K8sES: Kubernetes with Enhanced Storage Service-Level Objectives

immediate

Abstract

Kubernetes (k8s) is a system for managing containerized applications across multiple hosts. It offers automatic deployment, maintenance, scaling, and resource management for applications. Applications in k8s usually have storage requirements in the form of service-level objectives (SLOs). However, the current k8s storage management is insufficient in many aspects. The k8s administrators have to manually configure storage in advance, and users must know the configurations and capabilities of the provided storage. Users' storage SLOs can be easily violated in k8s.

In this paper, we design and implement a system, called k8sES (k8s Enhanced Storage), that efficiently supports applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment. We design and incorporate storage scheduling as part of the node scheduling process in k8s. Applications will be scheduled onto correct nodes and storage without intervention from either users or administrators. Proper storage resources will be dynamically carved and allocated to applications based on users' storage SLOs. In addition, we provide a tool to monitor the I/O activities of both applications and storage devices in k8sES. The evaluation shows that k8sES can better meet users' storage SLOs along with other requirements. At the same time, k8sES can achieve higher resource utilization efficiency with overhead similar to that of the current k8s.

1 Introduction

The use of Linux containers [9, 21] has boomed as many in industry transition from traditional virtual machines (VMs) to this light-weight virtualization technology using containers. For example, Google runs all its software in containers [20]. Linux containers are a virtualization method for running multiple applications in isolated systems (i.e., containers) on a host using a single Linux kernel. Linux containers were initially released in 2008, but their use soared after Docker's release in 2013 [4, 34]. Docker is a platform for creating, deploying, and running applications inside containers. In order to deploy and manage applications in Docker containers across multiple hosts, people use a container orchestrator to perform cluster management and orchestration (i.e., selecting which cluster nodes will host which containers). Kubernetes (k8s) is one prevalent container orchestrator. In k8s, a pod is the basic management unit. It includes a container, or a group of tightly coupled containers, with shared storage and network resources and a specification for how its containers should run.

Due to the success and popularity of containers and Kubernetes, various applications running in versatile environment are moving to containers and Kubernetes. For example, applications in traditional data center, private cloud [10], public cloud [1, 6], edge/IoT [7], etc. In all these different environment, stateful applications that store, provide and process data are critical. When deploying stateful applications, users usually have service-level objectives (SLOs) on storage as part of their service-level agreement (SLA) requirements. In the current k8s, users can specify their requirements on CPU, memory, affinities to nodes in the cluster or other pods, etc., in the pod configuration file when deploying a pod. If users want to specify storage requirements, they can refer to a StorageClass(SC) which they think can meet their storage requirements. SC is used to describe the class of a storage. For example, an administrator may classify the storage resources into three classes: gold, silver and bronze. StorageClasses must be pre-created by administrators in the cluster.

The current storage management in k8s has some limitations. First, storage resources are excluded from the host selection process. The current k8s scheduling process only selects hosts without considering the storage resources that each host can access. It assumes the selected host has the connectivity to enough storage resources, which may not be the case in today's versatile application environment. Hosts in the k8s cluster may only have access to a limited number of local storage or shared storage. Second, SC is static and cannot be used to efficiently schedule storage resources, since the actual performance of the storage changes with the utilization of the storage resources. Third, it is hard to decide a proper number of SCs that just satisfies users' requirements without wasting resources. If the number of SCs is small, people have few options and may have to pick SCs that provide more resources than they want. As the number of SCs in a system increases, the storage utilization efficiency might improve due to more user options, but it requires more effort to maintain. Fourth, modern applications may have advanced storage requirements such as rate limiting and caching policies [26, 36]. However, SC does not support these advanced storage requirements. In addition, from the users' point view, it takes extra effort to select a correct SC. More importantly, the SC selection process is error-prone, and it may cause storage SLO violations or wasted storage resources.

Storage resource management has been a research problem in VM environment for years. For example, Pesto [27] provides storage management for VMs running on VMware's vSphere [18]. But the advancement in VM storage resource management cannot be easily applied to containers. First, when placing a VM disk into storage, VMware uses a profile to describe the storage requirements of a VM. Then, a proper storage compatible with the VM profile will be selected. This process is similar to allocating storage by using SC in k8s. Second, VM storage management systems like Pesto are built upon Hypervisors [23, 29]. In the container environment, orchestration platforms (e.g., Kubernetes, Mesos, and Docker swarm) perform the role of Hypervisors. Providing a VM storage management system to k8s must follow the interface of k8s, and work as a storage provisioner [8], which relies on SC to describe the storage allocated to users. Therefore, Pesto will share the same limitations of SC. Third, in Pesto, each host is connected to the centrally managed storage. A VM can always access to the allocated storage resources, regardless which host it is running on, so that Pesto does not select storage in the process of scheduling VMs. This premise imposes a strong limitation on the application environment. However, today's application environment with k8s goes far beyond the centrally managed storage configuration (e.g., Azure IoT containers that store data locally [15]). The traditional VM storage management system cannot treat such various application environment, and will face the same issue of missing storage selection in the host selection process of k8s.

In order to overcome these limitations in the current storage management of k8s, we propose k8sES (k8s Enhanced Storage), a system that can efficiently support applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment. K8sES allows users to put their detailed storage requirements, including capacity, bandwidth, sharability, advanced policies, etc., directly in their configuration files. As a result, no modifications are made to the universal interface (*kubectl create -f* $\langle manifest \rangle$), which is used to create pods. We redesign the current scheduling and storage allocation mechanisms in k8s, so that it can dynamically allocate storage to applications based on users' storage requirements. At initial storage allocation, k8sES will select appropriate hosts and storage and automatically carve out storage resources to support the running of pods, based on users' requirements and the performance of each host and storage at real time. During the runtime of applications, k8sES can monitor the performance of both pods and storage, and adjust the storage resource allocation based on the storage SLOs compliance.

In summary, our key contributions are:

- K8sES overcomes the limitations of k8s in storage management by providing dynamic storage provisioning to k8s for the purpose of meeting users' storage SLOs.
- K8sES improves the storage utilization efficiency in k8s.
- K8sES saves effort for both k8s administrators and users by improving the complex and error-prone storage allocation mechanisms in the current k8s.
- K8sES provides new monitoring capabilities to monitor the I/O performance of both pods and storage devices in k8s.

2 Background and Motivation

2.1 Kubernetes Components

A Kubernetes cluster is composed of master components and node components. Master components provide a control plane to the cluster including a front-end for the control plane (kube-apiserver), a backing store for all cluster data (etcd), a scheduler that schedules pods (kube-scheduler), and a controller manager that runs controllers to watch and ensure the cluster state (kube-controller-manager). Typically, master components run on the same machine. Node components run on all nodes in the cluster. On each node, there is a daemon responsible for creating pods (kubelet), a proxy that forwards TCP or UDP traffic to the pods (kube-proxy), and a container runtime (e.g., Docker). The container runtime provides basic container management functions to k8s, e.g., creating, starting, and stopping containers, managing container images, etc.

2.2 Storage Support in k8s

Data in containers are ephemeral. To allow persistent user data, Kubernetes provides a volume abstraction to store data permanently. A volume in k8s outlives any containers that run within the pod, and data is preserved across container restarts. Essentially, a k8s volume is a directory on the host and is mounted into containers of a pod. The medium that backs a volume is determined by the particular volume type being used, e.g., local, iSCSI, awsElasticBlockStore (Amazon Web Services EBS), gcePersistentDisk (Google Compute Engine Persistent Disk), etc.

