Designing **CLUSTER SCHEDULERS** for Internet-Scale Services

DIPTANU CHOUDHURY AND TIMOTHY PERRETT

EMBRACING FAILURES FOR IMPROVING AVAILABILITY

In the past decade, cluster schedulers have become the foundation of Internet-scale services that span hundreds or thousands of machines across many geographically distributed data centers. Cluster schedulers have their origins in high-performance computing or managing services that run on warehouse-scale computing systems, which allow for efficient usage of computational resources. Schedulers also provide an API for application developers writing distributed applications such as Spark and MapReduce or building application management platforms such as DC/OS (Data Center Operating System). In the past few years open-source schedulers such as Kubernetes, HashiCorp's Nomad, and Apache Mesos have democratized scale, allowing many enterprises to adopt scheduler technologies that were previously accessible only to companies like Facebook, Google, and Twitter.

Despite this apparent ubiquity, operating and implementing scheduling software is an exceedingly tricky

To an extent, schedulers allow operators to ignore the distribution of compute resources.

task with many nuanced edge cases. This article highlights some of these cases based on the real-world experience of the authors designing, building, and operating a variety of schedulers for large Internet companies.

FAILURES ARE INEVITABLE

On a long enough timeline, everything will fail. Every day, many of the Internet services the world has come to rely upon have many small—even imperceptible—failures. Machines crash, APIs become intermittently latent, hard drives fail, and networks become saturated, but usually services powered by them don't fail to serve user requests. Though it may seem that recovering automatically from such failures is a solved problem, the reality is that many different processes are involved in orchestrating this recovery. Scheduling software is often the foundational infrastructure that allows a service to recover by interacting with various other data center services.

Modern distributed systems such as search, social networks, and cloud object stores consume many more resources than a handful of servers or even mainframes can provide. Such systems consume resources on the order of tens of thousands of machines, potentially spread across many data centers. This often means it is not possible to treat the data center as a collection of servers, and the notion of an individual server is simply less relevant.

Software developers care only about available resources such as RAM, CPUs, and available bandwidth for accessing networked systems. To an extent, schedulers allow operators to ignore the distribution of compute resources. Within these bounds, when a workload running

on a scheduler fails, the scheduler can simply look for resources within the available pool and reschedule that job for processing within the constraints imposed by the user. It is exactly at this moment of failure where things become interesting. Why did the workload fail? Was there an application problem? Or a machine-specific problem? Or was there perhaps a cluster-wide or otherwise environmental problem? More importantly, how does the architecture of a scheduler impact the ability and timeline of a workload to recover? The answers to these questions directly affect and dictate how effective a scheduler can be in recovering that failed workload.

One of the responsibilities of a cluster scheduler is to supervise an individual unit of work, and the most primitive form of remediation is to move that workload to a different, healthy node; doing so will frequently solve a given failure scenario. When using any kind of shared infrastructure, however, you must carefully evaluate the bulkheading options applied for that shared infrastructure, and objectively assess the opportunity for cascading failure. For example, if an I/O-intensive job is relocated to a node that already hosts another I/O-intensive job, they could potentially saturate the network links in the absence of any bulkheading of IOPs (I/O operations), resulting in a degraded QoS [quality of service] of other tenants on the node.

The following sections highlight the various failure domains within scheduling systems and touch upon some of the practical problems operators encounter with machine, scheduler, environmental, and cluster-wide failures. In addition, the article provides some answers to dealing with the failures.

CONSIDERATIONS FOR MACHINE FAILURES

Failures at the machine level are probably the most common. They have a variety of causes: hardware failures such as disks crashing; faults in network interfaces; software failures such as excessive logging and monitoring; and problems with containerizer daemons. All these failures result in degraded application performance or a potential incoherent cluster state. While all possible system failure modes are too numerous to mention in one article, within the realm of scheduling there are a handful of important factors to consider; the following sections cover details about the mechanics of the modern Linux operating system and how to mitigate the effects of typical failure modes encountered in the field.

Capacity planning

Regardless of where compute capacity is housed—public cloud or private data center—at some point capacity planning will be necessary to figure out how many machines are needed. Traditional methods⁸ for capacity planning make assumptions about compute resources being entirely dedicated, where a given machine has a sole tenant. While this is a common industry practice, it is often ineffective as application authors tend to be overly optimistic about their runtime performance (resulting in insufficient capacity and potential outage at runtime) or overly cautious about resource consumption, leading to a high cost of ownership with a large amount of waste⁵ when operating at scale.

