The evolution of cluster scheduler architectures.

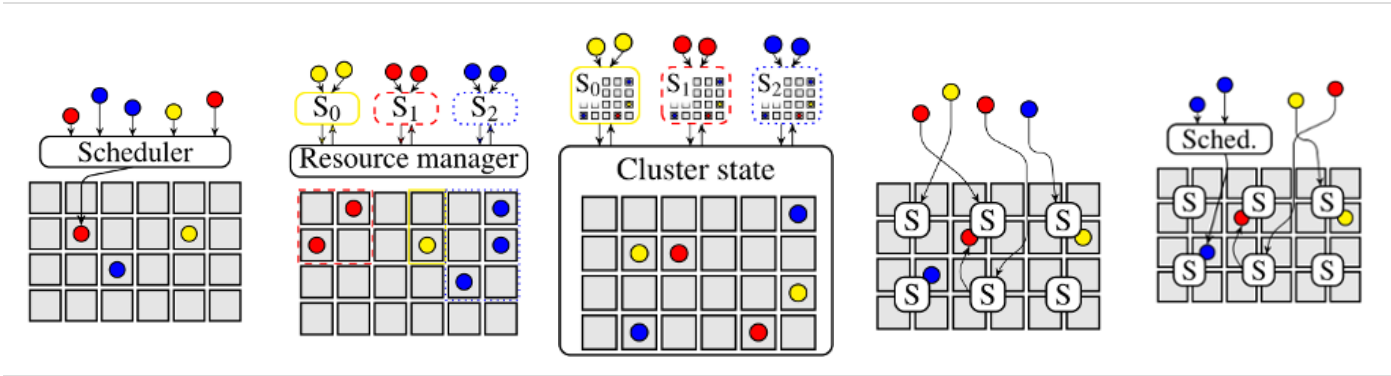
March 9, 2016 by Malte (https://twitter.com/ms705)

tl;dr: *cluster schedulers are an important component of modern infrastructure, and have evolved significantly in the last few years. Their architecture has moved from monolithic designs to much more flexible, disaggregated and distributed designs. However, many current open-source offerings are either still monolithic, or otherwise lack key features. These features matter to real-world users, as they are required to achieve good utilization.*

This post is our first in a series of posts about *task scheduling on large clusters*, such as those operated by internet companies like Amazon, Google, Facebook, Microsoft, or Yahoo!, but increasingly elsewhere too. Scheduling is an important topic because it directly affects the cost of operating a cluster: a poor scheduler results in low *utilization*, which costs money as expensive machines are left idle. High utilization, however, is not sufficient on its own: antagonistic workloads interfere with other workloads unless the decisions are made carefully.

Architectural evolution

This post discusses how scheduler architectures have evolved over the last few years, and why this happened. Figure 1 visualises the different approaches: a gray square corresponds to a machine, a coloured circle to a task, and a rounded rectangle with an "S" inside corresponds to a scheduler.[0] Arrows indicate placement decisions made by schedulers, and the three colours correspond to different workloads (e.g., web serving, batch analytics, and machine learning).



(a) Monolithic scheduler. **(b)** Two-level scheduling. **(c)** Shared-state scheduling. **(d)** Distributed scheduling. **(e)** Hybrid scheduling.

Figure 1: Different cluster scheduler architectures. Gray boxes represent cluster machines, circles correspond to tasks and S_i denotes scheduler *i*.

Many cluster schedulers – such as most high-performance computing (HPC) schedulers, the Borg scheduler (http://dl.acm.org/citation.cfm?id=2741964), various early Hadoop schedulers and the Kubernetes scheduler – are **monolithic**. A single scheduler process runs on one machine (e.g., the `JobTracker` in Hadoop v1, and `kube-scheduler` in Kubernetes) and assigns tasks to machines. All workloads are handled by the same

scheduler, and all tasks run through the same scheduling logic (Figure 1a). This is simple and uniform, and has led to increasingly sophisticated schedulers being developed. As an example, see the Paragon (http://dl.acm.org/citation.cfm?id=2451125) and Quasar (http://dl.acm.org/citation.cfm?id=2541941) schedulers, which use a machine learning approach to avoid negative interference between workloads competing for resources.

