Mesos

Flexible Resource Sharing for the Cloud

BENJAMIN HINDMAN, ANDY KONWINSKI, MATEI ZAHARIA, ALI GHODSI, ANTHONY D. JOSEPH, RANDY H. KATZ, SCOTT SHENKER, AND ION STOICA

Benjamin Hindman is a fourth-year PhD student at the University of California, Berkeley. Before working on resource management for cluster computing, he worked on resource management for single-node parallel computing. His interests include operating systems, distributed systems, programming languages, and all the ways they intersect.
benh@eecs.berkeley.edu

Andy Konwinski is a PhD student in computer science at the University of California, Berkeley, who has worked on tracing and scheduling in distributed systems such as Hadoop and Mesos.
andyk@berkeley.edu

Matei Zaharia is a fourth-year graduate student at the University of California, Berkeley, working with Scott Shenker and Ion Stoica on topics in cloud computing, operating systems, and networking. He is also a committer on Apache Hadoop. He got his undergraduate degree at the University of Waterloo in Canada.
matei@eecs.berkeley.edu

Ali Ghodsi got his PhD from KTH Royal Institute of Technology in 2007. He is on leave from his position as an assistant professor at KTH and has been visiting the University of California, Berkeley, since 2009. His interests include cloud computing, distributed computing, and micro-economic applications in computer science.
alig@cs.berkeley.edu

Anthony D. Joseph is a Chancellor's Associate Professor in Electrical Engineering and Computer Science at the University of California, Berkeley. He is developing adaptive techniques for cloud computing, network and computer security, and security defenses for machine-learning–based decision systems. He also co-leads the DETERlab testbed, a secure scalable testbed for conducting cybersecurity research.
adj@eecs.berkeley.edu

Randy H. Katz is the United Microelectronics Corporation Distinguished Professor in Electrical Engineering and Computer Science at the University of California, Berkeley, where he has been on the faculty since 1983. His current interests are the architecture and design of modern Internet Datacenters and related large-scale services.
randy@cs.Berkeley.edu

Scott Shenker spent his academic youth studying theoretical physics but soon gave up chaos theory for computer science. Continuing to display a remarkably short attention span, over the years he has wandered from computer performance modeling and computer networks research to game theory and economics. Unable to hold a steady job, he currently splits his time between the University of California, Berkeley, Computer Science Department and the International Computer Science Institute.
shenker@icsi.berkeley.edu

Ion Stoica is an Associate Professor in the EECS Department at the University of California, Berkeley, where he does research on cloud computing and networked computer systems. Past work includes the Chord DHT, Dynamic Packet State (DPS), Internet Indirection Infrastructure (i3), declarative networks, replay-debugging, and multi-layer tracing in distributed systems. His current research includes resource management and scheduling for data centers, cluster computing frameworks for iterative and interactive applications, and network architectures.
istoica@eecs.berkeley.edu

Clusters of commodity servers have become a major computing platform, powering both large Internet services and a growing number of data-intensive enterprise and scientific applications. To reduce the challenges of building distributed applications, researchers and practitioners have developed a diverse array of new software frameworks for clusters. For example, frameworks such as memcached

[4] make accessing large datasets more efficient, while frameworks such as Hadoop [1] and MPI [6] simplify distributed computation.

Unfortunately, sharing a cluster efficiently between two or more of these frameworks is difficult. Many operators statically partition their clusters at physical machine granularities, yielding poor overall resource utilization. Furthermore, static partitioning makes it expensive to share big datasets between two computing frameworks (e.g., Hadoop and MPI): one must either copy the data into a separate cluster for each framework, consuming extra storage, or have the frameworks read it across the network, reducing performance.

This article introduces Mesos, a platform that enables fine-grained, dynamic resource sharing across multiple frameworks in the same cluster. For example, using Mesos, an organization can simultaneously run Hadoop and MPI jobs on the same datasets, and have Hadoop use more resources when MPI is not using them and vice versa. Mesos gives these and other frameworks a common interface for accessing cluster resources to efficiently share both resources and data.