K8s provides a PersistentVolume (PV) subsystem to manage how backend storage is provisioned and consumed. A PV is a portion of the cluster's storage that an administrator has provisioned [12]. A PV will generally have a specific storage capacity set when it is created. Currently, storage size is the only PV resource attribute that can be set or requested. To avoid exposing users to the details of how volumes are implemented, an administrator usually uses StorageClass (SC) to categorize PVs, e.g., gold, silver, and bronze classes. A PV can be manually provisioned or dynamically provisioned. To manually provision a PV, cluster administrators have to make calls to their storage provider to create storage volumes in advance and then create PV objects to represent them in k8s. In the manifest (i.e., configuration file) of a PV, the administrator will assign an SC name to indicate the StorageClass of that PV. To dynamically provision a PV, cluster administrators first have to create StorageClass objects in k8s. Inside the manifest of each SC object, the administrators must then specify a set of properties of the volume and a provisioner that is able to provision such volumes. Each SC must be unique in the cluster. Regardless how PVs are created, administrators must classify the storage resources in the cluster by using SC in advance.

With PVs, a k8s user does not use the pod manifest to specify what storage should back the volume. Instead, k8s users provision volumes using a PersistentVolumeClaim (PVC). Storage size is the only attribute that can be used in a PVC request, but a user can choose the class of storage by specifying an SC name. After a PVC is created, k8s will look for a matching pre-provisioned PV, or create one if dynamically provisioning PVs, and bind the PVC with it.

When scheduling a pod, k8s only selects a host for the pod based on CPU, memory, affinities requirements, etc. It does not consider the storage resources that each host can access. After a destination host is determined, the kubelet daemon on the selected host will assign a pre-created PV to the pod, or call the corresponding storage provisioner to create one. It assumes that each host has access to enough storage resources.

First, storage selection is excluded from the host selection process, which does not consider the storage resources that each host can access. During the scheduling process, k8s just assumes the selected host will have access to enough storage resources, which may not be the case. Hosts in the k8s cluster may only have access to a limited number of local storage or shared storage resources. The accessibility to different storage resources, and the available storage resources should be considered during the pod scheduling process.

Given these concepts, the process of deploying an application with persistent storage in the current Kubernetes environment is described as follows: (1) The administrator classify storage resources in the cluster by using SC. (2) The administrator creates PV objects (in manual provisioning) or SC objects (in dynamic provisioning). (3) The user creates a PVC. (4) The user creates one or multiple pods and refers to the previously created PVC. (5) The kube-scheduler selects hosts. (6) The kubelet daemons on the selected hosts assign storage.

2.3 Limitations of Storage Support in k8s

With support for storage as it currently exists in k8s, administrators can configure their cluster's storage into different categories and present them to users as multiple StorageClasses. However, this mechanism suffer from several limitations. First, storage selection is not considered when scheduling pods in kube-scheduler. In case that hosts only have access to a limited number of local storage or shared storage, it is probable that the storage resources a selected host can access cannot meet the user's storage requirements. Second, SC is static, but the performance of a storage system dynamically changes as the workload running on it constantly evolves. For example, consider a system containing multiple HDDs and SSDs where the administrator configures two SCs. One SC is called "fast" with SSDs as the backend, and the other SC is called "slow" with HDDs as the backend. If too many workloads are running in the SSDs and make them congested while the HDDs have no workload, the "fast" SC can be slower than the "slow" SC. To avoid this problem, administrators have to monitor the performance of all types of storage and dynamically adjust the physical configuration under each SC. This monitoring and adjustment takes significant effort from the administrators to meet users' storage SLOs. Second, limited SC choices or inappropriate configurations may waste storage resources to meet users' SLOs while large numbers of SCs are hard to maintain. When deploying applications, users have to map their storage requirements into an existing SC, which usually provides more resources than they need, e.g., more bandwidth, more space, or unnecessary functions (encryption, deduplication, etc.). Otherwise, users will be in danger of an SLO violation. If the number of SCs are few in a cluster, it has a high chance that a user may have to pick an SC providing resources much more than what he/she actually needs. If administrators want to perfectly meet each user's storage requirements, they may have to create an SC for each user with different storage requirements. In this case, it requires a large number of SCs which is hard to maintain. Third, StorageClass does not support advanced storage requirements. Some modern applications have advanced storage requirements expressed as policies [26, 36]. These policies can be dynamic, where an action is triggered if conditions are met. For example, a policy that requires cache when the GET operations per second are greater than 5 can be expressed as:

$$\text{policy: WHEN GETS/s} > 5, \text{SET CACHING} \tag{2.1}$$

With these advanced storage requirements, it is hard to pick an appropriate SC.

2.4 Scope and Objectives

This paper focuses on guaranteeing users' storage requirements in Kubernetes environment. Users' applications are deployed in containers in k8s. We assume the total CPU, memory, and storage resources in the k8s cluster are fixed, and administrators will not add infinite resources due to the high cost and maintenance effort.

K8sES is designed based on several objectives to ensure users' storage requirements are met. First, the storage where a pod is running should meet the storage SLOs of the user. At the same time, the node where the pod is running should meet other k8s resource related requirements of the user, including CPU, memory requirements, etc. Second, all k8s non-resource requirements, including port, affinities with pods, etc., should be met. Third, the pod scheduling and storage allocation decisions should reduce the possibility of SLO violations. Fourth, resources in the cluster should be used efficiently. Fifth, it should be easy for administrators to configure. Sixth, the proposed design of k8sES should keep the same interface as standard k8s. To these ends, k8sES can accommodate different storage SLOs.

In this paper, we focus on several essential storage requirements. K8sES enables users to specify capacity, sustained bandwidth, storage sharability, and advanced storage policies [26] as stated directly in their pod configuration file.

3 Architecture

Figure 1 shows the architecture of k8sES, which follows the master-slave mode of Kubernetes. In Kubernetes, a user puts the specifications of an object that he/she wants to create in a configuration file (manifest) and calls a universal interface *create -f* $\langle manifest \rangle$ to create the object. Since this interface is easy and succinct, there is no need of modifying the interface; rather, we add a new section in the manifest that allows users to directly specify their detailed storage requirements (e.g., Figure 2). When users of k8sES are not certain about which StorageClass to pick, they can simply put their detailed storage requirements in the manifest and use the same standard interface to deploy an application.

After receiving a scheduling request, the k8sES-scheduler (a modified kube-scheduler) is responsible for selecting a proper host that can meet the user's standard computation, memory, and non-resource requirements (affinity, node health, port, etc.) as well as a proper storage device (or system) that can meet the user's storage requirements. The k8sES-scheduler will restrict that the selected host has access to the selected storage. The scheduling decision is sent to the kubelet on the selected host to launch the pods. We extend the kubelet function so it can automatically allocate storage resources for the pods on the selected storage device. After receiving the scheduling decisions, the kubelet will pass parameters about the users' storage requirements to our k8sES driver so it can create volumes with the requested resources on the selected storage. Finally, the kubelet calls the underlying container runtime to launch containers and mount the storage to these containers. The Monitor module monitors all the running pods and storage. It collects I/O related data from each running pod and each storage device. The collected data are used for both pod and storage management. If there is any storage SLO violation, the Monitor module will call the Migrator to migrate pods and data. When deleting a pod, its allocated storage will be retained if users have set the reclaim policy to "Retain" in the new storage section of the pod configuration file. Users can reuse a volume by referring to the name of the pod that this volume previously belongs to, plus the name of the volume itself. If the reclaim policy is "Delete" or omitted, the allocated storage will be deleted together with the pod.

K8sES also contains a Discovery module to detect the available storage resources in the cluster. The joining and leaving of storage devices, as well as storage failures, can be automatically detected by the Discovery module. Both the Discovery and Monitor modules track the remaining storage resources in the system. The k8sES-scheduler queries these two modules to determine the currently available resources on each storage device.

