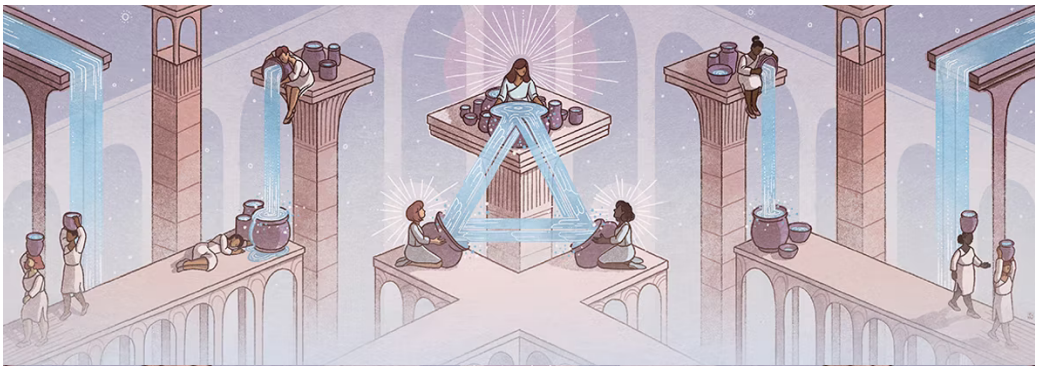
← Back to Blog

A Brief History of High Availability

Written by Sean Loiselle, and Jessica Edwards on August 3, 2022



Content

Where to from here?

I once went to a website that had “hours of operation,” and was only “open” when its brick and mortar counterpart had its lights on. I felt perplexed and a little frustrated; computers are capable of running all day every day, so why shouldn’t they? I’d been habituated to the internet’s incredible availability guarantees.

However, before the internet, 24/7 high availability wasn’t “a thing.” Availability was desirable, but not something to which we felt fundamentally entitled. We used computers only when we needed them; they weren’t waiting idly by on the off-chance a request came by. As the internet grew, those previously uncommon requests at 3am local time became prime business hours partway across the globe, and making sure that a computer could facilitate the request was important.

Many systems, though, relied on only one computer to facilitate these requests — a single point of failure — which we all know is a story that doesn’t end well. To keep things up and running, we needed to distribute the load among multiple computers that could fulfill our needs. However, distributed computation, for all its well-known upsides, has sharp edges: in particular, synchronization and tolerating partial failures (fault tolerance) within a system. Each generation of engineers has iterated on these solutions to fit the needs of their time.

How distribution came to databases is of particular interest because it is a difficult problem that has been much slower to develop than other areas of computer science. Certainly, software tracked the results of some distributed computation in a local database, but the state of the database itself was kept on a single machine. Why?

Replicating state across machines is hard.

In this post, we take a look at how distributed databases have historically handled fault tolerance and—at a high level—what high availability looks like. We also walk through different types of high availability systems.

Fault Tolerance vs High Availability

Before we dive deep into the colorful history of high availability I want to clarify the distinction between these two terms that are often thought of as synonyms. While they are very closely related, they are not the same.

Fault tolerance implies zero service interruptions. If there is a failure somewhere the system will instantly switch to the backup solution and service will continue without interruption. High availability, on the other hand, implies that services are, well, *highly available but not always available*. A system can be highly available but not fault tolerant. I generally consider high availability to be an aspect of fault tolerance. In that, it addresses a certain type of "fault" (availability), but doesn't necessarily talk about other aspects.

This is somewhat of a contrived example, but basically everyone watches streaming content, so let's consider a digital rights management service that determines whether a viewer can watch a particular video. The service could be configured to be highly available, in that it will always serve and return queries. However, it may not handle certain backend data correctly and get into a state where it returns errors, or denies all requests. In this case, it would be highly available (it is reachable and is returning an answer), but it is not fault tolerant, because something in the system has caused it to misbehave.

