Fail-fast Is Failing... Fast!

PAT HELLAND

**CHANGES IN
COMPUTE
ENVIRONMENTS ARE
PLACING PRESSURE
ON TRIED-AND-TRUE
DISTRIBUTED-
SYSTEMS SOLUTIONS.**

For more than 40 years, fail-fast has been the dominant way of achieving fault tolerance. In this approach, some mechanism is responsible for ensuring that each component is up, functioning, and responding to work. As long as it's still alive and healthy, you continue forward; when something goes awry with the component, it is removed from the system and the remaining components reorganize themselves to continue. For many (but not all) systems, this is how you ensure the system remains alive when pieces fail.

As the industry moves to leverage cloud computing, this is getting more challenging. First of all, I love cloud computing and think it is an essential step forward. Still, the way we create robust solutions is under pressure as the individual components experience emerging challenges called *gray failures*. In a gray failure, a server or part of the network doesn't fail fast but instead starts running slow. Running slow is WAY worse than failing fast. The slow component, sometimes running at less than one percent of its normal speed, may be healthy enough

*It's Not Your Grandmother's Database Anymore

to say, “I’m still here!” but slow enough to clog up all the work. This makes fail-fast schemes vulnerable.

BACK BEFORE EVERYTHING TURNED GRAY

Back in days of yore, hardware for your servers was in YOUR data center in tightly controlled environments. Some salesperson had convinced you that you needed the best, most expensive, and most resilient servers. You knew if you bought THOSE servers, you wouldn’t get fired. Connecting those servers meant you had a local network supporting only your stuff. You could ensure that the traffic running in your local network wasn’t TOO crowded. Because of this, the individual servers could respond predictably to high-priority messages such as, “How’re you doing, good buddy?” The messages flew through the overprovisioned network like a trip down Highway 101 through San Francisco at 4 a.m. on Sunday morning.

Leveraging these extremely probable answers to inquiries about health, it was easy for a supervising server to impose rules for health or sickness. There was an expected time for an answer to this health check. If the response took too long, you would try again. The longer you went without an answer, the more likely the lagging server was truly sick. After a reasonably small number of attempts, you could give up and declare the party pooper was excluded from the party. Yeah, the decision was probabilistic, but the odds were really, really good, and you could decide pretty fast.

Next, it became important to crisply evict the renegade node. Sending it a message to cease and

desist may or may not have made a difference. This is often called STONITH (shoot the other node in the head). Sometimes the other node has a hard head and STONITH doesn't work. Another trick is to ensure the banished node cannot do any harm. If you can guarantee the wayward node can't make any change *outside* the node, the rest of the team can move on with their lives. Let it rot in solitary.

In this fashion, you could detect the node had failed and ensure it had completely failed. In the past, it didn't take too long to do this.

IT'S HARD TO AVOID BEING AN ASS

There is a paradox called *Buridan's ass*, named after the 14th-century philosopher Jean Buridan⁷. Buridan's ass highlights an apparent contradiction inherent to the concept of determinism or the belief that every event is derived from previous events. The paradox proposes that a very hungry donkey is placed exactly halfway between two bales of hay. Assuming the donkey will go to the closest resource, it will starve to death.

In electronics, there is a phenomenon called *metastability*, which means the system may be in an unstable state for an unbounded time. To have a valid signal as an output to the circuit, it must reside within a certain voltage or current level. If the output lands in the middle gray area, wonky things happen to the next circuit. It, too, can do weird things.

This metastability is inherent within asynchronous circuits. Since you don't know the timing of the input

signals, some of the time the various inputs arrive simultaneously, and the circuit can't decide between the two bales of hay. Asynchronous digital systems usually add arbiters to the input signals to ensure the signals are ordered and, hence, avoid metastability. Within a clock domain on a synchronous device, the clock ensures the timing of the inputs provided to the logic circuit, avoiding metastability problems. As synchronous circuits receive incoming asynchronous signals, special synchronizers work to make the probability of metastability vanishingly small but still possible.

Doing a lift-and-shift of a distributed system onto a complex cloud-computing environment poses a bunch of new problems for the distributed-systems design. Running a server in a virtual machine provides a *lot* of valuable computing for a good price, but it may do so according to its own timeframe. The noisy neighbor problem refers to varying capacity for *your* virtual machine as it competes for resources with *other* virtual machines on the same physical servers. Multicore servers add to the fun as their coordination may or may not stall messages you hope to send. Trips through the cloud-networking infrastructure may experience congestion, causing communication emulating the US Postal Service. That makes timer-based fail-fast probabilistic. It always was probabilistic, but now the odds have shifted away from minuscule probabilities to simply very-hard-to-debug rare probabilities.

