Jepsen: Testing the Partition Tolerance of PostgreSQL, Redis, MongoDB and Riak

Posted by Kyle Kingsbury on Jun 20, 2013 / 4 Discuss

Distributed systems are characterized by exchanging state over high-latency or unreliable links. The system must be robust to both node and network failure if it is to operate reliably--however, not all systems satisfy the safety invariants we'd like. In this article, we'll explore some of the design considerations of distributed databases, and how they respond to network partitions.

IP networks may arbitrarily drop, delay, reorder, or duplicate messages send between nodes--so many distributed systems use TCP to prevent reordered and duplicate messages. However, TCP/IP is still fundamentally *asynchronous*: the network may arbitrarily delay messages, and connections may drop at any time. Moreover, failure detection is unreliable: it may be impossible to determine whether a node has died, the network connection has dropped, or things are just slower than expected.

This type of failure - where messages are arbitrarily delayed or dropped--is called a network partition. Partitions can occur in production networks for a variety of reasons: GC pressure, NIC failure, switch firmware bugs, misconfiguration, congestion, or backhoes, to name a few. Given that partitions occur, the CAP theorem restricts the maximally achievable guarantees of distributed systems. When messages are dropped, "consistent" (CP) systems preserve linearizability by rejecting some requests on some nodes. "Available" (AP) systems can handle requests on all nodes, but must sacrifice linearizability: different nodes can disagree about the order in which operations took place. Systems can be both consistent and available when the network is healthy, but since real networks partition, there are no fully CA systems.

It's also worth noting that CAP doesn't just apply to the database as a whole--but also to subsystems like tables, keys, columns, and even distinct operations. For example, a database can offer linearizability for each key independently, but not *between* keys. Making that tradeoff allows the system to handle a larger space of requests during a partition. Many databases offer tunable consistency levels for individual reads and writes, with commensurate performance and correctness tradeoffs.

Testing partitions

Theory bounds a design space, but real software may not achieve those bounds. We need to *test* a system's behavior to really understand how it behaves.

First, you'll need a collection of nodes to test. I'm using five LXC nodes on a Linux computer, but you could use Solaris zones, VMs, EC2 nodes, physical hardware, etc. You'll want the nodes to share a network of some kind--in my case, a single virtual bridge interface. I've named my nodes n1, n2, ... n5, and set up DNS between them and the host OS.

To cause a partition, you'll need a way to drop or delay messages: for instance, with firewall rules. On Linux you can use **iptables -A INPUT -s some-peer -j DROP** to cause a unidirectional partition, dropping messages from some-peer to the local node. By applying these rules on several hosts, you can build up arbitrary patterns of network loss.

Running these commands repeatably on several hosts takes a little bit of work. I'm using a tool I wrote called Salticid, but you could use CSSH or any other cluster automation system. The key factor is latency--

you want to be able to initiate and end the partition quickly, so Chef or other slow-converging systems are probably less useful.

Then, you'll need to set up the distributed system on those nodes, and design an application to test it. I've written a simple test: a Clojure program, running outside the cluster, with threads simulating five isolated clients. The clients concurrently add N integers to a set in the distributed system: one writes 0, 5, 10, ...; another writes 1, 6, 11, ...; and so on. Each client records a log of its writes, and whether they succeeded or failed. When all writes are complete, it waits for the cluster to converge, and check whether the client logs agree with the actual state of the database. This is a simple kind of consistency check, but can be adapted to test a variety of data models.

The client and config automation, including scripts for simulating partitions and setting up databases, is freely available. For instructions and code, click here.

PostgreSQL

A single-node PostgreSQL instance is a CP system; it can provide serializable consistency for transactions, at the cost of becoming unavailable when the node fails. However, the distributed system comprised of the server *and* a client may not be consistent.

Postgres' commit protocol is a special case of two-phase commit. In the first phase, the client votes to commit (or abort) the current transaction, and sends that message to the server. The server checks to see whether its consistency constraints allow the transaction to proceed, and if so, it votes to commit. It writes the transaction to storage and informs the client that the commit has taken place (or failed, as the case may be.) Now both the client and server agree on the outcome of the transaction.

