# Resource Revocation in Apache Mesos

Stephen Twigg, Huy Vo
{sdtwigg|huytbvo}@eecs.berkeley.edu
December 17, 2012

## 1. Abstract

*We demonstrate how adding resource revocation to Mesos allows the system to provide latency and resource guarantees to frameworks. Mesos, which uses dominant resource fairness to offers of new resources, was designed initially for primarly MapReduce-like and found to provide weak guarantees with more general workloads. This project resolved this issue by allowing frameworks to explicitely state resource minimums needed to constrain their task latency and then use these constraints to guide Mesos in revoking resources. This solution both minimizes work lost from revocation and leverages the pre-existing DRF algorithm in Mesos to already provide long-term fairness. Offer revocation after a timeout was employed to minimize resource waste due to indecisive frameworks. These adjustments were evaluated on a single machine using synthetic benchmarks designed to simulate problem scenarios that occur when running Mesos in production on large 1000+ node clusters. Evalutation demonstrated frameworks running latency sensitive tasks were able to start new jobs on a fully-utilized Mesos cluster employing revocation as if they were the only framework in the system even in the presence of resource-heavy or indecisive frameworks.*

## 2. Introduction and Motivation

As the data analytics and processing needs of companies grow, systems designers are looking more and more to using clusters in order to provide for those needs. Clustering of compute resources permits more efficient allocation of compute resources to tasks relative to naïve static partitioning by allowing unused resources to quickly and easily be shared for use by other tasks.

Clustering allows companies to more effectively deal with dynamic workloads by giving compute clients the ability to pickup or shed resources over time as their computational requirements shift. For example, during the day a retailer has to devote substantive processing power to ensure all transactions are completed in a timely manner such that the retailer can do real business. However, during the evening when transaction loads are light, the excess resources are better spent doing data analysis on the transaction data to track buying trends and better inform managers for making business decisions. These clusters need specific cluster management software that maps compute nodes to clients and provides administrators a convenient point to monitor the status and utilization of all agents in the system.

### 2.1. Apache Mesos

Apache Mesos is one such cluster manager that was originally developed by the UC Berkeley AMP Lab and is currently in use to manage clusters at Twitter and Conviva [1, 3]. Mesos uses an event-driven architecture and is written in C++ and built on top of the Libprocess and Zookeeper libraries. This paper will explain enough of the Mesos architecture to understand the change implemented in the project; however, please refer to the original paper [3] for the particular details on other parts of the system.

The Mesos API model does not require frameworks to specify the fine details of what resources the participating frameworks (the clients) need. Rather, Mesos compiles a listing of available resources at nodes into an offer that it then presents to a framework. The framework may then analyze the resources in the offer and then cherrypick the specific nodes and associated resource needs onto which Mesos should launch new tasks for the framework. This allows frameworks to fulfill arbitrarily complex resource allocation requirements without complicating the Mesos scheduler. For example, by tracking its previous allocations, frameworks could ensure its tasks are all located on the same rack in the system (for better IPC speed) or far away from each other (for better resiliency to failure), only launched on nodes with internet access (for webservers), or, for Hadoop-like tasks, preferentially launched on nodes already containing the data to be processed (to minimize data fetch time).

To enforce fairness amongst all frameworks, the allocator in Mesos implements dominant resource fairness [2] and only makes new offers to frameworks in order of increasing dominant share. Over time, this is expected to balance out resource use in the system amongst all active frameworks. Mesos has been found to be highly scalable (up to 50000 nodes) and competitive when run against static partitioning for various Hadoop and Spark workloads.

## 2.2. Fairness Problems with Mesos

Producing fairness by ordering offers to frameworks, unfortunately, does not work well in the general case. The original evaluation of Mesos focused on MapReduce or MapReduce-like jobs, which are generally characterized by having a small footprint and, critically, low execution times. Thus, for any small interval of time during those tests, tasks were ending thus freeing up resources on nodes allowing offers to be made to undersubscribed frameworks.

Unfortunately, this behavior is not guaranteed in general, particularly in systems containing frameworks with 'long' running tasks. Assuming these same frameworks are capable of spawning speculative execution (in order to avoid letting cluster resources go to waste during times of low interest), all system resources will be reserved for extended periods of time leaving new frameworks (or equivalently, other frameworks that suddenly need a bunch of resources to respond to a workload burst) unable to get resources in a timely manner. These high latencies may be unacceptable for interactive frameworks that must launch tasks immediately to respond to user queries. An example of offer timings unfarily impacting task latencies between frameworks is illustrated in Figure 1(a). In that example, even though both frameworks enter the system at roughly the same time, the first framework received an offer first thus allowing it to grab all remaining resources and completely starve the second framework.