Before, when distributed systems were composed of predictable servers and networks, you had a darned good idea how quickly your cohort would answer a health check. Using that expectation, you could remove sick nodes

When the old deployment of a distributed system is lifted-and-shifted into the cloud, it's like a trip to the fun zone at the carnival.

quickly while only *rarely* removing a healthy node. It was a probability game where the odds were *extremely* high that you would make the right decision. This is much like circuits working within a clock domain to avoid metastable behavior.

Servers and/or the messages to and from them don't always work at a predictable pace. It's much the same as removing the clocking within a synchronous circuit so it is an asynchronous digital system. The probability of dampening and removing metastability has become *way* lower.

Consider a team of jugglers handling dozens of balls on stage. If some of them are sent into slow motion in a noncorrelated way, it can be a problem. The jugglers have their own timeframes, and they see balls arrive when they arrive. For a short while this may make sense. It becomes virtually impossible to juggle time-based interactions across the team as each juggler's time speeds up and slows down unbeknownst to them.

When the old deployment of a distributed system is lifted-and-shifted into the cloud, it's like a trip to the fun zone at the carnival. Time and distance get distorted and fail-fast decisions become unpredictable. When servers don't march approximately to the beat of the same drummer, distributed-systems algorithms can become metastable.

Even worse, most systems depend on other systems in the data center. HDFS (Hadoop Distributed File System) depends on Apache ZooKeeper.¹ ZooKeeper and other services depend on DNS (Domain Name System). Each of these systems has timeouts and retries, which can lead to cascading delays, timeouts, and failures. This is another

form of metastability that can accentuate problems rather than dampen them.

When this metastability interferes with a primary or leader in your distributed system, you don't have rapid access to the "perfect truth" or linearizability that many systems require. Availability can suffer quite a bit.

DISTRIBUTED ALGORITHMS:

"PRETTY GOOD" IS BETTER THAN PERFECTION

Let's turn our attention to the importance of algorithms that don't mandate the perfect latest answer. When a request to a user is based on a pool of replicas containing stale state, any one of the replicas will suffice. The background work to update the replicas with new state might be quite a bit behind, and that's OK. An excellent example of "pretty good" can be found when you read a replica of the product catalog or of product reviews in online retail. Similarly, in a web search, you access all the shards for the search terms in the query. It is not essential that each request returns the absolutely most recent results. "Pretty good" is pretty darned good.

When sending work to a pool of servers with replicas of these caches, the client can time out and retry to bound the latency. It doesn't really care too much about the health of an individual server, as the client can get a good answer from another by retrying using a technique called *hedging*.³ The state of the unresponsive server is not too important for a while. Letting it be metastable while it lives in an uncertain state is OK. Frankly, you don't need a perfect answer about the server's membership in the cluster. Over time it will start to behave better or it will get

tossed out of the cluster. Demanding a perfect answer in a hurry is not necessary.

There are similar emerging tolerant algorithms for logging. Apache BookKeeper² is an open-source logging subsystem in which writes to append new records to the log do not need to get to all log servers containing replicas of the log. Getting to a required subset is enough. Similarly, Amazon Aurora,^{8,9} offered through AWS (Amazon Web Services), runs a database in a centralized server but sends its updates to a pool of storage servers. Because not all the storage servers need to respond in a timely fashion, the approach used by BookKeeper and Aurora is dramatically more resilient to delays in servers. An individual replica can live in its own time warp, and the broader distributed algorithm proceeds happily.

I think of this as water flowing around a rock in a river. Getting *enough work* through is dramatically more resilient than insisting *all the work* gets through.

WITHOUT FAIL-FAST, PRIMARY/BACKUP ALGORITHMS ARE DOOMED

When traditional algorithms depend on perfect knowledge of membership in a cluster, what can you do? Many systems are built expecting perfect knowledge to give perfect answers. For example, HDFS⁶ has a NameNode that is responsible for the allocation of data pages. Unless HDFS knows that you have *either zero or one primary NameNode at a time*, the file system may be corrupted.