Assuming an application has been benchmarked for performance and resource consumption, running that

Not provisioning an adequate amount of resources for system services inadvertently surfaces as busy-neighbor problems.

application on a scheduling system introduces additional challenges for capacity planning: how will shared components handle multitenancy? Common configurations have per-node utilities for routing traffic, monitoring, and logging (among other tasks); will these potentially impact those lab performance numbers? The probability is very high that they will have a (negative) impact.

Ensure that the capacity plan includes headroom for the operating system, file systems, logging agents, and anything else that will run as a shared component. Critically, anything that is a shared component should have well-defined limits (where possible) on compute resources. Not provisioning an adequate amount of resources for system services inadvertently surfaces as busy-neighbor problems. Many schedulers allow operators to reserve resources for running system components, and correctly configuring these resource reservations can dramatically improve the predictability of application performance.

➔ *File systems.* Learn about the overhead and resource usage of file systems. This is useful, for example, when using ZFS to limit the ARC (adaptive replacement cache) to an acceptable size, or when planning to turn on deduplication or compression to account for the CPU cycles that ZFS itself is going to use. Consider another example: two containers doing a lot of file-system I/O with a very limited cache would end up invalidating each other's caches and result in poor I/O performance. Limiting file-system IOPs is not straightforward in Linux, since the block I/O and memory controller cannot interact with each other to limit the writeback I/O with traditional cgroup v1. The next version of cgroup *can* properly limit I/O, but a

few controllers—such as the CPU controller—have not yet been merged.

➔ *Sidecars.* Logging, monitoring, or service meshes such as Envoy² can potentially use a considerable amount of resources, and this needs to be accounted for. For example, if a logging agent such as Fluentd³ is forwarding logs to a remote sink, then the network bandwidth for that process should be limited so that containers can get their expected amount of network resources for application traffic. Fair sharing of such resources is considerably difficult, and therefore it is sometimes easier to run sidecars for every allocation on a node rather than sharing them, so that their resources can be accounted for under the cgroup hierarchy of the allocation.

➔ *Administration.* Policies for system or component configurations—such as garbage collection—should be based on the units that the underlying resource understands. For example, log retention policies based on a number of days are not effective on a node where the storage is limited by number of bytes—rotating logs every three days is useless if the available bytes are consumed within a matter of hours. Systems administrators may often apply the same types of policies that they write for cluster-wide log-aggregation services for local nodes. This can have disastrous consequences at the cluster level where services are designed to scale out horizontally, where a workload might be spread across many nodes that have the same—or similar—hardware configurations.

These are some of the key elements to consider for capacity-planning purposes, but this is by no means an exhaustive set. Be sure to consider any environment-

specific capacity bounds that might apply to your case, always basing your plan on real data about actual usage collected in the field.

The OOM killer

The OOM (out of memory) killer in Linux steps in under extremely low memory conditions and kills processes to recover memory based on a set of rules and heuristics. The decisions made by the OOM killer are based on a so-called *oom_score*, which changes over time, based on certain rules, and is not deterministic in most situations.

The OOM killer is an important system component to keep in mind while designing schedulers that allow for oversubscription,¹ since they allow more tasks on a machine than actual resources. In such situations, it is important to design a good QoS module that actively tracks resource usage of tasks and kills said tasks proactively. If tasks consume more memory than they are allocated, the scheduler should kill the tasks before the overall resource utilization forces invocation of the system OOM killer. For example, QoS modules could implement their own strategy for releasing memory by listening for kernel notifications indicating memory pressure, and subsequently killing lower-priority tasks, which in turn prevent the kernel from invoking the OOM killer.

Having scheduler agents killing tasks allows for deterministic behavior and is easier to debug and troubleshoot. For example, in the Mesos cluster manager the Mesos agent runs a QoS controller that continuously monitors tasks that run with revocable resources and kills them if they interfere with normal tasks.

Leaking container resources

Since its introduction to the Linux kernel a decade ago, container technology has improved immensely. It is, however, still an imperfect world, and tools that have been built atop these foundations have added more complexity over time, opening the door to interesting and tricky-to-solve problems. One of the common runtime issues operators will encounter is the leaking of associated resources. For example, say you boot a container with a bridged networking mode; under the hood a virtual Ethernet adapter will be created. If the application crashes unexpectedly—and is not killed by an external agent—the container daemon can potentially leak virtual interfaces over time, which eventually causes a system problem when a moderate number of interfaces have been leaked. This causes new applications attempting to boot on that machine to fail, as they are unable to create virtual network adapters.