K8sES has high scalability. To avoid single-node limitations, both the k8sES master and worker nodes can be horizontally scaled in the same way as standard k8s [3, 14].

3.1 Selecting Storage Along with Nodes

The current k8s process of scheduling a pod on a node undergoes ***predicate***, ***priority***, and ***select*** steps. In the *predicate* step, the scheduler filters out nodes that cannot meet all the predefined predicates (i.e., quick yes/no rules). These predicates will check whether those requirements with definite numbers and those explicitly specified as "RequiredDuringScheduling" in the pod configuration files are met. In the *priority* step, the scheduler checks the priority of each node by scoring it based on a list of priority rules. Each priority rule has a weight and calculates a score from 0 to 10 for each node. The weighted summation of the scores from all priority rules is the final score of a node. Finally, the scheduler selects the node with the highest score and sends the decision to kube-apiserver. The kubelet on the selected node will launch the pod. If there are multiple nodes with the same highest score, the scheduler will select one in a round robin fashion.

In k8sES-scheduler, we select both storage and nodes for a pod. As storage and nodes are different resources, how to coordinate the scheduling to meet users' various requirements is an issue. Intuitively, we may select nodes first. In this case, however, the storage belonging to the selected nodes may not meet the storage requirements. Alternatively, we may select storage first. In this case, the selected storage may not belong to any eligible nodes. In k8sES, we first define users' storage SLOs (capacity, bandwidth, sharing, policies, etc.) as predicates. All other requirements used to reduce the possibility of SLO violations and optimize cloud resource efficiency are defined as storage priority rules (discussed in Sec. 3.2). In the *predicate* step, the scheduler will first filter a list of node candidates which can meet all node related requirements. Then, we check the storage predicates. We only check the storage that can be accessed by the filtered nodes. In the *priority* step, we calculate scores for each filtered node and storage device

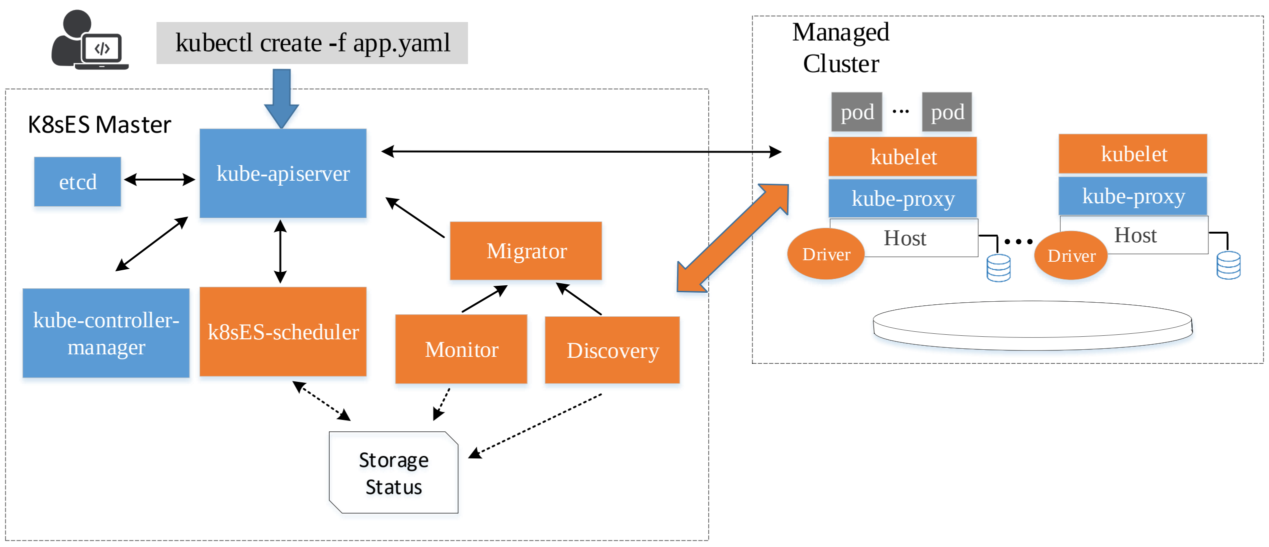


Figure 1: System architecture of k8sES.

```
<k8sES volume>
   size: x GB
   sustained bw: y MB/s
   sharing: False
   reclaim: Retain
   policy: WHEN GETS/s > z, SET CACHING
```

Figure 2: Example storage requirements in k8sES.

based on the node priority rules and storage priority rules. In the final *select* step, one option to select node and storage is selecting a node with the highest score calculated by the node priority rules and then picking storage with the highest score accessible by this node. However, this method downgrades the importance of storage priority rules. It is possible that all the storage devices that the selected node can access have very low scores (e.g., the available storage space and bandwidth resources are exactly the same as the requested resources). K8sES selection considers both node and storage priority rules fairly. After calculating the score of a node and the scores of its accessible storage, we pick the storage with the highest score and add its score to the score of the node. The final decision selects the node that has the highest combined score and then selects the storage with the highest score on that node. The decision tuple $\langle Node, Storage\rangle$ is then sent to the kubelet on the selected node where it will launch the pod on *Node* and create the volume on *Storage*.

3.2 Priority Rule

In k8sES, the *predicate* process ensures the storage SLOs can be met at initial allocation. However, as more applications are launched, the storage SLOs of existing pods may still be violated due to bad allocation from other sources. K8sES considers this possibility and gives preference to storage that has a lower possibility of future SLO violations. K8sES sets a *least_storage_usage* priority rule in the *priority* process that assigns a higher score to a storage device if its space and bandwidth usage would still be low after assigning the pod to it. In practice, the score (maximum is 10) of a storage device for this priority rule can be calculated as:

$$\begin{aligned} Score =& (10 \times \frac{Size_{total} - Size_{requested}}{Size_{total}} \\ &+ 10 \times \frac{Bandwidth_{total} - Bandwidth_{requested}}{Bandwidth_{total}})/2 \end{aligned}$$

where $Size_{total}$ is the total size of a storage device, $Size_{requested}$ is the requested storage size of all existing pods using that device plus the pod to be launched, $Bandwidth_{total}$ is the total bandwidth of the storage device, and $Bandwidth_{requested}$ is the requested storage bandwidth of all existing pods using that device plus the pod to be launched.

Moreover, storage and other resources in the cluster need to be balanced. Otherwise, an unbalanced utilization may waste resources. For example, assume there is a cluster of two nodes that each have local storage. Each node has 64GB of memory and 1TB of storage, and we only care about memory and storage space resources. Due to system and user activities, the current available resources are 38GB of memory and 900GB of storage in Node 1 and 58GB of memory and 400GB of storage in Node 2. Now assume a pod requires 32GB of memory and 100GB of storage. The scores given by the *least_storage_usage* rule on memory and storage are summarized in Table 1. As discussed in the previous section, we select Node 1 as it has a higher combined score. This pod will be launched on Node 1 and its local storage. After this allocation, only 6GB of memory is left while 800GB of storage is unused on Node 1. If all incoming pods require more than

Table 1: Example scores of nodes with different resources

Nodes	Memory	Storage Space	Combined
Node 1	1	8	9
Node 2	4	3	7

6GB of memory, the remaining storage resources on Node 1 will be wasted. Therefore, we set a *usage_leveling* priority rule which assigns a higher score to a storage device if CPU, memory, storage space, and storage bandwidth usages will be more balanced after assigning the pod to it. In practice, the score of a storage device under this priority rule can be calculated as:

$$Score = 10 - 10 \times \left| \frac{CPU_{requested}}{CPU_{total}} + \frac{Memory_{requested}}{Memory_{total}} - \frac{Size_{requested}}{Size_{total}} - \frac{Bandwidth_{requested}}{Bandwidth_{total}} \right|$$

Each priority rule is also associated with a weight (0 to 1). Administrators of k8sES can adjust the importance of each priority rule by adjusting the weights. When adding the storage score to the score of a node, we also have weights allowing administrators to adjust the importance of storage resources and non-storage resources. By setting these weights during the startup of k8sES, people can achieve different tradeoffs between storage resources, node resources, resource usage efficiency, risk of SLO violations, etc.

