2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)

Tromino: Demand and DRF Aware Multi-Tenant Queue Manager for Apache Mesos Cluster

Pankaj Saha, Angel Beltre, and Madhusudhan Govindaraju

Cloud and Big Data Laboratory, State University of New York (SUNY) at Binghamton
{psaha4, abeltre1, mgovinda}@binghamton.edu

***Abstract*—Apache Mesos, a two-level resource scheduler, provides resource sharing across multiple users in a multi-tenant cluster environment. Computational resources (i.e., CPU, memory, disk, etc.) are distributed according to the Dominant Resource Fairness (DRF) policy. Mesos frameworks (users) receive resources based on their current usage and are responsible for scheduling their tasks within the allocation. We have observed that multiple frameworks can cause fairness imbalance in a multi-user environment. For example, a greedy framework consuming more than its fair share of resources can deny resource fairness to others. The user with the least Dominant Share is considered first by the DRF module to get its resource allocation. However, the default DRF implementation, in Apache Mesos' Master allocation module, does not consider the overall resource demands of the tasks in the queue for each user/framework. This lack of awareness can result in users without any pending task receiving more resource offers while users with a queue of pending tasks starve due to their high dominant shares.**

In a multi-tenant environment, the characteristics of frameworks and workloads must be understood by cluster managers to be able to define fairness based on not only resource share but also resource demand and queue wait time. We have developed a policy driven queue manager, *Tromino*, for an Apache Mesos cluster where tasks for individual frameworks can be scheduled based on each framework's overall resource demands and current resource consumption. Dominant Share and demand awareness of Tromino and scheduling based on these attributes can reduce (1) the impact of unfairness due to a framework specific configuration, and (2) unfair waiting time due to higher resource demand in a pending task queue. In the best case, Tromino can significantly reduce the average waiting time of a framework by using the proposed Demand-DRF aware policy.

I. INTRODUCTION

In clouds and large clusters, several different types of applications are executed and multiple users/groups can demand difference resources to execute their tasks. In such shared environments, *Fairness* needs to be defined and maintained. Apache Mesos [1] is a data center Operating System that combines resources from all participating cluster nodes and provides a global view as a single giant pool of resources. Fairness for multiple resources in this multi-tenant environment is defined using the Dominant Resource Fairness (DRF) policy, introduced by Ghodsi et al. [2].

Apache Mesos acts as a resource manager and different Mesos frameworks act as resource consumers. One of the widely known frameworks, Apache Aurora [3], was developed by Twitter for running services and short-lived jobs. Mesosphere developed, Marathon [4], a framework for long-running services and container orchestration. The Chronos [5] framework was developed for periodic execution of cron jobs. In our previous work, we developed Scylla [6], which is a Mesos framework for running MPI jobs on cloud-based HPC systems. Apache Mesos has proven scalability of running on more than 10K nodes[7] in a production setup, and it seamlessly supports Docker [8] as its primary choice for containerized applications.

The introduction of Apache Mesos and its DRF based allocation module led to widespread acceptance by the cloud computing community, as workload fairness and optimal resource utilization are essential for multi-framework execution environments. In our previous work [9], we identified how resource allocation and fairness could be affected due to framework settings such as offer refusal period, resource holding period, task arrival rate, and second level scheduling. Each framework in a Mesos cluster is typically designed for a specific type of application, but its configuration properties can hinder fairness and induce starvation in a cluster.

To observe the unfairness in an Apache Mesos cluster, we set up a cluster environment of 4 nodes where each node contains $< 8\ CPU, 16\ GB\ memory >$ of resources. We orchestrated synthetic jobs, launched by Scylla and Marathon, wherein each required $< 1\ CPU, 2\ GB\ memory >$ of resources. In an ideal fair distribution scenario, each framework should be able to run 16 jobs each. As each job is identical in terms of resource requirements, the number of jobs launched by each framework is proportional to the amount of resources consumed by each framework.

In Figure 1, we can observe how Marathon utilizes more resources and launches more tasks in the cluster while Scylla uses comparatively low amount of resources. We measure the unfairness U_A to framework A by using the following formula proposed in earlier work [9]:

$$U_A = \left(\frac{Area_{i,j}\ by\ framework_A}{Area_{i,j}\ by\ fair\ graph}\right) * 100$$

$Area_{i,j}$ is the area under the curve from point i to j

In Figure 1, the dotted horizontal line shows the fairness baseline, which indicates the number of tasks each framework should be able to execute in a fair distribution manner. Two vertical dotted lines represent the beginning and end of the period for which we have calculated the fairness.

In Figure 4, the flow diagram shows how the Mesos allocation module distributes resources to multiple frameworks

978-1-5386-5504-7/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC.2018.00015

in a Mesos cluster. Apache Mesos' implementation of DRF does not consider the overall resource demands of all the tasks pending in each framework's queue. While allocating resources, it only considers the current resource consumption of each framework. This can lead to a situation where a framework with a higher number of tasks in its queue faces an extended waiting time for the tasks to be launched. This phenomenon can increase the cluster's overall waiting time by imposing unfair waiting time to frameworks with higher demands.While allocating resources, the Mesos Master picks agent nodes with available resources in a random order. It does not validate if the available resources are useful to a framework, or if they are aligned with the resource demands. We have used of-the-shelf allocation module of Apache Mesos to study this phenomenon and present schemes on how demand awareness can be achieved while allocating resources.

We have developed a queue manager, *Tromino*, which is aware of DRF and the dominant share of each framework in the cluster. Tromino controls the waiting task queue of each framework and releases tasks based on the dominant share for better fairness. We have also considered a situation where few frameworks in a cluster have higher demands compared to others. Releasing tasks only based on the dominant share may improve resource fairness but could also increase the total waiting time of tasks for that framework.

