Asynchronous replication with failover

2013/05/21
Software Network Distributed Systems Jepsen Redis

In response to my earlier post on Redis inconsistency, Antirez was kind enough to help clarify some points about Redis Sentinel's design.

First, I'd like to reiterate my respect for Redis. I've used Redis extensively in the past with good results. It's delightfully fast, simple to operate, and offers some of the best documentation in the field. Redis is operationally *predictable*. Data structures and their performance behave just how you'd expect. I hear nothing but good things about the clarity and quality of Antirez' C code. This guy *knows his programming*.

I think Antirez and I agree with each other, and we're both saying the same sorts of things. I'd just like to expand on some of these ideas a bit, and generalize to a broader class of systems.

First, the distributed system comprised of Redis and Redis Sentinel cannot be characterized as consistent. Nor can MongoDB with anything less than WriteConcern.MAJORITY, or MySQL with asynchronous replication, for that matter. Antirez writes:

> What I'm saying here is that just the goal of the system is:
>
> 1) To promote a slave into a master if the master fails.
> 2) To do so in a reliable way.

Redis Sentinel *does* reliably promote secondaries into primaries. It is so good at this that it can promote two, three, or all of your secondaries into primaries concurrently, and keep them in that state indefinitely. As we've seen, having causally unconnected primaries in this kind of distributed system allows for conflicts–and since Redis Sentinel will destroy the state on an old primary when it becomes visible to a quorum of Sentinels, this can lead to arbitrary loss of acknowledged writes to the system.

> Ok I just made clear enough that there is no such goal in Sentinel to turn N Redis instances into a distributed store,

If you use any kind of failover, your Redis system is a distributed store. Heck, reading from secondaries makes Redis a distributed store.

> So you can say, ok, Sentinel has a limited scope, but could you add a feature so that when the master feels in the minority it no longer accept writes? I don't think it's a good idea. What it means to be in the minority for a Redis master monitored by Sentinels (especially given that Redis and Sentinel are completely separated systems)?
>
> Do you want your Redis master stopping to accept writes when it is no longer able to replicate to its slaves?

Yes. This is *required* for a CP system with failover. If you don't do it, your system can and will lose data. You cannot achieve consistency in the face of a partition without sacrificing availability. If you want Redis to be AP, then don't destroy the data on the old primaries by demoting them. Preserve conflicts and surface them to the clients for merging.

You could do this as an application developer by setting every Redis node to be a primary, and writing a proxy layer which uses, say, consistent hashing and active anti-entropy to replicate writes between nodes. Take a look at Antirez's own experiments in this direction. If you want a CP system, you could follow Datomic's model and use immutable shared-structure values in Redis, combined with, say, Zookeeper for mutable state.

Why topology matters

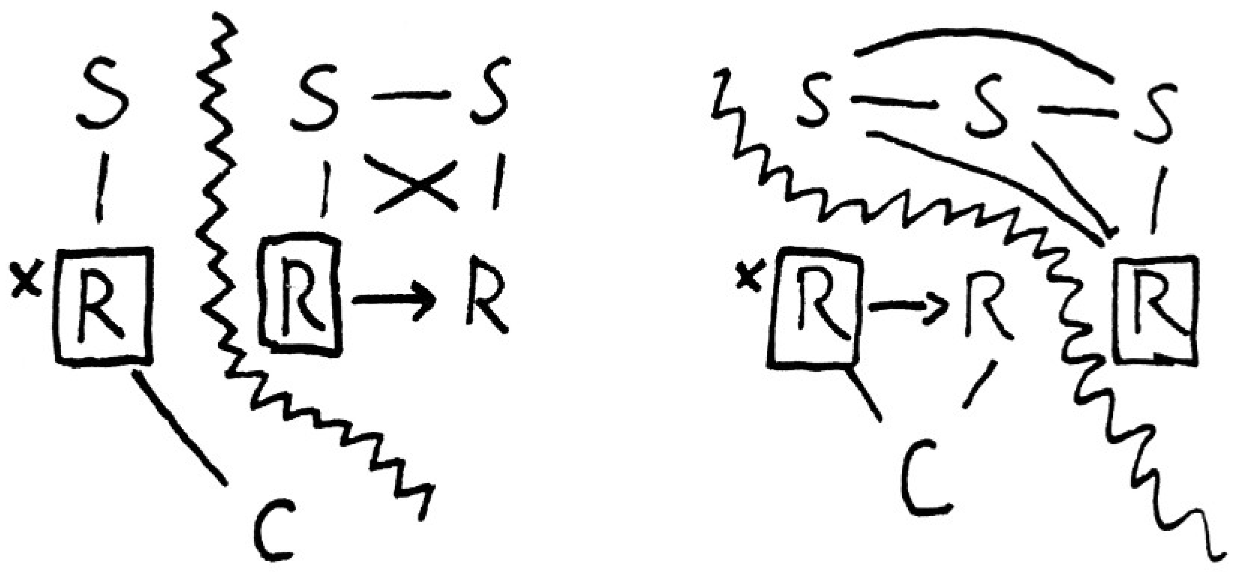
Antirez recommends a different approach to placing Sentinels than I used in my Redis experiments:

