acmqueue

Reliable Cron across the Planet

...or How I stopped worrying and learned to love time

Štěpán Davidovič, Kavita Guliani, Google

This article describes Google's implementation of a distributed Cron service, serving the vast majority of internal teams that need periodic scheduling of compute jobs. During its existence, we have learned many lessons on how to design and implement what might seem like a basic service. Here, we discuss the problems that distributed Crons face and outline some potential solutions.

Cron is a common Unix utility designed to periodically launch arbitrary jobs at user-defined times or intervals. We will first analyze the basic principles of Cron and its most common implementations and then review how an application such as Cron can work in a large, distributed environment, in order to increase the reliability of the system against single-machine failures. We describe a distributed Cron system that is deployed on a small number of machines, but is capable of launching Cron jobs on machines across an entire data center, in conjunction with a data center scheduling system.

Before we jump onto the topic of how to run a reliable Cron service for entire data centers, let us first check out Cron — introduce its properties and look at it from the perspective of a site reliability engineer.

Cron is designed so that both system administrators and ordinary users of a system can specify commands to be run and when to run them. Various kinds of jobs, including garbage collection and periodic data analysis, are executed. The most common time specification format is called "crontab." It supports simple intervals (e.g., "once a day at noon" or "every hour on the hour"). Complex intervals such as "every Saturday, and on the 30th day of the month" can also be configured.

Cron is usually implemented using a single component, commonly called `crond`. This is a daemon that loads the list of Cron jobs to be run and keeps them sorted based on their next execution time. The daemon then waits until the first item is scheduled to be executed. At that time, `crond` launches the given job or jobs, calculates the next time to launch them, and waits until the next scheduled execution time.

RELIABILITY

Viewing the service from a reliability perspective, there are several things to notice. First, the failure domain is essentially just one machine. If the machine is not running, neither the Cron scheduler nor the jobs it launches can run. (Failure of the individual jobs, for example because they cannot reach the network or they try to access a broken HDD, is beyond the scope of this analysis.) Consider a very simple distributed case with two machines, in which the Cron scheduler launches jobs on a separate machine (for example using SSH). We already have distinct failure domains that could impact our ability to launch jobs: either the scheduler or the destination machine could fail.

Another important aspect of Cron is that the only state that needs to persist across `crond` restarts (including machine reboots) is the crontab configuration itself. The Cron launches are fire-and-forget, and `crond` makes no attempt to track them.

A notable exception to this is `anacron`, which attempts to launch jobs that were scheduled to be launched when the system was down. This is limited to jobs that run daily or less frequently, but is very useful for running maintenance jobs on workstations and notebooks. Running these jobs is facilitated by keeping a file containing timestamps of the last launch for all registered Cron jobs.

CRON JOBS AND IDEMPOTENCY

Cron jobs are used to perform periodic work, but beyond that, it is hard to know their functions in advance. Let us digress for a bit, and discuss the actions of Cron jobs themselves, because understanding the variety of requirements of Cron jobs will be a theme throughout the rest of the text, and obviously affects our reliability requirements.

Some Cron jobs are idempotent, and in case of system malfunction, it is safe to launch them multiple times — for example, garbage collection. Other Cron jobs should not be launched twice, for example, a process that sends out an email newsletter with wide distribution.

To make matters more complicated, failure to launch is acceptable for some Cron jobs but not for others. A garbage collection Cron job scheduled to run every five minutes may be able to skip one launch, but a payroll job scheduled to run once a month probably should not.

This large variety of Cron jobs makes it difficult to reason about failure modes: for a service like Cron, there is no single answer that fits every situation. The approach we will be leaning towards in this article is to prefer skipping launches, instead of risking double launches, as much as the infrastructure allows. The Cron job owners can (and should!) monitor their Cron jobs, for example by having the Cron service expose state for the jobs it manages, or by setting up independent monitoring of the effect of the Cron jobs. In case of a skipped launch, Cron job owners can take action that matches the nature of the job. However, undoing a double launch may be difficult, and in some cases (consider the newsletter example) impossible. So, we prefer to “fail closed” (in engineering terminology) to avoid systemically creating bad state.

CRON AT LARGE SCALE

Moving away from single machines toward large-scale deployments requires some fundamental rethinking of how to make Cron work well in such an environment. Let us discuss those differences and the design changes they necessitated, before presenting the details of Google’s Cron solution.