The key contributions of this work are the following:

- We have designed and developed a queue manager, *Tromino*, on top of the Apache Mesos scheduler, which keeps track of incoming tasks of all the registered frameworks in the cluster and their current resource consumption.
- *Tromino* monitors the resource demands and resource consumption information from the waiting queue of jobs and Mesos Master respectively to control task dispatching.
- We have shown how our demand aware scheduling on top of Mesos' default DRF, can reduce the average waiting time across all frameworks.
- We present and show how different policies of task dispatching, based on the demand and dominant share, affect fairness in four different case scenarios.

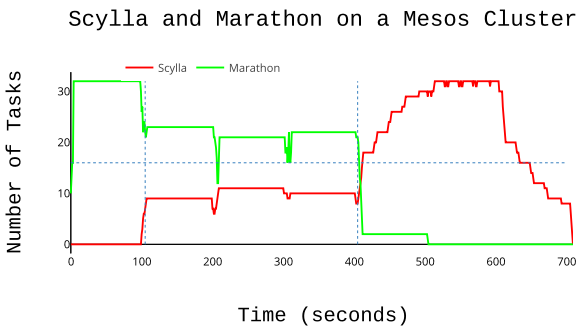


Figure 1. *Scylla and Marathon are chasing for resources in a Mesos cluster. Marathon is able to launch several more tasks than Scylla. Scylla's tasks face longer wait times due to unfair distribution.*

II. BACKGROUND

A. Apache Mesos

Figure 2 shows the architectural components of Apache Mesos. It consists of three major components. Mesos Agent, Mesos Framework, and Mesos Master. **Mesos Agent** consists of the computational resources like CPU, memory, disk, etc. that are required to execute tasks. Each Mesos Agent needs to have a Mesos Executor installed to receive task execution requests. **Mesos executor** is a program that resides in all the Mesos agent nodes and executes tasks upon requests from the Mesos master. **Mesos Frameworks** are the users that have a pending queue of tasks to be launched, along with user-defined resource demands. They also have a scheduler that decides the task that has to be launched on an each agent node after resources are offered to them. During framework registration, each framework needs to provide the executable path of the Mesos Executor, or else the default Mesos Executor takes care of the requested tasks. For example, Apache Aurora uses Thermos [10] as the Mesos Executor whereas Mesosphere Marathon [4], developed by the core developers of Apache Mesos, uses the default Mesos Executor. **Mesos Master** negotiates between the Mesos frameworks and Mesos agents to allocate resources based on current resource consumption by each framework. In a distributed production setup, multiple Mesos Masters are installed, and one of them is elected as a leader by zookeeper [11] to serve as the resource broker for the cluster. Mesos Master consists of a resource allocation module, which decides to allocate resources to each framework periodically based on the DRF policy [2]. The Mesos setup follows the following steps to allocate resources for executing tasks on the agent nodes.

- ***Step 1 - Advertising Resources***. At the beginning of an allocation cycle each Mesos Agent advertises its available resources like CPU, memory, disk, etc. to the Mesos Master.
- ***Step 2 - DRF based Resource Allocation***. Based on the current resource consumption by each framework, the Mesos Master's allocation module decides the resource allocation for each framework for executing the tasks. The Mesos Master does not take into account the resource needs of a framework before sending it offers. This step is considered as the 1st level scheduling in a Mesos setup.
- ***Steps 3 - Generating Matching List of Tasks and Agents***. Now, each framework decides how to schedule tasks across the resources in the agent nodes allocated to it. The framework takes into account the hardware, device, or other task specific constraints provided by the user to the framework. Once a framework makes its decision on using or rejecting the resources from each allocated agent, it makes a list of tasks matching with the Mesos agents and sends it to the Mesos Master. The matching of resources offered by the Mesos Master to the requirements of tasks in a framework is called the 2nd level scheduling.
- ***Step 4 - Assigning Tasks to Allocated Agents***. If the framework's request for resources for each task does not exceed the available resources in the agent nodes, the

Mesos Master request Mesos agents to execute the task. If the frameworks' resource requirements do not match the availability on the agent nodes, the execution request is not sent to the agents. The resources that are not used by each framework are returned to the pool of available resources and offered during the next allocation cycle as explained in Figure 4.

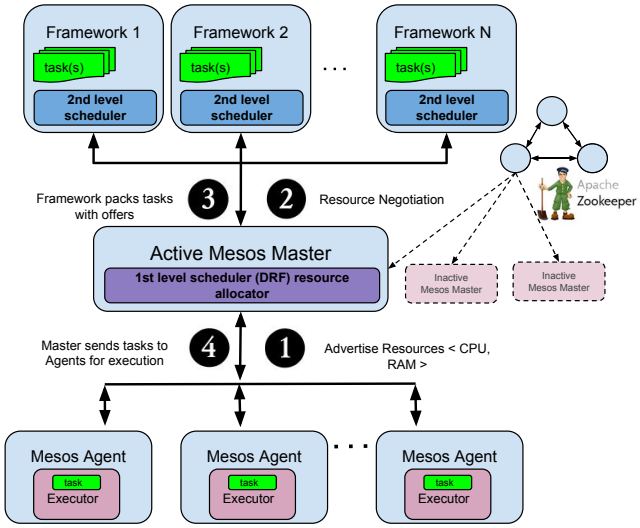


Figure 2. ***Apache Mesos Architecture***: *This diagram shows the primary architectural components and resource allocation steps from Mesos Agents to Mesos Framework (user) through DRF based resource allocation by Mesos Master.*

B. DRF and Apache Mesos

Resource allocation policies, such as Max-Min, or its more generalized version like the weighted Max-Min, can provide fairness to multiple users in a multi-tenant environment. However, they are designed for a single type of resource. For multiple resources, slot based allocation has became popular with YARN [12] for Hadoop and map-reduce tasks. However, the slot based allocation has a shortcoming of over or under allocation of resources. In cloud and modern cluster computing environments users can request different types of resources. Multiple jobs can be co-scheduled on the same physical node. The Dominant Resource Fairness (DRF) [2] algorithm was introduced to bring fairness among multiple users competing for various kinds of resources. Apache Mesos is one of the leading cluster resource managers to incorporate DRF. Its resource allocation module is based on DRF.