> … place your Sentinels and set your quorum so that you are defensive enough against partitions. This way the system will activate only when the issue is really the master node down, not a network problem. Fear data loss and partitions? Have 10 Linux boxes? Put a Sentinel in every box and set quorum to 8.

I… can't parse this statement in a way that makes sense. Adding more boxes to a distributed system doesn't reduce the probability of partitions–and more to the point, *trying to determine the state of a distributed system from outside the system itself is fundamentally flawed*.

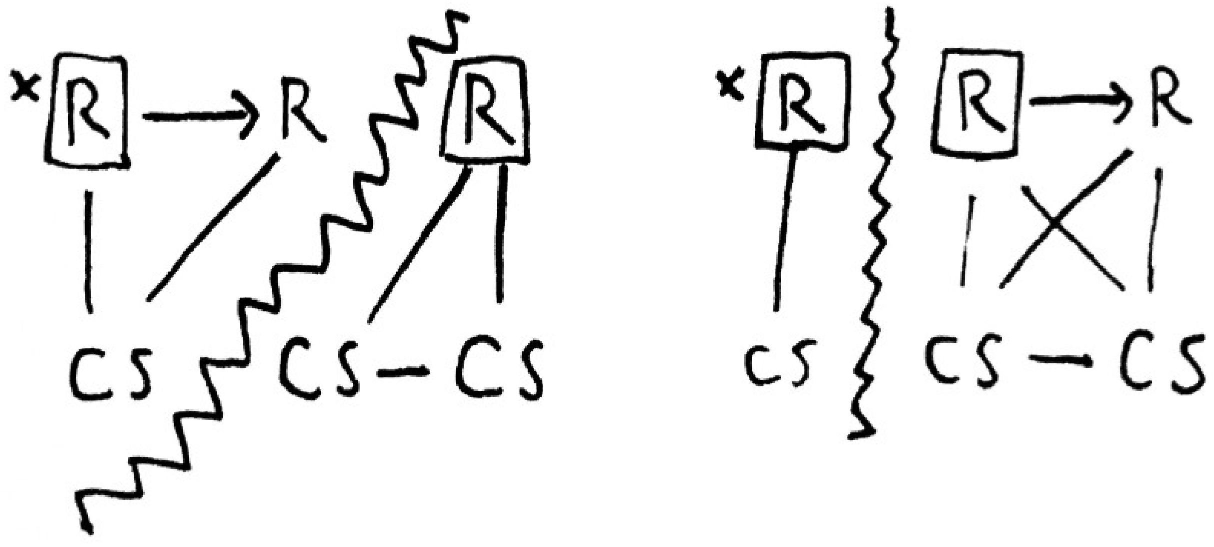
I mentioned that having the nodes which determine the cluster state (the Sentinels) be *separate* from the nodes which actually perform the replication (the Redis servers) can lead to *worse* kinds of partitions. I'd like to explain a little more, because I'm concerned that people might actually be doing this in production.

In this image, S stands for Sentinel, R stands for a Redis server, and C stands for Client. A box around an R indicates that node is a primary, and where it is able to replicate data to a secondary Redis server, an arrow is shown on that path. Lines show open network connections, and the jagged border shows a network partition.



Let's say we place our sentinels on 3 nodes to observe a three-node cluster. In the left-hand scenario, the majority of Sentinels are isolated, with two servers, from the clients. They promote node 2 to be a new primary, and it begins replicating to node 3. Node 1, however, is *still* a primary. Clients will continue writing to node 1, even though a.) its durability guarantees are greatly diminished–if it dies, all writes will be lost, and b.) the node doesn't have a quorum, so it cannot safely accept writes. When the partition resolves, the Sentinels will demote node 1 to a secondary and replace its data with the copy from N2, effectively destroying all writes during the partition.

On the right-hand side, a fully connected group of Sentinels can only see one Redis node. It's *not* safe to promote that node, because it doesn't have a majority and servers won't demote themselves when isolated, but the sentinels do it anyway. This scenario *could* be safely available to clients because a majority is present, but Redis Sentinel happily creates a split-brain and obliterates the data on the first node at some later time.



