KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters

Angel Beltre[1], Pankaj Saha[1], and Madhusudhan Govindaraju[1]

[1]SUNY Binghamton University, Binghamton, NY
Email: {abeltre1, psaha4, mgovinda}@binghamton.edu

***Abstract*—In a multi-tenant environment, users' resource demands must be understood by cluster administrators to efficiently and fairly share cluster resources without hindering performance. Kubernetes is a container orchestration system that enables users to share cluster resources, such as CPU, memory, and disk, for the execution of their tasks. Kubernetes provides a monolithic scheduler to make a scheduling decisions for all users in a multi-tenant shared cluster. Kube-batch enables Kubernetes to make scheduling decision based on a multi-resource fairness policy called Dominant Resource Fairness (DRF). DRF has been proven to be a successful mechanism for fine grained resource allocation. However, it does not incorporate other fairness aspects of a shared cluster. Our fairness metrics take into account the use of DRF along with a task's resource demand and average waiting time. We have developed a policy driven meta-scheduler, *KubeSphere*, for a Kubernetes cluster where tasks for individual users can be scheduled based on each user's overall resource demands and current resource consumption. Our experimental results show how the dominant share of a task along with the overall resource demand can improve fairness in a multi-tenant cluster.**
[1]

***Index Terms*—Kubernetes; Resource Fairness; scheduling; Multi-tenant.**

I. INTRODUCTION

It is a common practice for users to acquire a lease on nodes or Virtual Machines from cloud vendors. It is important to improve cluster utilization to fully take advantage of the acquired computing resources. Balancing both cluster utilization and fairness is challenging. In previous work, we identified how different Mesos frameworks [1], with different internal property configurations, can negatively affect performance and the overall resource fairness [2][3]. In Figure 1, we show an example of unfair resource usage in a cluster. User-First is able to launch more tasks at a faster rate and create an unfairness gap with User-Second. In *KubeSphere*, we improve the Kubernetes monolithic scheduler to a two-level scheduler. Our aim is to leverage the flexibility of a two-level scheduling scheme so that it can access the state of the entire cluster and thereby make fairness based scheduling decisions.

In our previous work [3], we provided insight on how overall cluster demand and waiting time can be used to provide fairness in an Apache Mesos environment. While the previous work provided fairness within Mesos' two level scheduling scheme, this paper provides fairness in Kubernetes, along with a new two level scheduling capability on top of Kubernetes' monolithic mechanism.

[1]This work was supported in part by NSF grant OAC-1740263.

The key contributions of the paper are the following:

- We have designed and developed, *KubeSphere*, to enable two-level scheduling in a Kubernetes environment.
- We demonstrate how different fairness policies, when used with the Kubernetes default scheduler, can reduce the average waiting time across all users.

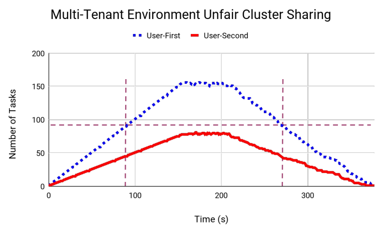


Figure 1. Unfairness: *Two users launch 200 and 100 tasks every 1 second and 2 seconds respectively. The area between the two users represents the unfairness.*

II. BACKGROUND

A. Cloud Scheduling Mechanisms

Task scheduling and resource allocation in cloud infrastructure is a well known and widely discussed topic. In **monolithic scheduling**, such as off-the-shelf Kubernetes, one scheduling component takes care of all incoming task requests and places them in the cluster based on a single policy. In a homogeneous environment, where all tasks are similar, such an approach works well.

Apache Mesos [4][5] [6] has **two level scheduling** wherein at the first level, based on the scheduling priority, a user receives resource offers from the cluster manager. Once resource offers are received, the user makes a second level scheduling decision based on the specific resource demands to accept and reject the resource offer. Apache Mesos incorporated the idea of multi-resource fairness to bring fine grained resource allocation in a cluster based on the Dominant Resource Fairness (DRF) [7] algorithm. However, in our studies [2][3], we show how DRF based fairness in Apache Mesos has several limitations.

In **shared state scheduling**, introduced by Omega [8], all users have similar access to all available resources. With knowledge of the shared resource state of the cluster, users can compete for resources based on their own custom scheduling policy.

