Skip to content

Martin Kleppmann

- About/Contact

How to do distributed locking

Published by Martin Kleppmann on 08 Feb 2016.

As part of the research for my book, I came across an algorithm called Redlock on the Redis website. The algorithm claims to implement fault-tolerant distributed locks (or rather, leases [1]) on top of Redis, and the page asks for feedback from people who are into distributed systems. The algorithm instinctively set off some alarm bells in the back of my mind, so I spent a bit of time thinking about it and writing up these notes.

Since there are already over 10 independent implementations of Redlock and we don't know who is already relying on this algorithm, I thought it would be worth sharing my notes publicly. I won't go into other aspects of Redis, some of which have already been critiqued elsewhere.

Before I go into the details of Redlock, let me say that I quite like Redis, and I have successfully used it in production in the past. I think it's a good fit in situations where you want to share some transient, approximate, fast-changing data between servers, and where it's not a big deal if you occasionally lose that data for whatever reason. For example, a good use case is maintaining request counters per IP address (for rate limiting purposes) and sets of distinct IP addresses per user ID (for abuse detection).

However, Redis has been gradually making inroads into areas of data management where there are stronger consistency and durability expectations – which worries me, because this is not what Redis is designed for. Arguably, distributed locking is one of those areas. Let's examine it in some more detail.

What are you using that lock for?

这是作者的大论点之一。从用锁来做什么入手：
– for efficiency，比如一些复杂计算不重复做
– for correctness，不允许多个进程同时运行某段代码

The purpose of a lock is to ensure that among several nodes that might try to do the same piece of work, only one actually does it (at least only one at a time). That work might be to write some data to a shared storage system, to perform some computation, to call some external API, or suchlike. At a high level, there are two reasons why you might want a lock in a distributed application: for efficiency or for correctness [2]. To distinguish these cases, you can ask what would happen if the lock failed:

- **Efficiency:** Taking a lock saves you from unnecessarily doing the same work twice (e.g. some expensive computation). If the lock fails and two nodes end up doing the same piece of work, the result is a minor increase in cost (you end up paying 5 cents more to AWS than you otherwise would have) or a minor inconvenience (e.g. a user ends up getting the same email notification twice).
- **Correctness:** Taking a lock prevents concurrent processes from stepping on each others' toes and messing up the state of your system. If the lock fails and two nodes concurrently work on the same piece of data, the result is a corrupted file, data loss, permanent inconsistency, the wrong dose of a drug administered to a patient, or some other serious problem.

Both are valid cases for wanting a lock, but you need to be very clear about which one of the two you are dealing with.

I will argue that if you are using locks merely for efficiency purposes, it is unnecessary to incur the cost and complexity of Redlock, running 5 Redis servers and checking for a majority to acquire your lock. You are better off just using a single Redis instance, perhaps with asynchronous replication to a secondary instance in case the primary crashes.

作者认为如果为了 efficiency，不必引入 Redlock 的复杂性，直接一套主从 Redis 就可以了。因为即使 Redis 挂掉了，部分锁丢了，大不了就是一些耗时计算多做几次而已。This "no big deal" scenario is where Redis shines.

If you use a single Redis instance, of course you will drop some locks if the power suddenly goes out on your Redis node, or something else goes wrong. But if you're only using the locks as an efficiency optimization, and the crashes don't happen too often, that's no big deal. This "no big deal" scenario is where Redis shines. At least if you're relying on a single Redis instance, it is clear to everyone who looks at the system that the locks are approximate, and only to be used for non-critical purposes.

Redlock 的 5 复本及 majority voting，暗示了它是个针对 correctness 的场景。但事实上它并不能保证 correctness。

On the other hand, the Redlock algorithm, with its 5 replicas and majority voting, looks at first glance as though it is suitable for situations in which your locking is important for *correctness*. I will argue in the following sections that it is *not* suitable for that purpose. For the rest of this article we will assume that your locks are important for correctness, and that it is a serious bug if two different nodes concurrently believe that they are holding the same lock.