What happens if the message acknowledging the commit is dropped before the client receives it? Then the client does't know whether the commit succeeded or not! The 2PC protocol says nodes must wait for the acknowledgement message to arrive in order to decide the outcome. If it doesn't arrive, 2PC fails to terminate. It's not a partition-tolerant protocol. Real systems can't wait forever, so at some point the client times out, leaving the commit protocol in an indeterminate state.

If I cause this type of partition, the JDBC Postgres client throws exceptions like

```
217 An I/O error occurred while sending to the backend.
Failure to execute query with SQL:
INSERT INTO "set_app" ("element") VALUES (?)  ::  [219]
PSQLException:
Message: An I/O error occured while sending to the backend.
SQLState: 08006
Error Code: 0
218 An I/O error occured while sending to the backend.
```

... which we might interpret as "the transactions for writes 217 and 218 failed". However, when the test app queries the database to find which write transactions were successful, it finds that two "failed" writes were actually present:

```
1000 total
950 acknowledged
952 survivors
2 unacknowledged writes found! ヽ(´ー`)ノ
(215 218)
```

```
0.95 ack rate
0.0 loss rate
0.002105263 unacknowledged but successful rate
```

Out of 1000 attempted writes, 950 were successfully acknowledged, and all 950 of those writes were present in the result set. However, two writes (215 and 218) succeeded, even though they threw an exception! Note that this exception doesn't guarantee that the write succeeded or failed: 217 also threw an I/O error while sending, but because the connection dropped before the client's commit message arrived at the server, the transaction never took place.

There is no way to reliably distinguish these cases from the client. A network partition--and indeed, most network errors--doesn't mean a failure. It means the *absence* of information. Without a partition-tolerant commit protocol, like extended three-phase commit, we cannot assert the state of these writes.

You can handle this indeterminacy by making your operations idempotent and retrying them blindly, or by writing the transaction ID as a part of the transaction itself, and querying for it after the partition heals.

Redis

Redis is a data structure server, typically deployed as a shared heap. Since it runs on one single-threaded server, it offers linearizable consistency by default: all operations happen in a single, well-defined order.

Redis also offers asynchronous primary->secondary replication. A single server is chosen as the primary, which can accept writes. It relays its state changes to secondary servers, which follow along. Asynchronous, in this context, means that clients *do not block* while the primary replicates a given operation - the write will "eventually" arrive on the secondaries.

To handle discovery, leader election, and failover, Redis includes a companion system: Redis Sentinel. Sentinel nodes gossip the state of the Redis servers they can see, and attempt to promote and demote nodes to maintain a single authoritative primary. In this test, I've installed Redis and Redis Sentinel on all five nodes. Initially all five clients read from the primary on n1, and n2--n5 are secondaries. Then we partition n1 and n2 away from n3, n4, and n5.

If Redis were a CP system, n1 and n2 would become unavailable during the partition, and a new primary in the majority component (n3, n4, n5) would be elected. This is *not* the case. Instead, writes continue to successfully complete against n1. After a few seconds, the Sentinel nodes begin to detect the partition, and elect, say, n5 as a new primary.

For the duration of the partition, there are *two* primary nodes--one in each component of the system--and both accept writes independently. This is a classic split-brain scenario--and it violates the C in CP. Writes (and reads) in this state are not linearizable, because clients will see a different state of the database depending on which node they're talking to.

What happens when the partition resolves? Redis used to leave *both* primaries running indefinitely. Any partition resulting in a promotion would cause permanent split-brain. That changed in Redis 2.6.13, which was released April 30th, 2013. Now, the sentinels will resolve the conflict by demoting the original primary, destroying a potentially unbounded set of writes in the process. For instance:

```
2000 total
1998 acknowledged
872 survivors 1126 acknowledged writes lost! (ノ°□°)ノ﹀ ┻┻
```

```
50 51 52 53 54 55 ... 1671 1675 1676 1680 1681 1685
0.999 ack rate
0.5635636 loss rate
0.0 unacknowledged but successful rate
```

Out of 2000 writes, Redis claimed that 1998 of them completed successfully. However, only 872 of those integers were present in the final set. Redis threw away 56% of the writes it claimed were successful.

There are two problems here. First, notice that all the clients lost writes at the beginning of the partition: (50, 51, 52, 53, ...). That's because they were all writing to n1 when the network dropped--and since n1 was demoted later, any writes made during that window were destroyed.