B. How Dominant Resource Fairness (DRF) works.

To characterize how DRF works, we illustrate with a pool of computing resources (i.e., <8 CPU, 16 GB of memory>) for which multiple users bid to launch their particular tasks. We have a pair of users named User-A and User-B, who can get access to the pool of resources. User-A consumes <5 CPU, 8 GB of memory>and User-B consumes <2 CPU, 6 GB of memory>in their respective queues of executing tasks. In this particular scenario, User-A consumes 62.5% and 50% from the CPU and memory pool respectively. In addition, User-B consumes 25% and 37.5% from CPU and memory pool respectively. As a result, User-A's dominant share is CPU and User-B's dominant share is memory with 62.5% and 37.5% respectively. The Dominant Share calculation was introduced by Ghodsi et al. [7] with the following equation:

$$DS_i = max_{j=1}^{n}(\frac{u_{i,j}}{r_j}) \tag{1}$$

III. Architecture and Resource Allocation

KubeSphere works as a middleware component in a Kubernetes (k8s) cluster. It receives tasks from users and dispatches them to a connected k8s cluster based on custom defined policy and fairness goals. In a conventional k8s cluster, a user submits tasks directly to a Kubernetes control plane to let k8s master decide when to launch each task. However, in KubeSphere, we provide another layer of control wherein the cluster admin can define and employ fairness policies to control the task dispatching for each user. *KubeSphere* provides three fairness policies, similar to the ones provided in our earlier work with Apache Mesos [3]: (1) *DRF-Aware*, (2) *Demand-Aware*, (3) *Demand-DRF-Aware*. *KubeSphere* also allows other pluggable policies to be used for different classes of applications.

A. KubeSphere Architecture Components

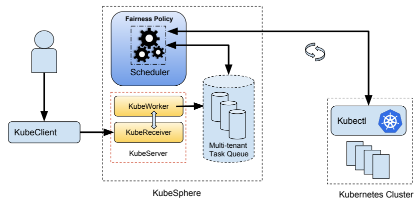


Figure 2. *KubeSphere* Architectural Components: *KubeSphere has a client-server architecture in which multiple users can submit their jobs to be processed based on a specific fairness policy.*

Figure 2 shows the architecture components of *KubeSphere*.

- ***KubeServer:*** KubeServer is the user facing component that receives tasks from users and if required instructs the KubeSphere scheduler to perform the necessary logistics for new users. Each user is mapped with a namespace in the k8s cluster. When a task is received, KubeServer places the task in the appropriate task queue associated with the user.
- ***Multi-tenant Task Queue:*** *KubeSphere* has a multi-tenant task queue for different users to keep track and maintain fairness. KubeServer can store tasks in appropriate queues and the scheduler can dispatch them as needed to meet fairness goals.
- ***Policy based Scheduler:*** The scheduler is placed between the task queue and k8s master. Periodically, the scheduler checks the resource availability form the k8s cluster and the status of the multi-tenant task queue. Based on the chosen custom fairness policy, along with the status of the cluster and task queue, the scheduler periodically dispatches tasks to the k8s cluster. Unlike a conventional k8s cluster, with help of *KubeSphere*, k8s does not have to hold a task for each user to schedule them later.

B. KubeSphere Task Allocation Cycle

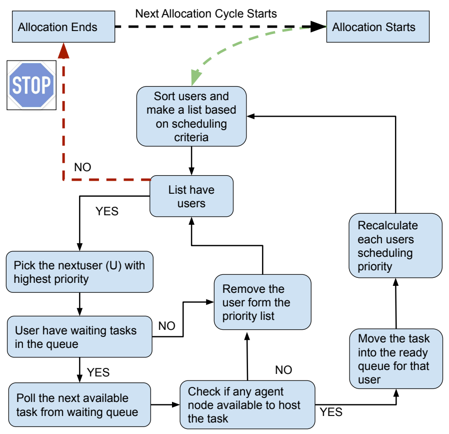

Allocation Ends
Next Allocation Cycle Starts
Allocation Starts
STOP
Sort users and make a list based on scheduling criteria
NO
List have users
YES
Recalculate each users scheduling priority
Pick the nextuser (U) with highest priority
Remove the user form the priority list
User have waiting tasks in the queue
NO
YES
NO
Move the task into the ready queue for that user
Poll the next available task from waiting queue
Check if any agent node available to host the task
YES

Figure 3. *KubeSphere* Allocation Cycle: *KubeSphere allocation based on fairness policies and the number of waiting tasks in their respective queues. Tasks from each user are scheduled on nodes with available resources.*

The primary goal of KubeSphere is to incorporate another layer of scheduling mechanism, on top of the k8s default scheduler, which is aware of cluster resources and task queues in order to incorporate the custom fairness policy. Note that while changing the Kubernetes allocation module is also an option, it requires constant community support for managing and building the code. Instead, *KubeSphere* is designed as an optional module that can be used along with off-the-shelf Kubernetes by interested users. Figure 3 presents a flow diagram that depicts the resource allocation cycles in *KubeSphere*.