Remediating these types of failures can be difficult; the issue must first be monitored to keep track of the resources being created and garbage collected over time, ensuring that the leaking is either kept to a minimum or effectively mitigated. Operators often find themselves writing agents to disable scheduling on a node until resources become available, to make sure a node isn't running under pressure, or preemptively redistributing work before the issue manifests itself into an outage. It is best to surface such problems to the operators even if automated mitigation procedures are in place, since they are usually a result of bugs in underlying container runtimes.

Naive scheduler policies might undersubscribe node-local resource usage but oversubscribe aggregate resources such as bandwidth.

Undersubscription of attached resources

Schedulers usually decide placements or bin-packing strategies for tasks based on node resources such as CPU, memory, disk, and capacity of the I/O subsystems. It is important, however, to consider the shared resources attached to a node, such as network storage, aggregated link layer bandwidth attached to the ToR (top of rack) switch, etc., to ensure such resources are allocated to a reasonable limit or are judiciously oversubscribed. Naive scheduler policies might undersubscribe node-local resource usage but oversubscribe aggregate resources such as bandwidth. In such situations, optimizing for cluster-level efficiency is better than local optimization strategies such as bin packing.

Multitenancy

Multitenancy is one of the hardest challenges for performance engineers to solve in an elastic, shared infrastructure. A cluster that is shared by many different services with varying resource usage patterns often shows so-called busy-neighbor problems. Performance of a service can become degraded because of the presence of other cotenant services. For example, on the Linux operating system, imposing QoS for the network can be complicated, so operators sometimes do not go through the effort of imposing traffic-shaping mechanisms for controlling throughput and bandwidth of network I/O in containers. If two network I/O-intensive applications run on the same node, they will adversely affect each other's performance.

Other common problems with multitenancy include

cgroup controllers not accounting for certain resources correctly, such as the VFS IOP, where services that are very disk I/O intensive will have degraded performance when colocated with similar services. Work has been ongoing in this area for the past five to six years to design new cgroup controllers⁹ on Linux that do better accounting, but not all these controllers have yet been put into production. When workloads use SIMD (single instruction multiple data) instructions such as those from Intel's AVX-512 instruction set, processors throttle the CPU clock speed to reduce power consumption, thereby slowing other workloads running on the same CPU cores that are running non-SIMD instructions.⁶

Fair sharing of resources is often the most common approach offered by schedulers, and shares of resources are often expressed via scalar values. Scalar values are easier to comprehend from an end-user perspective, but in practice they don't always work well because of interference.⁷ For example, if 100 units of IOPs are allocated to two workloads running on the same machine, the one doing sequential I/O may get a lot more throughput than the one performing random I/O.

CONSIDERATIONS FOR CLUSTER-LEVEL FAILURES

Most of the significant failures that wake operators up in the middle of the night are typically those that have affected entire clusters or racks of servers in a fleet. Cluster-level failures are usually triggered because of bad configuration changes, bad software deployment, or in some cases because of cascading failures in certain services that result in resource contention in a multitenant

environment. Most schedulers come with remediation steps for unhealthy tasks such as restarts or eviction of lower-priority tasks from a node to reduce resource contentions. Cluster-wide failures indicate a problem far bigger than local node-related problems that render local remediation techniques.

Such failures in most situations require paging on-call engineers for remediation actions; however, the scheduler can also play a role in remediation during such failures. The authors have written and deployed schedulers that have cluster-wide failure detectors and would prevent nodes from continuously restarting tasks locally. They also allow operators to define remediation strategies, such as reverting to a last known good version or decreasing the frequency of restarts, stopping evicting other tasks, etc., before the operators can debug possible causes of failures. Such failure-detection algorithms usually take into consideration the health of tasks cotenant on the same machine to differentiate service-level failures from other forms of infrastructure-related failures.

Cluster-wide failures should be taken seriously by scheduler developers; the authors have encountered failures that have generated so many cluster events that they saturated the scheduler's ability to react to failures. Therefore, sophisticated measures must be taken to ensure the events are sampled but don't lose context of the nature and magnitude of the underlying issues. Depending on the magnitude of failure, the saturation of events often brings operations to a standstill unless quickly mitigated. This section covers some of the most frequently used techniques for mitigating cluster-level failures.