EXTENDED INFRASTRUCTURE

Regular Cron is limited to a single machine. However, for large-scale system deployment, our Cron solution must not be tied to a single machine.

If we assume a single data center with exactly 1000 machines, a failure of just 1/1000th of the available machines would knock out the entire Cron service. For obvious reasons, this is not acceptable.

To solve this general problem, we decouple processes from machines. If you want to run a service, simply specify which data center it should run in and what it requires — the data center scheduling system (which itself should be reliable) takes care of figuring out which machine or machines to deploy it on, as well as handling machine deaths. Launching a job in a data center then effectively turns into sending one or more RPCs to the data center scheduler.

This process is not instantaneous, however. There are inherent delays associated with both discovering a dead machine (health-check timeouts) and rescheduling a job onto a different machine (software installation, startup time of the process).

Since moving a process to a different machine can mean loss of any local state stored on the old machine (unless live migration is employed), and the rescheduling time may exceed the smallest scheduling interval of one minute, we should be able to cope with this situation. One of the most obvious options is to simply persist the state on a distributed file system such as GFS, and use it during startup to identify jobs that failed to launch due to rescheduling. This solution falls short of timeliness expectations; however, if you run a Cron job every five minutes, a delay of a minute or two caused by the total overhead of rescheduling is a substantial portion of the interval.

The timeliness expectation might motivate keeping around hot spares, which could quickly jump in and resume operation.

EXTENDED REQUIREMENTS

Another substantial difference between deployment in a data center and on a single machine is how much more planning is required when it comes to resources such as CPU or RAM.

Single-machine systems typically just co-locate all running processes with limited isolation. While containers (such as Docker[3]) are now quite common, it's not necessary or common to use them to isolate absolutely everything, including `crond` and all the jobs it runs, for software deployed on a single machine.

Deployment at data center scale commonly means deployment into containers that enforce isolation. Isolation is necessary because the base expectation is that a running process in the data center should not negatively impact any other process. In order to enforce that, you need to know what resources to acquire upfront for any given process you want to run — including both the Cron system and the jobs it launches. As a logical consequence of this, a Cron job may be delayed if the data center does not have available resources matching the demands of that job. This, as well as the desire to monitor the Cron job launches, means that we need to track the full state of our Cron jobs, from scheduled launch through termination.

Because the Cron system has now decoupled the process launches from specific machines, as described above, we may experience partial launch failures. Thanks to the versatility of the job configuration, launching a new Cron job in a data center can necessitate multiple RPCs. This, unfortunately, means that we can reach the point where some RPCs succeeded, but others did not, for example, because the process sending them died in the middle. The recovery procedure has to handle these as well.

In terms of failure modes, a data center is substantially more complex than a single machine. The Cron service that started as a relatively simple binary on a single machine now has many obvious and less obvious dependencies in the data center case. For a service as basic as Cron, we want to ensure that even if the data center suffers a partial failure (for example, a partial power outage or problems with storage services), the service still functions. To improve reliability, we ensure that the data center scheduler locates replicas in diverse locations within the data center, to avoid, for example, a single power distribution unit taking out all the Cron processes.

It might be possible to deploy a single Cron service across the globe, but deploying Cron service within a data center has benefits such as shared fate with the data center scheduler (which is the core dependency) as well as low latency to it.

CRON AT GOOGLE AND HOW TO BUILD ONE

Let us address the problems that we must resolve in order to provide a reliable Cron in a large-scale distributed deployment and highlight some important decisions made for distributed Cron at Google.

TRACKING THE STATE OF CRON JOBS

As discussed above, we should hold some amount of state about the Cron jobs and be able to restore it quickly in case of failure. Moreover, the consistency of that state is paramount: it is more acceptable to not launch Cron jobs for a while than to mistakenly launch the same Cron job ten times instead of once. Recall that many Cron jobs are not idempotent, for example, a payroll run or sending out an email newsletter.

We have two options: store data externally in a generally available distributed storage, or store a small amount of state as part of the Cron service itself. When designing the distributed Cron, we opted for the second option. There are several reasons for this:

- Distributed file systems such as GFS and HDFS are often designed for the use case of very large files (for example, the output of web crawling programs), whereas the information we need to store about Cron jobs is very small. Small writes on such a distributed file system are very expensive and have high latency as the file system is not optimized for them.

- Base services with wide impact, such as Cron, should have very few dependencies. Even if parts of the data center go away, the Cron service should be able to function for at least some amount of time. This does not mean that the storage has to be part of the Cron process directly (that is essentially an implementation detail). However, it should be able to operate independently of downstream systems that cater to a large number of internal users.