KubeSphere consists of two cycles of scheduling resources. At the beginning of an outer cycle, it makes a list of users sorted by the fairness criteria and a list of available resources in all the nodes.

- ***Inner Cycle:*** The inner cycle picks one user from the sorted priority list. One pending task is picked from the user's task queue and it is determined if any node can host the task. If no nodes are available to host the task, the user is removed from the list and the inner cycle continues to the next user in the list.
- ***Outer Cycle:*** If a match is found between a task's resource requirement and a node's available resources, the user is

removed from the the inner cycle. At that time all users' scheduling priority is recalculated and a new list is generated for another round of scheduling through the inner cycle.

- ***End of Allocation:*** This process continues until all the resources are allocated, or users do not have any pending tasks, or no tasks can be fit into the available resources. At this time, the allocation cycle ends. The cycle restarts after a specified scheduling frequency.

IV. KubeSphere Fairness Scheduling Policies

We have implemented three fairness based scheduling policies, which were introduced in previous work [3] with Apache Mesos, to work with *KubeSphere* . The policies are *DRF-Aware*, *Demand-Aware*, *Demand-DRF-Aware*. In the Background section (see II-B), we discussed how we calculate Dominant Share (DS). In this section, we explain the Dominant Demand Share (DDS) policy.

1) Dominant Share (DS): As previously discussed, we use DS to calculate the demand of tasks before being dispatched by *KubeSphere* in the cluster for a given user. For example, assume that we have User-A executing 2 tasks each with a resource consumption of $< 2\, CPU\, 8\, GB\ of\ memory >$ and User-B is executing 7 tasks each with a resource consumption of $< 1\, CPU, 1\, GB\ of\ memory >$. User-A Eq. (2) and User-B Eq. (3) presents how the DS for a pair of users (i.e., DS_A and DS_B) is calculated. In the results, we have marked in bold the highest demand for each user. User-A's DS is memory and User-B's DS is CPU.

$$DS_A = max(\frac{2*2}{16}, \frac{2*8}{32}) = max(\frac{1}{4}, \mathbf{\frac{1}{2}}) \tag{2}$$

$$DS_B = max(\frac{7*1}{16}, \frac{7*1}{32}) = max(\mathbf{\frac{7}{16}}, \frac{7}{32}) \tag{3}$$

2) Dominant Demand Share (DDS): We use DDS to calculate the demand of tasks waiting in the queue for a given user before they are dispatched by *KubeSphere* to the cluster. Assume that we have User-A executing 8 tasks each with a resource consumption of $< 1\, CPU\, 3\, GB\ of\ memory >$ and User-B is executing 4 tasks each with a resource consumption of $< 2\ CPU, 1\ GB\ of\ memory >$. Then, User-A Eq. (4) and User-B Eq. (5) present how the DDS for a pair of users (i.e., DS_A and DS_B) are calculated. In the results, we have highlighted in bold the highest demand for each user. User-A's DS is memory and User-B's DS is CPU.

$$DDS_A = max(\frac{8*1}{16}, \frac{8*3}{32}) = max(\frac{1}{2}, \mathbf{\frac{3}{4}}) \tag{4}$$

$$DDS_B = max(\frac{4*2}{16}, \frac{4*1}{32}) = max(\mathbf{\frac{1}{2}}, \frac{7}{32}) \tag{5}$$

A. DRF-Aware Policy

DRF Aware policy gives higher priority to the user that has the lowest DS in a multi-tenant environment. Hence, a task gets released from a user's queue according to its dominant share calculation. Upon dispatching a task, *KubeSphere* registers a cycle, which leads to the recalculation of dominant share. Based on the results of each cycle,the preference for this policy is given to the user that has the lowest dominant share to release tasks.

B. Demand-Aware Policy

Demand Aware policy enables users that have higher demands to dispatch tasks before other users. After calculating DDS for each user, *KubeSphere* gives preference to the user that has the highest DDS as it is based on demand.

