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?

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.

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.

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.

Protecting a resource with a lock

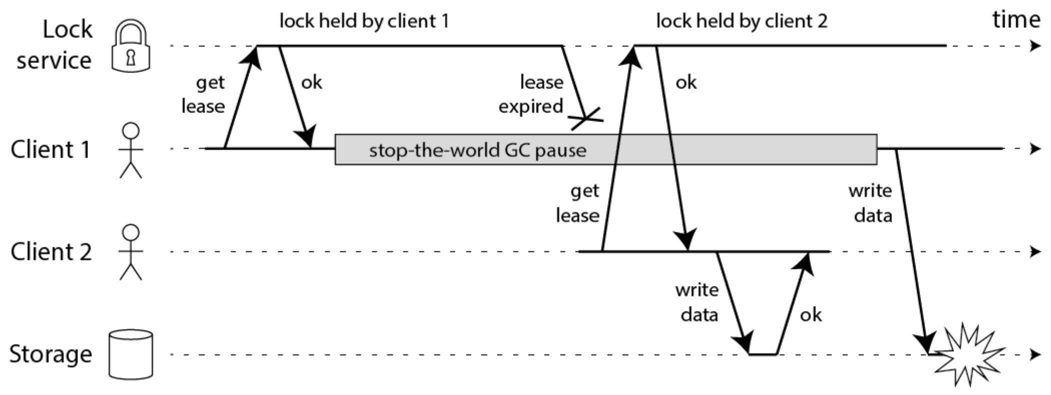
Let's leave the particulars of Redlock aside for a moment, and discuss how a distributed lock is used in general (independent of the particular locking algorithm used). It's important to remember that a lock in a distributed system is not like a mutex in a multi-threaded application. It's a more complicated 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:



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].

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).

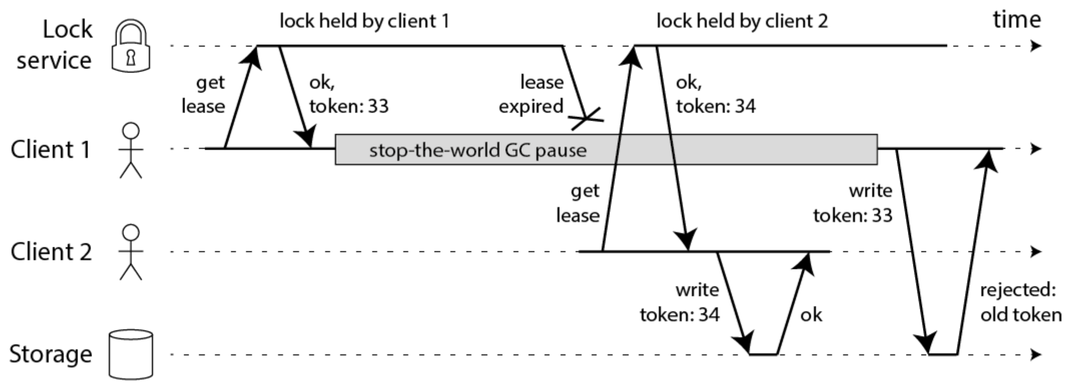
And if you're feeling smug because your programming language runtime doesn't have long GC pauses, there are many other reasons why your process might get paused. Maybe your process tried to read an address that is not yet loaded into memory, so it gets a page fault and is paused until the page is loaded 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.

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:



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].

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

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.

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 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.

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.

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.

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.

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

Join the discussion about this article on Hacker News.

28 Comments **Martin Kleppmann's Blog** 1 **Login**

Recommend 2 **Share** **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

李涛 • 3 months ago

Today, I learn and think about the "redlock:. I am very agree you!

1 • Reply • Share ›

Douglas Muth • 2 years ago

"Instead, please use a proper consensus system such as ZooKeeper, probably."

I literally skimmed the entire article for this part. :-)

That said, it's a good writeup, and very informative. Thanks for sharing!

1 ^ | ˅ • Reply • Share ›

서지웅 • 3 months ago

great post! may i translate this article to korean?

^ | ˅ • Reply • Share ›

Martin Kleppmann Mod ➦ 서지웅 • 2 months ago

Sure, all articles on this site are licensed under creative commons: https://creativecommons.org... — you're welcome to translate it, and please include a link back to the original here.