THE USE OF PAXOS

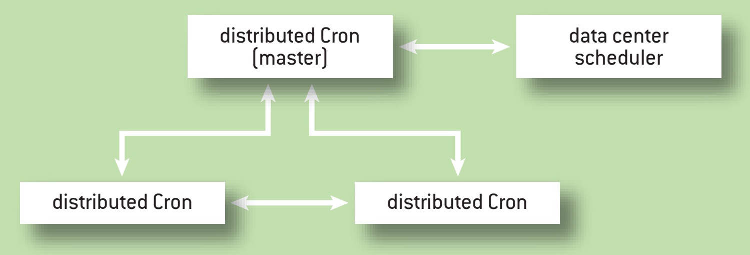
We deploy multiple replicas of the Cron service and use the Paxos algorithm to ensure consistent state among them.

The Paxos algorithm[5] and its successors (such as Zab[4] or Raft[7]) are commonplace in distributed systems nowadays (Chubby[1], Spanner[2], etc.). Describing Paxos in detail is beyond the scope of this article, but the basic idea is to achieve a consensus on state changes among multiple unreliable replicas. As long as the majority of Paxos group members are available, the distributed system, as a whole, can successfully process new state changes despite the failure of bounded subsets of the infrastructure.

The distributed Cron uses a single master job, shown in figure 1, which is the only replica that can modify the shared state, as well as the only replica that can launch Cron jobs. We take advantage of the fact that the variant of Paxos used, known as Fast Paxos[6], uses a master replica internally as an optimization—the Fast Paxos master replica also acts as the Cron service master.

If the master replica dies, the health checking mechanism of the Paxos group discovers this quickly (within seconds); with other Cron processes already started up and available, we can elect a new master. As soon as the new master is elected, we go through a master election protocol specific to the Cron service, which is responsible for taking over all the work left unfinished by the previous

FIGURE 1 **The Interactions Between Distributed Cron Replicas**



master. The master specific to Cron is the same as the Paxos master, but the Cron master needs to take additional action upon promotion. The fast reaction time for electing a new master allows us to stay well within a generally tolerable one-minute failover time.

The most important state we keep with Paxos is the information about what Cron jobs get launched. For each Cron job, we synchronously inform a quorum of replicas of the beginning and end of each scheduled launch.

THE ROLES OF THE MASTER AND THE SLAVE

As described above, our use of Paxos and its deployment in the Cron service has two assigned roles: the master and the slave. Let us go through the operation of each role.

THE MASTER

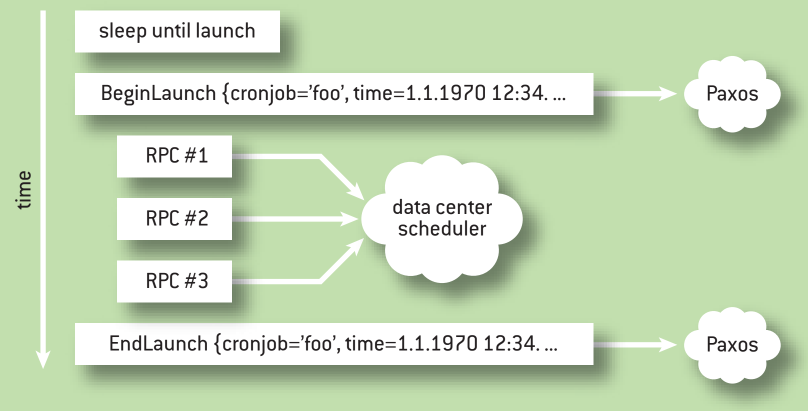
The master replica is the only replica actively launching Cron jobs. The master has an internal scheduler which, much like the simple `crond` described at the beginning, maintains the list of Cron jobs ordered by their scheduled launch times. The master replica waits until the scheduled launch time of the first element of this list.

When we reach the scheduled launch time, the master replica announces that it is about to launch this particular Cron job, and calculates the new scheduled launch time, just as a regular `crond` implementation would. Of course, as with regular `crond`, a Cron job launch specification may have changed since the last execution, and this launch specification must be kept in sync with the slaves as well. Simply identifying the Cron job is not enough: we should also uniquely identify the particular launch using the start time, to avoid ambiguity in Cron job launch tracking (especially likely with high-frequency Cron jobs, such as those running every minute). This communication is done over Paxos. Figure 2 illustrates the progress of a Cron job launch from the master's perspective.