C. Demand-DRF-Aware Policy

This policy combines *DRF-Aware* and *Demand-Aware* policies. Standalone Demand Aware Policy can enable a user with higher demands to consume the total amount of resources, which leads to unfairness in the cluster. By combining both policies, *KubeSphere* avoids starvation and it enables users with higher demands to work together while maximizing their task dispatching rate. Essentially, it can dispatch a number of tasks depending on a *dispatch-decision* value generated for a given cycle for the combination. The value is calculated by finding the maximum DS and minimum DDS, where DS is based on the computing resources and DDS is based on the task demands presented by each user.

V. Experimental setup, results, and evaluation

A. Experimental Setup

For the *KubeSphere* experimental setup, we used namespace as a synonym for user. So, each user's identifier is paired with a namespace, and each user can schedule all tasks in a single namespace. The Kubernetes cluster consists of 4 nodes each with 48 CPUs and 128 GB of memory. We implemented *KubeSphere* to receive tasks in a client server model from all the users created to launch tasks within the system. The tasks submitted for execution, by each user, are homogeneous, with each requesting $< 1\ CPU,\ 1\ GB\ of\ memory >$. As all the tasks are identical and each one consumes about 1 CPU, the execution peak is about 192 tasks.

B. ***Experiment 01***: *Users with Different Configurations of Tasks and Constant Arrival Rates for Each Individual User.*

Users	User-First	User-Second	User-Third
Configuration-1	800	900	1000
Configuration-2	900	900	900
Configuration-3	1000	900	800
Arrival Rate (s)	1	1.5	2

Table 1. Experiment 01 Default Configurations. *These configurations are aimed to extract the baseline for the different order of the number of tasks per user.*

In this experimental setup, we configured three users (User-1, User-2, and User-3) launching a different number of tasks, with different task arrival rates, into the Kubernetes cluster. In configuration-1, we have different task arrival rate and User-1 launches fewer tasks whereas User-3 launches more tasks. User-2 launches 100 tasks more than User-1 and 100 tasks less than User-3. In configuration-2, we have instrumented all three users wherein they are launching the same number of tasks but with different task arrival rates as mentioned in Table 1. In configuration-3, we have the same arrival rate as the previous configurations. However, User-1 launches a higher number of tasks than User-2, and User-2 launches more tasks than User-3. These configurations do not have any of the fairness policies enabled.

In Figure 4a, User-First has the lowest number of tasks with the highest arrival rate and User-Third has the largest number of tasks with slowest arrival rate. We can observe that in such a scenario the user with the lowest amount of tasks and the highest submission frequency has an advantage over the other users. For example, Figure 4a shows that the average number of tasks being executed by User-First and User-Second are close to each other as they are overlapping in some instances. However, User-Third faces unfairness when compared to the other users as most of its tasks are placed in the queue because it has a slower submission rate. In Figure 4b, all users have the same workload. User-First takes the lead, executing close to 100 tasks on average along the run. This is due to the fact that User-First has a large number of tasks and the highest arrival rate. While User-Second and User-Third have 1.5 and 2 seconds arrival rate respectively, the high arrival rate enables higher submission of tasks for User-First. In Figure 4c, we can observe a similar trend as the one shown in Figure 4b. User-First is ahead and maintains an average execution across the run of about 100 tasks. In addition, it can be observed that User-Second maintains an average execution of about 75 tasks. However, the amount of resources available in the system dictate the number of tasks that each user should execute at any given point, which is 64.

C. ***Experiment 02****: Lowest Number of Tasks paired with the Lowest Arrival Rates*

Users	User-First	User-Second	User-Third
Number of Tasks	800	900	1000
Arrival Rate (s)	1	1.5	2

Table 2. Experiment 02 Task Configuration. *Increasing number of tasks for each user paired with increasing arrival rate.*

Policies \ Users	User-First	User-Second	User-Third
DRF Aware	157.56%	-30.41%	-98.68%
Demand Aware	12.30%	0.48%	-10.27%
Demand DRF Aware	6.38%	8.76%	-13.00%

Table 3. Experiment 02 Results. *Increasing number of tasks for each user. The difference of average waiting time of each user in comparison to the average waiting time of the entire cluster for all KubeSphere policies.*

In this experiment, we have fewer tasks being launched by users with the fastest arrival rate. In Table 1, we show the number of tasks and the rate at which each user launches its workload.