The second problem was caused by split-brain: both n1 and n5 were primaries up until the partition healed. Depending on which node they were talking to, some clients might have their writes survive, and others have their writes lost. The last few numbers in the set (mod 5) are all 0 and 1--corresponding to clients which kept using n1 as a primary in the minority component.

In any replication scheme with failover, Redis provides neither high availability nor consistency. Only use Redis as a "best-effort" cache and shared heap where arbitrary data loss or corruption is acceptable.

MongoDB

MongoDB is a document-oriented database with a similar distribution design to Redis. In a replica set, there exists a single primary node which accepts writes and asynchronously replicates a log of its operations ("oplog") to N secondaries. However, there are a few key differences from Redis.

First, Mongo builds in its leader election and replicated state machine. There's no separate system which tries to observe a replica set in order to make decisions about what it should do. The replica set decides among itself which node should be primary, when to step down, how to replicate, etc. This is operationally simpler and eliminates whole classes of topology problems.

Second, Mongo allows you to ask that the primary confirm successful replication of a write by its disk log, or by secondary nodes. At the cost of latency, we can get stronger guarantees about whether or not a write was successful.

To test this, I've set up a five-node replica set, with the primary on n1. Clients make their writes to a single document (the unit of MongoDB consistency) via compare-and-set atomic updates. I then partition n1 and n2 away from the rest of the cluster, forcing Mongo to elect a new primary on the majority component, and to demote n1. I let the system run in a partitioned state for a short time, then reconnect the nodes, allowing Mongo to reconverge before the end of the test.

There are several consistency levels for operations against MongoDB, termed "write concerns". The defaults, up until recently, were to avoid checking for any type of failure at all. The Java driver calls this WriteConcern.UNACKNOWLEDGED. Naturally, this approach can lose any number of "successful" writes during a partition:

```
6000 total
5700 acknowledged
3319 survivors
2381 acknowledged writes lost! (ヾ°□°)ヾ﹀ ┻┻
469 474 479 484 489 494 ... 3166 3168 3171 3173 3178 3183
0.95 ack rate
0.4177193 loss rate
```

```
0.0 unacknowledged but successful rate
```

In this trial, 42% of writes, from 469 through 3183, were thrown away.

However, WriteConcern.SAFE, which confirms that data is successfully committed to the primary, also loses a large number of writes:

```
6000 total
5900 acknowledged
3692 survivors
2208 acknowledged writes lost! (╯°□°)╯︵ ┻━┻
458 463 468 473 478 483 ... 3075 3080 3085 3090 3095 3100 0.98333335 ack rate
0.3742373 loss rate
0.0 unacknowledged but successful rate
```

Because the replication protocol is asynchronous, writes continued to succeed against n1, even though n1 couldn't replicate those writes to the rest of the cluster. When n3 was later elected primary in the majority component, it came to power with an *old* version of history--causally disconnected from those writes on n1. The two evolved inconsistently for a time, before n1 realized it had to step down.

When the partition healed, Mongo tried to determine which node was authoritative. Of course, there *is* no authoritative node, because both accepted writes during the partition. Instead, MongoDB tries to find the node with the highest optime (a monotonic timestamp for each node's oplog). Then it forces the older primary (n1) to roll back to the last common point between the two, and re-applies n3's operations.

In a rollback, MongoDB dumps a snapshot of the current state of conflicting objects to disk, in a BSON file. An operator could later attempt to reconstruct the proper state of the document.

This system has several problems. First, there's a bug in the leader election code: MongoDB may promote a node which does *not* have the highest optime in the reachable set. Second, there's a bug in the rollback code. In my tests, rollback *only worked roughly 10% of the time*. In almost every case, MongoDB threw away the conflicting data altogether. Moreover, not all types of objects will be fully logged during a rollback: capped collections, for instance, throw away all conflicts *by design*. Third, even if these systems *do* work correctly, the rollback log is not sufficient to recover linearizability. Because the rollback version and the oplog do not share a well-defined causal order, only order-free merge functions (e.g. CRDTs) can reconstruct the correct state of the document in generality.