If you take Antirez' advice and colocate the sentinels with your clients, we can still get in to awful states. On the left, an uneven partition between clients and servers means we elect a *minority Redis server* as the primary, even though it can't replicate to any other nodes. The majority component of the servers can still accept writes, but they're doomed: when the clients are able to see those nodes again, they'll wipe out all the writes that took place on those 2 nodes.

On the right, we've got the same partition topology I demonstrated in the Redis post. Same deal: split brain means conflicting writes and throwing away data.

If you encounter intermittent or rolling partitions (which can happen in the event of congestion and network failover), shifting quorums coupled with the inability of servers to reason about their own cluster state could yield horrifying consequences, like *every* node being a primary at the same time. You might be able to destroy not only writes that took place during the partition, but all data ever written–not sure if the replication protocol allows this or if every node just shuts down.

Bottom line: if you're building a distributed system, you *must* measure connectivity in the distributed system itself, not by what you can see from the outside. Like we saw with MongoDB and Riak, it's not the wall-clock state that matters–it's the logical *messages* in the system. The further you get from those messages, the wider your windows for data loss.

It's not just Sentinel

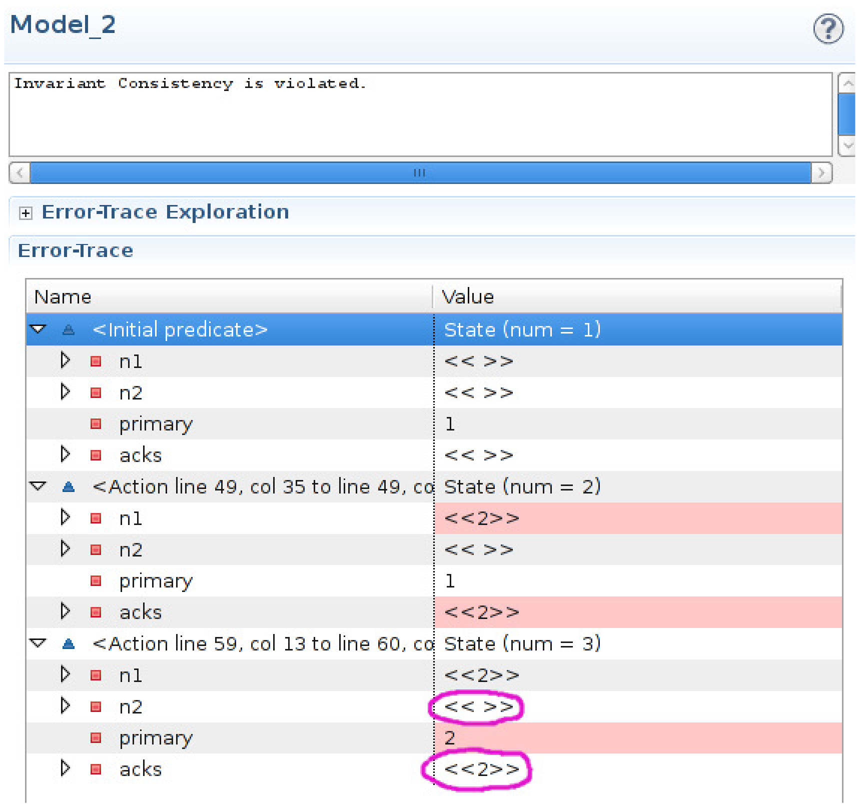
I assert that any system which uses asynchronous primary-secondary replication, and can change which node is the primary, is inconsistent. Why? If you write an operation to the primary, and then failover occurs *before* the operation is replicated to the node which is about to become the new primary, the new primary won't have that operation. If your replication strategy is to make secondaries look like the current primary, the system isn't just inconsistent, but can actually destroy acknowledged operations.