Bad software push

Most cluster-level job failures are the result of bad software pushes or configuration changes. It can often be useful to track the start time of such failures and correlate them with cluster events such as job submissions, updates, and configuration changes. Another common, yet simple, technique for reducing the likelihood of cluster-wide failures in the face of bad software pushes is a rolling update strategy that incrementally deploys new versions of software only after the new instances have proven to be healthy or working as expected from the perspective of key metrics. Schedulers such as Nomad and Kubernetes come with such provisions. They move on to deploying newer versions of software only when the current set of tasks passes health checks and stops deployments if they start encountering failures.

System software failure

System software, such as the Docker daemon and monitoring and logging software, is an integral part of the scheduler infrastructure and thus contributes to the health of the cluster. New versions of such software have often been deployed only for sysadmins to find that they cause failures after some period of time in a cluster. In one instance a specific Auto Scaling Group on AWS (Amazon Web Services) started misbehaving a few days after the cluster joined the scheduler infrastructure; it turned out that a new version of Docker had been rolled out, which had functional regressions.

In most cases, the best strategy for dealing with such failures is to disable scheduling on those machines and

drain the assigned work to force the relocation of the workload to elsewhere in the data center. Alternatively, you could introduce additional resource capacity with a working configuration of all system software, such that pending workloads could be scheduled successfully.

Such failures affect the tasks of all jobs in a specific cluster or resource pool; hence, schedulers should have good mechanisms for dealing with them. A robust scheduler design should ideally be able to detect an issue with a given cluster or resource pool and proactively stop scheduling workloads there. This kind of proactive remediation event should be included in the telemetry information being emitted by the scheduler so that on-call engineers can further debug and resolve the specific problem.

Shared resource failures

Modes of failure at the infrastructure level include fiber cuts; faulty power distribution for a machine, rack, or ToR (Top of Rack) switch; and many other environmental possibilities. In such cases, other than moving affected workloads to unaffected systems, a scheduler can do very little to mitigate the problem.

In some cluster schedulers, the default behavior when nodes become disconnected from the network is to begin killing tasks. Operationally, this can cause significant challenges when the nodes return to a healthy state. In most cases, it is preferable to delegate the complexity of guaranteeing a fixed number of currently operational tasks to the application itself. This typically makes the scheduling system easier to operate and allows the application to get

precisely the consistency and failure semantics it desires. Tasks should be allowed to join the cluster gracefully when the failures are mitigated. Such measures decrease the churn in the cluster and allow for its faster recovery.

Depletion of global resources

In addition to node resources, global resources such as aggregate bandwidth or power usage within the cluster should be tracked by the scheduler's resource allocator. Failure to track these global resources could result in placement decisions oversubscribing cluster resources, causing bottlenecks that create hotspots within the data center, thereby reducing the efficiency of the provisioned hardware.

For example, bin packing too many network I/O-intensive tasks in a single rack might saturate the links to the data center's backbone, creating contention, even though network links at the node level might not be saturated. Bin packing workloads very tightly in a specific part of the data center can also have interesting or unexpected side effects with respect to power consumption, thereby impacting the available cooling solutions.

Software-distribution mechanisms

It is very important to understand the bottlenecks of software-distribution mechanisms. For example, if the aggregate capacity of a distribution mechanism is 5 Gbps, then launching a job with tens of thousands of tasks could easily saturate the limit of the distribution mechanism or even of the shared backbone. This could have detrimental effects on the entire cluster and/or the running services.

Parallel deployments of other services can often be affected in such a failure mode; hence, the parallelism of task launches must be capped to ensure no additional bottlenecks are created when tasks are deployed or updated.

Keep in mind that distribution mechanisms that are centralized in nature, such as the Docker registry, are part of the availability equation. When these centralized systems fail, job submission or update requests fail as well, thereby putting services at risk of becoming unavailable if they, too, are updated. Extensive caching of artifacts on local nodes to reduce pressure on centralized distribution mechanisms can be an effective mitigation strategy against centralized distribution outages. In some instances, peer-to-peer distribution technologies such as BitTorrent can further increase the availability and robustness of such systems.

Backoff strategies for remediation actions

Certain workloads might not perform well on any node in the cluster and might be adversely affecting the health of the services and the nodes. In such cases, the schedulers must detect the trend while reallocating workloads or bring additional capacity to ensure they don't deplete global resources, such as API call limits of cloud providers, or adversely affect cotenant workload, thereby causing cascading failures.