Implementation details of Mesos also reveal other, more subtle fairness issues that are not properly resolved by the basic offer model. The resource allocator in Mesos takes a snapshot of all resources currently allotted to frameworks right as it must decide to whom the offer must be directed and always selects the one with the lowest dominant share at that instant. Thus, a framework can arrange to almost always capture the next offer by scheduling all tasks to end within Mesos' one second scan interval. Since Mesos does not restrict how much of an offer is taken, the greedy framework is thus able to relaunch onto all the resources currently used up plus anything else freed during the update. This problem behavior arises more commonly with frameworks that launch very large resource-heavy tasks (which, again, is in stark contrast to light MapReduce tasks).

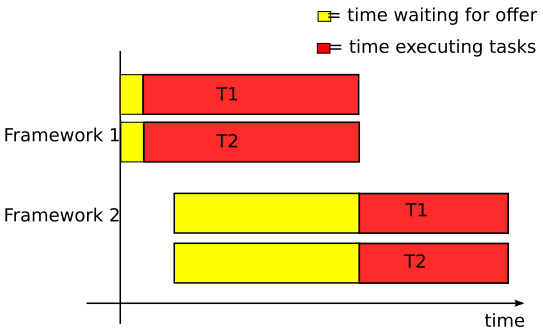
Another issue is related arises in the actual mechanics of how Mesos handles offers. In order to avoid race conditions in accepting offers, Mesos will not simultaneously offer the same resource to different frameworks. Since Mesos generally includes all available resources when constructing offers, frameworks that take an excessive period of time deciding on offers will end up wasting those offered resources in a contentious system.

While admittedly many of these issues would only seem to arise because of a 'maliciously' written framework, many times the difference between buggy or flawed design and malicious design is merely semantic. For example, an offer response may be inappropriately delayed due to a framework crash. Alternatively, greedy frameworks may have just been designed to always fill up large portions of an offer with speculative work under the assumption that the large offer means the system is unloaded and thus the resources would just to waste otherwise. Thus, even for completely closed deployments of Mesos, resolving these issues will lead to a more stable system overall.

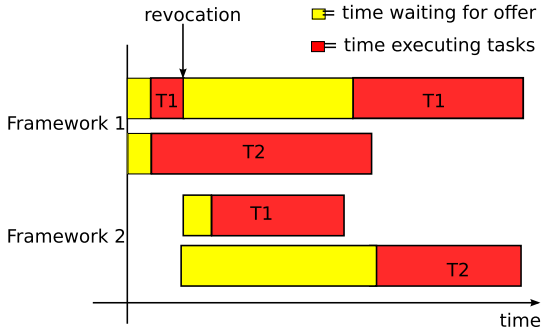
# 3. Methodology

## 3.1. Proposed Solution

Acknowledging that a core problem with the offer-based fairness solution of Mesos starts breaking down as the system fills and no resources are available, the




(a) Stock Mesos – No Revocation




(b) Mesos With Revocation

**Figure 1: Comparison of Mesos With and Without Revocation Enabled** Example Mesos cluster with two slots. Both example frameworks have two parallel tasks so they are able to fully utilize the entire cluster. Framework 2 arrives later than the Framework 1. Yellow bars corrrespond to the time that the framework has to wait for an offer from Mesos. Red bar corresponds to the time that framework is running a task on a node.

clear solution is to simply revoke resources from 'over-subscribed' frameworks in order to make resources available. Mesos is expected to be run on a cluster composed of somewhat unreliable nodes and thus frameworks already must be instrumented to cope with tasks failing abruptly. Even so, killing a task effectively causes what is commonly called 'goodput loss' since all time that task had previously spent with those resources is now effectively lost (although raw system utilization numbers may be maxed). Also, not all frameworks cope equally well with task-loss. MapReduce frameworks are able recover very quickly whereas data analysis frameworks using MPI to synchronize work amongst task processes on different machines may require all participating tasks to stall (or worse terminate without saving) when one task is lost. These considerations drive the need for conservative revocations that minimize the amount of revocation done.

Following this goal, this project partly extended the communication model of Mesos to allow frameworks to send a new type of message indicating resource needs. The documentation for this message makes a clear distinction between needs and desires by defining needs as the minimum possible resources that framework requires to function according to its defined specifications and SLAs. Resources desired for doing speculative execution or merely to get through the task queue faster are not and must be acquired through traditional means. Mesos administrators are encouraged to filter these resource need requests to ensure frameworks are making reasonable demands. The Mesos allocator tracks these demands and periodically checks to ensure all frameworks have their demand or, altarnatively, enough free resources exist in the system to cover the needs. If not, Mesos starts revoking tasks from frameworks using resources over their defined need. In order to minimize goodput loss, Mesos revokes tasks from a framework starting with the most recently launched task.