Most clusters run different types of applications today (as opposed to, say, just Hadoop MapReduce jobs in the early days). However, maintaining a single scheduler implementation that handles mixed (heterogeneous) workloads can be tricky, for several reasons:

1. It is quite reasonable to expect a scheduler to treat long-running service jobs and batch analytics jobs differently.
2. Since different applications have different needs, supporting them all keeps adding features to the scheduler, increasing the complexity of its logic and implementation.
3. The order in which the scheduler processes tasks becomes an issue: queueing effects (e.g., head-of-line blocking) and backlog can become an issue unless the scheduler is carefully designed.

Overall, this sounds like the makings of an engineering nightmare – and the never-ending lists of feature requests that scheduler maintainers receive attests to this.[1]

Two-level scheduling architectures address this problem by separating the concerns of *resource allocation* and *task placement*. This allows the task placement logic to be tailored towards specific applications, but also maintains the ability to share the cluster between them. The Mesos (http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf) cluster manager pioneered this approach, and YARN (http://dl.acm.org/citation.cfm?id=2523633) supports a limited version of it. In Mesos, resources are *offered* to application-level schedulers (which may pick and choose from them), while YARN allows the application-level schedulers to *request* resources (and receive allocations in return).[2] Figure 1b shows the general idea: workload-specific schedulers (S_0–S_2) interact with a resource manager that carves out dynamic partitions of the cluster resources for each workload. This is a very flexible approach that allows for custom, workload-specific scheduling policies.

Yet, the separation of concerns in two-level architectures comes with a drawback: the application-level schedulers lose *omniscience*, i.e., they cannot see *all* the possible placement options any more.[3] Instead, they merely see those options that correspond to resources offered (Mesos) or allocated (YARN) by the resource manager component. This has several disadvantages:

1. *Priority preemption* (higher priority tasks kick out lower priority ones) becomes difficult to implement: in an offer-based model, the resources occupied by running tasks aren't visible to the upper-level schedulers; in a request-based model, the lower-level resource manager must understand the preemption policy (which may be application-dependent).
2. Schedulers are unable to consider *interference from running workloads* that may degrade resource quality (e.g., "noisy neighbours" that saturate I/O bandwidth), since they cannot see them.
3. Application-specific schedulers care about many different aspects of the underlying resources, but their only means of choosing resources is the offer/request interface with the resource manager. This interface can easily become quite complex.

Shared-state architectures address this by moving to a semi-distributed model,[4] in which multiple replicas of cluster state are independently updated by application-level schedulers, as shown in Figure 1c. After the change is applied locally, the scheduler issues an optimistically concurrent transaction to update the shared cluster state. This transaction may fail, of course: another scheduler may have made a conflicting change in the meantime.

The most prominent examples of shared-state designs are Omega (http://dl.acm.org/citation.cfm?id=2465386) at Google, and Apollo (https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin) at Microsoft, as well as the Nomad (https://www.nomadproject.io/docs/internals/scheduling.html) container scheduler by Hashicorp. All of these materialise the *shared cluster state* in a single location: the "cell state" in Omega, the "resource monitor" in

Apollo, and the "plan queue" in Nomad.[5] Apollo differs from the other two as its shared-state is read-only, and the scheduling transactions are submitted directly to the cluster machines. The machines themselves check for conflicts and accept or reject the changes. This allows Apollo to make progress even if the shared-state is temporarily unavailable.[6]

A "logical" shared-state design can also be achieved without materialising the full cluster state anywhere. In this approach (somewhat similar to what Apollo does), each machine maintains its own state and sends updates to different interested agents such as schedulers, machine health monitors, and resource monitoring systems. Each machine's local view of its state now forms a "shard" of the global shared-state.

However, shared-state architectures have some drawbacks, too: they must work with stale information (unlike a centralized scheduler), and may experience degraded scheduler performance under high contention (although this can apply to other architectures as well).

Fully-distributed architectures take the disaggregation even further: they have no coordination between schedulers at all, and use many independent schedulers to service the incoming workload, as shown in Figure 1d. Each of these schedulers works purely with its local, partial, and often out-of-date view of the cluster. Jobs can typically be submitted to any scheduler, and each scheduler may place tasks anywhere in the cluster. Unlike with two-level schedulers, there are no partitions that each scheduler is responsible for. Instead, the overall schedule and resource partitioning are emergent consequences of statistical multiplexing and randomness in workload and scheduler decisions – similar to shared-state schedulers, albeit without any central control at all.