To explain how DRF works in Apache Mesos, we illustrate using a simple example how the dominant resource and dominant share are calculated. Figure 3 shows a pool of computing resources and two frameworks competing for different amount of resources for various tasks in their own queues. Framework A is currently consuming 4 CPUs and 6 GB of memory for all its running tasks. Similarly, Framework B's tasks are consuming 2 CPUs and 6 GB of memory. The

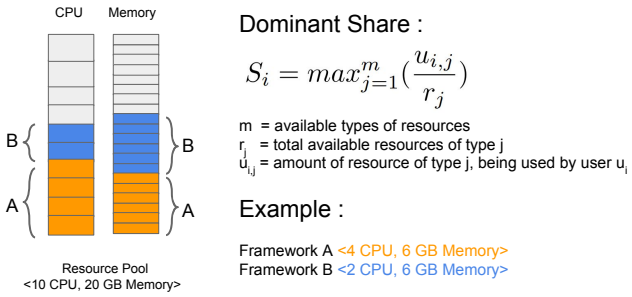


Figure 3. ***Dominant Share***: *This diagram pictorially represents the concept of Dominant Share, which decides the available resource allocation to each framework, while consuming multiple types of resources to execute pending tasks in the queue.*

total pool of resources in this example consists of 10CPUs and 20 GB memory. Figure 3 shows how the dominant share and dominant resource are determined for both the frameworks. The flowchart in Figure 4 explains how DRF is implemented in Apache Mesos and how the allocation cycle works.

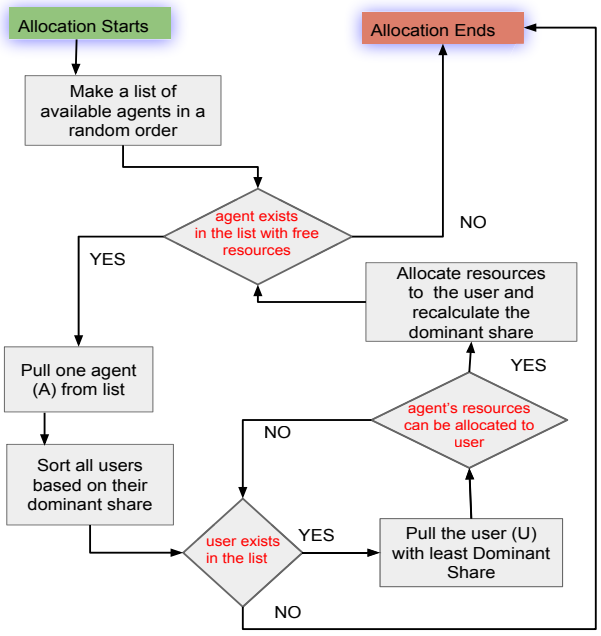


Figure 4. ***Resource Allocation Cycle***: *Periodic resource allocation cycle by Mesos Master's allocation module to allocate computational resources from Mesos Agents to Mesos Frameworks determined by dominant share of each framework (user).*

III. Tromino Architecture and Strategy

A. Tromino

Figure 5 shows how the *Tromino* queue manager fits in a Mesos setup to manage the task queues for several Mesos Frameworks. *Tromino* fetches cluster and task information periodically form the Mesos Master and keeps track of the tasks in the queue for each framework. In a conventional Mesos setup, the end user submits tasks directly through the frameworks, and each framework's tasks are executed using the steps listed in Section II-A. Unlike a conventional setup, in the presence of *Tromino*, a user submits tasks directly to *Tromino*. *Tromino* maintains separate queues for each framework. *Tromino* takes into account the following information to decide the task and framework to dispatch: (1) current resource consumption of each framework; (2) the total available resources in the cluster; and (3) the resource demands of tasks in each framework's queue.

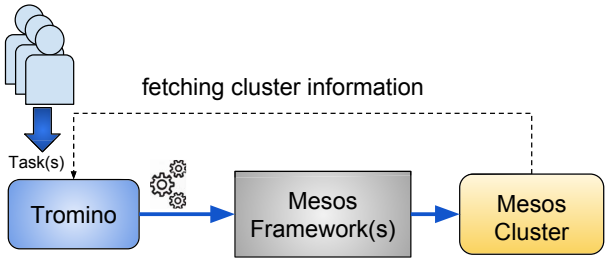


Figure 5. ***Tromino Architecture****: Tromino communicates with the Apache Mesos Master to understand the current resource consumption of each framework and based on the chosen policy it releases tasks from the queue associated with each framework.*

B. Tromino Manager

Figure 6 shows the components and flow involved in dispatching tasks through *Tromino*. *Tromino* consists of three major elements (1) Tromino Dispatcher, (2) Tromino Manager and (3) Tromino Scheduler.

Tromino Dispatcher consists of a dispatcher and a task queue for each framework registered with the Mesos cluster. Based on the user's preference for a framework as specified to the Tromino client, *Tromino* moves the task to the appropriate dispatcher. Each dispatcher collects information on all the resource demands of the tasks in its queue and the current dominant resource demand of the queue.

Tromino Manager periodically communicates with the Mesos Master to fetch information regarding resource consumption of all the frameworks, the dominant share of each framework, and the available resources in the cluster.

Tromino Scheduler controls the release of the tasks from each dispatcher's queue to the corresponding frameworks. The tasks are released based on the chosen scheduling policy (see Section III-C). It consults with the Tromino Manager to decide how many tasks need to be released.