3.3 Discovery

Unlike k8s, k8sES does not require administrators to create PersistentVolume or StorageClass resources to describe and categorize the storage capabilities in the cluster. We include a Discovery module to detect all available storage resources in the cluster automatically. First, when a node joins the cluster, it must register with the Discovery module to report its available storage resources including Name, Location, Size, Bandwidth, and Shareability. Whenever storage is added to or removed from a registered node, the node also needs to report the changes to the Discovery module. Expanding on the node health check capabilities in the current k8s, the Discovery module allows k8sES to also detect storage failures. Each node periodically sends heartbeat messages to the Discovery module. The module maintains a liveness map of each storage device and is quickly able to recognize storage failures. If there is a node or storage failure, the Discovery module will call the Migrator module to start the failover process.

3.4 Monitoring

The Monitor module in k8sES collects the I/O performance of both pods and storage devices. It collects the read and write I/O throughput of each pod in real time. It also monitors the collective I/O throughput and space utilization of each storage device. The collected data are used in four ways.

First, the collected data are used to identify any misbehaving pods. In case that a pod takes more I/O resources than it requested, the Monitor may throttle the I/Os, or it may log and report the event to the administrator for later auditing. The actions to be taken are determined during the initialization of k8sES.

Second, the collected data are used to enforce dynamic policies when their conditions are met. The Monitor module analyzes the I/O performance related statistics, e.g., read/write IOPS (I/Os per second) and read/write throughput. Once the metric defined in the dynamic policy of a pod meets the condition, the Monitor will call the corresponding functions or tools to take actions. For example, assume a pod defines a dynamic policy (2.1). Once the read IOPS on the storage allocated to that pod is greater than 5 for a significant amount of time (e.g., one minute), the Monitor will set up a dedicated cache for the pod. Note that such a cache is more useful for pods accessing remotely shared storage. For pods accessing local storage, such a policy may be ignored due to the existence of the file system cache. Currently, the functions in k8sES include caching and I/O throttling. In the future, we may support more metrics by collecting richer information in the cluster and support additional I/O related functions.

Third, the collected data are used to adjust the storage resource allocation based on the actual storage resource utilization. These adjustments are discussed in Sec. 3.5.

Fourth, the collected data are used to determine possible storage SLO violations of pods. In case of SLO violation, it will call the Migrator module to do migration. The details are discussed in Sec. 3.6.

3.5 Thin Provisioning, and Multiplexing

The Monitor module analyzes storage utilization related statistics, e.g., disk space used by each pod and read/write throughput of the storage on each host. It maintains two tables: one about the space usage at the pod level and one with bandwidth usage at the storage level. For each pod i running in the cluster, it records the current disk space used on the file system, S^i_{used}, and the requested storage space, S^i_{req}. After the scheduling, the kubelet on the selected node will not fully provision the volume with the requested space. Instead, it will only provision a volume with size $\rho \cdot S^i_{\text{req}}$ where $(0 < \rho \leq 1)$. ρ controls the initial storage space allocation. As time goes on, once the file system usage reaches a threshold θ, the Monitor module will issue a command to the host to expand the current storage space allocation by $\mu \cdot S^i_{\text{req}}$ where $(0 < \mu \leq 1)$. μ controls the speed of space allocation increase. This method of storage space allocation is called "thin provisioning." Thin provisioning is reasonable because users typically request more storage space than they actually use. Thus, thin provisioning can further save storage space resources of the cluster. There are still tradeoffs, however. Considering that increasing storage space of a volume while it is being used may influence the ongoing I/Os, a big ρ and μ will ensure more steady I/O performance but waste more storage space. A small ρ and μ can save more storage space but may hurt the I/O performance. The configurations of ρ and μ are at administrators' discretion and can be adjusted in k8sES.

For each storage device j detected by the Discovery module, the Monitor module records its throughput (TP in MB/s) at time t as TP_t^j, its total requested bandwidth as B_{req}^j, and its literal (overall maximum) total bandwidth as B_{total}^j. The Monitor module calculates the average throughput, $\overline{TP^j}$, over a time interval τ (e.g., six hours) from the TP_t^j measurements for each storage device. With these parameters, the Monitor module calculates the bandwidth utilization as:

$$\frac{1}{\alpha^j} = \frac{\overline{TP^j}}{B_{\text{req}}^j}$$

We call α^j the amplification factor of the bandwidth of storage device j. It indicates that we can allocate pods to storage device j as if it has a total bandwidth of $B_{\text{amlified}}^j = \alpha^j \cdot B_{\text{total}}^j$ without violating pods' storage bandwidth requirements. We use B_{amlified}^j to update the available bandwidth for storage device j, on which the scheduling process is based. Conceptually similar to thin provisioning, we call this scheme of storage bandwidth allocation "multiplexing." Given that users typically request more storage bandwidth than they actually use, this form of statistical multiplexing will further save storage bandwidth resources of the cluster.

Since α^j keeps changing as $\overline{TP^j}$ changes, we will calculate a new B_{amlified}^j only when the difference between the newly calculated amplification factor α^j and the one currently being used to calculate B_{amlified}^j is greater than a threshold γ (e.g., $\pm 10\%$). A large γ will lead to late updates of B_{amlified}^j and further cause either resource waste (due to decreased bandwidth utilization) or SLO violations (due to increased bandwidth utilization). A small γ will lead to frequent updates of B_{amlified}^j and a waste of computation power. In practice, we set γ to be 10%. In addition, it is possible that at some earlier time the bandwidth utilization of a storage device is very low, and a lot of pods are allocated to this storage. This results in a high ratio between B_{req}^j and B_{total}^j. Once a pod's throughput resumes to its requested bandwidth, there is a high chance that a lot of pods' storage SLOs will be violated. To reduce the chance of such SLO violations, we set a cap for α^j to be no more than 120%.

3.6 Migrator

Both thin provisioning and multiplexing improve the storage utilization efficiency but bring potential storage space or bandwidth SLO violations. To meet users' storage SLOs with high storage utilization efficiency, we design a Migrator to migrate pods along with storage. If the total space utilization of a storage device reaches a threshold (e.g. 90%), it will trigger the migration. This is because there is a chance that the next storage space expansion of a pod may fail due to insufficient storage space. The migration will also be triggered if TP_t^j equals B_{total}^j, which means the current stable throughput on storage device j has reached the literal maximum. In other cases like node and storage failure, pods and storage will also be migrated.

If the migration is triggered by either thin provisioning or multiplexing, the Migrator has to migrate the storage of one or more pods. When selecting candidates to migrate, several factors need to be considered. First, migrating the storage of a pod will freeze the pod's I/O for a period. The bigger the storage space allocated, the longer it takes to migrate. Second, migrating a pod with bigger storage will release more storage space. Third, migrating a pod with a higher throughput will release more bandwidth resources. Fourth, some pods have pod affinities that require them to stay on the same node. Our goal of migration is to reduce the down time of a pod and reduce future migration. Our migration candidate selection algorithm works as follows.

(1) If the migration is triggered by thin provisioning, migrate the pod(s) with the smallest storage space allocated. No matter which pod is migrated, some storage space will be released. Because migration is done early at the 90% threshold, no storage space SLO has been violated yet and the migration scheme can conservatively release storage space to reduce migration overhead. For a pod with pod affinity, we consider all those affinities as a group. We sum the allocated storage space of that group and compare it with other candidates. If the group is selected to migrate, we migrate the whole group.