As demonstrated in Figure 1(b) this prevents the subtle timing of offers to frameworks or framework entry in the system from unacceptably impacting the latency until a framework receives usable resources. Frameworks are permitted to adjust their needs over time (for example, interactive frameworks may relinquish demands for resources when no user queries are pending) and the system is designed to respond quickly to these changes. MPI-like frameworks that prefer not to lose tasks can thus insulate themselves from revocation by always keeping their task resource below their posted resource needs whereas MapReduce-like frameworks are still able to speculative execute but with the understanding this leaves them open to revocation and thus higher task-loss rates. The previous issue of frameworks tying up resources by not responding to offers was resolved by revoking offers from frameworks after an exponentially expanding timeout period.

Using resource revocation to only satisfy needs and not balance the system reflects the observation that offer-based DRF was working to balance the system in general only failing due to latency issues in select scenarios. This also relieves from the Mesos the responsibility of determining if a resource-light framework would even use resources revoked from a heavier framework. Whereas offer responses are very specific by detailing precisely which nodes to use resources on, need requests are by design kept general specifying only quantity of a resource type needed but not location. This allows Mesos administrators to more easily ensure all needs for their system are actually satisfiable. Finally, this solution is completely backwards compatible with frameworks designed to use the older communication model. If no frameworks ever submit a resource need request then the revocation subsystem will never turn on. If only a handful of frameworks submit requests, those that did not are still able to run but just more susceptible to being revoked. As a bonus, this provides an incentive (but not mandate) for framework maintainers to keep their code up to date.

### 3.2. Rejected Solutions

An alternative solution would have Mesos submit 'tenative offers' that include currently used resources to frameworks with low dominant shares. Resources in these offers would be specially marked to allow frameworks to show some restraint in asking for currently utilized resources otherwise a greedy framework doing speculation may immediately ask for everything causing a large amount of revocation and corresponding work loss. Already, this solution is less appealing as it requires frameworks to consider impact on other frameworks when responding to these tentaive offers. By comparison, the proposed solution only asks frameworks to construct messages about their own needs. Mesos is then able to use its global view of the system and framework resource needs to make reasoned decisions on who to revoke.

This solution also leads hard-to-resolve resource conflicts and possibly thrashing. If the tentative offers are too small including only resources from the frameworks with large dominant shares, they may not contain enough resources that the offer recipient is able to use the offer. If the tentative offers are too large, the receiving framework may ask for resources used by a only somewhat large framework when it would have been just as satisfied taking resources from the most dominant framework, a clearly more ideal case. Attempts to annotate the offer response wherein the responding framework includes a listing of alternatives would rapidly complicate the communication model(and is likely to cause some frameworks to require even more time to respond to offers). In either

case, this could lead to thrashing as the more dominant frameworks are forced to shrink down in size and then become eligible to receive tentative offers as well. Guaranteed shares could be used to avoid this situation but at that point becomes barely indistinguishable from the proposed solution.

For similar reasons, allowing frameworks to, in general without an offer, explicitely ask for resources on a specific node is not supported in order to prevent two frameworks from constantly thrashing over that specific resource. This may be resolvable as an extension to the proposed guaranteed-based solution. Revocation would only occur if the framework holding the resource is actually oversubscribed with the understanding that, since the framework is still above its guarantee, it can still make some forward progress. Due to time constraints and challenges constructed good test cases, actual implementation and testing of this solution is reserved for future work.

# 4. Implementation

In this paper, we contribute two types of revocation mechanisms to Mesos: offer revocation and resource revocation. The idea behind offer revocation is that Mesos will give frameworks adequate time to respond to an offer it has made. If a framework fails to respond in time, Mesos will forcibly take back the resources it had to tie down to make the offer. With resource revocation, Mesos periodically searches for frameworks that have gone over their guaranteed share. Then, if there are frameworks under their guaranteed share, Mesos will forcibly take the over allocated resources from the frameworks that are over their guaranteed share and give the resources to the frameworks under their guaranteed share. We go into a detailed discussion of our modifications to Mesos below.

## 4.1. Offer Revocation

Since offer revocation does not involve any acknowledgements from a framework, we did not have to change the communication interface between Mesos and a framework. On the Mesos Master side, an offer revocation looks very much like a situation in which a framework has rejected offer so we were able to resuse a lot of the mechanisms that were already in place on Mesos Master. The only major addition we made to Mesos Master is to have Mesos Master start a separate timer task whenever it makes an offer. When the timer goes off, it will place a offer revoked message onto Mesos Master's event queue. When the event handler processes the message, it will put the resources in the offer back on the list of allocatable offers. Note that there is zero possiblity for a race condition in which the offer revocation timer goes off at the same time that a framework responded to an offer. One of those messages, either the offer revocation message or the offer response message, will be enqueued first. If the event handler sees the offer response message first, then the offer revocation message is ignored. If it sees the offer revocation message first, then the offer response message is ignored and the framework is notified. On the framework side, the framework will see that its tasks were never launched onto a Mesos Slave. It is up to the framework to do the appropriate thing (e.g. add the task back to its task queue).