The caveat with this example is that there's a fine line between a "bug" and fault tolerance. But the idea of fault tolerance is that the system can handle unexpected events gracefully and continue providing an excellent user experience. (If this example is interesting to you, I recommend taking a look at how Netflix's chaos monkey randomly terminates instances in production to ensure that engineers implement their services to be resilient to instance failures).

Okay, let's get into some highly available database examples.

What are the Types of High Availability Databases?

High availability databases generally fall into two categories, with a third category becoming more common:

1. **Active-passive databases**: Where a database has an active node that processes requests with a hot spare that is ready to go in a disaster
2. **Active-active databases**: Where a database has active nodes that shard data and perform writes to the database
3. **Multi-active databases**: Where a database has at least three active nodes, each of which can perform reads and writes for any data in the cluster without generating conflicts.

What is Active-Passive Availability?

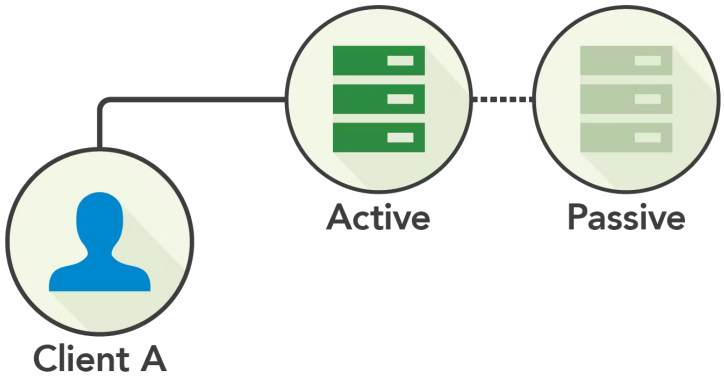
Active-passive availability means the database has an active node that processes requests with a hot spare that is ready to go in a disaster. The active-passive availability model works on the two-node

concept of one node receiving all requests that it then replicates to its follower.

In the days of yore, databases ran on single machines. There was only one node and it handled all reads and all writes. There was no such thing as a “partial failure”; the database was either up or down.

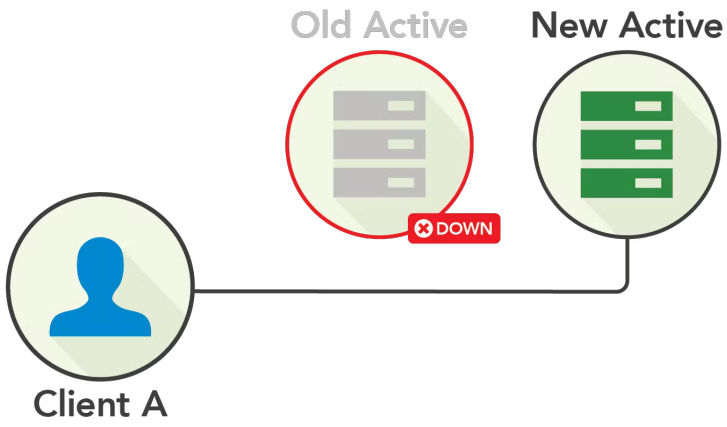
Total failure of a single database was a two-fold problem for the internet; first, computers were being accessed around the clock, so downtime was more likely to directly impact users; second, by placing computers under constant demand, they were more likely to fail. The obvious solution to this problem is to have more than one computer that can handle the request, and this is where the story of distributed databases truly begins.

Living in a single-node world, the most natural solution was to continue letting a single node serve reads and writes and simply sync its state onto a secondary, passive machine—and thus, Active-Passive replication was born.