This lack of linearizability also applies to FSYNC_SAFE, JOURNAL_SAFE, and even REPLICAS_SAFE, which ensures that writes are acknowledged by two replicas before the request succeeds:

```
6000 total
5695 acknowledged
3768 survivors
1927 acknowledged writes lost! (╯°□°)╯︵ ┻━┻
712 717 722 727 732 737 ... 2794 2799 2804 2809 2814 2819
0.94916666 ack rate
0.338367 loss rate
0.0 unacknowledged but successful rate
```

The only way to recover linearizability in MongoDB's model is by waiting for a *quorum* of nodes to respond. However, WriteConcern.MAJORITY is *still* inconsistent, dropping acknowledged writes and

recovering failed writes.

```
6000 total
5700 acknowledged
5701 survivors
2 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
(596 598)
3 unacknowledged writes found! ヽ(´ー`)ノ
(562 653 3818)
0.95 ack rate
1.754386E-4 loss rate
5.2631577E-4 unacknowledged but successful rate
```

Where UNSAFE, SAFE, and REPLICAS_SAFE can lose any or all writes made during a partition, MAJORITY can only only lose writes which were *in flight* when the partition started. When the primary steps down it signs off on all WriteConcern requests, setting OK to TRUE on each reply regardless of whether the WriteConcern was satisfied.

Moreover, MongoDB can emit any number of false negatives. In this trial, 3 unacknowledged writes were actually recovered in the final dataset. At least in version 2.4.1 and earlier, there is no way to prevent data loss during partition, at any consistency level.

If you need linearizability in MongoDB, use WriteConcern.MAJORITY. It won't actually be consistent, but it dramatically reduces the window of write loss.

Riak

As a Dynamo clone, Riak takes an AP approach to partition tolerance. Riak will detect causally divergent histories--whether due to a partition or normal concurrent writes--and present *all* diverging copies of an object to the client, who must then choose how to merge them together.

The default merge function in Riak is last-write-wins. Each write includes a timestamp, and merging values together is done by preserving *only* the version with the highest timestamp. If clocks are perfectly synchronized, this ensures Riak picks the most recent value.

Even in the absence of partitions and clock skew, causally concurrent writes means that last-write-wins can cause successful writes to be silently dropped:

```
2000 total
2000 acknowledged
566 survivors
1434 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
1 2 3 4 6 8 ... 1990 1991 1992 1995 1996 1997
1.0 ack rate
0.717 loss rate
```

In this case, a healthy cluster lost 71% of operations--because when two clients wrote to the value at roughly the same time, Riak simply picked the write with the higher timestamp and ignored the others--which might have added new numbers.

Often, people try to resolve this problem by adding a lock service to prevent concurrent. Since locks *must* be linearizable, the CAP theorem tells us that distributed lock systems cannot be fully available during a

partition--but even if they *were*, it wouldn't prevent write loss. Here's a Riak cluster with R=W=QUORUM, where all clients perform their reads+writes atomically using a mutex. When the partition occurs, Riak loses 91% of successful writes:

```
2000 total
1985 acknowledged
176 survivors
1815 acknowledged writes lost! (╯°□°)╯︵ ┻━┻
85 90 95 100 105 106 ... 1994 1995 1996 1997 1998 1999
6 unacknowledged writes found! ヽ(´ー`)ノ
(203 204 218 234 262 277)
0.9925 ack rate
0.91435766 loss rate
0.00302267 unacknowledged but successful rate
```

In fact, LWW can cause *unbounded data loss*, including the loss of information written *before* the partition occurred. This is possible because Dynamo (by design) allows for sloppy quorums, where fallback vnodes on both sides of the partition can satisfy R and W.

We can tell Riak to use a strict quorum using PR and PW--which succeeds only if a quorum of the *original* vnodes acknowledges the operation. This can still cause unbounded data loss if a partition occurs:

```
2000 total
1971 acknowledged
170 survivors
1807 acknowledged writes lost! (╯°□°)╯︵ ┻━┻
86 91 95 96 100 101 ... 1994 1995 1996 1997 1998 1999
6 unacknowledged writes found! ヽ(´ー`)ノ
(193 208 219 237 249 252)
0.9855 ack rate 0.9167935 loss rate
0.00304414 unacknowledged but successful rate
```

Dynamo is designed to preserve writes as much as possible. Even though a node might return "PW val unsatisfied" when it can't replicate to the primary vnodes for a key, it may have been able to write to one primary vnode--or any number of fallback vnodes. Those values will still be exchanged during read-repair, considered as conflicts, and the timestamp used to discard the older value--meaning all writes from one side of the cluster.

This means the minority component's "failed" writes can destroy all of the majority component's successful writes.