CONTROL-PLANE FAILURE CONSIDERATIONS

Control planes within schedulers have a different set of failure considerations than compute nodes and clusters, as

the control plane must react to changes in the cluster as they happen, including various forms of failure. Software engineers writing such systems should understand the user interaction, scale, and SLA (service-level agreement) for workloads and then derive an appropriate design that encompasses handling failures in the control plane. This section looks at some of the important design considerations for control-plane developers.

Reliable cluster state reconciliation

At the end of the day, most schedulers are just managing cluster state, supervising tasks running on the cluster, and ensuring QoS for them. Schedulers usually track the cluster state and maintain an internal finite-state machine for all the cluster objects they manage, such as clusters, nodes, jobs, and tasks. The two main ways of cluster state reconciliation are level- and edge-triggered mechanisms. The former are employed by schedulers such as Kubernetes, which periodically looks for unplaced work and tries to schedule that work. These kinds of schedulers often suffer from having a fixed baseline latency for reconciliation.

Edge-triggered scheduling is more common. Most schedulers, such as Mesos and Nomad, work on this model. Events are generated when something changes in the cluster infrastructure, such as a task failing, node failing, node joining, etc. Schedulers must react to these events, updating the finite state machine of the cluster objects and modifying the cluster state accordingly. For example, when a task fails in Mesos, the framework gets a `TASK_LOST` message from the master and reacts to that event based

on certain rules, such as restarting the task elsewhere on the cluster or marking a job as dead or complete. Nomad is similar: it invokes a scheduler based on the type of the allocation that died, and the scheduler then decides whether the allocation needs to be replaced.

While event-driven schedulers are faster and more responsive in practice, guaranteeing correctness can be harder since the schedulers have no room to drop or miss the processing of an event. Dropping cluster events will result in the cluster not converging to the right state; jobs might not be in their expected state or have the right number of tasks running. Schedulers usually deal with such problems by making the agents or the source of the cluster event resend the event until they get an acknowledgment from the consumer that the events have persisted.

Quotas for schedulers

Schedulers are usually offered to various teams in an organization for consuming compute infrastructure in the data center. Schedulers usually implement quotas, which ensure that various jobs have the right amount of resources on the clusters during resource contention. Besides quotas for compute resources on the compute clusters, scheduler developers also must consider how much time schedulers spend doing scheduling per job. For example, the amount of time it would take to schedule a batch job with 15,000 tasks would be much more than for a job with 10 tasks. Alternatively, a job might have a few tasks but very rigorous placement constraints. The scheduler might spend varying amounts of time serving various jobs from various teams based on constraints and volume of

the tasks or churn in the cluster. Monolithic schedulers, which centralize all the scheduling work, are more prone to these kinds of problems than two-level schedulers such as Mesos, where operators can run multiple frameworks to ensure various schedulers are serving a single purpose and thereby not sharing scheduling time for anything else.

With monolithic schedulers it is important to develop concepts such as quotas for various types of jobs or teams. Another possibility for scaling schedulers is to do parallel scheduling in a similar manner to Nomad, where the operators can run many scheduler instances that work in parallel and can decide how many scheduler processes they want to run for a certain job type.

Recovering cluster state from failures

Scheduler operators want AP (CAP available, partition tolerant) systems in practice because they prefer availability and operability over consistency. The convergence of the cluster state eventually must be guaranteed after all the cluster events have been processed or by some form of reconciliation mechanism. Most real-world schedulers, however, are built on top of highly consistent coordination systems such as ZooKeeper or etcd, because building and reasoning about such distributed systems are easier when the data store provides guarantees of linearizability. It is not unheard of for schedulers to lose their entire database for a few hours. One such instance was when AWS had a Dynamo outage, and a large scheduler operating on top of AWS was using Dynamo for storing cluster state. There isn't a lot that can be done in such situations, but scheduler

developers have to consider this scenario and develop with the goal of causing the least impact to running services on the cluster.

Some schedulers such as Mesos allow operators to configure a duration after which an agent that is disconnected from the scheduler starts killing the tasks running on a machine. Usually this is done with the assumption that a scheduler is disconnected from the nodes because of failures such as network partitions; since the scheduler thinks the node is offline, it has already restarted the tasks on that machine somewhere else in the cluster. This does not work when schedulers are experiencing outages or have failed in an unrecoverable manner. It is better to design scheduler agents that don't kill tasks when the agent disconnects from the scheduler, but instead allow the tasks to run and even restart them if a long-running service fails. Once the agents rejoin the cluster, the reconciliation mechanisms should converge the state of the cluster to an expected state.