In designing Mesos, we sought to make the system both flexible enough to support a wide range of frameworks (and maximize utilization by pooling resources across all these frameworks), and highly scalable and reliable (to be able to manage large production clusters). Specifically, we had four goals:

High utilization: share resources dynamically as the demand of each application changes
Scalability: support tens of thousands of machines and hundreds of concurrent jobs
Reliability: recover from machine failures within seconds
Flexibility: support a wide array of frameworks with diverse scheduling needs

Mesos achieves these goals by adopting an *application-controlled* scheduling model. The Mesos core is only responsible for deciding how many resources each framework should receive (based on an operator-selected policy such as priority or fair sharing), while frameworks decide which resources to use and which computations to run on them, using a mechanism called *resource offers*. This design has the dual benefit of giving frameworks the flexibility to schedule work based on their needs and letting the Mesos core be simple, scalable, and robust. Indeed, we show that Mesos scales to 50,000 nodes, recovers from master failures in less than 10 seconds, and lets applications achieve nearly perfect data locality in scheduling their computations.

Finally, Mesos provides important benefits even to organizations that only use one cluster computing framework. First, an organization can use Mesos to run multiple, isolated instances of the framework on the same cluster (e.g., to isolate production and experimental Hadoop workloads), as well as multiple versions of the framework (e.g., to test a new version). Second, Mesos allows developers to build *specialized* frameworks for applications where general abstractions like MapReduce are inefficient, and have them coexist with current systems. Later in this article we describe a specialized framework we developed for iterative applications and interactive data mining called Spark, which can outperform Hadoop by a factor of 30 for these workloads. We hope that other organizations also leverage Mesos to experiment with new cluster programming models.

Mesos began as a research project at UC Berkeley and is now open source under the Apache Incubator. It is actively being used at Twitter, Conviva, UC Berkeley, and UC San Francisco.

Mesos Architecture

Mesos enables efficient resource sharing across frameworks by giving them a common API to launch units of work, called *tasks*, on the cluster. A task typically runs on a slice of a machine, within a resource allocation chosen by the framework (e.g., 1 CPU core and 2 GB RAM). Mesos isolates tasks from each other using OS facilities like Linux Containers [2] to ensure that a runaway task will not affect other applications.

To support a wide range of frameworks while remaining scalable and robust, Mesos employs an application-controlled scheduling model. Mesos decides how many resources each framework should receive according to an organization-defined policy such as fair sharing. However, each framework is responsible for dividing its work into tasks, deciding which tasks to run on each machine, and, as we shall explain, selecting which machines to use. This lets the frameworks perform application-specific placement optimizations: for example, a MapReduce framework can place its map tasks on nodes that contain their input data.

Figure 1 shows the architecture of Mesos. The system has a fault-tolerant *master* process that controls *slave* daemons on each node. Each framework that uses Mesos has a *scheduler* process that registers with the master. Schedulers launch tasks on their allocated resources by providing *task descriptions*. Mesos passes these descriptions to a framework-specific *executor* process that it launches on slave nodes. Executors are also reused for subsequent tasks that run on the same node, to amortize initialization costs. Finally, Mesos passes *status updates* about tasks to schedulers, including notification if a task fails or a node is lost.

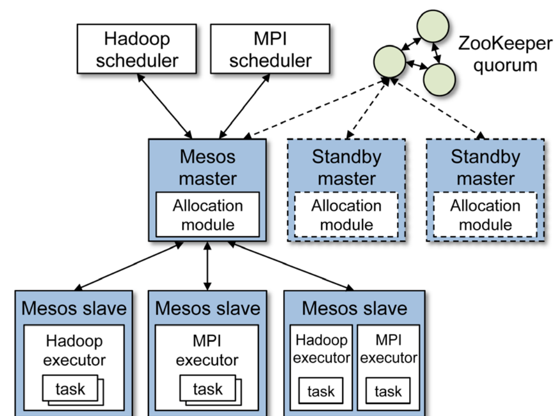

Figure 1: Mesos architecture, showing two running frameworks (Hadoop and MPI)

Mesos uses a mechanism called *resource offers* to let frameworks choose which resources to use. When resources on a machine become free, Mesos offers them to each framework scheduler in turn, in an order defined by the cluster's allocation policy (e.g., starting with the framework furthest below its fair share). Each framework may accept the resources and launch a task using some of them, or reject the resources if, for example, it has no data on that machine. Refusing resources keeps the framework at the front of the allocation queue, ensuring that it is offered future resources before other frameworks. While it may seem counterintuitive that refusing resources can help frameworks, we found that a simple policy where frameworks wait a short time for local resources achieves near-perfect data locality in typical cluster workloads.