The fairness exploration for the increasing number of tasks for each user is presented in Figure 5a, 5b, 5c. In essence, we enabled *KubeSphere* with each of the task dispatching policies implemented in the cluster. For DRF-Aware policy, User-First yields the highest waiting time in comparison to the User-Second and User-Third. In addition, User-First yielded a 157.56% higher waiting time in comparison to the cluster's average for all tasks. However, for Demand-Aware policy User-First yielded a 12.30% higher average waiting time than the cluster's waiting time. In addition, User-Second was below 1% and User-Third experienced 10.27% lower average waiting time than the cluster's total. Moreover, for Demand-DRF-Aware policy both User-First and User-Second experienced higher waiting time in comparison to the cluster of 6.38% and 8.76% respectively. For User-Third, the average waiting time for Demand-DRF Aware encountered was 13.00% below the cluster's average waiting time.

For waiting time, Figure 6a shows the total waiting time of each user when executed with the different policies. Figure 6b has the average waiting time of each user for the different *KubeSphere* policies. Lastly, Figure 6c compares the total waiting time for each policy for all the tasks in the cluster. In Table 3, we show average waiting time difference for all the users and the different policy configurations compared to the cluster's average waiting time.

D. ***Experiment 03****: Large number of tasks with higher arrival rates, and lower number of tasks with slower arrival rates.*

Users	User-First	User-Second	User-Third
Number of Tasks	900	900	900
Arrival Rate (s)	1	1.5	2

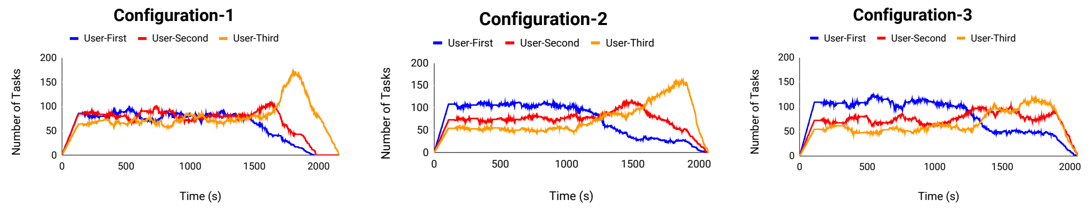
Table 4. Experiment 03 Task Configuration. *Users with the same number of tasks paired with increasing arrival rate.*

Policies \ Users	User-First	User-Second	User-Third
DRF Aware	54.23%	3.90%	-58.13%
Demand Aware	-0.14%	0.41%	-0.26%
DemandDRFAware	3.22%	4.57%	-7.79%

Table 5. Experiment 03 Results. *The difference of average waiting time of each user in comparison to the average waiting time of the entire cluster for all KubeSphere policies*

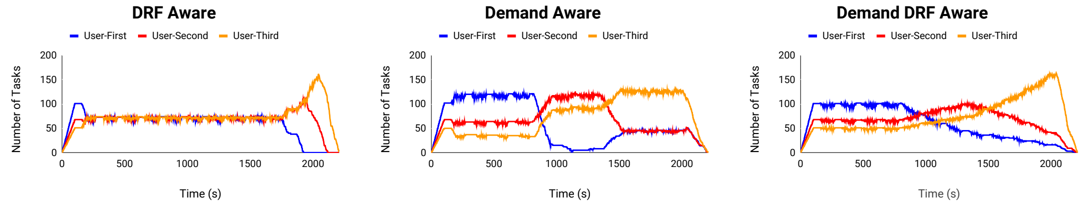
In this experiment, *KubeSphere* launches tasks for all the users at a rate of 1, 1.5, and 2 seconds for User-First, User-Second, User-Third respectively. In Table 4, we present that all the users have the same workload of 900 tasks. To guarantee the reproducibility of the experiment, we kept all the resources fixed for all the tasks with the following resource configuration: $< 1\ CPU, 1\ GB\ of\ memory >$. At any given point, the fair number of tasks, given the total amount of resources, for each user is 64.