Here's a formal model of a simple system which maintains a log of operations. At any stage, one of three things can happen: we can write an operation to the primary, replicate the log of the primary to the secondary, or fail over:

```
------------------------------ MODULE failover ------------------------------

EXTENDS Naturals, Sequences, TLC

CONSTANT Ops

\* N1 and N2 are the list of writes made against each node
VARIABLES n1, n2
\* The list of writes acknowledged to the client
```

```
VARIABLE acks

\* The current primary node
VARIABLE primary

\* The types we allow variables to take on
TypeInvariant == /\ primary \in {1, 2}
                 /\ n1 \in Seq(Ops)
                 /\ n2 \in Seq(Ops)
                 /\ acks \in Seq(Ops)

\* An operation is acknowledged if it has an index somewhere in acks.
IsAcked(op) == \E i \in DOMAIN acks : acks[i] = op

\* The system is *consistent* if every acknowledged operation appears,
\* in order, in the current primary's oplog:
Consistency == acks = SelectSeq((IF primary = 1 THEN n1 ELSE n2), IsAcked)

\* We'll say the system is *potentially consistent* if at least one node
\* has a superset of our acknowledged writes in order.
PotentialConsistency == \/ acks = SelectSeq(n1, IsAcked)
                        \/ acks = SelectSeq(n2, IsAcked)

\* To start out, all oplogs are empty, and the primary is n1.
Init == /\ primary = 1
        /\ n1 = <<>>
        /\ n2 = <<>>
        /\ acks = <<>>

\* A client can send an operation to the primary. The write is immediately
\* stored on the primary and acknowledged to the client.
Write(op) == IF primary = 1 THEN /\ n1' = Append(n1, op)
                                 /\ acks' = Append(acks, op)
                                 /\ UNCHANGED <<n2, primary>>
                            ELSE /\ n2' = Append(n2, op)
                                 /\ acks' = Append(acks, op)
                                 /\ UNCHANGED <<n1, primary>>

\* For clarity, we'll have the client issues unique writes
WriteSomething == \E op \in Ops : ~IsAcked(op) /\ Write(op)

\* The primary can *replicate* its state by forcing another node
\* into conformance with its oplog
Replicate == IF primary = 1 THEN /\ n2' = n1
                                 /\ UNCHANGED <<n1, acks, primary>>
                            ELSE /\ n1' = n2
                                 /\ UNCHANGED <<n2, acks, primary>>

\* Or we can failover to a new primary.
Failover == /\ IF primary = 1 THEN primary' = 2 ELSE primary = 1
            /\ UNCHANGED <<n1, n2, acks>>

\* At each step, we allow the system to either write, replicate, or fail over
Next == \/ WriteSomething
        \/ Replicate
        \/ Failover
```

This is written in the TLA+ language for describing algorithms, which encodes a good subset of ZF axiomatic set theory with first-order logic and the Temporal Law of Actions. We can explore this specification with the TLC model checker, which takes our initial state and evolves it by executing every possible state transition until it hits an error:



```
Invariant Consistency is violated.
```

Name	Value
<Initial predicate>	State (num = 1)
n1	<< >>
n2	<< >>
primary	1
acks	<< >>
<Action line 49, col 35 to line 49, co	State (num = 2)
n1	<<2>>
n2	<< >>
primary	1
acks	<<2>>
<Action line 59, col 13 to line 60, co	State (num = 3)
n1	<<2>>
n2	<< >>
primary	2
acks	<<2>>

This protocol is inconsistent. The fields in red show the state changes during each transition: in the third step, the primary is n2, but n2's oplog is empty, instead of containing the list <<2>>. In fact, this model fails the PotentiallyConsistent invariant shortly thereafter, if replication or a write occurs. We can also test for the total loss of writes; it fails that invariant too.

That doesn't mean primary-secondary failover systems must be inconsistent. You just have to ensure that writes are replicated before they're acknowledged:

```
\* We can recover consistency by making the write protocol synchronous
SyncWrite(op) == /\ n1' = Append(n1, op)
                 /\ n2' = Append(n2, op)
                 /\ acks' = Append(acks, op)
                 /\ UNCHANGED primary

\* This new state transition satisfies both consistency constraints
SyncNext == \/ \E op \in Ops : SyncWrite(op)
            \/ Replicate
            \/ Failover
```

And in fact, we don't have to replicate to all nodes before ack to achieve consistency–we can get away with only writing to a quorum, if we're willing to use a more complex protocol like Paxos.

The important bit

So you skimmed the proof; big deal, right? The important thing that it doesn't matter *how* you actually decide to do the failover: Sentinel, Mongo's gossip protocol, Heartbeat, Corosync, Byzantine Paxos, or a human being flipping the switch. Redis Sentinel happens to be more complicated than it needs to be, and it leaves much larger windows for write loss than it has to, but *even if it were perfect* the underlying Redis replication model is fundamentally inconsistent. We saw the same problem in MongoDB when we wrote with less than WriteConcern.MAJORITY. This affects asynchronous replication in MySQL and Postgres. It affects DRBD (yeaaaahhh, this can happen to your *filesystem*). If you use any of this software, you are building an asynchronous distributed system, and there are eventualities that have to be acknowledged.