^ | ˅ • Reply • Share ›

Huaqing Li • 3 months ago

Hi **@Martin Kleppmann** , there is one confusion I'd like to hear your thoughts about. In the red lock solution, a client is considered to successfully acquire the lock should get locks from the majority of the redis masters(that is at least N/2+1 out of N redis instances). I'm expecting a more clear definition of the 'majority'. As in the scenario where there are 5 redis instances and 3 of them are down, then for all the clients, it is impossible that anyone would get the more than half of the locks. So should we always check how are running redis instances before we acquire a lock? But if so, every client may get different result at different time.

^ | ˅ • Reply • Share ›

Martin Kleppmann Mod ➦ Huaqing Li • 2 months ago

The definition of "majority" assumes that you know how many redis instances you have (N), which would typically be part of your fixed configuration of the system. Given that you know N, a majority requires that you get votes from more than half of the instances. If 3 out of 5 instances are down, that does not mean N=2; it is still the case that N=5. In this case you will not be able to get the three required votes, and so you will not be able to acquire the lock. You don't need to explicitly check whether an instance is down, because a crashed instance is handled exactly the same way as an instance you cannot reach due to a network problem, or an instance that fails to vote for any other reason.

4 ^ | ˅ • Reply • Share ›

Huaqing Li ➦ Martin Kleppmann • 2 months ago

That makes sense to me. I probably missed something about the solution to it, is it introduced in your article? Thx!

^ | ˅ • Reply • Share ›

John Mullaney • a year ago

Why should the fencing token need to be monotonically increasing?

Wouldn't it just need to be different on each operation? E.g., couldn't a UUID be used? The key thing is checking whether the write token value is equal to the current token value. Less-thans and greater-thans wouldn't come into it.

• Reply • Share ›

Martin Kleppmann Mod ➦ John Mullaney • a year ago

The storage system that checks the fencing tokens doesn't know when a new client is granted the lock, it can only go by the token. If you used a UUID and a request turned up with a UUID that has never been seen before, how would the storage system know what to do? Perhaps the lock has been granted to a new client, so the new UUID should be accepted and the old UUID should be blocked. But perhaps the UUID is from a client that acquired the lock and then immediately paused before it was able to make any requests; in that case, a third client will have already acquired the lock, and the previously-unseen UUID should be blocked. Using monotonically increasing fencing tokens solves this conundrum.

• Reply • Share ›

John Mullaney ➦ Martin Kleppmann • a year ago

Thanks, I see. hm... though now I wonder if locks/leases aren't needlessly complex for correctness. If I'm understanding this, the locking assumes the storage system provides (a) a reliable check of a token value coming in with the write request and (b) does the check as part of an atomic check-write operation.

With that, a storage system could ensure correctness without a lock service like this: it maintains a current token value for a resource. A client reads this token value along with the resource. With the write operation the client sends along a two part token: the first part is the original token value and second part is a new unique value. It must be unique so it is a UUID. The storage system, in its check-write atomic operation checks that the first part of the token matches the resource's current token value. If it matches the the second part of the incoming token is written as the new current token value of the resource (along with the rest of the write operation). If it doesn't match, the write fails.

I think that's correctness where the only assumptions are the uniqueness of the tokens provided by the clients with the write operations and the atomicity of the storage service's check-write operation -- no locking/leasing assumed.

see more

• Reply • Share ›

Martin Kleppmann Mod ➦ John Mullaney • a year ago

What you describe is a viable approach, and it is quite similar to the causal context used in Riak, for example. If all you care about is managing writes to a storage system, you're right that a lock/lease may be overkill.

In general, I'd say that people call for a leasing system if there needs to be one instance of some process for some reason. For example, perhaps you have a process that sends out a bunch of emails, and you only want exactly one instance of that process running (zero instances would mean no emails sent, multiple instances would mean duplicate emails).

In this blog post, I have tried to draw attention to the fact that in the presence of process pauses, you have to think about the interaction between different stateful systems. The nature of these interactions depends on the particulars of your application, of course.

These blog comments aren't really a great medium for such subtle discussion, so for further details I'd like to politely refer you to my book http://dataintensive.net/ :)

• Reply • Share ›

idelvall • a year ago

Great post! But I think a clarification has to be made regarding fencing tokens: What happens in the (unlikely) case that client 2 also suffers a STW pause and they write with increasing successive tokens?