这一节解释了为什么 Redlock 是无法保证 correctness 的。主要见下面图片。

Protecting a resource with a lock

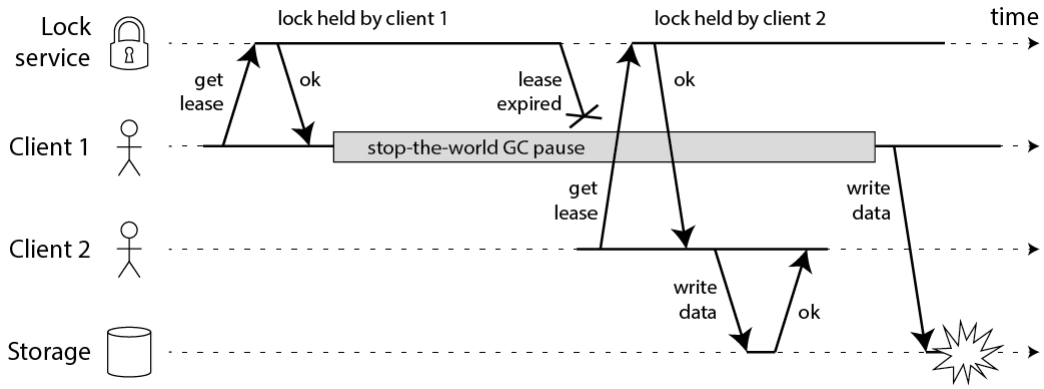
Let's leave the particulars of Redlock aside for a moment, and discuss how a distributed lock is used in general (independent of the particular locking algorithm used). It's important to remember that a lock in a distributed system is not like a mutex in a multi-threaded application. It's a more complicated beast, due to the problem that different nodes and the network can all fail independently in various ways.

For example, say you have an application in which a client needs to update a file in shared storage (e.g. HDFS or S3). A client first acquires the lock, then reads the file, makes some changes, writes the modified file back, and finally releases the lock. The lock prevents two clients from performing this read-modify-write cycle concurrently, which would result in lost updates. The code might look something like this:

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

Unfortunately, even if you have a perfect lock service, the code above is broken. The following diagram shows how you can end up with corrupted data:



Redlock 的主要问题是：

只要某 client 在 lock 的租期到之前未完成相应的代码执行，就必然会发生 lost update。

引起代码执行超过预计时间的原因有多种：

- Stop-the-world GC
- 网络延时
- CPU 调度问题等等

In this example, the client that acquired the lock is paused for an extended period of time while holding the lock – for example because the garbage collector (GC) kicked in. The lock has a timeout (i.e. it is a lease), which is always a good idea (otherwise a crashed client could end up holding a lock forever and never releasing it). However, if the GC pause lasts longer than the lease expiry period, and the client doesn't realise that it has expired, it may go ahead and make some unsafe change.

This bug is not theoretical: HBase used to have this problem [3,4]. Normally, GC pauses are quite short, but "stop-the-world" GC pauses have sometimes been known to last for several minutes [5] – certainly long enough for a lease to expire. Even so-called "concurrent" garbage collectors like the HotSpot JVM's CMS cannot fully run in parallel with the application code – even they need to stop the world from time to time [6].

因为程序可能在任一时刻停住较长时间，因此你在写回存储时再检查一次 lock 有效性，也是起不到保护作用的。

You cannot fix this problem by inserting a check on the lock expiry just before writing back to storage. Remember that GC can pause a running thread at *any point*, including the point that is maximally inconvenient for you (between the last check and the write operation).

And if you're feeling smug because your programming language runtime doesn't have long GC pauses, there are many other reasons why your process might get paused. Maybe your process tried to read an address that is not yet loaded into memory, so it gets a page fault and is paused until the page is loaded from disk. Maybe your disk is actually EBS, and so reading a variable unwittingly turned into a synchronous network request over Amazon's congested network. Maybe there are many other processes contending for CPU, and you hit a black node in your scheduler tree. Maybe someone accidentally sent SIGSTOP to the process. Whatever. Your processes will get paused.

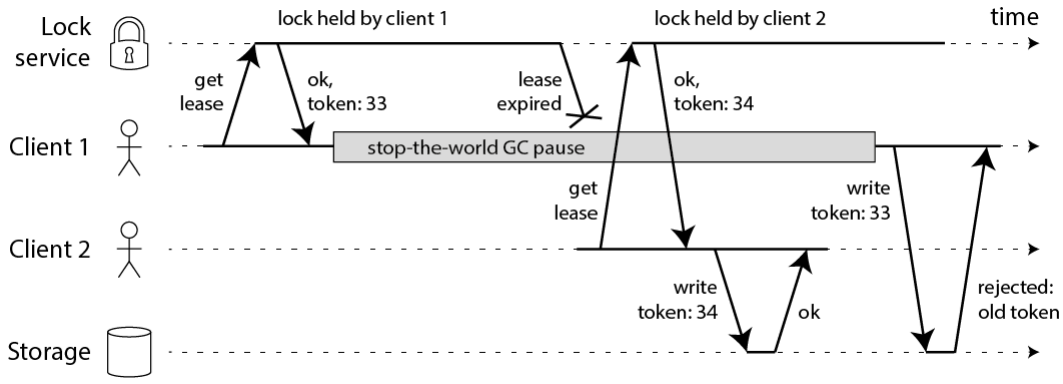
If you still don't believe me about process pauses, then consider instead that the file-writing request may get delayed in the network before reaching the storage service. Packet networks such as Ethernet and IP may delay packets *arbitrarily*, and they do [7]: in a famous incident at GitHub, packets were delayed in the network for approximately 90 seconds [8]. This means that an application process may send a write request, and it may reach the storage server a minute later when the lease has already expired.