Our fairness exploration for the same number of tasks is presented in Figure , , and . We enabled *KubeSphere* with each of the task dispatching policies implemented in the cluster. For DRF-Aware policy, User-First yields the highest waiting time in comparison to the User-Second and User-Third. In addition, User-First has a 54.23% waiting time in comparison



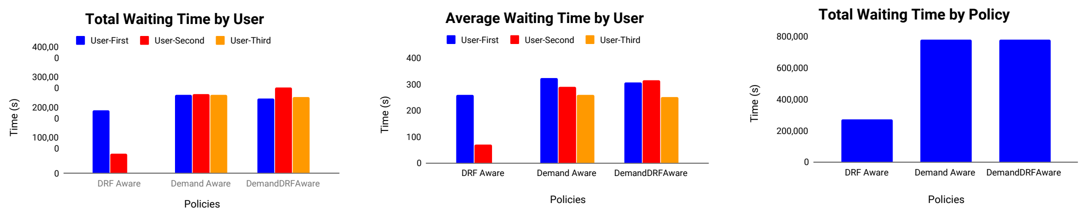
(a) (b) (c)

Figure 4. Experiment 01 Fairness. *(a) Default for 800, 900, and 1000 tasks for User-First, User-Second, User-Third respectively. (b) Default for 900, 900, and 900 tasks for User-First, User-Second, User-Third respectively. (c) Default for 1000, 900, and 800 tasks for User-First, User-Second, User-Third respectively. All runs were executed using 1, 1.5, and 2 seconds for each user and its number of tasks respectively.*



(a) (b) (c)

Figure 5. Experiment 02 Fairness. *(a) DRF-Aware, (b) Demand Aware, and (c) Demand DRF Aware for 800, 900, and 1000 tasks for User-First, User-Second, User-Third respectively. All runs were executed using 1, 1.5, 2 seconds for each user and its number of tasks respectively.*



(a) (b) (c)

Figure 6. Experiment 02 Results. *(a) Total Waiting Time by User, (b) Average Waiting Time by User, (c) Total waiting Time by Policy. KubeSphere receives a higher number of tasks from User-First at the higher arrival rate than User-Second and User-Third.*

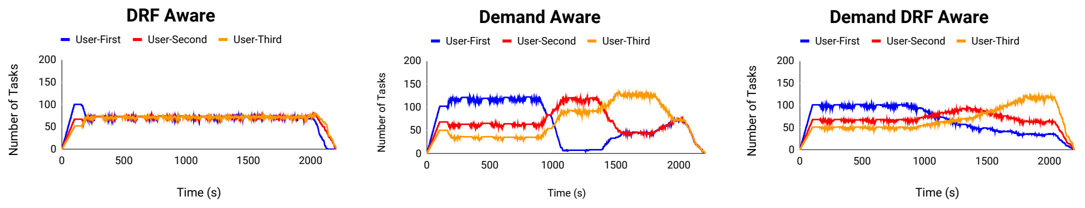
to the cluster's average for all tasks. However, for Demand-Aware policy, all the users stay within 1% of the cluster's average waiting time. Moreover, Demand-DRF-Aware policy for User-First experienced a 3.22% higher waiting time in comparison to the cluster's average. For User-Second, all the policies waiting time stayed below 5%. However, User-Third experienced the lowest waiting time for all the policies. Our metadata collected from the experiment is presented in Figure 8a, 8b, and 8c. Figure 8a has the total waiting time of each user when executed with the different policies. Figure 8b has the average waiting time of each user for the different *KubeSphere* policies. Lastly, Figure 8c compares the total waiting time for each policy for all the tasks in the cluster.

Users	User-First	User-Second	User-Third
Number of Tasks	1000	900	800
Arrival Rate (s)	1	1.5	2

Table 6. Experiment 4 Task Configuration. *KubeSphere receives a higher number of tasks and a faster rate from User-First. Also, it receives a reduced number of tasks at a slower rate from User-Third.*

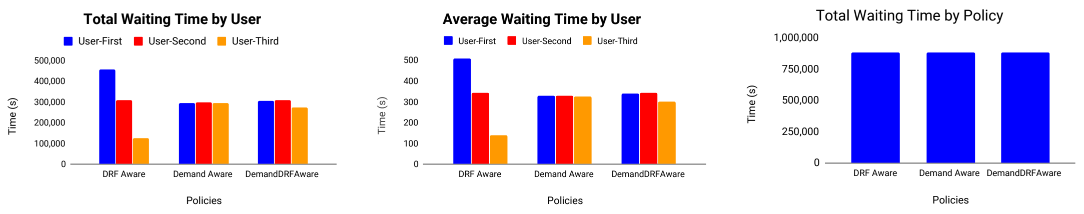
Users / Policies	User-First	User-Second	User-Third
DRF Aware	57.06%	-5.45%	-65.19%
Demand Aware	-9.80%	0.28%	11.93%
DemandDRFAware	1.21%	1.86%	-3.61%