(2) If the migration is triggered by multiplexing, we sort pods based on space allocated and current throughput, respectively. We assign scores based on the rank. A smaller allocated storage space has a higher score, and a higher throughput has a higher score. Then, these two scores are summed and the pod with the highest sum will be migrated. For a pod which has pod affinity, we will treat all those affinities as a group.

The migration destination is determined by the kube-scheduler as a new scheduling process except that the originally selected storage is excluded.

4 Implementation

We implement k8sES based on Kubernetes Release-1.7. The current k8s implementation is decoupled into multiple binaries. Each module of k8s is an individual binary as are the new/enhanced k8sES modules.

Our implementation of the Discovery module and Monitor module in k8sES uses master/slave mode. Each k8sES node runs a Discovery slave and a Monitor slave daemon. The slaves collect the performance measurements for each pod and the storage information on each node, and they send the data to the Discovery master and Monitor master. If the masters make some decisions, they call the corresponding module to carry out the operations. In the current implementation, the Discovery slaves run *lsblk* to get the storage capabilities. The Monitor slaves run *iotop* on each node to monitor the I/O performance of each pod and run *dstat* to monitor the

I/O performance of each storage device. People may extend the functions of k8sES by calling other tools. The Discovery master and Monitor master run in the same node as the k8sES master. They maintain a data structure recording the storage configuration map of all nodes in the cluster. This data structure is shared with the k8sES-scheduler module. In addition, the Monitor master also maintains a map of all running pods containing the current I/O performance and the dynamic policies of each pod. It will call the corresponding slaves to execute the actions defined in the policies if the conditions are met. Since the Monitor master is an individual binary, people can easily extend the support of storage policies without affecting other functionalities of k8sES. The Discovery master will call the Migrator module if any node or storage is inaccessible. The Monitor master will call the Migrator module if there are storage SLO violations.

The k8sES-scheduler receives the storage configuration map from the Discovery and Monitor modules. It maintains the map by recording the currently available resources of each storage device. We implement our k8sESdriver as a Flexvolume plugin. Flexvolume lets users write their own drivers to make Kubernetes support their volumes [5]. Once called by the kubelet, the k8sESdriver will first call the interface of the selected storage device or system to create volumes with the required size. For example, if LVM is the backend storage and storage groups are presented, the volume driver will create logical volumes to be used by the pod. It then optionally creates a file system (if required in the pod configuration file) and mounts the volume as a k8s volume. If there are sustained bandwidth requirements that need to be met, it will also set cgroups [33] to limit the bandwidth of the pod accessing this volume. After the volume driver mounts the created volume to the mount point defined in the pod configuration file, the kubelet then starts the containers defined in the pod.

5 Prototype Evaluation and Comparison

This section evaluates a prototype of k8sES. Sec. 5.2 validates the effectiveness of k8sES in meeting users' storage SLOs. We compare the scheduling results of k8sES with standard k8s. Sec. 5.3 shows the effectiveness of I/O throttling in k8sES and its benefits when sharing storage. We show the I/O monitoring abilities, storage resource savings, and migration capabilities of k8sES in Sec 5.4. Sec. 5.5 tests the storage usage efficiency under different circumstances. We discuss the overhead of k8sES compared with k8s in Sec. 5.6.

5.1 Experiment Setup

In order to control the CPU, memory and storage resources allocated to each node with flexibility, we use virtual machines (VMs) as nodes in our experimental k8s/k8sES cluster. Our k8s cluster consists of 41 virtual machines (VMs) running on 11 physical servers. Each server has two six-core Intel Xeon 2.40GHz E5-2620 v3 CPUs, 64GB of memory, 1TB Seagate ST1000NM0033-9ZM173 SATA hard disk, and is connected to an HP ProCurve 5406zl switch through a 1Gb/s Broadcom NetXtreme BCM5720 NIC port. Among these VMs, 40 run as k8s workers, and one runs as k8s master. One server hosts the master and the other servers each host 4 workers. All physical servers and VMs are installed Ubuntu 18.04. For each worker node, we allocate both local and shared storage resources. The storage capabilities of each worker are configured according to different experiments.

Table 2: The configuration of workers used for validation

	Worker 1	Worker 2
CPU	3	3
Memory	3 GB	2 GB
Local Storage Size	70 GB	50 GB
Local Storage Bandwidth	20 MB/s	50 MB/s
Shared Storage	Share same 100 GB at 50 MB/s	

5.2 Validation

We first verify scenarios where k8sES is able to schedule pods to correct nodes and storage to meet the pods' various requirements compared with k8s. As the scale of the cluster does not affect the correctness of the scheduling, we restrain the pod scheduling onto two workers for simplicity. The configuration of these 2 workers are listed in Table 3a.

5.2.1 Storage capacity

In the first scenario, we focus on the storage capacity requirements. Assume we have 4 applications sequentially deployed with pods A1, A2, A3, and B. Pods A1, A2, and A3 each require 5GB of non-shared storage and 2MB/s of storage bandwidth ($\langle 5GB, 2MB/s, \text{non-sharing}\rangle$). Pod B requires $\langle 50GB, 10MB/s, \text{non-sharing}\rangle$ storage. They do not have other resource requirements. We deploy A1, A2, and A3 first and check the scheduling result of pod B. The scheduling results of these four pods in k8s and k8sES are listed in Table 3a and 3b, respectively. In the tables, the remaining storage capacity and remaining storage bandwidth of the workers are given before Pod B is deployed. Because the current k8s scheduler does not consider storage resources, it will always pass the predicate step and generate equal priority scores in the priority step when deploying these four pods. Then, k8s schedules these pods in a round robin fashion, and schedules Pod B on Worker 2. However, Worker 2 only has 45GB of storage capacity available before scheduling B. The storage capacity requirement of Pod B will be violated with such scheduling decision. In contrast, k8sES-scheduler considers the storage resources. Although both workers can pass the predicate step of k8sES-scheduler, the priority step favors Worker 2 when scheduling pods A1, A2, and A3 as it has more balanced capacity and bandwidth resources. When deploying Pod B, Worker 2 will fail the predicate check with not enough storage capacity available. Pod B will be deployed on Worker 1, and the storage requirements can be met.

5.2.2 Storage bandwidth

The second scenario verifies that k8sES can correctly schedule pods with storage bandwidth requirements. Assume we

Table 3: Comparing meeting storage capacity requirement

Workers	Capacity (before B)	Bandwidth (before B)	Result
Worker 1	60GB	16MB/s	A1,A3
Worker 2	45GB	48MB/s	A2,B (✗)

(a) k8s scheduling result

Workers	Capacity (before B)	Bandwidth (before B)	Result
Worker 1	70GB	20MB/s	B
Worker 2	35GB	44MB/s	A1,A2,A3

(b) k8sES scheduling result

Table 4: Comparing meeting storage bandwidth requirement

Workers	Capacity (before D)	Bandwidth (before D)	Result
Worker 1	60GB	0	C1,D (✗)
Worker 2	40GB	30MB/s	C2

(a) k8s scheduling result

Workers	Capacity (before D)	Bandwidth (before D)	Result
Worker 1	60GB	0	C2
Worker 2	40GB	30MB/s	C1,D

(b) k8sES scheduling result

have applications sequentially deployed with pods C1, C2, and D. They each require $\langle 10GB, 20MB/s, \text{non-sharing} \rangle$ storage. No other resources are required. Tables 4a and 4b show the available storage resources before scheduling pod D and the scheduling results in k8s and k8sES, respectively. Similar to the results in Table 3, the scheduling decision made by k8sES is able to meet the storage requirements of all pods while k8s causes a violation of the bandwidth requirement of Pod D.