Even in well-managed networks, this kind of thing can happen. You simply cannot make any assumptions about timing, which is why the code above is fundamentally unsafe, no matter what lock service you use.

重点之一。使用 Fencing token 来保证 correctness。
实现看下图。

Making the lock safe with fencing

The fix for this problem is actually pretty simple: you need to include a *fencing token* with every write request to the storage service. In this context, a fencing token is simply a number that increases (e.g. incremented by the lock service) every time a client acquires the lock. This is illustrated in the following diagram:



看这个图。Fencing token 是 lock service 返回给 client 的一个自增的 ID。
Client 1 获得锁时，得到 token 33；它写入存储时，需要带着这个 token 33 给存储服务。此时存储服务记得 token 34 的请求已经写入，因此拒绝 33 的写入。

我觉得这里未提到的问题是：Storage 需要保证写入内容和记录最后 token 这两个操作一起是原子的，不能被中断。

Client 1 acquires the lease and gets a token of 33, but then it goes into a long pause and the lease expires. Client 2 acquires the lease, gets a token of 34 (the number always increases), and then sends its write to the storage service, including the token of 34. Later, client 1 comes back to life and sends its write to the storage service, including its token value 33. However, the storage server remembers that it has already processed a write with a higher token number (34), and so it rejects the request with token 33.

Note this requires the storage server to take an active role in checking tokens, and rejecting any writes on which the token has gone backwards. But this is not particularly hard, once you know the trick. And provided that the lock service generates strictly monotonically increasing tokens, this makes the lock safe. For example, if you are using ZooKeeper as lock service, you can use the `zxid` or the znode version number as fencing token, and you're in good shape [3].

实现上的两个重点：Storage server 检查 token；Lock service 提供单调增长 token。比如 ZooKeeper 可以作为 lock service。

However, this leads us to the first big problem with Redlock: *it does not have any facility for generating fencing tokens*. The algorithm does not produce any number that is guaranteed to increase every time a client acquires a lock. This means that even if the algorithm were otherwise perfect, it would not be safe to use, because you cannot prevent the race condition between clients in the case where one client is paused or its packets are delayed.

And it's not obvious to me how one would change the Redlock algorithm to start generating fencing tokens. The unique random value it uses does not provide the required monotonicity. Simply keeping a counter on one Redis node would not be sufficient, because that node may fail. Keeping counters on several nodes would mean they would go out of sync. It's likely that you would need a consensus algorithm just to generate the fencing tokens. (If only incrementing a counter was simple.)

Using time to solve consensus

这段在讲 Redlock 除了可能有 lost update 问题外，它的 timing assumption 也可能导致问题。

The fact that Redlock fails to generate fencing tokens should already be sufficient reason not to use it in situations where correctness depends on the lock. But there are some further problems that are worth discussing.

In the academic literature, the most practical system model for this kind of algorithm is the asynchronous model with unreliable failure detectors [9]. In plain English, this means that the algorithms make no assumptions about timing: processes may pause for arbitrary lengths of time, packets may be arbitrarily delayed in the network, and clocks may be arbitrarily wrong – and the algorithm is nevertheless expected to do the right thing. Given what we discussed above, these are very reasonable assumptions.

这类领域的算法，应该对时间没有假设：进程可能中断任意长的时间，网络包可能有任意长的延时，甚至本地时钟也可能是错的。比如 gettimeofday 的结果就可能跳来跳去（在 admin 手动设置或者跟 NTP server 同步时）

The only purpose for which algorithms may use clocks is to generate timeouts, to avoid waiting forever if a node is down. But timeouts do not have to be accurate: just because a request times out, that doesn't mean that the other node is definitely down – it could just as well be that there is a large delay in the network, or that your local clock is wrong. When used as a failure detector, timeouts are just a guess that something is wrong. (If they could, distributed

Clock 应该只用来生成超时。

algorithms would do without clocks entirely, but then consensus becomes impossible [10]. Acquiring a lock is like a compare-and-set operation, which requires consensus [11].)