One natural concern with resource offers is whether a framework will need to wait for a large number of offers to find a resource that it wants. To prevent this scenario, Mesos also provides an API for *requests* that lets frameworks specify which resources they wish to be offered. For example, a framework might provide a minimum amount of memory it needs, or a whitelist of nodes to run on. One important benefit of the resource offer model, however, is that frameworks whose needs cannot be expressed using requests can still achieve good task placement. That is, requests are an optimization, while resource offers guarantee correctness and allow the system to support arbitrary framework placement preferences.

More importantly, Mesos's application-controlled scheduling model also helps make the system extremely simple, scalable, and robust. Here is how Mesos achieves each of the four goals outlined in the introduction:

High utilization: Each framework is only allocated the resources to run its current tasks, as opposed to a static partition of the cluster.
Scalability: The Mesos master only makes inter-framework scheduling decisions (to pick which framework has priority for new offers), which are much simpler than the intra-framework decisions required for many applications (e.g., to achieve data locality). Our optimized C++ implementation can make thousands of decisions per second with sub-second latency and manage tens of thousands of nodes.
Reliability: The Mesos master only needs to store *soft state*: the list of currently active frameworks and tasks. Therefore, if the master crashes, a standby master can take over and repopulate its state within seconds when the frameworks and slaves connect to it.
Flexibility: Resource offers allow each framework to control its scheduling, while requests represent an extensible and efficient mechanism for frameworks to indicate their placement needs to the master.

Example Framework: Computing Pi

The Mesos team has already ported several popular frameworks, like Hadoop and MPI, to run on Mesos, but one of our main goals with Mesos was to let users easily develop other cluster applications that can run alongside existing frameworks. To show you how a Mesos framework looks from a programmer's perspective, Figure 2 illustrates a simple Python framework that computes π. Mesos also has APIs in C++ and Java.

The framework is composed of a scheduler, which launches tasks, and an executor, which runs them. The scheduler launches NUM_TASKS independent tasks, each of which computes an estimate of π and then averages the results. Each task uses an inefficient, but easy to explain method to estimate π: it picks random points in the unit square (from (0,0) to (1,1)) and counts what fraction of them fall in the unit circle. This fraction should be $\pi/4$, because one quarter of the unit circle is inside this square, so we multiply the result by 4. The tasks return their results in the data field of a Mesos status update. Note that the executor runs each task in a separate thread, in case a single machine is given multiple tasks.

Thanks to building on top of Mesos, this application does not need to implement infrastructure for launching work on the cluster or for communicating between tasks and the main program. It can just implement a few callbacks, such as `resourceOffer` and `statusUpdate`, to run on the Mesos-managed cluster.

```python
class MyScheduler(mesos.Scheduler):
  def resourceOffer(self, driver, id, offers):
    tasks = []
    for offer in offers:
      if self.tasksStarted < NUM_TASKS:
        self.tasksStarted += 1
        task = createTask(offer.slave_id,
                          {"cpus": 1, "mem": 32})
        tasks.append(task)
    driver.replyToOffer(id, tasks, {})

  def statusUpdate(self, driver, update):
    if update.state == TASK_FINISHED:
      self.resultSum += float(update.data)
      self.tasksDone += 1
      if self.tasksDone == NUM_TASKS:
        driver.stop()
        result = self.resultSum / NUM_TASKS
        print "Pi is roughly %f" % result
```

```python
class MyExecutor(mesos.Executor):
  def launchTask(self, driver, task):
    # Create a thread to run the task
    thread = Thread(target = self.runTask,
                    args = (driver, task))
    thread.start()

  def runTask(self, driver, task):
    NUM_SAMPLES = 1000000
    count = 0.0
    for i in range(1, NUM_SAMPLES):
      x = random()
      y = random()
      if x*x + y*y < 1:
        count += 1
    result = 4 * count / NUM_SAMPLES
    driver.sendStatusUpdate(
      task.task_id, TASK_FINISHED, str(result))
```

Figure 2: A sample Mesos framework, in Python, for computing π. The scheduler (left) launches NUM TASKS tasks and averages their results, while the executor (right) runs a separate estimation of π in a thread for each task. We omit some boilerplate initialization code.