When one server fails fast, it is replaced, data sloshes around the cluster, and pretty soon you're back in business. HDFS depends on fail-fast as its model for liveness. I

worry about the continued ability to use fail-fast and still follow the motto of that old Timex watch commercial: *“It takes a licking but keeps on ticking.”*

HAVE YOU EVER MET A STABLE DISTRIBUTED ALGORITHM?

Well... some algorithms are, indeed, stable when running on top of a surrealistic world. Some, however, are not. Any algorithm dependent on a fixed set of responsive servers conveying strongly consistent linearizable data will increasingly see challenges if the trend toward metastable clusters continues.

There are a number of reasons why parts of a cluster may perceive timing differences with other parts:

- ➔ Servers or VMs not getting the expected amount of the computation for a period of time.
- ➔ Multicore servers internally coordinating or stalling ahead of answering your question.
- ➔ Networks not providing a pair of servers their fair share of the bandwidth, causing delays.
- ➔ Hardware in the data center behaving unpredictably, especially as the price of these components is driven down.⁵
- ➔ Increasing dependencies on other services (e.g., ZooKeeper or DNS) that can cause cascading wonkiness because of timeouts.

All of these challenges compound in large, complex, and intertwined environments.

I want to emphasize that I see this as a great thing. While, in my dreams, it would be great to have a dedicated freeway lane for my commute home from

the office, it's not really practical. Instead, we share freeway lanes in spite of the inherent risk of delay in an open queuing network that allows lots of incoming traffic without any constraints. Not many of us can build a freeway dedicated to our personal use; hence, we share. *Cloud computing is sharing.*

The same is true in real life. If you live in a small town with a walking commute home, you can predictably walk in the front door at 6:23 every evening. As you move to the big city with elevators, parking garages, and freeways to traverse, it's harder to time your arrival precisely.

OPEN QUESTIONS TO PONDER

- ➔ How can we compensate for these issues of metastability in cloud environments? Does adding resources help?
- ➔ Socially, humans have a tendency to overconsume resources until there's a problem. Will we always succumb to the tragedy of the commons and suck up all slack until we are metastable?
- ➔ What algorithms can give us both crisp and clear linearizable updates AND rapid latency 99.9 percent of the time? How about 99.999 percent of the time? How do these respond when the environment is under stress?
- ➔ How can we evolve away from algorithms that are sensitive to metastability into a world where we tolerate and dampen most metastability?

References

1. Apache Software Foundation. 2010-2020. Welcome to Apache ZooKeeper; <https://zookeeper.apache.org>.

2. Apache Software Foundation. 2016-2021. BookKeeper concepts and architecture; <http://bookkeeper.apache.org/docs/4.5.1/getting-started/concepts/>.
3. Dean, J., Barroso, L. A. 2013. The tail at scale. *Communications of the ACM* 56(2), 74-80; <https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext>.
4. Gray, J. 1985. Why Do Computers Stop and What Can Be Done About It?. *Tandem Technical Report TR 85.7*; <https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf>
5. Gunawi, H. S., et al. 2018. Fail-slow at scale: evidence of hardware performance faults in large production systems. *Proceedings of the 16th Usenix Conference on File and Storage Technologies*; <https://www.usenix.org/system/files/conference/fast18/fast18-gunawi.pdf>.
6. Hadoop. 2021. HDFS architecture guide; https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
7. Lamport, L. 2012. Buridan's Principle. *Foundations of Physics* 42; <https://lamport.azurewebsites.net/pubs/buridan.pdf>
8. Verbitski, A., et al. 2017. Amazon Aurora: design considerations for high throughput cloud-native relational databases. *Proceedings of the ACM International Conference on Management of Data*, 1041-1052; <https://dl.acm.org/doi/pdf/10.1145/3035918.3056101>.
9. Verbitski, A., et al. 2018. Amazon Aurora: on avoiding distributed consensus for I/Os, commits, and membership changes. *Proceedings of the ACM International Conference on Management of Data*, 789-796; <http://pages.cs.wisc.edu/~yxy/lcs839-s20/papers/aurora-sigmod-18.pdf>.

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He works at Salesforce. His blog is at pathelland.substack.com.*

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

We're more than computational theorists, database managers, UX mavens, coders and developers. We're on a mission to solve tomorrow. ACM gives us the resources, the access and the tools to invent the future.

Join ACM today at acm.org/join

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

Association for
Computing Machinery

Advancing Computing as a Science & Profession