Table 7. Experiment 04 Results. *Difference average waiting time of each user in comparison to the cluster's average waiting time for all the policies in KubeSphere.*



(a) (b) (c)

Figure 7. Experiment 03 Fairness. *(a) DRF-Aware, (b) Demand-Aware, and (c) Demand-DRF-Aware for 900, 900, and 900 tasks for User-First, User-Second, User-Third respectively. All runs were executed using 1, 1.5, 2 seconds for each user.*



(a) (b) (c)

Figure 8. Experiment 03 Results. *(a) Total Waiting Time by User, (b) Average Waiting Time by User, (c) Total waiting Time by Policy. KubeSphere receives a higher number of tasks from User-First at the higher arrival rate than the others.*

*E. **Experiment 04:** Highest Number of Tasks Paired with Fastest Arrival Rate.*

For this experiment, we assigned the workloads for each user in descending order and the arrival rates in ascending order for User-First, User-Second, and User-Third respectively as presented in Table 6. In addition, the fairness observations and the waiting time results are presented in Figure 9 and 10 respectively.

We have consolidated all the graphical representation of our results in Figure 9 in Table 7. In Figure 10, we present the different times presented by each user with the different policies.

For completeness, we changed the User-First to have the largest number of tasks, but with the fastest arrival rates. In Figure 9a for *DRF-Aware* policy, we observe that despite having the fastest arrival rate User-First faces an high average waiting time of 57.06% in comparison to the average waiting time of the cluster. While User-Second and User-Third fell below the average waiting time of the entire cluster with a 5.45% and 65.19% respectively. On the other hand, in Figure 9b, User-First was able to capitalize on the fact that the *Demand-Aware* policy focuses on the workload of each individual user as a measure of fairness. The average waiting time of User-First was 9.80% below average waiting time of the entire cluster. In addition, for both User-Second and User-Third the average waiting time overhead presented is between 1% and 12% respectively. The combination of both *DRF-Aware* and *Demand-Aware* (see 9c) policies was able to keep the overhead below 4% and the best fairness performance 2% below cluster average waiting time.

VI. Related Work

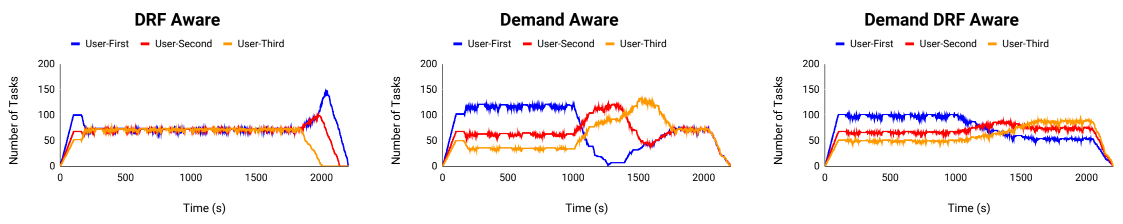
In this work, we implemented fairness policies within a Kubernetes cluster. Khaled et al. [9] worked on improving execution time of scientific workflows by implementing a Resource Demand Aware scheduling mechanism. With RDAS, Khaled et al. consider the structure of workflows and resource demands, which in turn enables the optimization of the system for better throughput. In the *KubeSphere* cluster, we have considered tasks from users and their requirements and scheduled them based on the total current demand from each user.

In an attempt to increase overall throughput, Boyang et al. [10] developed, *R-Storm*, a resource demand aware framework. R-Storm proved that in the presence of Storm, applications' performance can be improved.

Chowdhury et al. [11] identified tradeoffs between fixed and elastic demands for multiple resources and introduced *High Utilization with Guarantees* (HUG). In non-cooperative runtime environments, HUG achieves maximum network utilization, but not optimal isolation. In cooperative run time environments such as private datacenter networks, it can achieve optimal isolation guarantee. In our own previous work [3][12], we implemented the same policies but in an Apache Mesos setup. We have extended the work by re-implementing the policies in Kubernetes, adding a two-level scheduling mechanism, and quantifying the fairness benefits.

