – Current design does not solve the issue of pods conflicts.

ii. *Firmament-Poseidon* – offers an option in the form of a configurable API policy to write the scheduling policy that avoids task and pod collocation interference [29]. The collocation interference avoidance feature partially exists in Firmament-Poseidon, whereas the Kubernetes Default Scheduler provides extensive support [18].

iii. *Hybrid share-state proposed scheduler* – This issue is solved in our approach calling the SC function which assigns a weight to the interference time derived from the wait-time retrieved from the RNs for each of the tasks and the average estimated task runtime.

In this manner we differentiate the pod priorities by the calculated job execution time.

(F) **High-availability** – the service high-availability scenario mentioned in Section 3 where multiple pod replicas might be scheduled on the same node without taking into consideration service availability shows that the Kubernetes Default Scheduler does not have a mechanism adapted to the cluster dynamicity.
   i. *Descheduler* – The *Remove Duplicates* strategy ensures there is only one pod associated with a Replica Set, Deployment, or Job running on the same node. If there are more, those duplicate pods are evicted in order to better spread the pods in the cluster [20].
   ii. *Firmament-Poseidon* – currently this scheduling framework does not provide any strategy to ensure service availability and workload balance across the cluster in case of node failure or recovery.
   iii. *Hybrid share-state proposed scheduler* – When a node recovers, it will send a notification to the MSA which will forward an updated copy of the current cluster state to all the SAs so they will avoid scheduling service duplicates on the recovered node.

(G) **Priority preemption** – In case a pod cannot be scheduled, the scheduler tries to evict the lower priority pods in order to reschedule the available pods. The pods need to be created with `priorityClassName` so the scheduler orders the pending pods by their priorities in the scheduling queue. If none of the nodes satisfies the specified requirements of the pod, a priority-based preemption is triggered for that pod.
   i. *Descheduler* – It respects the following principle: *Best efforts* pods (with the lowest priority) are evicted before *Burstable* pods (that hold some form of minimal resource guarantee) and *Guaranteed* (top-priority) pods.
   ii. *Firmament-Poseidon* – A preemption cost is employed. Partially exists in Poseidon-Firmament [29] versus extensive support in Kubernetes Default Scheduler.
   iii. *Hybrid share-state proposed scheduler* – The pods with the lowest priority which have not been scheduled are evicted from the SAs queues and they will be send to the SC queue instead. In a StatefulSet we need to terminate completely it before its replacement. Thus, this job category is treated as unprioritized by the SC function. Therefore, based on its score, one of the following possibilities will apply: optimal scheduling, sub-optimal or rescheduling.

(H) **Inherent Rescheduling** – When a node is terminated, the pods previously running on that node will be rescheduled on the available nodes. If a new node is created with the exact resources configuration as the terminated node, the pods will not be rescheduled on this new node. Moreover, if one of the nodes where the pods have been scheduled happens to fail, the cluster will experience an outage [12].
   i. *Descheduler* – Will only evict the pod duplicates, it has no support for the inherent rescheduling.
   ii. The *Poseidon-Firmament scheduler* – Supports workload rescheduling as it functions based on the (MCMF) optimization and the scheduler looks at the entire cluster workload. This means that in each scheduling iteration it considers all the pods, hence it can migrate or evict pods from certain nodes without producing any outage in the cluster.
   iii. *Hybrid share-state proposed scheduler* – Since all the schedulers are whole time aware of the cluster state, this scenario is unlikely to happen. The SAs permanently synchronize their state with the MSA and an optimal scheduling is being run for each of the jobs according to the resources requirements.

7 CONCLUSIONS AND FUTURE WORK

This paper presents a novel scheduling framework for the Kubernetes system. The proposed hybrid-state scheduler provides scheduling corrections (SC function) for the processing of unscheduled and unprioritized jobs and assigns distributed agents (SAs) to optimize locally the main tasks, whereas the application-level scheduler (MSA) synchronizes the cluster state across all agents. This approach is meant to solve encountered problems like the behaviour of an average workload running in our deployed Kubernetes cluster under different scenarios, where we deployed 50 services in more than 75 pods. In order to classify and address the scheduling issues of other existing scheduling frameworks, we conducted a thorough survey of the main scheduling mechanisms along with their differences in Section 2.

Since a centralized model previously proved not to perform in all scenarios like resiliency and fault-tolerance scheduling, we made an in-depth analysis of our proposed hybrid shared-state scheduler capabilities. Our findings show that a shared-state scheduler integrated with the Kubernetes Default Scheduler will solve problems like collocation interference, priority preemption, high-availability or baseline scheduling that we encountered in our deployed Kubernetes cluster. In contrast to the other existing schedulers that have been integrated with Kubernetes system, our approach should perform an optimal scheduling and rescheduling due to the synchronized state of the cluster and the manner in which the SC function processes task execution time for each of the pods.

As a future work we propose a code implementation for our proposed hybrid shared-state scheduler and the integration with Kubernetes scheduler as well as testing our aforementioned discussed scenarios.