5.2.3 Storage with other resources

In the third case, we verify that k8sES is able to meet storage requirements along with other requirements. Assume we have applications sequentially deployed with pods E1, E2, and F. They each require $\langle 10GB, 20MB/s, \text{non-sharing} \rangle$ storage. In addition, each pod requires one CPU core and 1GB of memory. Tables 5a and 5b show the available CPU, memory, and storage resources before scheduling Pod F and the scheduling results in k8s and k8sES, respectively. The schedule decisions of k8s are made solely based on the CPU and memory resources. After deploying Pod E1 and E2, it schedules Pod F on Worker 1 as it has more balanced CPU and memory resources based on the internal *BalancedResourceAllocation* priority rule. However, Worker 1 has no more allocable storage bandwidth and thus violates the storage bandwidth requirement of Pod F. In contrast, k8sES-scheduler considers various requirements and is able to pick Worker 2, which can meet the CPU, memory, and storage requirements of Pod F.

These are just three possible scenarios where k8sES meets SLOs that k8s cannot. Since users may have storage requirements with any values, SLO violations in k8s may be very common without k8sES.

5.3 I/O Throttling

In k8sES, the administrator can configure the Monitor to throttle I/Os or just report the events for future audit when a pod consumes more storage bandwidth than it requested. In this experiment, we show the effectiveness of k8sES in throttling I/Os and preventing interference between different applications. We run containerized Nginx [?] (a popular web server, version 1.15.9), and OpenStack Swift [?] (a popular object storage, version 2.20.0) in k8s and k8sES.

We first show k8sES is able to throttle I/Os according to users' requirements. We set up an HTTP client to download files from the Nginx server. Without any limitation, it will try to download files and generate I/Os on Nginx as fast as possible. We set the Nginx pod to require local storage and sustained storage bandwidth of 40MB/s and 30MB/s in two tests. Figure 3 shows the I/O throughput of Nginx when being scheduled on Worker 2 in Sec. 5.2. We can see in k8s, the actual I/O throughput of Nginx is bounded to 50MB/s, the maximum bandwidth of the local storage in Worker 2. When running in k8sES, the actual throughput is throttled successfully as requested.

The I/O throttling capabilities of k8sES can prevent not only excessive I/O resource consumption, but also interference from misbehaved applications when multiple applications use shared storage. In this test, we deploy Nginx and OpenStack Swift in the same shared storage, which has 100GB size and 50MB/s I/O bandwidth. We use the default benchmark tool of OpenStack Swift, *ssbench* [?], to generate storage requests. It performs CREATE, READ, WRITE, DELETE of an object based on a configuration file called scenario. In the scenario, it mostly generates READ requests with 20 workers issuing requests concurrently. At the same time, we use the HTTP client to download files from Nginx. When deploying, we set Swift to require 10MB/s bandwidth and Nginx to require 40MB/s bandwidth. Ideally, Nginx should see a throughput equal to its requested bandwidth, as the HTTP client tries to download files as fast as possible. But in Figure 4(a), we can see that the Nginx is often running below the requested bandwidth in k8s, and sometimes can only deliver files at the speed of 3/4 of the requested bandwidth. Its throughput is fluctuating due to unstable performance of Swift. When Swift tries to generates I/Os more than its requested bandwidth (misbehaved), the storage SLO of Nginx is violated. In comparison, with k8sES (Figure 4(b)), the I/O throughput of Swift never exceeds its requested bandwidth, and thus the Nginx can always deliver data at a speed that matches the client requirement.

Table 5: Comparing meeting CPU+Memory+Storage requirement

Workers	CPU (before F)	Memory (before F)	Remaining Capacity (before F)	Remaining Bandwidth (before F)	k8s Scheduling Results
Worker 1	2	2	60GB	0	E1,F (✗)
Worker 2	2	1	40GB	30MB/s	E2

(a) k8s scheduling result

Workers	CPU (before F)	Memory (before F)	Remaining Capacity (before F)	Remaining Bandwidth (before F)	k8s Scheduling Results
Worker 1	2	2	60GB	0	E1
Worker 2	2	1	40GB	30MB/s	E2,F

(b) k8sES scheduling result

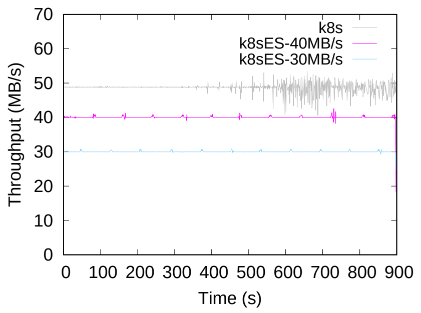
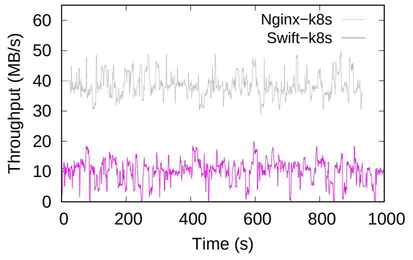
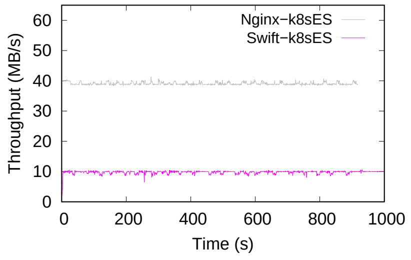


Figure 3: I/O throttling in k8s and k8sES.



(a) k8s



(b) k8sES

Figure 4: The effects of misbehaved applications on well-behaved applications.

5.4 Monitoring, Thin Provisioning, Multiplexing, and Migration

As monitoring capabilities are essential for ensuring storage SLOs and enhancing storage utilization efficiency, we perform an experiment to show these capabilities at both pod and storage granularities. Through this experiment, we also show the effects of thin provisioning, multiplexing, and pod migration, which rely on the monitoring capabilities. In this experiment, we mainly observe the I/O activities of three simulated applications in three pods, namely Pod A, Pod B, and Pod C. Pod A requires $\langle 10GB, 50MB/s, \text{non-sharing}\rangle$ storage. Pod B requires $\langle 2GB, 40MB/s, \text{non-sharing}\rangle$ storage. Pod C requires $\langle 5GB, 30MB/s, \text{non-sharing}\rangle$ storage. Each pod will generate I/Os bounded by its requested bandwidth for 40 minutes. The I/Os follow four normal distributions, each lasting ten minutes. In the first ten minutes, the average throughput of each pod equals half of its requested storage bandwidth. In the third ten minutes, the throughput of each pod will reach its maximum (the requested bandwidth). During the other times (10-20 and 30-40 minutes), the distribution has random mean and standard deviation.

At the beginning of the experiment, Pod A, Pod B, and Pod C were deployed on the same node (Worker 3) which has a $\langle 20GB, 100MB/s\rangle$ local storage based on the status of the cluster of that time. We set the time interval τ to be five minutes, which is the length of time used to calculate the average I/O throughput, $\overline{TP}$, of a storage device. Pod B and Pod C were deployed after Pod A had run for five minutes.