Note that Redis uses `gettimeofday`, not a monotonic clock, to determine the expiry of keys. The man page for `gettimeofday` explicitly says that the time it returns is subject to discontinuous jumps in system time – that is, it might suddenly jump forwards by a few minutes, or even jump back in time (e.g. if the clock is stepped by NTP because it differs from a NTP server by too much, or if the clock is manually adjusted by an administrator). Thus, if the system clock is doing weird things, it could easily happen that the expiry of a key in Redis is much faster or much slower than expected.

For algorithms in the asynchronous model this is not a big problem: these algorithms generally ensure that their *safety* properties always hold, without making any timing assumptions [12]. Only *liveness* properties depend on timeouts or some other failure detector. In plain English, this means that even if the timings in the system are all over the place (processes pausing, networks delaying, clocks jumping forwards and backwards), the performance of an algorithm might go to hell, but the algorithm will never make an incorrect decision.

异步模型的算法不做 timing assumption，假定上文提到的各种中断和延时都发生，也能保证算法是正确的，即使它的性能会差到爆。

However, Redlock is not like this. Its safety depends on a lot of timing assumptions: it assumes that all Redis nodes hold keys for approximately the right length of time before expiring; that the network delay is small compared to the expiry duration; and that process pauses are much shorter than the expiry duration.

但 Redlock 做了很多 timing assumption，这导致它必然是不够安全的，只要你构造精细的 timing 就可以找出问题。

Breaking Redlock with bad timings

Let's look at some examples to demonstrate Redlock's reliance on timing assumptions. Say the system has five Redis nodes (A, B, C, D and E), and two clients (1 and 2). What happens if a clock on one of the Redis nodes jumps forward?

1. Client 1 acquires lock on nodes A, B, C. Due to a network issue, D and E cannot be reached.
2. The clock on node C jumps forward, causing the lock to expire.
3. Client 2 acquires lock on nodes C, D, E. Due to a network issue, A and B cannot be reached.
4. Clients 1 and 2 now both believe they hold the lock.

A similar issue could happen if C crashes before persisting the lock to disk, and immediately restarts. For this reason, the Redlock documentation recommends delaying restarts of crashed nodes for at least the time-to-live of the longest-lived lock. But this restart delay again relies on a reasonably accurate measurement of time, and would fail if the clock jumps.

Okay, so maybe you think that a clock jump is unrealistic, because you're very confident in having correctly configured NTP to only ever slew the clock. In that case, let's look at an example of how a process pause may cause the algorithm to fail:

1. Client 1 requests lock on nodes A, B, C, D, E.
2. While the responses to client 1 are in flight, client 1 goes into stop-the-world GC.
3. Locks expire on all Redis nodes.
4. Client 2 acquires lock on nodes A, B, C, D, E.
5. Client 1 finishes GC, and receives the responses from Redis nodes indicating that it successfully acquired the lock (they were held in client 1's kernel network buffers while the process was paused).
6. Clients 1 and 2 now both believe they hold the lock.

Note that even though Redis is written in C, and thus doesn't have GC, that doesn't help us here: any system in which the *clients* may experience a GC pause has this problem. You can only make this safe by preventing client 1 from performing any operations under the lock after client 2 has acquired the lock, for example using the fencing approach above.

A long network delay can produce the same effect as the process pause. It perhaps depends on your TCP user timeout – if you make the timeout significantly shorter than the Redis TTL, perhaps the delayed network packets would be ignored, but we'd have to look in detail at the TCP implementation to be sure. Also, with the timeout we're back down to accuracy of time measurement again!

The synchrony assumptions of Redlock

These examples show that Redlock works correctly only if you assume a *synchronous* system model – that is, a system with the following properties:

- bounded network delay (you can guarantee that packets always arrive within some guaranteed maximum delay),
- bounded process pauses (in other words, hard real-time constraints, which you typically only find in car airbag systems and suchlike), and
- bounded clock error (cross your fingers that you don't get your time from a bad NTP server).