The recent distributed scheduler movement probably started with the Sparrow (http://dl.acm.org/citation.cfm?id=2522716) paper, although the underlying concept (power of multiple random choices) first appeared in 1996 (http://www.eecs.harvard.edu/~michaelm/postscripts/mythesis.pdf). The key premise of Sparrow is a hypothesis that the tasks we run on clusters are becoming ever shorter in duration, supported by an argument (http://dl.acm.org/citation.cfm?id=2490497) that fine-grained tasks have many benefits. Consequently, the authors assume that tasks are becoming more numerous, meaning that a higher decision throughput must be supported by the scheduler. Since a single scheduler may not be able to keep up with this throughput (assumed to be a million tasks per second!), Sparrow spreads the load across many schedulers.

This makes perfect sense: and the lack of central control can be conceptually appealing, and it suits some workloads very well – more on this in a future post. For the moment, it suffices to note that since the distributed schedulers are uncoordinated, they apply significantly simpler logic than advanced monolithic, two-level, or shared-state schedulers. For example:

1. Distributed schedulers are typically based on a simple "slot" concept that chops each machine into *n* uniform slots, and places up to *n* parallel tasks. This simplifies over the fact that tasks' resource requirements are not uniform.
2. They also use worker-side queues with simple service disciplines (e.g., FIFO in Sparrow), which restricts scheduling flexibility, as the scheduler can merely choose at which machine to enqueue a task.
3. Distributed schedulers have difficulty enforcing global invariants (e.g., fairness policies or strict priority precedence), since there is no central control.
4. Since they are designed for rapid decisions based on minimal knowledge, distributed schedulers cannot support or afford complex or application-specific scheduling policies. Avoiding interference between tasks, for example, becomes tricky.

Hybrid architectures are a recent (mostly academic) invention that seeks to address these drawbacks of fully distributed architectures by combining them with monolithic or shared-state designs. The way this typically works – e.g., in Tarcil (http://dl.acm.org/citation.cfm?id=2806779), Mercury (https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos), and Hawk (https://www.usenix.org/conference/atc15/technical-session/presentation/delgado) – is that there really are two scheduling paths: a distributed one for part of the workload (e.g., very short tasks, or low-priority batch

workloads), and a centralized one for the rest. Figure 1e illustrates this design. The behaviour of each constituent part of a hybrid scheduler is identical to the part's architecture described above. In practice, no hybrid schedulers have been deployed in production settings yet, however, as far as I know.

What does this mean in practice?

Discussion about the relative merits of different scheduler architectures is not merely an academic topic, although it naturally revolves around research papers. For an extensive discussion of the Borg, Mesos and Omega papers from an industry perspective, for example, see Andrew Wang's excellent blog post (http://umbrant.com/blog/2015/mesos_omega_borg_survey.html). Moreover, many of the systems discussed are deployed in production settings at large enterprises (e.g., Apollo at Microsoft, Borg at Google, and Mesos at Apple), and they have in turn inspired other systems that are available as open source projects.

These days, many clusters run containerised workloads, and consequently a variety of contained-focused "orchestration frameworks" have appeared. These are similar to what Google and others call "cluster managers". However, there are few detailed discussions of the schedulers within these frameworks and their design principles, and they typically focus more on the user-facing scheduler APIs (e.g., this report by Armand Grillet (http://armand.gr/static/files/htise.pdf), which compares Docker Swarm, Mesos/Marathon, and the Kubernetes default scheduler). Moreover, many users neither know what difference the scheduler architecture makes, nor which one is most suitable for their applications.

Figure 2 shows an overview of a selection of open-source orchestration frameworks, their architecture and the features supported by their schedulers. At the bottom of the table, We also include closed-source systems at Google and Microsoft for reference. The resource granularity column indicates whether the scheduler assigns tasks to fixed-size slots, or whether it allocates resources in multiple dimensions (e.g., CPU, memory, disk I/O bandwidth, network bandwidth, etc.).