Look gals, there's nothing new here. This is an old proof and many mature software projects (for instance, DRBD or RabbitMQ) explain the inconsistency and data-loss consequences of a partition in their documentation. However, not everyone knows. In fact, a good number of people seem *shocked*.

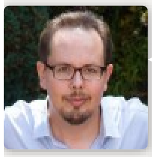
Why is this? I think it might be because software engineering is a really permeable field. You can start out learning Rails, and in two years wind up running four distributed databases *by accident*. Not everyone chose or could afford formal education, or was lucky enough to have a curmudgeonly mentor, or happened to read the right academic papers or find the right blogs. Now they might be using Redis as a lock server, or storing financial information in MongoDB. Is this dangerous? I honestly don't know. Depends on how they're using the system.

I don't view this so much as an engineering problem as a *cultural* one. Knives still come with sharp ends. Instruments are still hard for beginners to play. Not everything can or should be perfectly safe–or accessible. But I think we *should* warn people about what can happen, up front.

Tangentially: like many cultures, much of our collective understanding about what is desirable or achievable in distributed systems is driven by advertising. Yeah, MongoDB. That means you. ;-)

Bottom line

I don't mean to be a downer about all this. Inconsistency and even full-out data loss aren't the end of the world. Asynchronous replication is a good deal faster, both in bulk throughput and client latencies. I just think we lose sight, occasionally, of what that *means* for our production systems. My goal in writing Jepsen has been to push folks to consider their consistency properties carefully, and to explain them clearly to others. I think that'll help us all build safer systems. :)



Jasper A. Visser, on 2013/05/22

Thanks, this is an excellent read. I'm gonna have to digest the TLA+ bit for a bit longer to understand it. :)

For the purpose of continuity of discussion, antirez posted another reply here: http://antirez.com/news/56

Matteo, on 2013/05/30

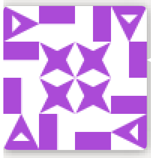
Thank you for the excellent post!

Could you please recommend some paper on the subject? I was curious to know which are "the right academic papers" you are referring to.

eas, on 2013/06/03

Just a guess at one of the "right academic papers": http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

Thanks for writing this series of articles.



Tanios, on 2013/10/24

What is CP system and AP system?

vp_arth, on 2013/12/26

Tanios, read here: http://henryr.github.io/cap-faq/

P - able to partitioning C - consistency A - availability all distributed systems have P, but they can't to be fully A and C both. if data is not very urgent we can sacrifice consistency and use AP system, and otherwise.

jmartin, on 2014/02/19

Kyle, could you recommend a good way to pick up TLA+? I looked it over once but found it dense (but maybe I dense).

pron, on 2014/03/12

I think more care should be taken with the definition of "consistent". You define consistency in the strictest of senses: all nodes see all acknowledged operation in the same order. For many (most?) systems, a more useful definition of consistency is the integrity of data constraints specific to the application (e.g. the sum of all accounts in the system must remain constant). That is the meaning of consistency in ACID. Ordering guarantees a stronger consistency, which is a strict subset of ACID consistency, but is not required by many systems. In the asynchronous replication model, if you only allow reads from the master, then while you may lose data upon failover, the data may still be consistent in the ACID sense; what you lose is ACID's "D": durability.

JensG, on 2015/11/29

I'd like to thank you for testing and analyzing all these systems in the way you do it. Excellent work and in its own way very unique. Thank you!

anonymous, on 2017/10/24

What about Postgres/MySQL clusters with synchronous replication? Can they have their own problems in regards with strong consistency (CP), exactly-once, transactions, isolation levels and all the properties that could discern between a single database or a distributed one, from outside? I could not find Jepsen test results for them. I know CockroachDB is now strong and has Jepsen tests to prove it maintains its invariants and promises while partitions occur. How real are the possibilities or errors happening in synchronous replication clusters compared to Raft consensus systems such as CockroachDB? Thank you

Aphyr, on 2017/10/25

Hi anon. Yes, in general, synchronously replicated systems can have their own consistency problems, but without testing that particular system, it's hard for me to say what those might be. It's gonna depend on what "synchronous" means for that system, and how promotion works.

Post a Comment

Name

Email

Http

Supports github-flavored markdown for [links](http://foo.com/), *emphasis*, _underline_, `code`, and > blockquotes. Use ```clj on its own line to start a Clojure code block, and ``` to end the block.

Post Comment

Copyright © 2018 Kyle Kingsbury.
Non-commercial re-use with attribution encouraged; all other rights reserved.
Comments are the property of respective posters.