Figure 5 shows the I/O throughput of each application. Figure 6(a) shows the I/O throughput on the local storage of Worker 3. Note that the sum of the requested bandwidth of the three pods is 120MB/s, which is greater than the 100MB/s literal storage bandwidth of Worker 3. This is because the storage bandwidth utilization of Worker 3 is only half of the requested bandwidth at the scheduling of Pod B and C, as Pod A only generates I/Os at half of its requested bandwidth during the first 10 minutes. It results in the amplification factor $\alpha = 2$. Due to the 120% cap of α, the thin provisioning and multiplexing mechanism enables allocation of pods to Worker 3 as if it has a storage bandwidth of 120MB/s, which is 120% of its literal bandwidth. We can see that the I/O throughput on Worker 3 never reaches its literal maximum until 20 minutes after the startup of Pod B. At that point, circled in Figure 5 and Figure 6(a), the I/O throughput of each pod starts to fully reach its requested bandwidth. When all pods reach their maximums, the I/O throughput on Worker 3 exceeds its literal maximum (We deliberately spare more storage bandwidth for Work 3 over the literal 100MB/s to allow for better figure presentation). This triggers the migration process. Because Pod B has the highest migration score, the Migrator decides to migrate Pod B to another worker, namely Worker 4, which has a $\langle 50GB, 50MB/s\rangle$ local storage. The circled region in Figure 5 also shows the migration process. The I/Os of Pod B first drop to zero and then resume to its maximum after restarting on

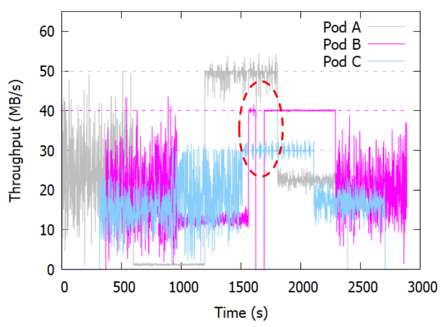
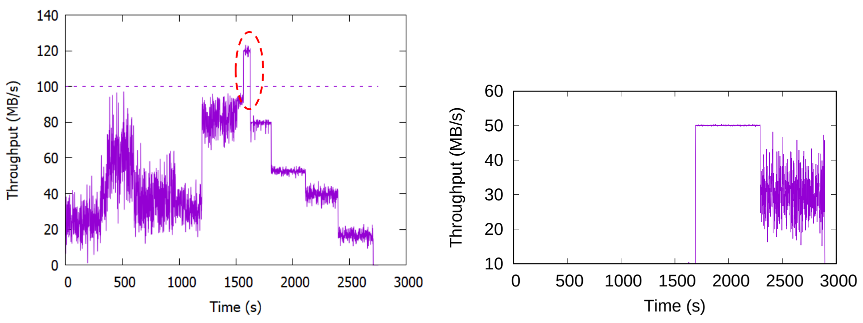


Figure 5: Throughput of applications over their lifetime.



(a) Worker 3

(b) Worker 4

Figure 6: Monitored I/O throughput on Worker 3 and 4.

Table 6: The storage configuration of workers on each server used for resource usage experiment.

	Storage Size	Storage Bandwidth
Worker 1	4000 GB	100 MB/s
Worker 2	500 GB	200 MB/s
Worker 3	500 GB	200 MB/s
Worker 4	250 GB	2000 MB/s

Worker 4. After the migration, the I/O throughput on Worker 3 immediately drops below its literal bandwidth, while the I/O throughput on Worker 4 (shown in Figure 6(b)) increases due to I/Os from Pod B.

5.5 Resource Usage Efficiency

In this experiment, we compare the storage resource utilization of k8sES with the current storage allocation mechanism in k8s. On each server, we allocate 1 NVMe SSD, 2 HDD and 1 SMR to the four workers. We configure the storage capabilities of the four workers on each server as in Table 6.

To make a fair comparison, we turn off the thin provisioning and multiplexing functionalities. For k8s we create one SC for each storage device in the cluster to allow for maximal resource utilization. We assume the administrator divides the resources of each SC evenly into multiple PVs. We also assume users have full knowledge of the configuration and capabilities of each SC and are always able to make the best decision of choosing SC that can meet their storage SLOs with minimal storage resources.

We deploy 5 applications in the cluster. One is the log server ELK, requiring $\langle 500GB, 10MB/s \rangle$ storage featuring high capacity low bandwidth requirement. Three of them are Nginx servers. Nginx 1 requires $\langle 100GB, 50MB/s \rangle$ storage. Nginx 2 requires $\langle 200GB, 100MB/s \rangle$ storage. Nginx 3 requires two storage volumes, $\langle 100GB, 50MB/s \rangle$ and $\langle 50GB, 100MB/s \rangle$. The last one is a read only Swift server requiring $\langle 50GB, 500MB/s \rangle$ storage.

Figure 7 shows the number of instances we can deploy for each application under different configurations. 1 PV and 2 PVs mean we divide each SC into 1 PV or evenly into 2 PVs. "Optimal" shows the maximum number of instances that can be deployed if we evenly divide the SC. For different applications, the number of PVs at "optimal" is different. "Optimal+1" shows the case where we have one more PV than "optimal" under each SC. "K8sES-no-leveling" shows the case where we do not set the *usage_leveling* priority rule in k8sES.

This figure shows that k8sES can deploy the most instances for each type of application. It has a higher utilization efficiency than the optimal case of using SCs divided into even-sized PVs. If the SCs and PVs can be created arbitrarily, either automatically or manually, the optimal result is then the same as k8sES without thin provisioning and multiplexing. This is because k8sES allocates storage on the fly based on users' requests. No resources are pre-allocated or pre-created. Furthermore, k8sES can be more resource efficient with thin provisioning and multiplexing enabled. From Nginx 3, we see that k8sES has a higher resource utilization efficiency with the *usage_leveling* priority rule enabled than "k8sES-no-leveling". This is because "k8sES-no-leveling" tends to schedule pods onto storage with extremely high storage size (e.g., Worker 1 on each server) with priority. As a result, the bandwidth resource will be quickly eaten up by k8s volumes with higher bandwidth requirements. The k8s volumes which could be placed in this storage (e.g., with high capacity low bandwidth requirements) cannot be placed any more. In contrast, k8sES is trying to consume different resources at a similar rate, and thus it results in more application instances deployment.

5.6 Computation Overhead

Fast creation time is a major advantage of containers over VMs. In k8s, containers are created during pod creation process. In this subsection, we evaluate the performance influence of selecting storage on creating pods. We deploy Nginx pods in both k8s and k8sES, and measure the time from issuing the "kube create" command till the pod entering "Running" status. We set the pod to request different sizes of storage. The measurement of the pod creation time is repeated 100 times for each pod configuration in both k8s and k8sES. To make a fair comparison, we install the same driver that we

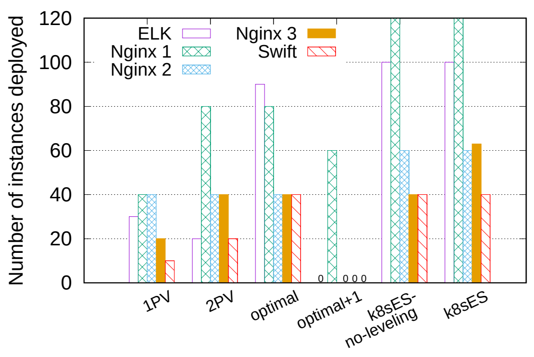


Figure 7: Number of applications that can be deployed.

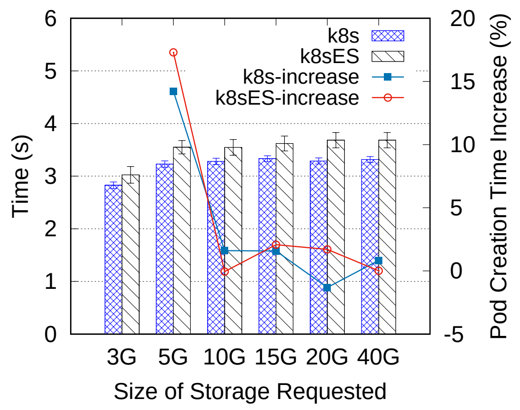


Figure 8: Pod creation time with 95% confidence interval.

implemented for k8sES in k8s so that volumes will be automatically provisioned in both systems. The difference is that there is no storage selection process in k8s. In k8s, we statically select a storage device for each node that has enough resources for the pod we create. Figure 8 shows the average time to create a pod with the 95% confidence interval, and the percentage of pod creation time increase with the storage size. Overall, the storage selection process in k8sES brings only 7%-12% addition latency to pod creation. In addition, there is no evidence that the pod creation time will increase with storage size, as the latency increase of pod creation bounces around 0 in the experiment.