	Framework	Architecture	Resource granularity	Multi-scheduler
OPEN	Kubernetes	monolithic	multi-dimensional	N[v1.2, DD (https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/r schedulers.md), Issue (https://github.com/kubernetes/kubernetes/issues/
	Swarm	monolithic	multi-dimensional	N
	YARN	monolithic/ two-level	RAM/CPU slots	Y
	Mesos	two-level	multi-dimensional	Y
	Nomad	shared-state	multi-dimensional	Y
	Sparrow	fully-distributed	fixed slots	Y
CLOSED	Borg	monolithic[7]	multi-dimensional	N[7]
	Omega	shared-state	multi-dimensional	Y
	Apollo	shared-state	multi-dimensional	Y

Figure 2: Architectural classifaction and feature matrix of widely-used orchestration frameworks, compared to closed-source systems.

One key aspect that helps determine an appropriate scheduler architecture is whether your cluster runs a *heterogeneous* (i.e., mixed) workload. This is the case, for example, when combining production front-end services (e.g., load-balanced web servers and memcached) with batch data analytics (e.g., MapReduce or Spark). Such combinations make sense in order to improve utilization, but the different applications have different scheduling needs. In a mixed setting, a monolithic scheduler likely results in sub-optimal assignments, since the logic cannot be diversified on a per-application basis. A two-level or shared-state scheduler will likely offer benefits here. [8]

Most user-facing service workloads run with resource allocations sized to serve peak demand expected of each container, but in practice they typically under-utilize their allocations substantially. In this situation, being able to opportunistically over-subscribe the resources with lower-priority workloads (while maintaining QoS guarantees) is the key to an efficient cluster. Mesos is currently the only open-source system that ships support for such over-subscription, although Kubernetes has a fairly mature proposal (http://kubernetes.io/v1.1/docs/proposals/resource-qos.html) for adding it. We should expect more activity in this space in the future, since the utilization of most clusters is still substantially lower than the 60-70% reported (http://dl.acm.org/citation.cfm?id=2741964) for Google's Borg clusters. We will focus on resource estimation, over-subscription and efficient machine utilization in a future post in this series.

Finally, specific analytics and OLAP-style applications (for example, Dremel or SparkSQL queries) can benefit from fully-distributed schedulers. However, fully-distributed schedulers (like e.g., Sparrow) come with fairly restricted feature sets, and thus work best when the workload is homogeneous (i.e., all tasks run for roughly the same time), set-up times are low (i.e., tasks are scheduled to long-running workers, as e.g., with MapReduce application-level tasks in YARN), and task churn is very high (i.e., many scheduling decisions must be made in a short time). We will talk more about these conditions and why fully-distributed schedulers – and the distributed components of hybrid schedulers – only make sense for these applications in the next blog post in this series. For now, it sufficies to observe that distributed schedulers are substantially simpler than others, and do not support multiple resource dimensions, over-subscription, or re-scheduling.

Overall, the table in Figure 2 is evidence that the open-source frameworks still have some way to go until they match the feature sets of advanced, but closed-source systems. This should serve as a call to action: as a result of missing features, utilization suffers, task performance is unpredictable, noisy neighbours cause pagers to go off, and elaborate hacks are required to coerce schedulers into supporting some user needs.

However, there are some good news: while many frameworks have monolithic schedulers today, many are also moving towards more flexible designs. Kubernetes already supports pluggable schedulers (the `kube-scheduler` pod can be replaced by another API-compatible scheduler pod), multiple schedulers from v1.2 (https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/multiple-schedulers.md), and has ongoing work on "extenders" to supply custom policies (https://github.com/kubernetes/kubernetes/blob/master/docs/design/scheduler_extender.md). Docker Swarm may – to my understanding – also gain pluggable scheduler support in the future.

What's next?

The next blog post in this series will look at the question of whether fully distributed architectures are the key innovation required to *scale* cluster schedulers further (spoiler: not necessarily). After that, we will also look at resource-fitting strategies (essential for good utilisation), and finally discuss how our Firmament scheduling platform combines many of the benefits of a shared state architecture with the scheduling quality of monolithic schedulers and the speed of fully-distributed schedulers.

Follow us on Twitter (http://twitter.com/CamSysAtScale) to find out about new posts.