We did not settle on a offer timeout time since we were not sure how long it would take a well behaved framework to respond to an offer. Instead, we start the timeout time at some small time. If a framework fails to respond to resource offer before the timer goes off, we gradually increase the timeout time up to a maximum.

## 4.2. Resource Revocation

Resource revocation is much more complicted. Figure 2 shows the new communication paths that we added to Mesos. At every heartbeat, stock Mesos will perform DRF to determine which framework to make a resource offer to. We modified Mesos to also check the resource utlizations of all the frameworks at every heartbeat. Mesos will calculate the amount of resources that are allocatable ($R_a$), the amount of resources by which each framework is over its guaranteed share ($R_o$), and the amount of resources by which each framework is under its guaranteed share ($R_u$). Then if

$$R_u > R_a$$

Mesos will begin to revoke from the all the frameworks that have gone over their guaranteed share until the total amount of resources revoked ($R_r$) is such that

$$R_u = R_a + R_r.$$

Keep in mind three things. First, Mesos will only revoke resources from frameworks to bring them down to their guaranteed share and never under their guaranteed share. Second, Mesos will allow a framework to be over its guaranteed share as long as all the other frameworks in their system have met their guaranteed share. Finally, we assume that it is possible to revoke resources to bring everyone to at least meet their guarantee. If this is impossible to do, then that means the system is oversubscribed, a problem that resource revocation is not meant to solve.

Once Mesos has determined the amount of resources to revoke from each framework, it will then determine which slave these frameworks are currently running tasks on. It will then figure out how many tasks to kill on each slave such that enough resources are freed. To make this a fast process, we simply have Mesos kill the most recently launched tasks. Once it has figured out which tasks to kill, Mesos sends kill task messages to the slaves. Upon

receiving these messages, the slaves will immediately kill the corresponding task and notify Mesos Master and corresponding framework that a task has been killed. On receipt of this message, Mesos Master will add the resources that the task was using back to the list of allocatable resources.

## 5. Results

### 5.1. Experimental Setup

We ran our experimentation on 8-core 2.67 GHz Intel Xeon processor with 49 GB of usable memory. In all of our experiments, we used 1 instance of a Mesos Master managing 1 Slave in charge of a 8 slot "cluster". Due to compute resource limitations, the experimental setup is not meant to be representative of a real Mesos cluster. However, since we designed our synthetic frameworks to highlight specific problem scenarios in a Mesos cluster and not for big-picture catch-all experiments, the experimental results are still highly beneficial.

In our experiments, we define goodput to be the amount of time that a framework was able to use a resource slot to make progress that is eventually saved. For example, if a task occupies a slot for 30 seconds and after 30 seconds it has completed, then that task has made 30 seconds of goodput over the past 30 seconds. If a task occupies a slot for 30 second and after 30 seconds it was prematurely terminated (because it was killed, for example), then that task has made 0 second of goodput over the past 30 seconds. We measure latency to the difference in time between when a task enters a task queue and when the task completes. If, for example, a task enters framework's task queue at time 100 seconds, is launched on a slaved at time 102 seconds, and then completes at time 130 seconds, it latency will be 30 seconds (130 seconds - 100 seconds).

### 5.2. Offer Revocation

The goal behind offer revocation is to minimize the impact that malicious or poorly coded frameworks have on the system. In our experiments, if all the frameworks in the cluster are well behaved, then offer revocation has no effect on the cluster at all, which is the desired behavior. We also introduced frameworks that take a lengthy amount of time to make a decision on an offer as well as frameworks that never respond to offers. With offer revocation, well behaved frameworks were able to make forward progress.

### 5.3. Resource Revocation

For this experiment, we designed two frameworks. The first framework, Framework 1, models a greedy data analysis framework that wants as much compute resources as possible. It launches long lived tasks that lasts 3 minutes. If allowed to run in isolation, it will easily take up a large