6 Related Work

Containers offer an efficient way to run applications as microservices. They are the essential building blocks of clustering tools like Kubernetes. A performance analysis by IBM shows that containers perform equal to or better than VMs in CPU, memory, network, and storage related tests [25]. Most container research focuses on Docker containers. DRR [24] tries to improve the copy-on-write performance in Docker. Research by Tarasov et al. [35] focuses on the choice of Docker storage driver. Other studies like Slacker [28] and Anwar et al. [19] focus on the Docker registry and the image pulling process. Makin et al. [30] study Docker live migration.

Kubernetes is an open-source container orchestration engine developed by Google and evolved from their prior work on Borg [38] and Omega [22]. Since its v1.0 launch in 2015 [2], Kubernetes has become one of the most prevalent container management systems and motivated a handful of academic studies.. Víctor et al. [32] analyze the performance of the Kubernetes system and study its adaptive application scheduling [31]. Xu et al. [39] attempt to manage network bandwidth for Kubernetes. Tsai et al. apply Kubernetes in fog computing platforms for IoT (Internet of Things) [37].

Storage management in Kubernetes is still underexplored. Some third parties provide storage support in Kubernetes. REX-Ray [13] provides a vendor agnostic storage orchestration engine aiming to provide persistent storage for Docker, Kubernetes, and Mesos. Any volume that is to be used by a Kubernetes resource must be previously created and discoverable by REX-Ray. This is similar to the manual provisioning of PVs in k8s. NetApp's Trident [11, 16] provides persistent storage support to k8s. Users can specify Trident as a storage provisioner in StorageClass, so PVs can be dynamically provisioned from supported NetApp storage systems. However, Trident and similar provisioners still suffer from the issues in PV and SC. Our study targets the PV and SC issues in k8s. Trident and similar provisioners can better ensure users' storage SLOs as storage provisioners of k8sES.

There are storage management systems in VM environment for a period of time. Pesto [27] is implemented as part of VMware's Storage DRS [17] component of vSphere [18]. It provides an automated storage management system that can model and estimate storage performance, and recommend VM disk placement and migration in order to balance space and I/O resources across the datastores in VMware environment. However, in order to apply the Storage DRS to Kubernetes, a storage provisioner which supports Storage DRS must be developed based on the PV abstraction and SC of Kubernetes. In this way, it still suffers from the limitations of PV and SC we discussed in this paper. In addition, Pesto only works in an environment where each compute node is connected to a centrally managed storage. In today's versatile application environment with k8s, storage selection with consideration of the storage connectivity of each compute node is essential in application scheduling.

7 Conclusion

This paper presents k8sES, a system that can efficiently support applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment. With k8sES, users can put their storage requirements directly in their configuration files when deploying applications. K8sES will schedule a pod to the right node and storage, and it automatically creates volumes with the required resources on the selected storage. Users' storage SLOs can be ensured together with all other requirements. In addition, k8sES improves the cluster resource usage efficiency in the cloud and provides I/O monitoring capabilities to Kubernetes.

References

[1] Amazon elastic kubernetes service. https://aws.amazon.com/eks/.

[2] Announcing kubernetes 1.0. http://kuberneteslaunch.com/.

[3] Building large clusters. https://kubernetes.io/docs/setup/cluster-large/#size-of-master-and-master-components.

[4] Docker documentation. https://docs.docker.com/.

[5] Flexvolume. https://github.com/kubernetes/community/blob/master/contributors/devel/flexvolume.md.

[6] Google cloud kubernetes. https://cloud.google.com/kubernetes-engine/.

[7] Kubeedge, a kubernetes native edge computing framework. https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro/.

[8] Kubernetes provisioner. https://kubernetes.io/docs/concepts/storage/storage-classes/#provisioner.

[9] Lxc. https://help.ubuntu.com/lts/serverguide/lxc.html.

[10] Netapp kubernetes service. https://cloud.netapp.com/kubernetes-service.

[11] Netapp/trident github. https://github.com/NetApp/trident.

[12] Persistent volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[13] Rex-ray. https://rexray.readthedocs.io/en/stable/.

[14] Scaling. https://kubernetes.io/docs/getting-started-guides/ubuntu/scaling/.

[15] Store data at the edge with azure blob storage on iot edge. https://docs.microsoft.com/en-us/azure/iot-edge/how-to-store-data-blob.

[16] Trident. https://netapp-trident.readthedocs.io/en/stable-v18.10/.

[17] Vmware storage drs. https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.resmgmt.doc/GUID-827DBD6D-08B7-4411-9214-9E126671457F.html. Accessed: 2019-3-28.

[18] Vmware vsphere. https://docs.vmware.com/en/VMware-vSphere/index.html#com.vmware.vsphere.doc. Accessed: 2019-2-26.

[19] Anwar, A., Mohamed, M., Tarasov, V., Littley, M., Rupprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A. S., Ludwig, H., et al. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies* (2018), p. 265.

[20] Beda, J. Containers at scale. http://slides.eightypercent.net/GlueCon%202014%20-%20Containers%20At%20Scale.pdf.

[21] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[22] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 10.

[23] Chisnall, D. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[24] Dell, E. Improving copy-on-write performance in container storage drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

[25] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On* (2015), IEEE, pp. 171–172.

[26] Gracia-Tinedo, R., Sampé, J., Zamora, E., Sánchez-Artigas, M., García-López, P., Moatti, Y., and Rom, E. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (2017), USENIX Association, pp. 243–256.

[27] Gulati, A., Shanmuganathan, G., Ahmad, I., Waldspurger, C., and Uysal, M. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 19.

[28] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *FAST* (2016), vol. 16, pp. 181–195.

[29] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[30] Makin, G., Kantor, K., Wen, H., Cao, Z., and Mehta, V. Systems and methods for performing live migrations of software containers, Dec. 25 2018. US Patent App. 10/162,559.

[31] Medel, V., Rana, O., Bañares, J. Á., and Arronategui, U. Adaptive application scheduling under interference in kubernetes. In *Utility and Cloud Computing (UCC), 2016 IEEE/ACM 9th International Conference on* (2016), IEEE, pp. 426–427.

[32] Medel, V., Rana, O., Bañares, J. Á., and Arronategui, U. Modelling performance & resource management in kubernetes. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)* (2016), IEEE, pp. 257–262.

[33] Menage, P., Jackson, P., and Lameter, C. Cgroups. *Available on-line at: http://www.mjmwired.net/kernel/Documentation/cgroups.txt* (2008).

[34] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

[35] Tarasov, V., Rupprecht, L., Skourtis, D., Warke, A., Hildebrand, D., Mohamed, M., Mandagere, N., Li, W., Rangaswami, R., and Zhao, M. In search of the ideal storage configuration for docker containers. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on* (2017), IEEE, pp. 199–206.

[36] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 182–196.

[37] Tsai, P.-H., Hong, H.-J., Cheng, A.-C., and Hsu, C.-H. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific* (2017), IEEE, pp. 145–150.

[38] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[39] Xu, C., Rajamani, K., and Felter, W. Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry* (2018), ACM, pp. 32–38.