Note that a synchronous model does not mean exactly synchronised clocks: it means you are assuming a *known, fixed upper bound* on network delay, pauses and clock drift [12]. Redlock assumes that delays, pauses and drift are all small relative to the time-to-live of a lock; if the timing issues become as large as the time-to-live, the algorithm fails. Redlock 本质的问题在于对时间同步性的预设：网络延迟、process pauses 等相对 TTL 是比较小的。

In a reasonably well-behaved datacenter environment, the timing assumptions will be satisfied *most* of the time – this is known as a partially synchronous system [12]. But is that good enough? As soon as those timing assumptions are broken, Redlock may violate its safety properties, e.g. granting a lease to one client before another has expired. If you're depending on your lock for correctness, "most of the time" is not enough – you need it to *always* be correct.

There is plenty of evidence that it is not safe to assume a synchronous system model for most practical system environments [7,8]. Keep reminding yourself of the GitHub incident with the 90-second packet delay. It is unlikely that Redlock would survive a Jepsen test.

On the other hand, a consensus algorithm designed for a partially synchronous system model (or asynchronous model with failure detector) actually has a chance of working. Raft, Viewstamped Replication, Zab and Paxos all fall in this category. Such an algorithm must let go of all timing assumptions. That's hard: it's so tempting to assume networks, processes and clocks are more reliable than they really are. But in the messy reality of distributed systems, you have to be very careful with your assumptions.

Conclusion

I think the Redlock algorithm is a poor choice because it is "neither fish nor fowl": it is unnecessarily heavyweight and expensive for efficiency-optimization locks, but it is not sufficiently safe for situations in which correctness depends on the lock.

In particular, the algorithm makes dangerous assumptions about timing and system clocks (essentially assuming a synchronous system with bounded network delay and bounded execution time for operations), and it violates safety

properties if those assumptions are not met. Moreover, it lacks a facility for generating fencing tokens (which protect a system against long delays in the network or in paused processes).

If you need locks only on a best-effort basis (as an efficiency optimization, not for correctness), I would recommend sticking with the straightforward single-node locking algorithm for Redis (conditional set-if-not-exists to obtain a lock, atomic delete-if-value-matches to release a lock), and documenting very clearly in your code that the locks are only approximate and may occasionally fail. Don't bother with setting up a cluster of five Redis nodes.

On the other hand, if you need locks for correctness, please don't use Redlock. Instead, please use a proper consensus system such as ZooKeeper, probably via one of the Curator recipes that implements a lock. (At the very least, use a database with reasonable transactional guarantees.) And please enforce use of fencing tokens on all resource accesses under the lock.

这里的材料推荐值得一看。Curator recipes 和 ZooKeeper 的内容。但看到这里还不知道 fencing token 怎么实现。需要额外补充下这块内容。

As I said at the beginning, Redis is an excellent tool if you use it correctly. None of the above diminishes the usefulness of Redis for its intended purposes. Salvatore has been very dedicated to the project for years, and its success is well deserved. But every tool has limitations, and it is important to know them and to plan accordingly.

If you want to learn more, I explain this topic in greater detail in chapters 8 and 9 of my book, now available in Early Release from O'Reilly. (The diagrams above are taken from my book.) For learning how to use ZooKeeper, I recommend Junqueira and Reed's book [3]. For a good introduction to the theory of distributed systems, I recommend Cachin, Guerraoui and Rodrigues' textbook [13].

Thank you to Kyle Kingsbury, Camille Fournier, Flavio Junqueira, and Salvatore Sanfilippo for reviewing a draft of this article. Any errors are mine, of course.

Update 9 Feb 2016: Salvatore, the original author of Redlock, has posted a rebuttal to this article (see also HN discussion). He makes some good points, but I stand by my conclusions. I may elaborate in a follow-up post if I have time, but please form your own opinions – and please consult the references below, many of which have received rigorous academic peer review (unlike either of our blog posts).

References

[1] Cary G Gray and David R Cheriton: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," at *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:10.1145/74850.74870

[2] Mike Burrows: "The Chubby lock service for loosely-coupled distributed systems," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[3] Flavio P Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, November 2013. ISBN: 978-1-4493-6130-3

[4] Enis Söztutar: "HBase and HDFS: Understanding filesystem usage in HBase," at *HBaseCon*, June 2013.

[5] Todd Lipcon: "Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1," blog.cloudera.com, 24 February 2011.

[6] Martin Thompson: "Java Garbage Collection Distilled," mechanical-sympathy.blogspot.co.uk, 16 July 2013.