The process of restoring a cluster's state when a scheduler loses all its data is complicated, and the design depends largely on the architecture and data model of the scheduler. On Apache Mesos, the scheduler frameworks¹¹ can query statuses of tasks for known task IDs. The Mesos master would respond with the current state of the nonterminal tasks. On Nomad, the cluster state is captured in the raft stores of the schedulers, and there is no good way to back up the cluster state and restore from a snapshot. Users are expected to resubmit the jobs. Nomad can then reconcile the cluster state, which creates a lot of churn in services.

CONCLUSION

Designing for failures in all aspects of a distributed cluster scheduler is a must for operational stability and reliability. Scheduler agents should be developed with the understanding that only finite amounts of resources exist on a given system. Processes could leak resources or consume more resources than they were intended to, resulting in unexpected performance degradation caused by resource contention. These scheduler agents must also be able to converge on a good state by using robust reconciliation mechanisms during a given failure (or set of failures), even when particular failure modes could inundate the scheduler

with cluster events—for example, the loss of many agent systems caused by a power failure.

Engineers looking to build scheduling systems should consider all failure modes of the underlying infrastructure they use and consider how operators of scheduling systems can configure remediation strategies, while aiding in keeping tenant systems as stable as possible during periods of troubleshooting by owners of said tenant systems.

The cutting-edge nature of this field of engineering makes it one of the most exciting areas in which to

work, enabling workload mobility, uniform scalability, and self-healing systems to become widespread.

Related articles

Hadoop Superlinear Scalability

The perpetual motion of parallel performance
Neil Gunther, Paul Puglia, and
Kristofer Tomasette

<https://queue.acm.org/detail.cfm?id=2789974>

A Conversation with Phil Smoot

The challenges of managing a megaservice
<https://queue.acm.org/detail.cfm?id=1113332>

The Network is Reliable

An informal survey of real-world
communications failures

Peter Bailis and Kyle Kingsbury

<https://queue.acm.org/detail.cfm?id=2655736>

References

1. Apache Mesos; <http://mesos.apache.org/documentation/latest/frameworks/>.
2. Envoy; <https://www.envoyproxy.io/>.
3. Fluentd; <https://www.fluentd.org/>.
4. Ionel, G., Schwarzkopf, M., Gleave, A., Watson, R. N. M., Hand, S. 2016. Firmament: fast, centralized cluster scheduling at scale. Proceedings of the 12th Usenix Symposium on Operating Systems Design and Implementation; <https://research.google.com/pubs/pub45746.html>.
5. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A. 2009. Quincy: fair scheduling for distributed computing clusters. *Proceedings of the 22nd ACM SIGOPS Symposium on Operating System Principles*: 261-276; <https://dl.acm.org/citation.cfm?id=1629601>.
6. Krasnov, V. 2017. On the dangers of Intel's frequency scaling. Cloudflare; <https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/>.
7. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., Kozyrakis, C. 2016. Improving resource efficiency at scale with Heracles. *ACM Transactions on Computer Systems* 34(2); <http://dl.acm.org/citation.cfm?id=2882783>.
8. Microsoft System Center. 2013. Methods and formula used to determine server capacity. TechNet Library; <https://technet.microsoft.com/en-us/library/cc181325.aspx>.
9. Rosen, R. 2016. Understanding the new control groups API. LWN.net; <https://lwn.net/Articles/679786/>.
10. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M.,

Wilkes, J. 2013. Omega: flexible, scalable schedulers for large compute clusters. SIGOPS European Conference on Computer Systems; <https://research.google.com/pubs/pub41684.html>.

11. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. Wilkes, J. 2015. Large-scale cluster management at Google with Borg. *Proceedings of the 10th European Conference on Computer Systems*; <https://dl.acm.org/citation.cfm?id=2741964>.

Diptanu Gon Choudhury (@diptanu) works at Facebook on large-scale distributed systems. He is one of the maintainers of the open source cluster scheduler called Nomad and previously worked on cluster schedulers on top of Apache Mesos at Netflix.

Timothy Perrett (@timperrett) is an infrastructure engineering veteran, published author, and speaker and has led engineering teams at a range of blue chip companies. He is primarily interested in scheduling systems, programming language theory, security systems and changing the way industry approaches software engineering.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.