It is important to keep this Paxos communication synchronous and not proceed with the actual Cron job launch until there is a confirmation that the Paxos quorum has received the launch notification. The Cron service needs to know whether each Cron job has launched, to decide the next course of action in case of master failover. Not performing this synchronization could mean that the entire Cron job launch happened on the master, but the slave replicas were not aware of it. In case of failover, they might attempt to perform the same launch again simply because they were

FIGURE 2

Progress of a Cron Job Launch


sleep until launch
BeginLaunch {cronjob='foo', time=1.1.1970 12:34. ...
Paxos
RPC #1
RPC #2
RPC #3
data center scheduler
time
EndLaunch {cronjob='foo', time=1.1.1970 12:34. ...
Paxos

not informed that it had already happened.

The completion of a Cron job launch is announced via Paxos to the other replicas, also synchronously. Note that it does not matter whether the launch succeeded or failed for external reasons (for example, if the data center scheduler was unavailable). Here, we are keeping track of the fact that the Cron service has attempted the launch at the scheduled time. We also need to be able to resolve failures of the Cron system in the middle of this operation, which we discuss below.

Another important feature of the master is that as soon as it loses its mastership for any reason, it must immediately stop interacting with the data center scheduler. Holding the mastership should guarantee mutual exclusion of access to the data center scheduler. If it does not, the old and new masters might perform conflicting actions on the data center scheduler.

THE SLAVE

The slave replicas keep track of the state of the world, as provided by the master, in order to take over at a moment's notice if needed. All the state changes that slave replicas track are coming, via Paxos, from the master replica. Much like the master, they also maintain a list of all Cron jobs in the system. This list must be kept consistent among the replicas (through the use of Paxos).

Upon receiving notification about a launch, the slave replica updates its local next scheduled launch time for the given Cron job. This important state change (recall that it is done synchronously) ensures that the Cron job schedules within the system are consistent. We keep track of all open launches, that is, launches where we have been notified of their beginning, but not of their end.

If a master replica dies or otherwise malfunctions (e.g., gets partitioned away from the other replicas on the network), a slave should be elected as a new master. This election process must happen in less than one minute, to avoid the risk of missing (or unreasonably delaying) a Cron job launch. Once a master is elected, all the open launches (i.e., partial failures) must be concluded. This can be a complicated process, imposing additional requirements on both the Cron system and the data center infrastructure, and it deserves more detailed explanation.

RESOLVING PARTIAL FAILURES

As mentioned above, the interaction between the master replica and the data center scheduler can fail in between sending multiple RPCs that describe a single logical Cron job launch, and we should be able to handle this condition as well.

Recall that every Cron job launch has two synchronization points: when we are about to perform the launch and when we have finished it. This allows us to delimit the launch. Even if the launch consists of a single RPC, how do we know whether the RPC was actually sent? We know that the scheduled launch started, but we were not notified of its completion before the master replica died.

To achieve this condition, all operations on external systems, which we may need to continue upon re-election, either have to be idempotent (i.e., we can safely do them again), or we need to be able to look up their state and see whether they completed or not, unambiguously.

These conditions impose significant constraints, and may be difficult to implement, but they are fundamental to the accurate operation of a Cron service in a distributed environment that could suffer partial failures. Not handling this appropriately can lead to missed launches or a double launch of the same Cron job.

Most infrastructure that launches logical jobs in data centers (for example Mesos) provides naming for those jobs, making it possible to look up their state, stop them, or perform other maintenance. A reasonable solution to the idempotency problem is to construct these names ahead of time — without causing any mutating operations on the data center scheduler — and distribute them to all replicas of the Cron service. If the Cron service master dies during launch, the new master simply looks up the states of all the precomputed names and launches the missing jobs.

Recall that we track the scheduled launch time when keeping the internal state between the replicas. Similarly, we need to disambiguate our interaction with the data center scheduler, also by using the scheduled launch time. For example, consider a short-lived but frequently run Cron job. The Cron job was launched, but before this was communicated to all replicas, the master crashed and an unusually long failover — long enough that the Cron job finished successfully — took place. The new master looks up the state of the job, observes that it finished, and attempts to launch it. If the time had been included, the master would have known that the job on the data center scheduler was the result of this particular Cron job launch, and the double launch would not have happened.