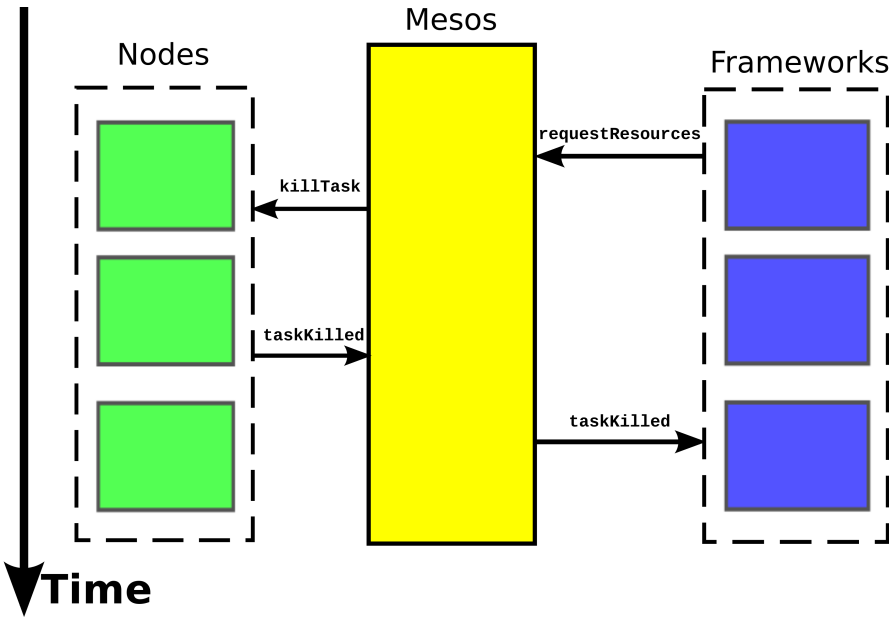
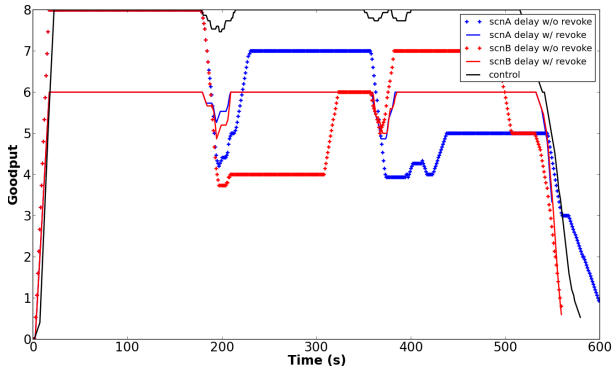



**Figure 2: Timeline of Communication From Frameworks to Mesos to Nodes** Time flows top to bottom, that is, communication arrows closer to the top happen before communication arrows closer to the bottom. When a framework enters the Mesos cluster, it will send a resourceRequest message to Mesos. Mesos then determines if it needs to perform revocation. If it does, then it will send killTasks messages to the nodes. Once the nodes have killed the tasks, it will then send a taskKilled message to Mesos. Finally, a Mesos sends a killedTask message to the corresponding framework to inform it that its task was killed.

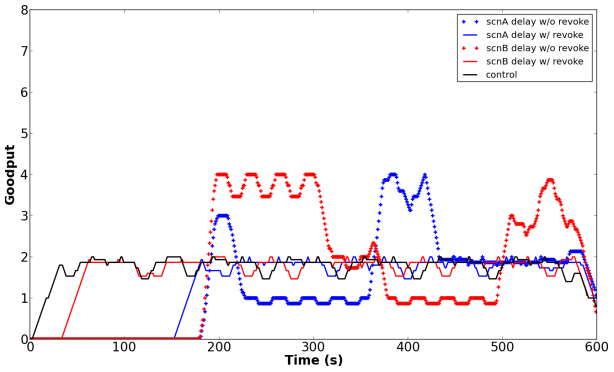
share of the cluster. In our experimental setup, we allowed Framework 1 to take up all 8 slots. Framework 2, on the other hand, models reactive realtime jobs. Every 15 seconds it generates and launches 2 short-lived tasks spanning 30 seconds each. If allowed to run in isolation, this framework will only use up two slots in the cluster.

Before diving into our experiments, we first allow Frameworks A and B to run in the cluster isolation to verify that they behave as intended and to have a good reference point for our experiments. The solid black lines in Figure 3(a) and Figure 3(b) correspond to achievable goodput results when the frameworks are allowed to run by themselves. As expected, Framework 1 will eat up all 8 slots while Framework 2 will never take up more than 2 slots even if Mesos offers it more resources. The solid black lines in Figure 4(b) and Figure 5(b) corresponds to the achievable latency when Framework 2 is allowed to run by itself in the cluster.

We closely examined two extreme corners of operation. Figure 4 documents our results for Scenario A. In this scenario, we have Framework 1 enter the system first. Since it is the only framework in the system, it quickly takes up all 8 slots in the system. We then have Framework 2 enter the system 150 seconds later. We first examine how our system behaves without resource revocation. We'll then show how resource revocation improves the performance of our system. Recall that Framework's A task each last 3 minutes. This means that without re-