It is possible to preserve data in an AP system by using CRDTs. If we use sets as the data structure, with union as the merge function, we can preserve all writes even in the face of arbitrary partitions:

```
2000 total
1948 acknowledged
2000 survivors All
2000 writes succeeded. :-D
```

This is *not* linearizable consistency--and not all data structures can be expressed as CRDTs. It also does not prevent false negatives--Riak can still time out or report failures--but it does guarantee safe convergence

for acknowledged writes.

If you're using Riak, use CRDTs, or write as much of a merge function as you can. There are only a few cases (e.g. immutable data) where LWW is appropriate; avoid it everywhere else.

Measure assumptions

We've seen that distributed systems can behave in unexpected ways under partition--but the nature of that failure depends on many factors. The probability of data loss and inconsistency depends on your application, network, client topology, timing, the nature of the failure, and so on. Instead of telling you to choose a particular database, I encourage you to think carefully about the invariants you need, the risks you're willing to accept, and to design your system accordingly.

A key part of building that design is *measuring* it. First, establish the boundaries for the system - places where it interacts with the user, the internet, or other services. Determine the guarantees which must hold at those boundaries.

Then, write a program which makes requests of the system from just outside that boundary, and measures the external consequences. Record whether HTTP requests returned 200 or 503s, or the list of comments on a post at each point. While running that program, cause a failure: kill a process, unmount a disk, or firewall nodes away from one another.

Finally, compare logs to verify the system's guarantees. For instance, if acknowledged chat messages should be delivered at least once to their recipients, check to see whether the messages were *actually* delivered.

The results may be surprising; use them to investigate the system's design, implementation, and dependencies. Consider designing the system to continuously measure its critical safety guarantees, much like you instrument performance.

Lessons

Even if you don't use MongoDB or Riak, there are some general lessons you can take away from these examples.

First, clients are an important part of the distributed system, not objective observers. Network errors mean "I don't know," not "It failed." Make the difference between success, failure, and indeterminacy explicit in your code and APIs. Consider extending consistency algorithms through the boundaries of your systems: hand TCP clients ETags or vector clocks, or extend CRDTs to the browser.

Even well-known algorithms like two-phase commit have some caveats, like false negatives. SQL transactional consistency comes in several levels. If you use the stronger consistency levels, remember that conflict handling is essential.

Certain problems are hard to solve well, like maintaining an authoritative primary with failover. Consistency is a property of the data, not of the nodes. Avoid systems which assume node state consensus implies data consistency.

Wall clocks are only useful for ensuring responsiveness in the face of deadlock, and even then they're not a positive guarantee of correctness. In these tests, all clocks were well-synchronized with NTP, and we still lost data. Even worse things can happen if a clock gets out of sync, or a node pauses for a while. Use logical clocks on your data. Distrust systems which rely on the system time, unless you're running GPS or atomic clocks on your nodes. Measure your clock skew anyway.

Where correctness matters, rely on techniques with a formal proof and review in the literature. There's a huge gulf between theoretically correct algorithm and live software--especially with respect to latency--but a buggy implementation of a correct algorithm is typically better than a correct implementation of a terrible algorithm. Bugs you can fix. Designs are much harder to re-evaluate.

Choose the right design for your problem space. Some parts of your architecture demand strong consistency. Other parts can sacrifice linearizability while remaining correct, as with CRDTs. Sometimes you can afford to lose data entirely. There is often a tradeoff between performance and correctness: think, experiment, and find out.

Restricting your system with particular rules can make it easier to attain safety. Immutability is an incredibly useful property, and can be combined with a mutable CP data store for powerful hybrid systems. Use idempotent operations as much as possible: it enables all sorts of queuing and retry semantics. Go one step further, if practical, and use full CRDTs.

Preventing write loss in databases like MongoDB requires a significant latency tradeoff. It might be faster to just use Postgres. Sometimes buying more reliable network and power infrastructure is cheaper than scaling out. Sometimes not.

Distributed state is a difficult problem, but with a little work we can make our systems significantly more reliable. For more on the consequences of network partitions, including examples of production failures, see this.

About the Author

Kyle Kingsbury is an engineer at [Factual], and writes here. He's also the author of [Riemann], an open source event-driven monitoring system; and [Timelike], an experiment in network simulation. He lives in San Francisco, California, and has no idea how computers work.