Use Cases

Mesos Usage at Twitter

Twitter has been using Mesos internally as an end-to-end framework for deploying some of their application services. Using Mesos for some of their services appealed to Twitter for many reasons, including:

Flexible deployment: Statically configuring where services should run makes it difficult for different teams within Twitter to operate autonomously. By leveraging Mesos, engineering teams can focus on doing code deploys against a generic pool of resources, while the operations team can focus on the operating system and hardware (e.g., rebooting machines with new kernels, replacing disks, etc).

Increased utilization: Many services within the cluster are sharded for better fault-tolerance and do not (or cannot) fully utilize a modern server with up to 16 CPU cores and 64+ GB of memory. Mesos enables Twitter to treat machines as a pool of resources and run multiple services on the same machine, yielding better overall cluster utilization.

Elasticity: Certain services might want to "scale up" during peak or unexpected events when traffic and load has increased. Using Mesos, it's easy for different services to consume more or less resources as they are needed.

Using Mesos to facilitate normal datacenter maintenance and upgrades has been especially compelling at Twitter. Because Mesos notifies frameworks when machines fail, operators can easily remove machines from the cluster (provided there is enough general capacity). Frameworks simply react to these "failures" and reschedule their computations as needed.

Because of Mesos's two-level scheduling design, Twitter can provide its own organizational policies for how resources should be allocated to frameworks. For example, some machines can have most of their resources dedicated to applications serving user requests (e.g., Web servers and databases), allowing unused "slack" resources to be used for lower-priority applications. Twitter uses Linux Containers [2] to isolate services running on the same machine from one another.

Using Mesos, engineers at Twitter have been able to easily experiment with building new services, including spam detectors, load testers, distributed tracing frameworks, and service quality monitors, among others. Twitter continues to experiment with using Mesos for deploying more services in their clusters.

Managing Hadoop Clusters

Running the popular Hadoop framework on Mesos has many advantages. In current versions of Hadoop, a single master process (the job tracker) manages an entire cluster, which creates a single point of failure and leads to poor isolation between workloads (for example, a single user submitting too large a job may crash the job tracker). Mesos has been designed to support many concurrent frameworks, so it can run each Hadoop job separately, with its own job tracker, isolating MapReduce applications from each other. Mesos also provides stronger isolation of the resources on each machine through Linux Containers. Finally, from an operations viewpoint, an important advantage of running Hadoop on Mesos is that it enables organizations to experiment with different versions of Hadoop in one cluster, or to gradually upgrade from an older version to a newer one.

More recently, the next-generation Hadoop design was announced, which refactors the current Hadoop job tracker into a simpler resource manager and a separate application master for each job to achieve similar isolation benefits [7]. These new, lightweight application masters fit cleanly as framework schedulers in the Mesos model, and we are working to port them to run on top of Mesos to let Hadoop share resources with the other frameworks supported by Mesos.

Spark: A Framework for Low-Latency In-Memory Cluster Computing

One of our main goals with Mesos was to enable the development of new analytics frameworks that complement the popular MapReduce programming model. As an example, we developed Spark, a framework for iterative applications and interactive data mining that provides primitives for in-memory cluster computing. Unlike frameworks based on acyclic data flow, such as MapReduce and Dryad, Spark allows programmers to create in-memory *distributed datasets* and reuse them efficiently in multiple parallel operations. This makes Spark especially suitable for iterative algorithms that reuse the same data repeatedly, such as machine learning and graph applications, and for interactive data mining, where a user can load a dataset into memory and query it repeatedly. As previously mentioned, Spark can outperform Hadoop by a factor of 30 in these tasks.

Spark provides a language-integrated programming interface, similar to Microsoft's DryadLINQ [9], in Scala [5], a high-level language for the Java VM. This means that users can write functions in a single program that automatically get sent to a cluster for execution. For example, the following code snippet implements the π estimation algorithm from earlier in this article:

```
val count = spark.parallelize(1 to NUM_SAMPLES).map(i =>
  val x = Math.random
  val y = Math.random
  if (x*x + y*y < 1) 1.0 else 0.0
).reduce(_ + _)
println("Pi is roughly " + 4 * count / NUM_SAMPLES)
```

Here, the arguments to `map` and `reduce` are Scala function literals (closures) that are automatically shipped to the Mesos cluster for parallel execution. The `_ + _` syntax means a function to add two numbers.