REFERENCES

[1] Apache. Hadoop On Demand 2007. Scheduler. Retrieved March 13, 2019 from https://hadoop.apache.org/docs/r1.2.1/hod$_s$cheduler.html

[2] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 285–300. http://dl.acm.org/citation.cfm?id=2685048.2685071

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. (2016), 50–57. https://doi.org/10.1145/2890784

[4] B. Burns and C. Tracey. 2018. *Managing Kubernetes: Operating Kubernetes Clusters in the Real World*. O'Reilly Media. https://books.google.ro/books?id=UG15DwAAQBAJ

[5] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah. 2017. A New Docker Swarm Scheduling Strategy. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*. 112–117. https://doi.org/10.1109/SC2.2017.24

[6] Oleg Chunikhin. 2018. Implementing Advanced Scheduling Techniques with Kubernetes. Retrieved March 13, 2019 from https://thenewstack.io/implementing-advanced-scheduling-techniques-with-kubernetes/

[7] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, New York, NY, USA, 97–110. https://doi.org/10.1145/2806777.2806779

[8] Docker 2019. What is a Container? Retrieved March 12, 2019 from https://www.docker.com/resources/what-container

[9] Dominik Ernst, David Bermbach, and Stefan Tai. 2016. Understanding the Container Ecosystem : A Taxonomy of Building Blocks for Container Lifecycle and Cluster Management.

[10] Etcd 2018. Etcd. Retrieved March 16, 2019 from https://etcd.io/

[11] Fluentd 2018. Fluentd Project. Retrieved March 16, 2019 from https://www.fluentd.org/

[12] GitHub Kubernetes 2017. Automatic pod rescheduling. Retrieved March 13, 2019 from https://github.com/kubernetes/kubernetes/issues/47965

[13] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 99–115. http://dl.acm.org/citation.cfm?id=3026877.3026886

[14] Armand Grillet. 2016. Comparison of Container Schedulers. Retrieved March 12, 2019 from https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421

[15] Hashicorp. Nomad Project 2007. Nomad. Retrieved March 13, 2019 from https://www.nomadproject.io/

[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[17] Toye Idowu. 2018. What is a Kubernetes Deployment? Deployment in K8S Explained. Retrieved March 16, 2019 from https://www.bmc.com/blogs/kubernetes-deployment/

[18] Kubernetes 2019. Kubernetes Documentation. Retrieved March 12, 2019 from https://kubernetes.io/docs/

[19] Kubernetes 2019. Unofficial Kubernetes. Retrieved March 16, 2019 from https://unofficial-kubernetes.readthedocs.io/en/latest/getting-started-guides/docker-multinode/

[20] Kubernetes Incubator 2019. Descheduler for Kubernetes. Retrieved March 12, 2019 from https://github.com/kubernetes-incubator/descheduler

[21] Kubernetes Repository 2018. Scheduler Performance Tuning. Retrieved March 13, 2019 from https://github.com/kubernetes/website/blob/master/content/en/docs/concepts/configuration/scheduler-perf-tuning.md

[22] Ian Lewis and David Oppenheimer. 2017. Advanced Scheduling in Kubernetes. Retrieved March 13, 2019 from https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/

[23] Sean Loiselle. 2018. Kubernetes: The State of Stateful Apps. Retrieved March 13, 2019 from https://www.cockroachlabs.com/blog/kubernetes-state-of-stateful-apps/

[24] Schwarzkopf Malte, Konwinski Andy, Abd-El-Malek Michael, and Wilkes John. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[25] Tarek Menouer and Christophe Cérin. 2017. Scheduling and Resource Management Allocation System Combined with an Economic Model. In *ISPA/IUCC*. IEEE, 807–813.

[26] Mesos Apache Org. 2018. The Apache Software Foundation. Apache Mesos. Retrieved March 12, 2019 from http://mesos.apache.org/

[27] Ngnix 2019. NGINX Ingress Controller for Kubernetes. Retrieved April 30, 2019 from https://github.com/kubernetes/ingress-nginx

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[29] Poseidon 2017. A Firmament-based Kubernetes scheduler. Retrieved March 13, 2019 from https://github.com/kubernetes-sigs/poseidon

[30] Malte Schwarzkopf. 2016. The evolution of cluster scheduler architectures. Retrieved March 13, 2019 from http://firmament.io/blog/scheduler-architectures.html#fig1e

[31] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*. ACM, New York, NY, USA, 1:1–1:13. https://doi.org/10.1145/2988336.2988337

[32] Swarm 2017. Swarm: a Docker-native clustering system. Retrieved March 12, 2019 from https://github.com/docker/swarm

[33] The Helm Project 2019. Heapster. Retrieved March 13, 2019 from https://github.com/helm/charts/tree/master/stable/heapster

[34] Dominik Tornow. 2018. The Kubernetes Scheduler. Retrieved March 13, 2019 from https://medium.com/@dominik.tornow/the-kubernetes-scheduler-cd429abac02f

[35] César Tron-Lozai. 2018. Keep your Kubernetes cluster balanced: the secret to High Availability. Retrieved March 13, 2019 from https://itnext.io/keep-you-kubernetes-cluster-balanced-the-secret-to-high-availability-17edf60d9cb7

[36] Vault Project 2018. Kubernetes Auth Method. Retrieved March 16, 2019 from https://www.vaultproject.io/docs/auth/kubernetes.html

[37] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, New York, NY, USA, 5:1–5:16. https://doi.org/10.1145/2523616.2523633