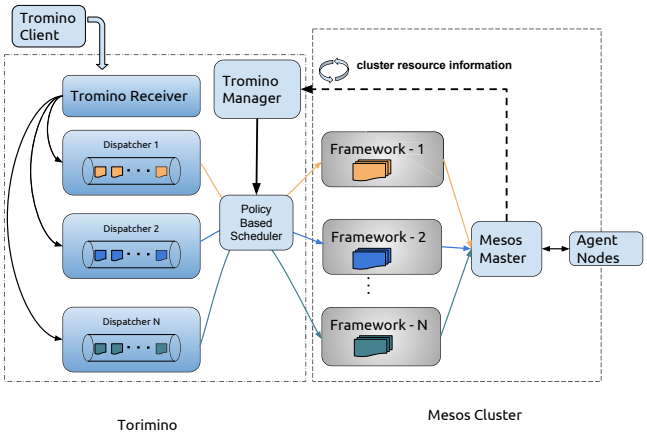


Figure 6. ***Tromino Manager****: Tromino Managers consists of multiple Tromino Dispatchers, one for each framework, and it can communicate with Mesos Master to get information about current resource consumption of each registered active framework. Tromino Manager also communicates with each dispatcher to understand the current resource demand to make decisions regarding the release of tasks from each dispatcher.*

C. Tromino Policies

We have designed three scheduling policies for the Tromino Scheduler: DRF Aware Policy, Demand Aware Policy, and Demand-DRF Aware Policy. These policies can be extended further based on the scheduling needs of users and applications. In Section II-B, we discused how the Dominant Share (DS) is calculated for any DRF based algorithm. We introduce the Dominant Demand Share (DDS) attribute in this section. Later in this section, we explain how the Tromino policies use the DS and DDS values.

For example, let us consider a cluster with a total of 20 CPUs and 40 GB of memory, where two frameworks (Framework A and Framework B) are competing for shared resources. Each of the frameworks can have a different number of tasks waiting in their queues to be dispatched. In this example, Framework A has 10 tasks each with $< 1\ CPU, 4\ GB\ memory >$ as the resource demands. Framework B has a total of 5 tasks each with $< 2\ CPU, 1\ GB\ memory >$ demand waiting in the queue to be dispatched. In Table 1, we present how the calculation is carried out for the Dominant Demand Shares (DDS_A *and* DDS_B) and the dominant demand of each framework.

$$DDS_A = max[(10{*}1)/20, (10{*}4)/40] = max[0.5,\ \mathbf{1.0}]$$
$$DDS_B = max[(5{*}2)/20, (5{*}1)/40] = max[\mathbf{0.5},\ 0.125]$$

Table 1. *Dominant Demand Share (DDS) calculation for both frameworks in the example, before Tromino starts dispatching any tasks to the Mesos cluster.*

Also, let us consider that Framework A is executing 3 tasks each consuming $< 1\ CPU, 4\ GB\ memory >$ of resources, and Framework B is executing 5 tasks wherein each is consuming $< 2\ CPU, 1\ GB\ memory >$ of resources. Now, the Dominant Shares (DS_A *and* DS_B) for both frameworks and their dominant resources are shown in Table 2.

$$DS_A = max[(3*1)/20, (3*4)/40] = max[0.15,\ \mathbf{0.3}]$$
$$DS_B = max[(5*2)/20, (5*1)/40] = max[\mathbf{0.5},\ 0.125]$$

Table 2. *Dominant Share (DS) calculation for both frameworks in the example, before Tromino starts dispatching any tasks to the Mesos cluster.*

In Table 1 and 2, we show the calculation of DDS and DS for both the frameworks. For Framework A, the values are 1.0 and 0.3 respectively. Similarly, for Framework B, the values are 0.5 and 0.5 for DDS and DS respectively.

- ***DRF Aware Policy.*** In this policy, we assign a higher priority to the framework with lesser dominant share and let its corresponding dispatcher release a task. After a task is dispatched, *Tromino* recalculates the dominant share and decides the next dispatcher from which a task can be released. For example, as shown in Table 1, Tromino allows Framework A to release the task. After the first task is dispatched, the DS for Framework B becomes **0.4**. Subsequently, *Tromino* allows another two tasks to be released from Framework A's dispatcher until its dominant share becomes 0.6, which is higher than the dominant share of Framework B. Now, *Tromino* allows two more tasks

$$DS_A = max[(6*1)/20, (6*4)/40] = max[0.3,\ \mathbf{0.6}]$$
$$DS_B = max[(5*2)/20, (5*1)/40] = max[\mathbf{0.5},\ 0.12]$$

Table 3. *Dominant Share of both frameworks after Tromino dispatches 3 tasks from Framework A's dispatcher.*

from Framework B's dispatcher to be released and then the dominant share for both the frameworks is as shown in Table 4. At this point, *Tromino* stops from any further

$$DS_A = max[(6*1)/20, (6*4)/40] = max[0.3,\ \mathbf{0.6}]$$
$$DS_B = max[(7*2)/20, (7*1)/40] = max[\mathbf{0.7},\ 0.15]$$

Table 4. *Dominant Share after Tromino dispatches 2 more tasks from Framework B's dispatcher until no more resources are available in the cluster.*

dispatching as there are no more resources available in the cluster. Finally, *Tromino* follows the same steps in the next dispatching cycle if more resources become available.

- ***Demand Aware Policy.*** In this policy, we consider the Dominant Demand Share (DDS) to control the dispatching of tasks from each framework's dispatcher. The framework that has more demand in terms of Dominant Demand Share is given higher priority to dispatch its tasks first. Then, every time a task is dispatched, *Tromino* recalculates the DDS and decides which dispatcher gets a chance to release the next task. We observe that in the example discussed in Table 1, Framework A has higher demand compared to Framework B. In that particular case scenario, *Tromino* allows the dispatcher corresponding to Framework A to dispatch the task. It cycles until Framework A dispatches 5 more tasks from its dispatcher queue. At this point, Table 5 shows the DDS for both Frameworks. Now, both the Frameworks have similar DDS, but Framework A cannot launch any tasks as its resource demands cannot be satisfied with the available resources in the cluster. Thus, Framework B's dispatcher dispatches one task from the queue. *Tromino* stops this cycle and waits for resources to once again become available so that they could be used in the next cycle. After this cycle, the DDS for both frameworks are presented in Table 6. In the next dispatching cycle, Framework A may get priority if it still has a higher DDS than Framework B. The DDS of Framework B may go up if it gets new tasks before the next cycle.