As a more interesting example, the code below implements logistic regression [3], an iterative machine learning algorithm for classification (e.g., identifying spam). We build an in-memory distributed dataset called points by loading the data in a text file, then run map and reduce operations on it repeatedly to perform a gradient descent. Loading points into memory allows subsequent iterations to be much faster than the first and lets Spark outperform Hadoop for this application.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D)
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating parameter: " + w)
```

Spark can also be used interactively from a modified Scala interpreter to build and query distributed datasets. We have used Spark to analyze several large traces in the course of our research.

Spark is being used by several groups of machine learning researchers at Berkeley, for projects including traffic estimation and spam detection on social networks. It is also being used at Conviva, an online video distribution company, to run analytics on large Hadoop and Hive datasets. The system has grown into a research project of its own, and is open source at http://www.spark-project.org.

Experimental Results

We evaluated Mesos through a series of experiments included in our NSDI '11 paper [8]. We sketch three of them here.

Job performance in a shared cluster: In the first experiment, we wanted to compare Mesos's performance with a static partitioning of a cluster, where each partition ran a separate framework. For this, we ran a 100-node cluster on Amazon EC2 and concurrently ran four frameworks: (1) a mixed Hadoop workload based on the workload at Facebook, (2) a Hadoop batch workload, (3) a Spark instance running machine learning jobs, and (4) the popular Torque scheduler running MPI jobs. Table 1 compares job completion times for Mesos and static partitioning. As seen, most jobs speed up when using Mesos. Note that the Torque framework was configured to never use more than a fourth of the cluster. It is therefore expected not to see any speedup. The slight slowdown for Torque was due to a slow machine on EC2. The speedups are due to frameworks scaling up and down dynamically to use other resources when another framework's demand is low. In contrast, with static partitioning, frameworks are confined to a fixed fraction of the cluster machines.

Scalability: The second experiment investigated how the Mesos master scales with the cluster size. We ran 200 frameworks filling the whole cluster with tasks that on average took 30 seconds to finish. Thus, the Mesos master was busy making scheduling decisions as the tasks were continuously finishing and being launched by the frameworks. We then launched one additional framework that

ran one task and measured the overhead of scheduling this task. The result was that the scheduling overhead remained on average under one second for up to 50,000 slave daemons (which we ran as separate processes on up to 200 physical machines), showing that the master can manage large clusters with heavy workloads. Much of the system's scalability stems from our use of C++ and efficient I/O mechanisms in the master.

Reliability: In the final experiment, we wanted to measure how fast Mesos recovered from master failures. As in the scalability experiment, we filled the cluster with tasks. We then killed the master node and measured how long it took for the system to elect a new master node and repopulate its state. For a 4000-node cluster, the whole system recovered within 10 seconds.

Framework	Sum of Exec Times w/ Static Partitioning (s)	Sum of Exec Times with Mesos (s)	Speedup
Facebook Hadoop Mix	7235	6319	**1.14**
Large Hadoop Mix	3143	1494	**2.10**
Spark	1684	1338	**1.26**
Torque / MPI	3210	3352	**0.96**

Table 1: Aggregate performance of each framework in the macro-benchmark (sum of running times of all the jobs in the framework). The speedup column shows the relative gain on Mesos.

Conclusion

As the number of software frameworks for clusters grows, it is becoming increasingly important to dynamically share resources between these frameworks. We have presented Mesos, a scalable and reliable platform that enables efficient, fine-grained sharing of clusters among diverse frameworks by giving frameworks control over their scheduling. Mesos can currently run Hadoop, MPI, the Torque resource manager, and a new framework, called Spark, for fast in-memory parallel computing. We hope that Mesos also encourages the development of other frameworks that can coexist with these. Mesos is open source at http://www.mesosproject.org.

References

[1] Apache Hadoop. : http://lucene.apache.org/hadoop.

[2] Linux containers Containers (LXC) overview document.: http://lxc.sourceforge.net/lxc.htmlhttp://lxc.sourceforge.net/.

[3] Logistic regression—Wikipedia. : http://en.wikipedia.org/wiki/Logistic_regression.

[4] memcached—a distributed object caching system. : http://memcached.org/.

[5] Scala programming language. : http://www.scala-lang.org/.

[6] The Message Passing Interface (MPI) Standard. : http://www.mcs.anl.gov/research/projects/mpi.

[7] The Next Generation of Apache Hadoop MapReduce. : http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen.

[8] 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," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*.

[9] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 1–14.