[7] Peter Bailis and Kyle Kingsbury: "The Network is Reliable," *ACM Queue*, volume 12, number 7, July 2014. doi:10.1145/2639988.2639988

[8] Mark Imbriaco: "Downtime last Saturday," github.com, 26 December 2012.

[9] Tushar Deepak Chandra and Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647

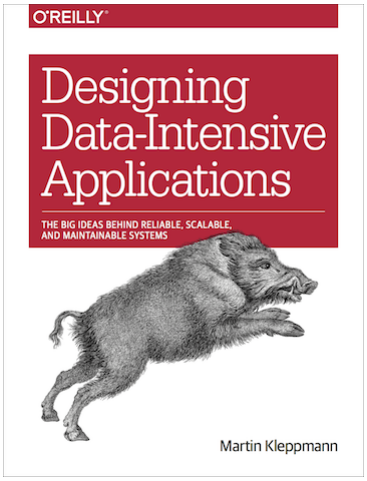
[10] Michael J Fischer, Nancy Lynch, and Michael S Paterson: "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, volume 32, number 2, pages 374–382, April 1985. doi:10.1145/3149.214121

[11] Maurice P Herlihy: "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems*, volume 13, number 1, pages 124–149, January 1991. doi:10.1145/114005.102808

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[13] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, Second Edition. Springer, February 2011. ISBN: 978-3-642-15259-7, doi:10.1007/978-3-642-15260-3

My book



My book, *Designing Data-Intensive Applications*, has received thousands of five-star reviews.

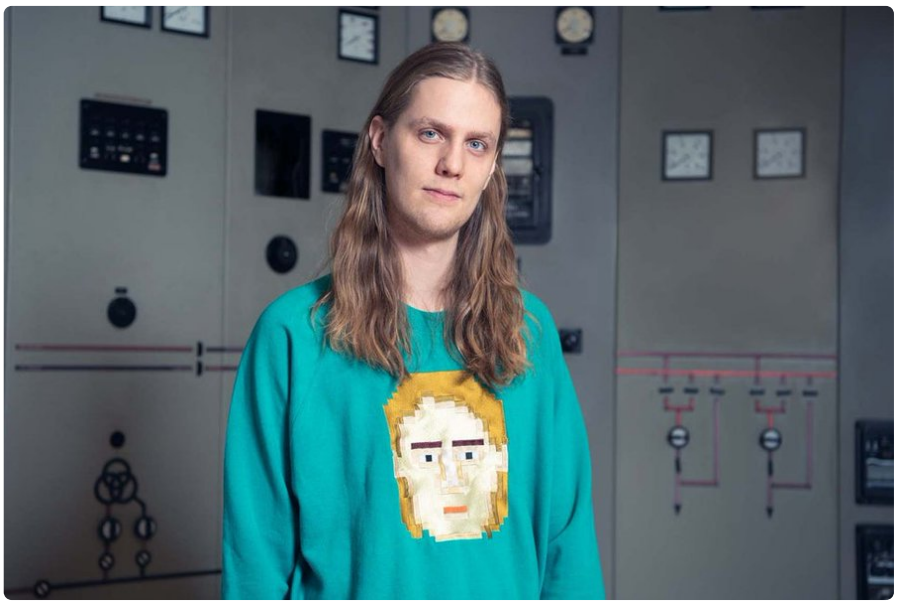
I am a researcher at the University of Cambridge, working on the TRVE DATA project at the intersection of databases, distributed systems, and information security.

If you find my work useful, please support me on Patreon.

Tweets by @martinkl

Martin Kleppmann
@martinkl

I have been informed that I can now have a second career as lookalike of Daði Freyr, Iceland's entry at Eurovision



1h

Martin Kleppmann
@martinkl

Embed View on Twitter

Recent posts

- 14 Apr 2021: It's time to say goodbye to the GPL
- 23 Feb 2021: Building the future of computing, with your help
- 13 Jan 2021: Decentralised content moderation
- 02 Dec 2020: Using Bloom filters to efficiently synchronise hash graphs
- 18 Nov 2020: New courses on distributed systems and elliptic curve cryptography
- Full archive

Conference talks

- 02 Jul 2021 at 15th ACM International Conference on Distributed and Event-based Systems (DEBS)
- 04 Jun 2021 at Craft Conference
- 04 May 2021 at Emerging Technologies for the Enterprise (ETE)
- 06 Jul 2020 at Hydra distributed computing conference
- 27 Apr 2020 at 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)
- Full archive