$$DDS_A = max[(5*1)/20, (5*4)/40] = max[0.25,\ \mathbf{0.5}]$$
$$DDS_B = max[(5*2)/20, (5*1)/40] = max[\mathbf{0.5},\ 0.12]$$

Table 5. *Dominant Demand Share after Tromino dispatches 5 more tasks from Framework A's dispatcher.*

$$DDS_A = max[(5*1)/20, (5*4)/40] = max[0.25,\ \mathbf{0.5}]$$
$$DDS_B = max[(4*2)/20, (4*1)/40] = max[\mathbf{0.4},\ 0.1]$$

Table 6. *Dominant Demand Share after Tromino dispatches 5 tasks from Framework A's dispatcher and 1 task from Framework B's dispatcher.*

- ***Demand and DRF Aware Policy.*** In this approach, we consider both the demands of each framework and their dominant share. Scheduling based just on the demand may cause unfairness. A framework could end up consuming the entire cluster due to its higher demand while another framework that has significantly fewer number of tasks to execute could starve for resources. We have combined both the dominant share and dominant demand share to generate a Demand-DRF factor in each cycle to decide the number of tasks to be dispatched from each framework's dispatcher.

Software	Version
Ubuntu	Ubuntu 16.04.2 LTS (Xenial)
Apache Aurora	17.06.0-ce
Marathon	1.4.0
Apache Mesos	1.3.0

Table 7. *Software Stack and Version*

IV. Experimental Results and Evaluation

For our experimental setup, we have considered two widely known Mesos frameworks, Apache Aurora and Marathon, along with Scylla, a framework developed by our team. The cluster consists of 8 nodes each with 8 CPUs and 16 GB of memory. We have instrumented *Tromino* to receive tasks for Apache Aurora, Marathon, and Scylla at a different task arrival rate. We have kept the resource requirements of each task identical (i.e., $< 0.5\ CPU, 1\ GB\ memory >$). The cluster at its peak utilization can execute 128 tasks with such requirements. As all the tasks are identical, the number of tasks that each framework is executing at any instance of time is proportional to the amount of resources consumed by that framework. Our aim with the experiments is to understand the way resource fairness and task awaiting time varies in different case scenarios. Also, we want to examine how *Tromino* policies can achieve better cluster-wide fairness and a reduced average waiting time, over Mesos' default DRF implementation. Our experimental results show the unfairness caused in a Mesos cluster and quantify the fairness in terms of average waiting time for different case scenarios.

A. ***Experiment 1.*** *Framework with default configurations and different arrival rates.*

In this experiment, we present a case scenario where Mesos' default DRF based allocation fails to provide cluster-wide fairness due to each framework's varying attributes and task arrival rates. We have instrumented *Tromino* to receive tasks for Aurora at a slower rate and receive tasks for Scylla at a higher frequency. Aurora's default implementation enforces holding resources for a period of time for better scheduling of tasks. On the other hand, Marathon is configured with a relatively greedier second level scheduling policy compared to Aurora. The second level scheduling can significantly affect a framework's individual resource utilization. In our particular case scenario, due to a greedier scheduling policy, Marathon is able to orchestrate more tasks upon receiving resources offers from the Mesos Master. So, Marathon's greedier scheduling policy, and Aurora's characteristic of holding on to resources, affects Aurora because it struggles to launch a fair number of tasks. For Aurora, holding resources makes its Dominant Share stay higher, even though the resources are not used for scheduling tasks. Unlike Aurora, Scylla does not hold resources, and the second level scheduling policy is less greedy compared to Marathon. In Table 8, we show the number of tasks for each framework along with task arrival rate and attributes that can impact the resource distribution.

	# of tasks	Arrival Rate (sec)	Attribute
Marathon	1000	1	Bin-Packing
Scylla	700	1.5	–
Aurora	500	2	Holds resources

Table 8. *Configuration: Aurora, Marathon and Scylla with different task arrival rate along with attributes that may affect the resource fairness in the cluster.*

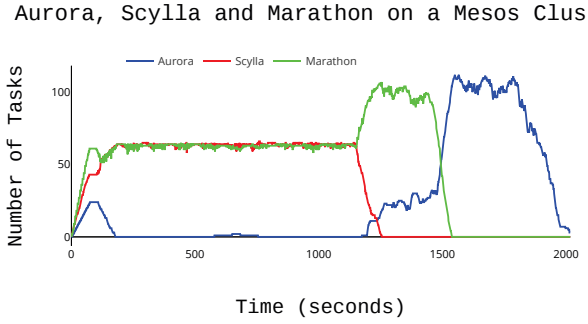


Figure 7. *Aurora is not able to launch its pending tasks until Marathon and Scylla are done with executing their tasks.*

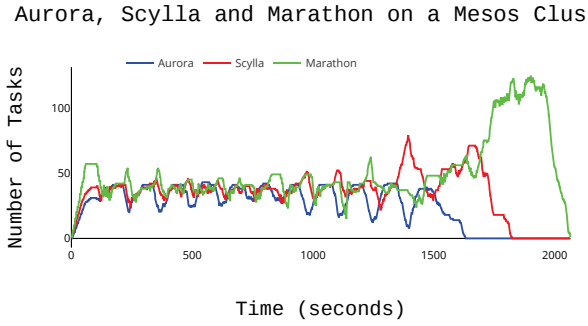


Figure 8. *Tromino improves the fairness by incorporating DRF-Aware policy in the task dispatcher.*

Figure 7 shows the fairness graphs of Aurora, Marathon, and Scylla competing for resources. Aurora could not launch a fair number of tasks. This can be attributed both to its default configuration of holding on to offers without using them for a long period of time and the other competing framework's greedy second level scheduling. We have incorporated DRF-Aware scheduling in *Tromino* to address such scenarios.