Posted on *March 9, 2016* by Malte (https://twitter.com/ms705).

Correction: March 10, 2016
An earlier version of the text incorrectly reported the implementation status of some Kubernetes features. We amended the table in Figure 2 and the text to clarify that scheduler extenders are implemented, and that over-subscription is supported although automatic resource estimation is not. We also added a footnote explaining that a single scheduler can serve a mixed workload, but that its complexity will be high.

Correction: March 15, 2016
An earlier version of the text suggested that YARN and Mesos are two-level designs in an equal sense. However, YARN's application-level scheduling is substantially less powerful than Mesos's. This is now clearer in the text, and clarified further in footnote 2.

[0] – This figure simplifies things a bit: of course, in practice each machine runs more than one task, and many schedulers fit tasks in multiple resource dimensions, rather than into simple slots.

[1] – As an illustrative example, `kube-scheduler` in Kubernetes currently has outstanding feature requests for re-scheduling (pod migration) (https://github.com/kubernetes/kubernetes/pull/22217), priority preemption (https://github.com/kubernetes/kubernetes/issues/22212), and resource oversubscription (https://github.com/kubernetes/kubernetes/pull/14943) in its monolithic scheduler.

[2] – YARN's approach is restricted compared to Mesos because the application-level logic cannot choose resources (unless it requests much more than it needs from the resource manager), but it can only place application-level "tasks" to pre-existing containers that represent cluster-level tasks.
This is a good fit for a system like Hadoop MapReduce, in which application-level tasks (maps and reduces) must be assigned to a dynamic collection of workers in an application-specific way (e.g., optimised for data locality and per-job). It is less suited to building a more general, multi-application scheduler on top – for example, a service scheduler like the "Marathon" framework for Mesos.
Monolithic schedulers like the Kubernetes one do not have an "upper half" and rely on the application to do its own work scheduling (e.g., running a Spark "worker controller" as a long-running service (http://kubernetes.io/v1.0/examples/spark/README.html)). Consequently, there are efforts to put Kubernetes on top of YARN (http://hortonworks.com/blog/docker-kubernetes-apache-hadoop-yarn/) via a special `YARNScheduler` extension – requiring two complex systems to be administered. However, there are also long-term efforts to improve native "big data" batch processing support in Kubernetes (https://docs.google.com/document/d/1YhNLN39f5oZ4AHn_g7vBp0LQd7k37azL7FkWG8CEDrE/edit#heading=h.ukbaidczvy3r).

[3] – In the Omega paper, this problem was referred to as "information hiding".

[4] – Curiously, the literature does not appear to be quite sure in agreement about whether to consider shared-state schedulers centralized or distributed: the Hawk paper (https://www.usenix.org/conference/atc15/technical-session/presentation/delgado) treats them as examples of distributed schedulers, while the Mercury paper (https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos) refers to them as examples of a centralized architecture!

[5] – Nomad actually uses a slightly different approach to Omega and Apollo: while multiple independent schedulers exist, jobs are not submitted directly to them, but instead arrive via a centralised "evaluation broker" queue (https://www.nomadproject.io/docs/internals/scheduling.html).

[6] – It's worth noting that the same optimisation – taking the shared state off the critical path to enacting scheduling decisions – can be applied to Omega, but *not* to Nomad (in its current design): Omega can ship deltas directly to machines and update the cell state out-of-band, while Nomad's design is premised on the leader reconciling changes in the plan queue.

[7] – The table entry reflects the original Borg, but the Borg paper (http://dl.acm.org/citation.cfm?id=2741964) and the recent ACM Queue paper (http://queue.acm.org/detail.cfm?id=2898444) note that multi-scheduler support and other features have been back-ported into from Omega into Borg.

[8] – That said, having multiple schedulers is not a *necessary* precondition for serving mixed workloads: the Borg scheduler (http://dl.acm.org/citation.cfm?id=2741964) is a case in point that a sophisticated single scheduler can serve both long-running service and batch workloads. However, this comes at the expense of higher scheduler implementation complexity – a key motivation for Omega's multi-scheduler design (http://dl.acm.org/citation.cfm?id=2465386).

Firmament is a CamSaS (http://camsas.org) project. – Starry Sky background by Westin Lohe (https://github.com/westin/sky).