The actual implementation has a more complicated system for state lookup, driven by the implementation details of the underlying infrastructure. However, the description above covers the implementation-independent requirements of any such system. Depending on the available infrastructure, you may also need to consider the tradeoff between risking a double launch and risking skipping a launch.

STORING THE STATE

Using Paxos to achieve consensus is only one part of the problem of how to handle the state. Paxos is essentially a continuous log of state changes, appended to synchronously with the state changes. This has two implications: first, the log needs to be compacted, to prevent it from growing infinitely; second, the log itself must be stored somewhere.

In order to prevent infinite growth, we simply take a snapshot of the current state, so we can reconstruct the state without needing to replay all state change log entries leading to it. For example, if our state changes stored in logs are “Increment a counter by 1,” then after a thousand iterations, we have a thousand log entries which can be easily changed to a snapshot of “Set counter to 1000.”

In case of loss of logs, we would only lose the state since the last snapshot. Snapshots are in fact our most critical state — if we lose our snapshots, we essentially are starting from zero because we’ve lost our internal state. Losing logs, on the other hand, just causes a *bounded* loss of state and sends the Cron system back in time to when the latest snapshot was taken.

We have two main options for storing our data: externally in a generally available distributed storage, or in a system for storing the small volume of state as part of the Cron service itself. When designing the system, we opted for a bit of both.

We store the Paxos logs on local disk of the machine where Cron service replicas are scheduled. Having three replicas in default operation implies that we have three copies of the logs. We store the snapshots on local disk as well. However, since they are critical, we also back them up onto a distributed file system, thus protecting against failures affecting all three machines.

We are not storing logs on our distributed file system, as we consciously decided that losing logs, representing a small amount of the most recent state changes, is an acceptable risk. Storing logs on a distributed file system can come with a substantial performance penalty caused by frequent small writes. The simultaneous loss of all three machines is unlikely, but in case it happens, we will automatically restore from the snapshot and lose only a small amount of logs since the last snapshot, which we perform at configurable intervals. Of course, these tradeoffs may be different depending on the details of the infrastructure, as well as the requirements for a given Cron system.

In addition to the logs and snapshots stored on the local disk and snapshot backups on the distributed file system, a freshly started replica can fetch the state snapshot and all logs from an already running replica over the network. This makes replica startup independent of any state on the local machine, and makes rescheduling of a replica to a different machine upon restart (or machine death) essentially a non-issue for the reliability of the service.

RUNNING A LARGE CRON

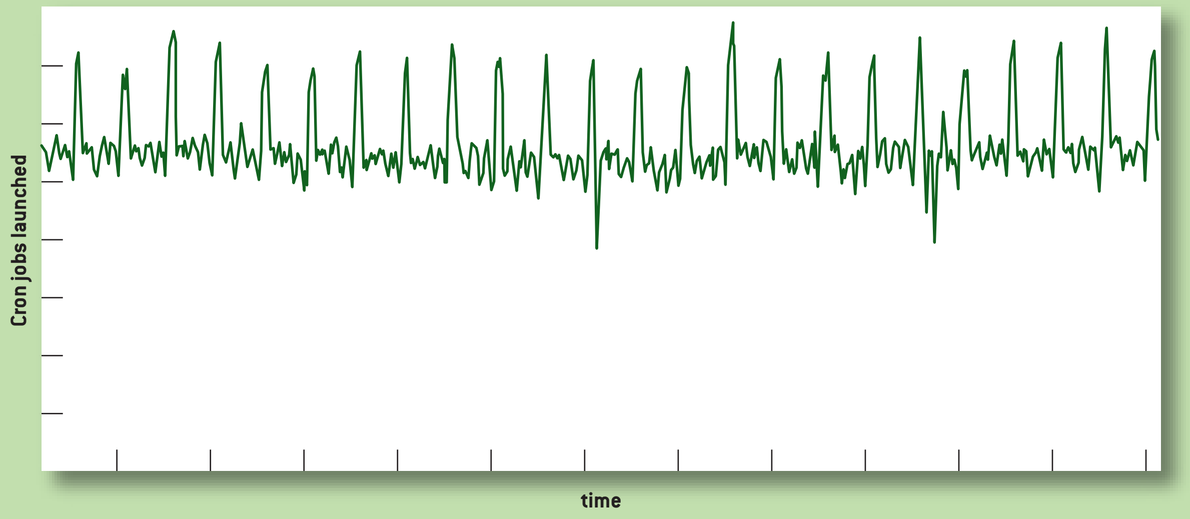
There are other, smaller but no less interesting, implications of running a large Cron deployment. A traditional Cron is small: it probably contains at most tens of Cron jobs. However, if you run a Cron service for thousands of machines in a data center, your usage will match that, and you will run into consequent problems.