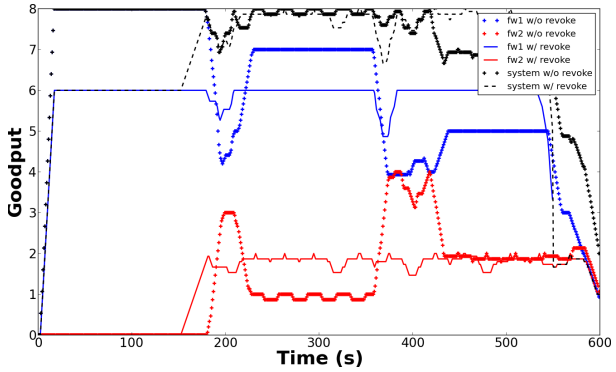



(a) Goodput Results for Framework 1

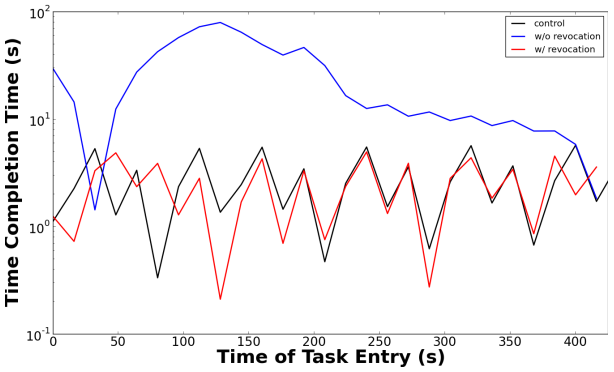



(b) Latency Results for Framework 2

**Figure 3: Goodput Results For Frameworks 1 and 2** Figure 3(a) and Figure 3(b) plots the goodput results for Framework 1 and Framework 2 respectively. The black lines correspond to the case where the frameworks are running in isolation. The blue lines correspond to Scenario A while the red lines correspond to Scenario B. Solid lines correspond to the case where resource revocation is used while + lines correspond to the case when no resource revocation is used.




(a) Goodput Results for Scenario A



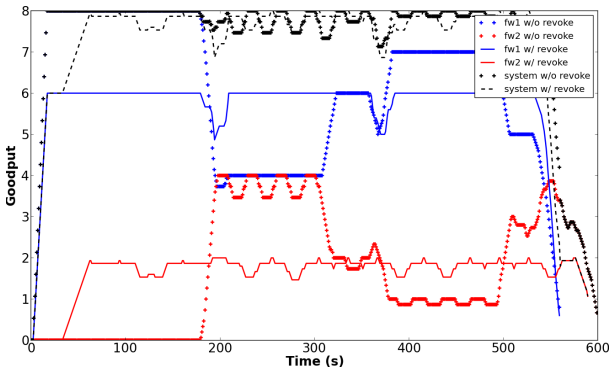

(b) Latency Results for Scenario A

**Figure 4: Goodput and Latency Results for Scenario A** In Figure 4(a), black lines correspond to the system's goodput, blue lines correspond to Framework 1's goodput, and red lines correspond to Framework 2's goodput. For the system goodput result, the + lines correspond to the case without resource revocation and the dashed line corresponds to the cae with resource revocation. For the frameworks, the + lines correspond to the case without resource revocation while the solid lines correspond to the case with resource revocation. Figure 4(b) plots Framework B's task latency when it is running in isolation (the black line), when it is running without resource revocation (the blue line), and when it is running with resource revocation (the red line).

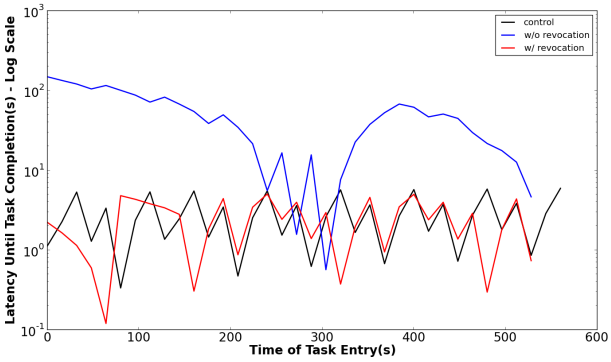
source revocation, Framework 2 would only have to wait 30 seconds for Framework 1's task to finish at which point Mesos can then make an offer to Framework 2. Furthermore, since Framework 2 generates tasks every 15 seconds, regardless of whether or not Mesos is able to make it a resource offer, by the time Mesos is able to make it an offer, Framework 2 will have a pile up of tasks in its task queue. It will thus grab a larger share of the system. This behavior can be seen by the red + line in Figure 4(a). Notice that at 180s, this line will jump up to 3. It will then quickly drop to 1. This is because Framework 2 only requires 2 slots to keep its task queue empty and having more slots means that it will complete tasks faster than it can generate tasks. It will therefore relinquish two of its three slots to Framework 1. This will cause the tasks in its task queue to pile up again. So then at time 360, when Framework 1's second set of tasks completes, Framework 2 will take up four slots in the system. This behavior can also be seen by Framework 2's latency (the solid blue line in Figure 4(b)). Notice how initially Framework 2's tasks have latency an order of magnitude larger than the control case (the solid black line). However, this latency will quickly drop (because Framework 2 has a much larger share of the system than it needs) before spiking up again (because it has a smaller share of the system than it needs).