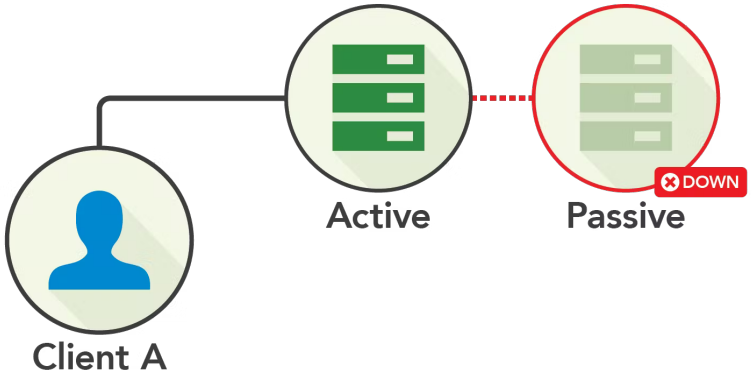


Active-Passive was an early step towards high availability with an up-to-date backup. In cases where the active node failed, you could simply start directing traffic to the passive node, thereby promoting it to being active. Whenever you could, you would replace the

downed server with a new passive machine (and hope the active one did not fail in the interim).



At first, replication from the active to the passive node was a synchronous procedure, i.e., transformations were not committed until the Passive node acknowledged them. However, it was unclear what to do if the passive node went down. It certainly didn't make sense for the entire system to go down if the backup system wasn't available—but with synchronous replication, that's what would happen.



To further improve availability, data could instead be replicated asynchronously. While its architecture looks the same, it was capable

of handling either the active or the passive node going down without impacting the database's availability.

While asynchronous Active-Passive was another step forward, there were still significant downsides:

- When the active node died, any data that wasn't yet replicated to the passive node could be lost—despite the fact that the client was led to believe the data was fully committed.
- By relying on a single machine to handle traffic, you were still bound to the maximum available resources of a single machine.

Chasing Five 9s High Availability: Scale to Many Machines

As the Internet proliferated, business' needs grew in scale and complexity. For databases this meant that they needed the ability to handle more traffic than any single node could handle, and that providing "always on" high availability became a mandate.

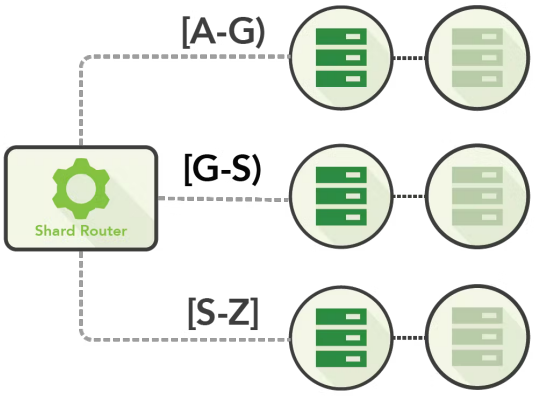
Given that swaths of engineers now had experience working on other distributed technologies, it was clear that databases could move beyond single-node Active-Passive setups and distribute a database across many machines.

Sharding

Again, the easiest place to start is adapting what you currently have, so engineers adapted Active-Passive replication into something more scalable by developing sharding.

In this scheme, you split up a cluster's data by some value (such as a number of rows or unique values in a primary key) and distributed those segments among a number of sites, each of which has an Active-Passive pair. You then add some kind of routing technology in

front of the cluster to direct clients to the correct site for their requests.



Sharding lets you distribute your workload among many machines, improving throughput, as well as creating even greater resilience by tolerating a greater number of partial failures and eliminating single points of failure.

Despite these upsides, sharding a system was complex and posed a substantial operational burden on teams. The deliberate accounting of shards could grow so onerous that the routing ended up creeping into an application's business logic. And worse, if you needed to modify the way a system was sharded (such as a schema change), it often posed a significant (or even monumental) amount of engineering to achieve.

Single-node Active-Passive systems had also provided transactional support (even if not strong consistency). However, the difficulty of coordinating transactions across shards was so knotted and complex, many sharded systems decided to forgo them completely.