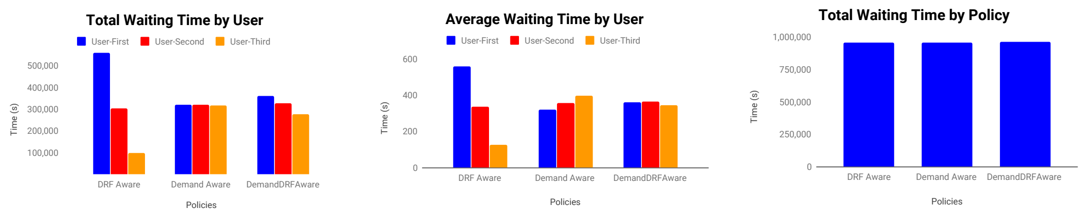
VII. Conclusion

Our work focuses on enabling resource fairness for containerized workloads on Kubernetes. We identified that the off-the-shelf implementation does not consider overall resource demands for all the tasks or the average waiting time of tasks



(a) (b) (c)

Figure 9. Experiment 04 Fairness. *(a) DRF-Aware, (b) Demand-Aware, and (c) Demand-DRF-Aware for 1000, 900, and 800 tasks for User-First, User-Second, User-Third respectively. All runs were executed using 1, 1.5, 2 seconds for each user and its number of tasks respectively.*



(a) (b) (c)

Figure 10. Experiment 04 Results. *(a) Total Waiting Time by User, (b) Average Waiting Time by User, (c) Total waiting Time by Policy. KubeSphere receives a higher number of tasks from User-First at the higher arrival rate than others.*

before execution in a cluster. Hence, we implemented *KubeSphere* on top of Kubernetes to enable seamless scheduling of tasks based on their resource demands and their waiting time, before allocation of resources. Kubernetes provides a monolithic scheduler where scheduling decisions are made at a single point by Kube-Master. For a diverse set of tasks and different resource requirements, a monolithic scheduler is not suitable for multi-tenant environments. In contrast, Apache Mesos provides two level scheduling where individual user's scheduling decision and resource constraints are honored. However, due to user specific constraints and scheduling properties, Mesos' DRF based scheduling falls short to provide fairness in a multi-tenant environment. We provide *KubeSphere*, which incorporates another scheduling layer on top of Kubernetes' monolithic scheduler to improve resource fairness based on DRF, user resource demand, and their combination. Our results show how *KubeSphere*'s custom policies improve the fairness over the default monolithic scheduling mechanism. Our results show that *KubeSphere*'s scheduling provides fairness among competing users. For instance DRF Aware, yielded the lowest total waiting time for Experiment V-C. Demand Aware and Demand DRF Aware both yield similar total waiting time. *KubeSphere* shows that it is essential to understand user demands to improve their average waiting time. *KubeSphere* can be used by system administrators to get insights on how a Kubernetes cluster performs and improve the overall average waiting time for all users in a cluster.

References

[1] P. Saha, A. Beltre, and M. Govindaraju, "Scylla: A Mesos Framework for Container Based MPI Jobs," in *MTAGS17: 10th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, Denver, 2017.

[2] ——, "Exploring the fairness and resource distribution in an apache mesos environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 434–441. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00061

[3] ——, "Tromino: Demand and DRF aware multi-tenant queue manager for apache mesos cluster," in *11th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2018, Zurich, Switzerland, December 17-20, 2018*, 2018, pp. 63–72. [Online]. Available: https://doi.org/10.1109/UCC.2018.00015

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[5] P. Saha, M. Govindaraju, S. Marru, and M. Pierce, "Integrating apache airavata with docker, marathon, and mesos," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 1952–1959, 2016.

[6] ——, "Multicloud resource management using apache mesos with apache airavata," *arXiv preprint arXiv:1906.07312*, 2019.

[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *Nsdi*, vol. 11, no. 2011, 2011, pp. 24–24.

[8] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf

[9] K. Almi'ani, Y. C. Lee, and B. Mans, "Resource demand aware scheduling for workflows in clouds," in *Network Computing and Applications (NCA), 2017 IEEE 16th International Symposium on*. IEEE, 2017, pp. 1–5.

[10] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 149–161.

[11] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "{HUG}: Multi-Resource fairness for correlated and elastic demands," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. usenix.org, 2016, pp. 407–424.

[12] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proceedings of the Practice and Experience on Advanced Research Computing*. ACM, 2018, p. 11.