Now we'll examine what happens when we allow Mesos to revoke resources. The solid blue and red lines indicate the behaviors of Framework 1 and Framework 2 with revocation respectively. First, notice that Framework 2 immediately starts at time 150, when it enters the system whereas before it was not able to start until time 180, as indicated by the red + line. Furthermore, Framework 2's goodput remains steady throughout its run, fully utilizing its two slots in the system. And finally, Framework 2's tasks experiences latency that matches closely with the control case (compare the red line to the black line in Figure 4(b)). Although resource revocation improves Framework 2's performance, it also introduces some drawbacks into the system that were not previously there. Compare the system's goodput with revocation (the dashed black line in Figure 4(a)) to the system's goodput without resource revocation (the + black line in Figure 4(a)). Notice that there is a drop in goodput from time 0 to time 150. This is due to the fact that two of Framework 1's tasks are killed at time 150. Thus any work that those two tasks made from time 0 to time 150 is lost. However, this is the only time that the system loses goodput.

In our second scenario, Scenario B, we again had Framework 1 come into the cluster first and allow it to take up the entire cluster. The difference this time is that we had Framework 2 come into the system much earlier, at 30 seconds. Our results for this experiment are in Figure 5. Again, we first examine the performance of our system without resource revocation. Since Framework 2 enters the system at 30 seconds and since Framework 1's tasks each take 3 minutes to complete, Framework B has to wait 150 seconds before it is receives an offer from Mesos. At this time, Framework 2 will have built up a lot of tasks in its task queue. So when Framework 2 finally gets an offer from Mesos at time 180 seconds, it will take up four slots, demonstrated by the red + line in Figure 5(a). Similar to the behavior observed in Scenario A, Framework 2 will again only hold on to this large share of the cluster for a small period of time. Note; however,




(a) Goodput Results for Scenario B




(b) Latency Results for Scenario B

**Figure 5: Goodput and Latency Results for Scenario B** In Figure 5(a), black lines correspond to the system's goodput, blue lines correspond to Framework 1's goodput, and red lines correspond to Framework 2's goodput. For the system goodput result, the + lines correspond to the case without resource revocation and the dashed line corresponds to the cae with resource revocation. For the frameworks, the + lines correspond to the case without resource revocation while the solid lines correspond to the case with resource revocation. Figure 5(b) plots Framework B's task latency when it is running in isolation (the black line), when it is running without resource revocation (the blue line), and when it is running with resource revocation (the red line).

that in Scenario B, Framework 2 will hold onto this larger share of the cluster for a much longer period of time due to the fact that more tasks will have accumulated in its task queue. Finally, in both scenarios, Framework 2's tasks will experience similar latency behaviors. The major difference is that in Scenario B, Framework 2's tasks will have latency that are 2 orders of magnitude larger than the control (compare solid black lines in Figure 4(b) and Figure 5(b)).

With resource revocation, Framework 2 will have much better performance in terms of goodput and latency. Examing the solid red line in Figure 5(b), we can see that Framework 2 starts making goodput right as it enters the cluster instead of having to wait until time 180 seconds when Framework A's tasks will complete. Figure 5(b) also shows that Framework 2's task's latency matches closely with that of the control (compare the black line to the red line). Again, resource revocation does come with a drawback. Comparing the dashed black line to the + black line in Figure 5(a), we can see that system does lose goodput. However, the amount loss is very neglible compared to the amount lost in Scenario A. This is due to the fact that the tasks that Mesos killed have only made 30 seconds of progress so only 30 seconds of work was lost.

The idea behind these two experiments was to have Scenario A show the case when resource revocation could possibly hurt the system more than help while Scenario B would highlight the case when the system would desperately need resource revocation to ensure fairness. However, our results show that resource revocation is beneficial in both cases. Specifically, resource revocation keeps the system in a much more stable state, as demonstrated by Figure 3. Notice that the solid lines in both Figure 3(a) and Figure 3(b) are stable while the + lines in both figures tend to oscillate. Futhermore, resource revocation keeps the latency of later arriving frameworks identical to the latency they would experience if they were running by themselves in the cluster. These performance gains outweigh the loss caused by the transient goodput drop introduced when the newer frameworks enter the cluster.

## 6. Related Work

These considerations are extensions of the considerations made in the original Mesos paper and focus primarily on the impact of revocation. Please refer to [3] for discussion of how basic Mesos compare other cluster managament solutions.