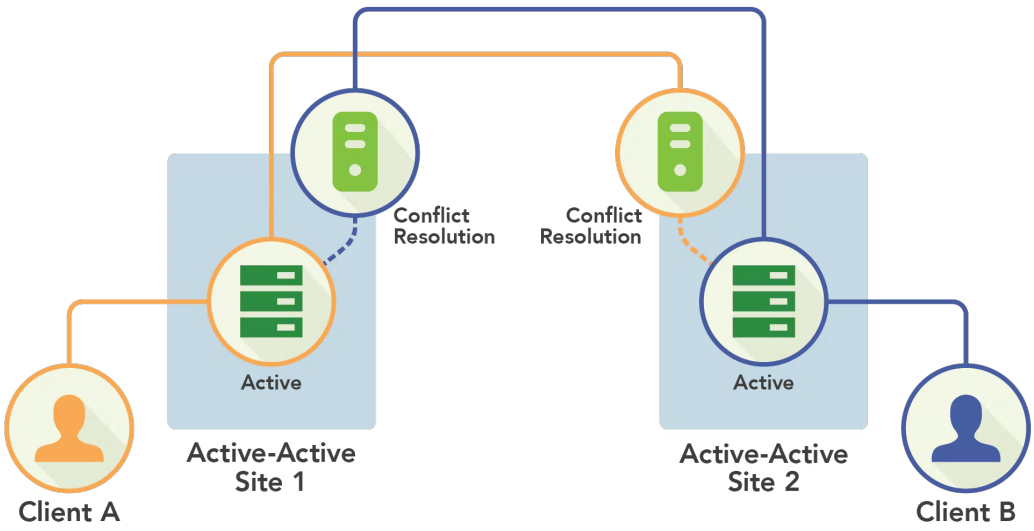
What is Active-Active Availability?

Active-active availability means a database has at least two active

nodes that shard data and perform writes to the database. Active-active availability represents an evolution from active-passive, enabling databases to scale beyond single machines by letting nodes in a cluster serve reads and writes.

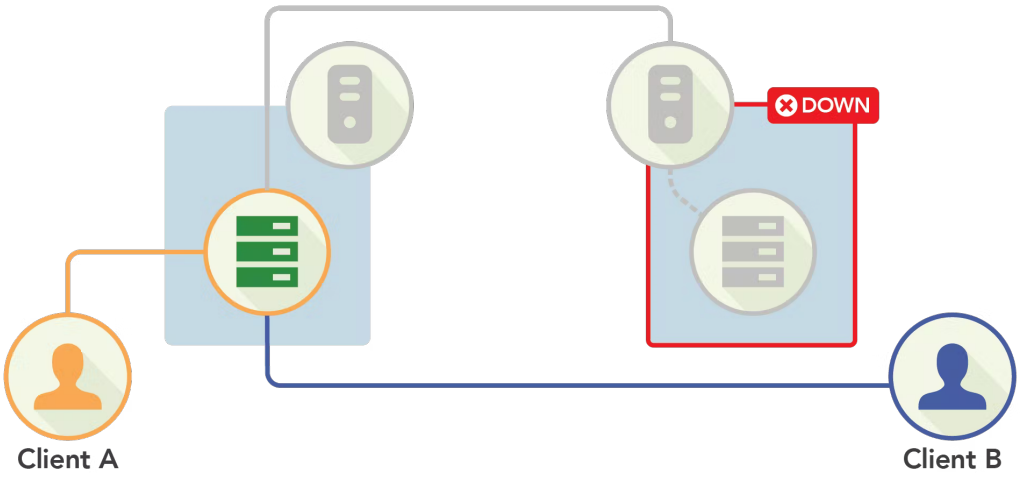
Given that sharded databases were difficult to manage and not fully featured, engineers began developing systems that would at least solve one of the problems. What emerged were systems that still didn't support transactions, but were dramatically easier to manage. With the increased demand on applications' uptime, it was a sensible decision to help teams meet their SLAs.