The results in Figure 8 show that each of the frameworks can launch close to a fair number of tasks, which is 42 in the cluster. In the following experiments, we configured *Tromino* with DRF awareness as the baseline to compare other *Tromino* policies in different case scenarios.

B. ***Experiment 2****: Frameworks with equal number of tasks, but with fast and slow arrival rates.*

In this experimental setup, we have instrumented Aurora, Marathon, and Scylla to launch tasks in our experimental

Mesos cluster. Tromino receives tasks for Aurora at a faster rate than Scylla, and Marathon at a slower rate than Scylla as shown in the Table 9. All three frameworks receive an equal number of tasks to be executed. Each task is identical in terms of resource requirement as mentioned in section IV. The fair number of tasks at any point in time for each framework is 42.

	# of tasks	Arrival Interval (sec)
Aurora	733	1
Marathon	733	1.5
Scylla	733	2

Table 9. *Configuration: Aurora, Marathon and Scylla with different task arrival rate for launching same number of tasks.*

Figure 9a, 9b, and 9c show the fairness plots when *Tromino* is configured with different task dispatching policies in the cluster. *Tromino* receives Aurora's tasks at a faster rate than Marathon and Scylla's tasks. During DRF-Aware policy configuration, Aurora faces a higher waiting time compared to Marathon and Scylla. Aurora is affected by a 44% higher waiting time compared to the cluster's average for all tasks. For the Demand-Aware policy, Aurora's average waiting time is reduced by 30% below the cluster's average. However, due to the lower task demand from Scylla, *Tromino* increased its waiting time to 27% above the cluster average. For Marathon, both the policies' average waiting time stays within 10% the cluster's average. In Demand-DRF-Aware policy, the average waiting time of the other two frameworks is within 2% of the cluster's average. Figure 10a presents the total waiting time for all three frameworks for different *Tromino* policies. Similarly, Figure 10b shows and compares the average waiting time per every 100 tasks to be scheduled by each framework for each *Tromino* policy. Lastly, Figure 10c compares the total waiting time for each policy for all the tasks in the cluster. Table10 provides the results for the above mentioned figures.

	Aurora	Marathon	Scylla
DRF Aware	44.24%	-6.37%	-37.87%
Demand Aware	-30.42%	2.57%	27.85%
Demand-DRF Aware	**-1.06%**	**1.19%**	**-0.13%**

Table 10. *Result: Difference between average waiting time of each framework from average waiting time of the cluster for different Tromino policies in Experiment 2.*

C. ***Experiment 3****: Large number of tasks with higher arrival rates, and lower number of tasks with slower arrival rates.*

In this experimental setup, *Tromino* receives fast arriving tasks for Aurora, slow arriving tasks for Scylla, and Marathon's task arrival rate is in between Aurora and Scylla's rate. Tromino receives a higher number of tasks for Aurora and fewer tasks for Scylla compared to the number of tasks received for Marathon. The task arrival rate and the number of tasks for each framework is mentioned in Table 11. *Tromino* is configured with all three policies as discussed in section III-C. In Figures 11 and 12, we present our observations about resource fairness and how waiting time varies for all the policies.

	# of tasks	Arrival Interval (sec)
Aurora	1000	1
Marathon	700	1.5
Scylla	500	2

Table 11. *Configuration: Tromino receives more tasks and at a fast rate for Aurora, and lesser number of tasks at a slower rate for Scylla.*

Figure 11 shows the resource fairness for all three frameworks after configuring *Tromino* with all three policies. In DRF aware policy configuration, Aurora's average waiting time is 73% more than the overall average waiting time of the cluster. For Scylla, with slow arriving tasks, the waiting time is 55% less. After changing the configuration to follow the Demand-Aware policy, the average task waiting time difference changed to 31% less and 34% more for Aurora and Scylla respectively. The average waiting time difference for all three frameworks is aligned better with the cluster's average when *Tromino* is configured with Demand-DRF aware policy.

Figure 12a presents the total waiting time for all three frameworks for different *Tromino* policies. Similarly, Figure 12b shows and compares the average waiting time per every 100 tasks to be scheduled by each framework for each *Tromino* policy. Lastly, Figure 12c compares the total waiting time for each policy for all the tasks in the cluster. Table 12 provides the results for the mentioned figures.

	Aurora	Marathon	Scylla
DRF Aware	73.33%	-18.16%	-55.17%
Demand Aware	-31.07%	-3.30%	34.37%
Demand-DRF Aware	**2.30%**	**-1.42%**	**-0.88%**

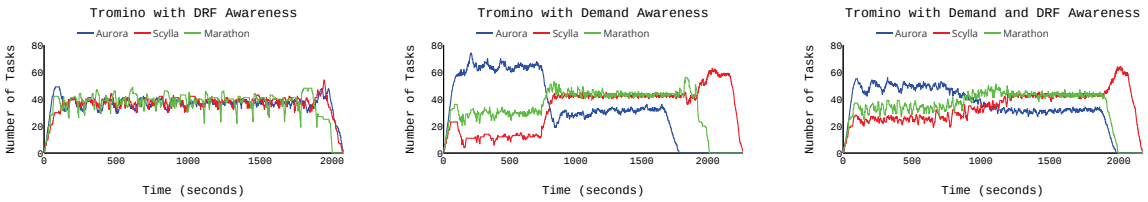
Table 12. *Results: Difference of average waiting time of each framework compared to average waiting time of the cluster for different Tromino policies in Experiment 3.*

D. ***Experiment 4****: Large number of tasks with slower arrival rates, and lower number of tasks with faster arrival rates.*

In this experimental setup, a fewer number of Aurora tasks are received by *Tromino* at a faster arrival rate, and unlike the previous experimental setup, *Tromino* receives more Scylla tasks at a slower rate. In Table 13, we present the number of tasks received for each framework and the arrival rate.