**HPC and Grid Schedulers.** These systems generally do not do any form of unscheduled revocation. Jobs submitted are run until finished or killed after a defined timeout period. If resources are not available to execute a job then the job is placed into a queue and forced to wait until it reaches the head of the queue and enough resources are available. This can lead to significant unpredictable latency for jobs that is highly dependent on system load. Priority in the system for jobs can only affect whether jobs are able to skip past other, lower-priority jobs in the queue. Some HPC systems will reserve nodes for jobs that are interactive or high-priority but this is inefficient when there are not enough high priority jobs are running in the system to use all the reserved slots.

**Clouds.** Features of virtual machines provide convenient alternatives to direct revocation. Virtual machines instead may be suspended, saved, and re-executed later during when the system is less overloaded or moved to less congested parts of the cluster, in theory. Even so, the overhead of launching and destroying virtual machines would make users less reluctant to actually spin off work to . Also, in practice, clusters such as AMazon EC2 are not instrumented to know certain instances are speculative, instead being forced to treat them all equally. would rely on the client to detect the congestion itself and suspend its own speculative instances. Thus, most responsibility falls to the client to police its own tasks and suspend speculative instances during times of congestion (which may be hard to detect at the client level). Undersubscribed frameworks running in congested parts of the system are simply forced to deal with the corresponding performance penalty.

**Quincy.** Quincy is designed specifically for running MapReduce-like tasks. This constrained set of applications allows the system to make more assumptions on framework behavior than Mesos is capable of making. Since the Quincy cluster manager knows directly which tasks are speculative, that jobs are highly resilient to task loss, and if jobs are willing to accept new resources, much better revocation decisions can be made and thus Quincy does use task preemption to enforce fairness in general. Unfortunately, not all frameworks run on Mesos follow the model of computation assumed by Quincy thus restricting the applicability of these solutions.

**Condor.** Now known as HTCondor, this system is basically an extension of the batch schedulers used in HPC and grids. Thus, like those systems, Condor cannot provide strong latency guarantees.

## 7. Conclusions

We have introduced offer and resource revocation in order to improve fairness when using Apache Mesos to manage clusters. Deploying offer revocation to timeout excessively indecisive frameworks permits the system to recover otherwise wasted resources and provide more offers to more frameworks. We implemented resource revocation to allow frameworks to obtain more reliable guarantees of system resources. This extension allows frame-

works to put hard limits on task latencies and thus be able to fulfill SLAs. We found that since offer based DRF in Mesos already did a reasonable job in providing fairness of resources, it was sufficient to focus revocation on fulfilling needs and not desires of frameworks. This ultimately minimized goodput loss due to revocation.

We evaluated our revocation mechanisms on a small cluster using synthetic frameworks designed to model problem scenarios that arise in a production cluster. Our results show that our revocation strategy works well to ensure fairness by minimizing task latencies at the cost of minimal goodput loss. Resource revocation was also able to maintain better resource stability and consistencies to participating frameworks. Greedy frameworks were prevented from locking out undersubscribed frameworks. Undersubscribed frameworks are able to get a resonable slice of the system without fear of being edged out the system by greedy frameworks.

## 8. Future Work

In this paper we chose the simplest resource revocation scheme. To continue this project, we would examine different revocation policies. One possible idea would be to reduce the frequency of revocations (since currently we check for revokation at every Mesos heartbeat). Reducing the frequency would obviously increase task latency but could also possibly reduce goodput loss by preventing Mesos from overreacting to transient utilization spikes. Other possible extensions explore broad adjustments to the revocation algorithm. Currently, the revocation routine iterates through oversubscribed rameworks in an arbitrary order until all resource needs are satisfied. A possible alternative would be to only revoke a handful of tasks every cycle and focus specifically on the most oversubscribed framework. This would possibly spread out the goodput impact amongst multiple oversubscribed frameworks providing a fairer punishment for oversubscription at the expense of taking more time to fulfill all needs.

Adding a voluntary revocation stage before forcing Mesos to directly revoke tasks would also be beneficial. Here, the master allocator would first send a warning message to oversubscribed frameworks giving them a chance to voluntarily terminate excess tasks (allowing those frameworks to pick a better termination order than youngest first) before Mesos initiated revocation. Since not all frameworks are like to respond (for example, legacy frameworks not updated to the new Mesos communication model), there is some design space in deciding how many extra resources Mesos should ask frameworks to voluntarily surrender.

## 9. Acknowledgements

Special thanks to Benjamin Hindman and the other Mesos developers at UC Berkeley for providing support in understanding the Mesos codebase and discussing possible implementation details for resource revocation. Thank you to Professors Anthony Joseph and John Kubiatowicz for providing feedback on the project progress and contributing general implementation ideas.

## References

[1] Apache mesos: Dynamic resource sharing for clusters. http://incubator.apache.org/mesos/index.html.

[2] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. May 2011.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. May 2011.