• Reply • Share ›

Martin Kleppmann Mod ➦ idelvall • a year ago

The storage system simply maintains the 'ratchet' that the token can only stay the same or increase, but not decrease. Thus, if client 2 pauses and client 3 acquires the lock, client 2 will have a lesser token. If client 3 has already made a request to the storage service, client 1 and 2 will both be blocked.

• Reply • Share ›

idelvall ➦ Martin Kleppmann • a year ago

what I mean is:
client 1 acquires lock
client 1 stops
client 1 lock expires
client 2 acquires lock
client 2 stops
client 1 resumes and writes to storage
client 2 resumes and writes to storage

This is more unlikely than the scenario you presented, but still possible, and breaks the desired "correctness" since both writes are accepted.

• Reply • Share ›

Martin Kleppmann Mod ➦ idelvall • a year ago

Oh, I see what you mean now. Yes, you have a good point — that scenario does look risky. To reason about it properly, I think we would need to make some assumptions about the semantics of the write that occurs under the lock; I think it would probably turn out to be safe in some cases, but unsafe in others. I will think about it some more.

In consensus algorithms that use epochs/ballot numbers/round numbers (which have a similar function to the fencing token), the algorithm works because the type of write is constrained. Thus, Paxos for example can maintain the invariant that if one node decides x, no other node will decide a value other than x, even in the presence of arbitrary pauses. If unconstrained writes were allowed, the safety property could be violated.

Perhaps it would be useful to regard a storage system with fencing token support as participating in an approximation of a consensus algorithm, but further protocol constraints would be required to make it safe in all circumstances?

• Reply • Share ›

idelvall Martin Kleppmann • a year ago

After some thinking, this is how I would do it:

In lock manager:
- If since last released lock, no other (later) has been expired, then next returned token is an "ordinary token" (incrementing the previous one)
- Otherwise, the next returned token is a "paired token" containing major/minor information, being major: the current token numbering, and minor: the numbering of the first token not released at this time

In lock-aware resources:
- Keep record of the highest accepted token
- If the current token is ordinary then behave as usual (rejecting when it's not greater than the highest)
- If the current token is paired (granted after some others expiration) then accept only if its minor number is greater than highest known

This would be consistent with my previous example:
-client 1 acquires lock (token: "1")
-client 1 stops

see more

• Reply • Share ›

Martin Kleppmann Mod idelvall • a year ago

Interesting idea. Seems plausible at first glance, but it's the kind of subtle protocol that would benefit from a formal proof of correctness. In these distributed systems topics it's terribly easy to accidentally overlook some edge-case.

• Reply • Share ›

Jeff Jeff Martin Kleppmann • 7 months ago

This actually looks a lot like the algorithm proposed in the paper "On Optimistic Methods for Concurrency Control" by Kung and Robinson, 1981 (http://www.eecs.harvard.edu...

I believe that this paper addresses the exact issues that @idelvall mentioned and also includes a formal proof. Additionally, in the case proposed as long as client 1 and client 2 have no conflicts in what they are writing then both would still be permitted, however in this case if there was a conflict then client 2 would be rejected prior to starting its write. Would be interested in hearing your thoughts on this though.

^ | ˅ • Reply • Share ›

idelvall ➦ Martin Kleppmann • a year ago

Agreed, just an idea. I'll take a look into Chubby's paper and see how they handle this

^ | ˅ • Reply • Share ›

Jaimie • a year ago

This is a really good resource if someone is learning for distributed locking. Here's a good example on how to use it in a distributed cache.
http://blogs.alachisoft.com...

^ | ˅ • Reply • Share ›

antirez • 2 years ago

Note for the readers: that there is an error in the way the Redlock algorithm is used in the blog post: the final step after the majority is acquired, is to check if the total time elapsed is already over the lock TTL, and in such a case the client does not consider the lock as valid. This makes Redlock immune from client <-> lock-server delays in the messages, and makes every other delay *after* the lock validity is tested as any other GC pause during the processing of the locked resource. This is also equivalent to what happens, when using a remote lock server, if the "OK, you have the lock" reply from the server remains in the kernel buffers since the socket pauses before reading it. So where in this blog post its assumed that network delays or GC pauses during the lock acquisition stage are a problem, there is an error.