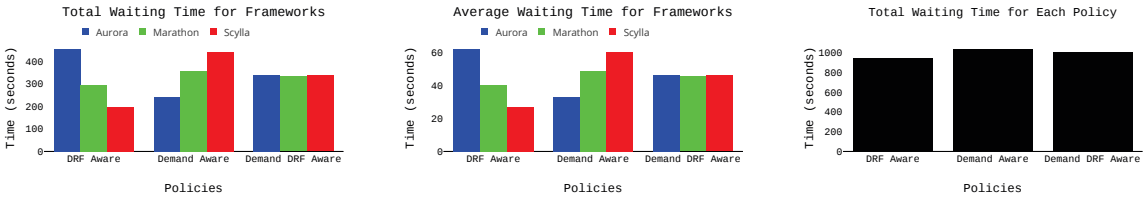
	# of tasks	Arrival Interval (sec)
Aurora	500	1
Marathon	700	1.5
Scylla	900	2

Table 13. *Configuration: Fewer tasks, but at a faster rate, for Auroras tasks; and a larger number of slow arriving Scylla tasks for Experiment 4.*



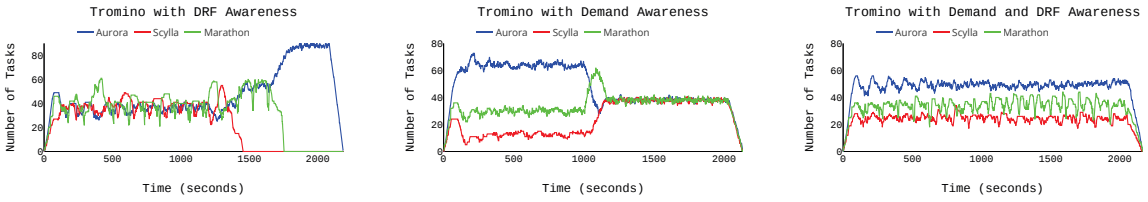
(a) *Fairness Graph when Tromino is DRF aware* (b) *Fairness Graph when Tromino is Demand Aware* (c) *Fairness Graph when Tromino is Demand and DRF aware*

Figure 9. *Resource Fairness for Experiment 2: Results show the fairness obtained by the cluster when Tromino is configured with different policies and equal number of tasks are launched in the cluster with different task arrival rates for each framework.*



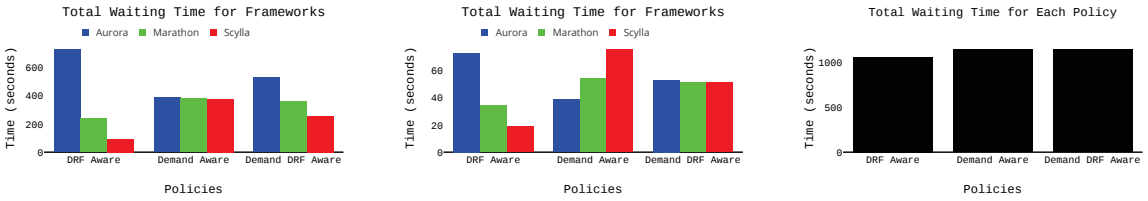
(a) *Waiting Time for Each Framework.* (b) *Average Waiting Time of 100 Tasks for Each Framework* (c) *Total Waiting Time for Each Policy*

Figure 10. *Results for Experiment 2: Equal number of tasks are launched by Aurora, Marathon and Scylla in a Mesos cluster. Experimental results show how total waiting time and average waiting time varies for Aurora, Marathon and Scylla when Tromino is configured with different policies.*



(a) *DRF Aware.* (b) *Demand Aware* (c) *Demand and DRF aware.*

Figure 11. *Resource Fairness for Experiment 3: Resource fairness obtained when Tromino is configured for different policies. Tromino receives more tasks for Aurora at a fast rate whereas Scylla's tasks arrive at a slower rate and are lesser in number.*



(a) *Waiting Time for Each Framework.* (b) *Average Waiting Time for Each Framework* (c) *Total Waiting Time of 100 Tasks for Each Policy*

Figure 12. *Results for Experiment 3: Results show how total waiting time and average waiting time varies for Aurora, Marathon and Scylla when Tromino is configured with different policies. Tromino receives higher number of tasks for Aurora in a higher arrival rate than Marathon and Scylla (Table: 11).*

Figure 14a presents the total waiting time for all three frameworks for different Tromino policies. Similarly, Figure 14b shows and compares the average waiting time per every 100 tasks to be scheduled by each framework for each *Tromino* policy. Lastly, Figure 14c compares the total waiting time for each policy for all the tasks in the cluster. Table 14 shows the difference of average waiting time of each framework with each policy configuration compared to the overall cluster's average waiting time.

	Aurora	**Marathon**	**Scylla**
DRF Aware	16.67%	7.61%	-24.28%
Demand Aware	-35.93%	8.78%	27.15%
Demand-DRF Aware	**-10.70%**	**4.03%**	**6.67%**

Table 14. *Result: Difference of average waiting time of each framework compared to average waiting time of the cluster for different Tromino policies in Experiment 4.*

V. Related Work

We have proposed a few policies to evaluate the fairness of an Apache Mesos cluster based on average waiting time. In our previous work [13] [14] we have shown how Apache Mesos can be integrated with scientific workflow managers like Apache Airavata [15] to run science application through Docker containers [16]. The community can take advantage of Mesos based fairness to distribute resources across users.

Khaled et al. [17] designed and developed Resource Demand Aware Scheduling (RDAS) for scientific workflows to reduce the overall completion time. RDAS considers the structure of workflows and based on the resource demands of each stage it tries to optimize the resource allocation for better throughput. However, in our Mesos cluster, we have considered short living tasks from different users with specific resource requirements and scheduled them based on the overall demand from each user.