The motivating idea behind these systems was that each site could contain some (or all) of a cluster's data and serve reads and writes for it. Whenever a node received a write it would propagate the change to all other nodes that would need a copy of it. To handle situations where two nodes received writes for the same key, other nodes' transformations were fed into a conflict resolution algorithm before committing. Given that each site was "active", it was dubbed Active-Active.



Because each server could handle reads and writes for all of its data, sharding was easier to accomplish algorithmically and made deployments easier to manage.

In terms of availability, Active-Active was excellent. If a node failed, clients just needed to be redirected to another node that did contain the data. As long as a single replica of the data was live, you could serve both reads and writes for it.



While this scheme is fantastic for high availability, its design is fundamentally at odds with consistency and data correctness. Because each site can handle writes for a key (and would in a failover scenario), it's incredibly difficult to keep data totally synchronized as it is being processed. Instead, the approach is generally to mediate conflicts between sites through the conflict resolution algorithm that makes coarse-grained decisions about how to "smooth out" inconsistencies.

Because that resolution is done post hoc, after a client has already received an answer about a procedure—and has theoretically executed other business logic based on the response—it's easy for active-active replication to generate anomalies in your data.

Given the premium on uptime, though, the cost of downtime was deemed greater than the cost of potential anomalies, so Active-Active became the dominant replication type.

Correctness at Scale: Consensus

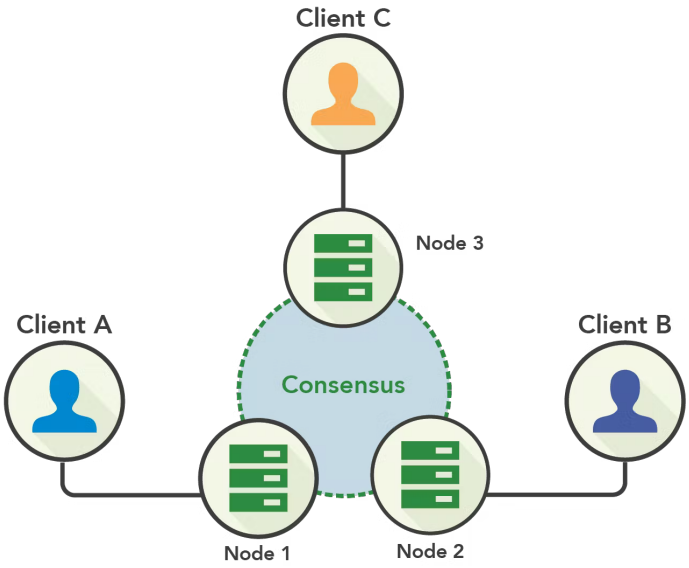
and Multi-Active Availability

Active-Active seemed like it addressed the major problem facing infrastructure — providing high availability. But it had only done so by forgoing transactions, which left systems that also required strong consistency without a compelling choice.

For example, Google used a massive and complex sharded MySQL system for its advertising business, which heavily relied on SQL's expressiveness to arbitrarily query the database. Because these queries often relied on secondary indexes to improve performance, they had to be kept totally consistent with the data they were derived from.

