cbtnuggets.com

AWS Architecture: High Availability vs Fault Tolerance

by Erik Mikac

8-10 minutes

Many of you have heard of—and thoroughly understand—the idea that time is money. For every minute that a product is unable to be created, advertised, or sold the bottom line will suffer. This concept is more important than ever in IT. Every minute a system goes down is loss of revenue and productivity. In fact, it is estimated that 98% of companies lose $100,000 for every hour of downtime. That's a lot of dough.

With AWS, teams can build highly resilient web applications whose downtime is next to never. They just have to make sure their systems are at least highly available or at most fault tolerant, both of which are disaster recovery methods.

It takes time and resources to ensure your systems are sufficiently resilient. However, that all pales in comparison to the potential losses incurred by downtime. That is why it is critical that your system is either highly available or fault tolerant. Let's explore how they work and how they are different.

What is High Availability?

High availability means that a system will almost always maintain uptime, albeit sometimes in a degraded state. With regard to AWS, a system has high availability when it has 99.999% uptime, also

known as "five nines". To put that in perspective, the system would be down for a mere five minutes and fifteen seconds a year. And yes, that is possible — and even routine — for AWS.

High availability is architected by the complete removing single points of failure by using system redundancy. For instance, if you had five computers connected to one server, that would be a single point of failure. If that server room flooded and destroyed the server, then you're out luck.

To mitigate this situation however, a backup server can be switched on in case of emergencies. This is adding redundancy to remove single points of failure. So, high availability removes single points of failure by adding redundancy.

In pre-AWS days, this was an expensive and costly situation to maintain. It was typically done by configuring complex RAIDs to ensure database redundancy. On top of that, hardware would have to be placed in temperature controlled, bomb-shelter-like structures that were expensive to maintain.

However, the abundance of cloud services have made high availability an affordable and realistic option for just about any organization. Let's take a deep dive on how we can use AWS to eliminate single points of failure and maximize our system's redundancy.

Example: A Mobile Banking App

Let's say that you are in charge of creating a mobile banking app. This app would enable users to view their account balance, make transfers, and withdraw funds. Optimally, the user should be able to do all three of these operations all the time. But in the real world something will fail — and that is where high availability architecture comes in.

In a highly available system, servers for our banking app will span

multiple availability zones. Within these availability zones are multiple servers that are configured in the same manner. (Servers on AWS are generally referred to as EC2 Instances.) That way, if one availability zone goes down, the mobile app will point to the other EC2 instances on the backup availability zone. This will eliminate any single point of failure.

What about the user data? That needs to be backed up too. As part of a redundant system, it ensures that the primary database is being replicated on a read-only database in a separate availability zone. If the primary availability zone goes down, users will be able to at least view their account balances. However they will be unable to withdraw funds or make transfers. That is because those are write operations, and the backup database is only set up for read operations.

Remember, high availability guarantees essential services, not full functionality. Telling people that once or twice a year they may encounter a situation where they can't deposit a check isn't too bad, though. Additionally, some of their banking data might not be as up to date, because our backup database only replicates once a day.

But what if a higher level of recovery is required? This is where fault tolerance comes into play.

What is Fault Tolerance?

Think of fault tolerance as high availability's older brother. Fault tolerance means that a system will almost always maintain uptime — and users will not notice anything different during a primary system outage.

If high availability was expensive in pre-AWS days, fault tolerance was exceedingly expensive. At a bare minimum, multiple servers would have to be load balanced, databases would have to be consistently replicated, and availability would have to span multiple

regions. All of this would need to be maintained by the company itself. Not only would the company have to foot the bill for all the hardware and expertise to run it, but they would have to follow esoteric IT security standards. Most non-IT companies simply aren't designed to handle this level of complexity.

AWS, however, takes care of all of these infrastructure requirements, allowing for a fault tolerant system without having to worry about the hardware itself. Let's revisit our mobile banking app and see how it can be architected for fault tolerance.

The Bank App Revisited

Currently, our mobile banking app has EC2 instances spanning multiple availability zones. The database has a read-only backup that is replicated once a day. This may be good enough for high availability, but fault tolerance sets a higher bar.

Instead of having a read-only database in another availability zone, an exact replica of the database would be housed there and possibly replicated to other regions. Every write operation to the database would essentially occur in parallel, ensuring the database is always up to date. In a fault tolerant system, every aspect of the production environment's state is maintained in a partitioned system. That way it is always ready in case of a disaster.

Another aspect of fault tolerance on AWS is to implement SQS (Simple Queue Service) to your database. Instead of writing directly to the database, API requests are placed into a queue. This will mitigate possible deadlocks and timeout to the Database, thus reducing any possibilities of a single point of failure. Remember that this, too, will have to be replicated in a separate environment to adhere to fault tolerance.

As you can imagine, having a replicated production environment can take significant time and resources to set up and maintain.

However the cost-benefit analysis will be in the favor of a fault tolerant system if the business deems the system highly critical.

High Availability vs. Fault Tolerance

Whether or not to utilize high availability over fault tolerance depends on your budget, and consequently the importance of the system. If you are running an e-commerce website with millions of hits a day, a fault tolerant system is your best bet.

According to Google, sites that load within 5 seconds have 70% longer sessions. That being said, high availability might not cut it, and you will need to architect a fault tolerant website. If the system is running in a degraded state, there is a good chance that you will lose customers either way. So, you may as well up the budget to accommodate fault tolerance capabilities.

On the other hand, let's say you are in charge of architecting a website for your employees, and it is not accessible from the internet. It's only accessible from the company's intranet. On this site, employees need to look up data 80% of the time and write to a database 20% of the time. In a situation like this, high availability would be perfectly acceptable.

In the event that the database server goes down, it would crossover to a read-only database. Now the employees can conduct 80% of their business unhindered until the primary system comes back up. As mentioned previously, SQS would be useful in this situation as well. The employees would be able to write to the database, but the request would simply be queued until the primary database is back up and running.

We need our employees to maintain productivity, so some amount of disaster recovery is vital. However it may not be worth the time or money to maintain a level of fault tolerance.

Final Thoughts

Verner Vogels, the CTO of Amazon, is quoted as saying "Failures are a given, and everything will eventually fail over time." This quote is brilliant in its simplicity. A good solutions architect embraces the fact that failure of systems is inevitable. Therefore disaster recovery should not be treated as a contingency plan, but as a matter of course.