Boyang et al. [18] developed *R-Storm*, which is aware of the resource demand and availability in a Storm based stream processing environment to increase the overall throughput of the cluster. Multiple Storm applications in a cluster yield better performance in the presence of R-Storm than the default Apache Storm configuration. Fahad R et al. [19] developed *Baarat*, a task aware scheduler over the network, which dynamically schedules multiple tasks together based on the task's network bandwidth requirements. It dynamically changes the level of multiplexing in the network to optimize the average and tail completion time for data center applications.

VI. Conclusion

- Individual framework configuration and attributes such as offer holding period and second level scheduling policy can impose unfairness in a Mesos cluster. DRF aware task dispatching by *Tromino* can overcome the unfairness and establish better fairness distribution in the cluster.
- A Framework with a higher task demand needs to get more resources than a framework with a lesser demand to keep the overall waiting time low. *Tromino* can schedule tasks based on the resource demand and current resource consumption of frameworks in the cluster.
- We orchestrated frameworks with different task arrival rates and different number of tasks to execute. Demand awareness is vital to optimize the average waiting time for each framework.
- Demand and DRF awareness on top of Mesos' default DRF based resource allocation can decrease the average waiting time for a framework.

VII. Future Work

In the scope of our current work, we have developed a queue manager, *Tromino*, external to Apache Mesos, which can dispatch tasks based on the dominant share and demands by monitoring the cluster information and pending task queues. Mesos' allocation module does consider the resource demands from each user. However, it can be extended and a new allocation module can be designed that checks the available resources on each agent and can allocate resources based on the demands. In a production environment, where thousands of nodes are configured, scanning through all the nodes with its available resources can take longer time for a single allocation cycle. It will be useful to study the trade-offs between demand aware allocation for meeting better resource constraints against current random resource allocation that may not fit and wait for future cycles to get a better allocation.

We would like to investigate and develop new policies to consider not only total resource demands but also the demand of each task at a more finer granularity. Tasks of different frameworks can have different resource requirements that can differ in the magnitude and may come with priorities. We would like to develop and test new policies to consider all such task constraints of a data center for further improvement and better resource allocation.

VIII. Acknowledgements

This work is partially supported by National Science Foundation, through the OAC-1740263 award.

References

[1] 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.

[2] 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.

[3] "Apache Aurora." [Online]. Available: http://aurora.apache.org/

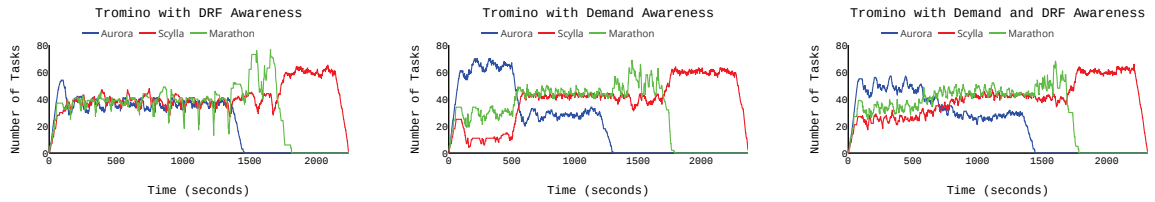
[4] "Marathon: A container orchestration platform for Mesos and DC/OS." [Online]. Available: https://mesosphere.github.io/marathon/

[5] "Chronos: Fault tolerant job scheduler for Mesos." [Online]. Available: https://mesos.github.io/chronos/

[6] 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.

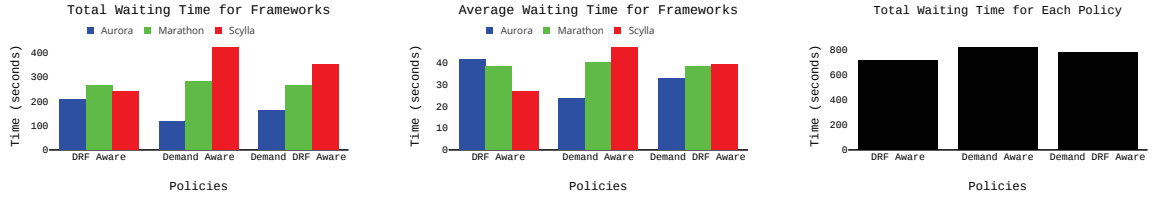
[7] "Mesos has been simulated to scale to 50,000 nodes, although it is not clear ho... — Hacker News." [Online]. Available: https://news.ycombinator.com/item?id=10228820

[8] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.



(a) *DRF Aware.* (b) *Demand Aware* (c) *Demand and DRF aware.*

Figure 13. *Resource Fairness for Experiment 4: Resource fairness of the Mesos cluster for different Tromino policies. Tromino receives fewer tasks for Aurora at a fast rate whereas Scylla's tasks arrive at a slower rate but are more in number.*



(a) *Waiting Time for Each Framework.* (b) *Average Waiting Time of 100 Tasks for Each Framework* (c) *Total Waiting Time for Each Policy*

Figure 14. *Results for Experiment 4: Results show how total waiting time and average waiting time varies for Aurora, Marathon and Scylla when Tromino is configured with different policies and tasks arrive at different arrival rates (Table 13).*

[9] P. Saha, A. Beltre, and M. Govindaraju, "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

[10] "Apache Aurora Thermos Runner." [Online]. Available: http://http://aurora.apache.org/documentation/latest/development/thermos/

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[13] 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.

[14] ——, "Multicloud resource management using apache mesos with apache airavata," 1 2017. [Online]. Available: https://figshare.com/articles/MultiCloud_Resource_Management_using Apache_Mesos_with_Apache_Airavata/4491629

[15] S. Marru, R. Gardler, A. Slominski, A. Douma, S. Perera, S. Weerawarana, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, and E. Chinthaka, "Apache airavata," in *Proceedings of the 2011 ACM workshop on Gateway computing environments - GCE '11*. New York, New York, USA: ACM Press, 2011, p. 21. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2110486.2110490

[16] 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.

[17] 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.

[18] 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.

[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 431–442.