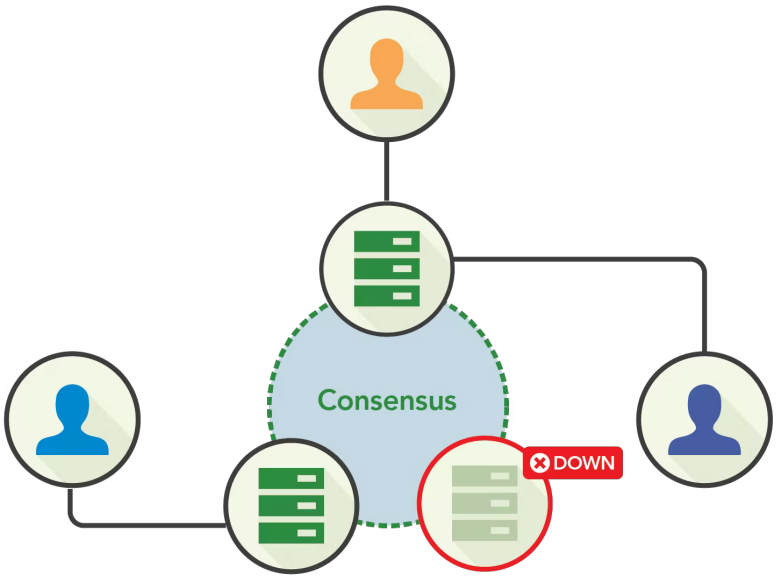
Eventually, the system grew large enough in size that it began causing problems for sharded MySQL (Spencer Kimball discusses his first-hand experience with sharded MySQL and AdWords on this podcast), so their engineers began imagining how they could solve the problem of having both a massively scalable system that could also offer the strong consistency their business required. Active-Active's lack of transactional support meant it wasn't an option, so they had to design something new. What they ended up with was a system based around consensus replication, which would guarantee consistency, but would also provide high availability.

Using consensus replication, writes are proposed to a node, and are then replicated to some number of other nodes. Once a majority of the nodes have acknowledge the write, it can be committed.



Consensus and High Availability

The lynch-pin notion here is that consensus replication lies in a sweet spot between synchronous and asynchronous replication: you need some arbitrary number of nodes to behave synchronously, but it doesn't matter which nodes those are. This means the cluster can tolerate a minority of nodes going down without impacting the system's availability. (Caveats made for handling the downed machines' traffic, etc.)



The cost of consensus, though, is that it requires nodes to communicate with others to perform writes. While there are steps

you can take to reduce the latency incurred between nodes, such as placing them in the same availability zone, this runs into trade-offs with high availability.

For example, if all of the nodes are in the same datacenter, it's fast for them to communicate with one another, but you cannot survive an entire datacenter going offline. Spreading your nodes out to multiple datacenters may increase the latency required for writes, but can improve your availability by letting an entire datacenter going offline without bringing down your application.

What is Multi-Active Availability?

Multi-active availability requires that a database has at least three active nodes, each of which can perform reads and writes for any data in the cluster without generating conflicts.

CockroachDB implements much of the learnings from the Google Spanner paper (though, notably, without requiring atomic clocks), including those features beyond consensus replication that make availability much simpler. To describe how this works and differentiate it from Active-Active, we've coined the term Multi-Active Availability.

Active-Active vs. Multi-Active

Active-Active achieves availability by letting any node in your cluster serve reads and writes for its keys, but propagates any changes it accepts to other nodes only *after* committing writes.

Multi-Active Availability, on the other hand, lets any node serve reads and writes, but ensures that a majority of replicas are kept in sync on writes (docs), and only serves reads from replicas of the latest version (docs).

In terms of high availability, Active-Active only requires a single replica to be available to serve both reads of writes, while Multi-Active requires a majority of replicas to be online to achieve consensus (which still allows for partial failures within the system).

Downstream of these databases' availability, though, is a difference of consistency. Active-Active databases work hard to accept writes in most situations, but then don't make guarantees about the ability for a client to then read that data now or in the future. On the other hand, Multi-Active databases accept writes only when it can guarantee that the data can later be read in a way that's consistent.

Where to from here?

Over the last 30 years, database replication and availability have taken major strides and now supports globe-spanning deployments that feel like they never go down. The field's first forays laid important groundwork through Active-Passive replication but eventually, we needed better availability and greater scale.

From there, the industry has developed two predominant paradigms of databases: Active-Active for applications whose primary concern is accepting writes quickly, and Multi-Active for those that require consistency.

May we all look forward to the day when we can harness quantum entanglement and move to the next paradigm in managing distributed state.

If defining the next phase of database replication and availability is your coffee-break daydream, then check out our open positions here.

Illustration by Christina Chung