^ | ˅ • Reply • Share ›

Martin Kleppmann Mod ➦ antirez • 2 years ago

This is correct, I had overlooked that additional clock check after messages are received. However, I believe that the additional check does not substantially alter the properties of the algorithm:

- Large network delay between the application and the shared resource (the thing that is protected by the lock) can still cause the resource to receive a message from an application process that no longer holds the lock, so fencing is still required.

- A GC pause between the final clock check and the resource access will not be caught by the clock check. As I point out in the article: "Remember that GC can pause a running thread at any point, including the point that is maximally inconvenient for you".

- All the dependencies on accuracy of clock measurement still hold.

• Reply • Share ›

antirez ➦ Martin Kleppmann • 2 years ago

Hello Martin, thanks for your reply. Network delays between the app and the shared resource, and a GC pause *after* the check, but before doing the actual work, are all conceptually exactly the same as the "point 1" of your argument, that is, GC pauses (or other pauses) make the algo require an incremental token. So, regarding the safety of the algorithm itself, the only remaining thing would be the dependency on clock drifts, that can be argued depending on point of view. So I'm sorry to have to say that IMHO the current version of the article, by showing the wrong implementation of the algorithm, and not citing the equivalence of GC pauses processing the shared resource, with GC pauses immediately after the token is acquired, does not provide a fair picture.

• Reply • Share ›

MarutSingh ➦ antirez • a year ago

Bottom line is for an application programmer like me this implementation looks doubtful enough not to use it in production systems. If this does not work perfectly then can introduce bugs which will be impossible to fix.

• Reply • Share ›

Russell • 2 years ago

"that that" in paragraph

"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 that the network delay is small compared to the expiry duration; and that process pauses are much shorter than the expiry duration."

Great article though, I'm just being an editor :P

• Reply • Share ›

Martin Kleppmann Mod ➦ Russell • 2 years ago

Thanks! Fixed.

• Reply • Share ›

Srdjan • 2 years ago

Great write up. Especially in terms of distilling the theory into examples. Perhaps it would be worth reiterating (in the paragraph before the conclusion) that paxos, raft et al are still safe even if the system degenerates into the async model, but progress is no longer guaranteed (i.e. Likeness)

• Reply • Share ›

Srdjan ➦ Srdjan • 2 years ago

Autocorrect killed me :) likeness = liveness

• Reply • Share ›

Subscribe

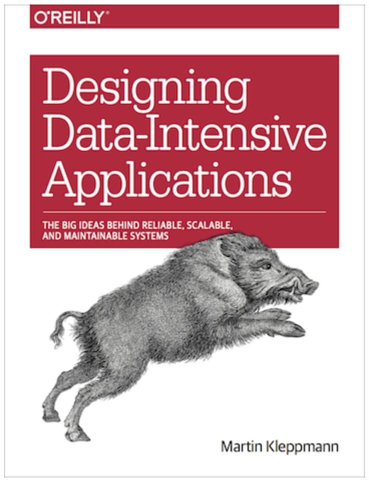
Site RSS feed

To get notified when I write something new, follow me on Twitter, subscribe to the RSS feed, or enter your email address:

Email address: Subscribe

I won't give your address to anyone else, won't send you any spam, and you can unsubscribe at any time.

My book


O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann

My book, *Designing Data-Intensive Applications*, was published by O'Reilly in March 2017.

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.

Tweets by @martinkl

Martin Kleppmann @martinkl
Posters and books from the Kickstarter campaign are packed and ready to go. Packing was surprisingly time-consuming!

11h

Martin Kleppmann @martinkl

Embed View on Twitter

Recent posts

- 15 Mar 2017: Drawing a map of distributed data systems
- 26 Jan 2017: The probability of data loss in large clusters
- 15 Apr 2016: Announcing TRVE DATA: Placing a bit less trust in the cloud
- 30 Mar 2016: Device security and the FBI
- 18 Feb 2016: Should law enforcement services have a backdoor into smartphones?
- Full archive

Conference talks

- 19 Jun 2017 at Curry On
- 15 Nov 2016 at GOTO Berlin
- 03 Nov 2016 at Code Mesh
- 26 Oct 2016 at University of Cambridge Computer Laboratory
- 16 Sep 2016 at Strange Loop
- Full archive