A substantial problem is the distributed system’s classic problem of the thundering herd, where the Cron service (based on user configuration) can cause substantial spikes in data center usage. When most people think of a daily Cron job, they immediately think of running it at midnight, and that’s how they configure their Cron jobs. This works just fine if the Cron job launches on the same machine, but what if your job can spawn a MapReduce with thousands of workers? And what if thirty different teams decide to run daily Cron jobs like this in the same data center? To solve this problem, we introduced an extension to the crontab format.

In the ordinary crontab, users specify the minute, hour, day of the month (or week), and month when the Cron job should launch, or asterisk to mean every value. Running at midnight, daily, would then have crontab specification of “0 0 * * *” (i.e., zeroth minute, zeroth hour, every day of the month, every month, and every day of the week). We have introduced the use of the question mark, which means that any value is acceptable, and the Cron system is given the freedom to *choose* the value. This value is chosen by hashing the Cron job configuration over the given time range (e.g., 0…23 for hour), thus distributing those launches more evenly.

FIGURE 3

The Number of Cron Jobs Launched Globally



Despite this change, the load caused by the Cron jobs is still very spiky. The graph in figure 3 illustrates the aggregate number of launches of Cron jobs at Google, globally. This highlights the frequent spikes, which are often caused by jobs that need to be launched at a specific time, for example, due to temporal dependency on external events.

SUMMARY

A Cron service has been a fundamental feature in UNIX systems for decades. The industry move toward large distributed systems, where a data center may be the smallest unit of hardware, requires changes in large portions of the stack, and Cron is no exception. A careful look at the required properties of a Cron service and the requirements of Cron jobs drives our new design.

We have discussed the new constraints and a possible design of a Cron service in a distributed system environment, based on the Google solution. The solution requires strong consistency guarantees in the distributed environment. The core of the distributed Cron implementation is therefore Paxos, a common algorithm for reaching consensus in an unreliable environment. The use of Paxos and correct analysis of new failure modes of Cron jobs in a large-scale, distributed environment allowed us to build a robust Cron service that is heavily used at Google.

ACKNOWLEDGMENTS

Many thanks to Dermot Duffy and Gabriel Krabbe (the designers of the foundation of the system described in this article), Derek Jackson, Chris Jones, Laura Nolan, Niall Murphy, and all other contributors and reviewers for their valuable input on this article.

REFERENCES

1. Burrows, M. 2006. The Chubby lock service for loosely-coupled distributed systems. Proceedings of

the 7th Symposium on Operating Systems Design and Implementation: 335-350. http://research.google.com/archive/chubby-osdi06.pdf
2. Corbett, J. C., et al. 2012. Spanner: Google's globally-distributed database, Proceedings of OSDI'12. Tenth Symposium on Operating System Design and Implementation. http://research.google.com/archive/spanner-osdi2012.pdf
3. Docker. https://www.docker.com/
4. Junqueira, F. P., Reed, B. C., Serafini, M. 2011. Zab: High-performance broadcast for primary-backup systems. Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference: 245-256. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958223&tag=1
5. Lamport, L. 2001. Paxos made simple. ACM SIGACT News 32 (4): 18-25, http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple
6. Lamport, L. 2006. Fast Paxos. Distributed Computing 19 (2): 79-103, http://research.microsoft.com/pubs/64624/tr-2005-112.pdf
7. Ongaro, D., Ousterhout, J. 2014. In search of an understandable consensus algorithm (extended version). https://ramcloud.stanford.edu/raft.pdf

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

ŠTĚPÁN DAVIDOVIČ is a Site Reliability Engineer at Google on the Ads Serving SRE team charged with the reliability of the AdSense product. He also works on a wide range of shared infrastructure projects at Google. He obtained his bachelor's degree from Czech Technical University, Prague, in 2010.

KAVITA GULIANI is a Technical Writer for technical infrastructure and site reliability engineers at Google Mountain View. Before working at Google, Kavita worked for companies like Symantec, Cisco, and Lam Research Corporation. She holds a degree in English from Delhi University with Technical Writing education from San José State University.